CS2020 – Data Structures and Algorithms Accelerated

Lecture 19 – All-Pairs Shortest Paths

stevenhalim@gmail.com

National University of Singapore

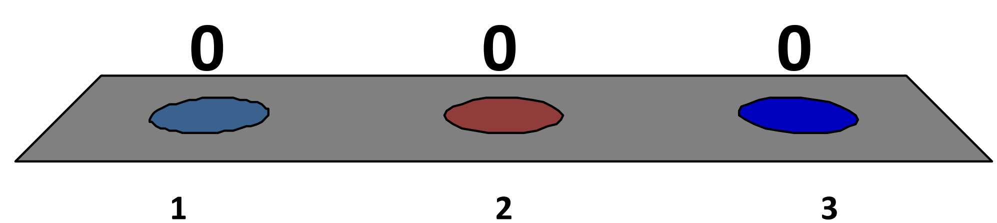
School of Computing

Outline

- What are we going to learn in this lecture?
  - Quick Review: the SSSP Problem
  - The All-Pairs Shortest Paths Problem
    - Some motivating examples
  - Floyd Warshall's Dynamic Programming algorithm
    - The code first ☺
    - The DP formulation (long one)
  - Some Interesting Variants

The SSSP problem is about…

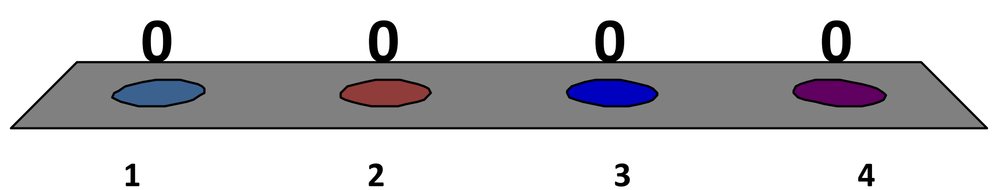
1. Finding the shortest path between a pair of vertices in the graph
2. Finding the shortest paths between any pair of vertices
3. Finding the shortest paths between one vertex to the other vertices in the graph



0 of 5

What is the best SSSP algorithm on (+ or -) weighted general graph but without non-negative weight cycle?

1. BFS
2. Original Dijkstra's
3. Modified Dijkstra's as shown in Lecture17
4. Bellman Ford's



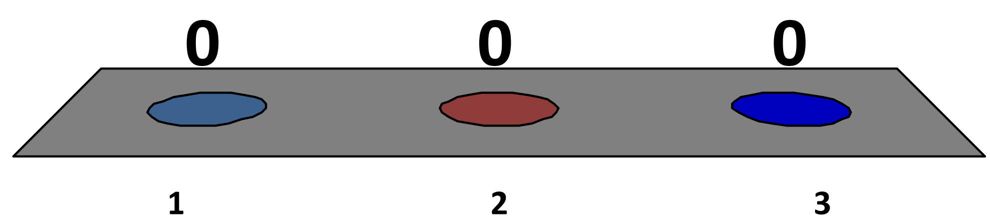
0 of 5

Let’s move on the the next topic

ALL-PAIRS SHORTEST PATHS

What is your knowledge level about APSP now?

1. I have not heard about this APSP problem or its solution before
2. I know this problem and its four liner Floyd Warshall's solution
3. I know how Floyd Warshall's algorithm works, not just how to code that four lines…

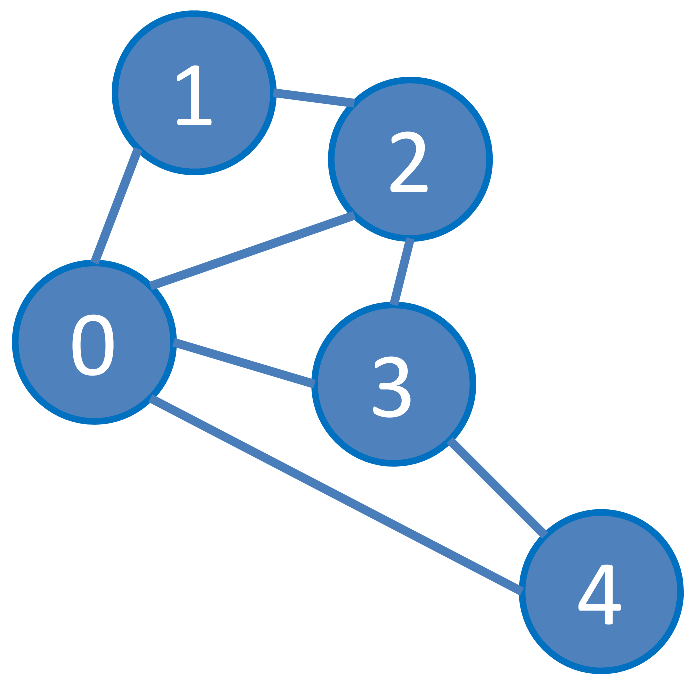


0 of 5

Motivating Problem 1

Diameter of a Graph

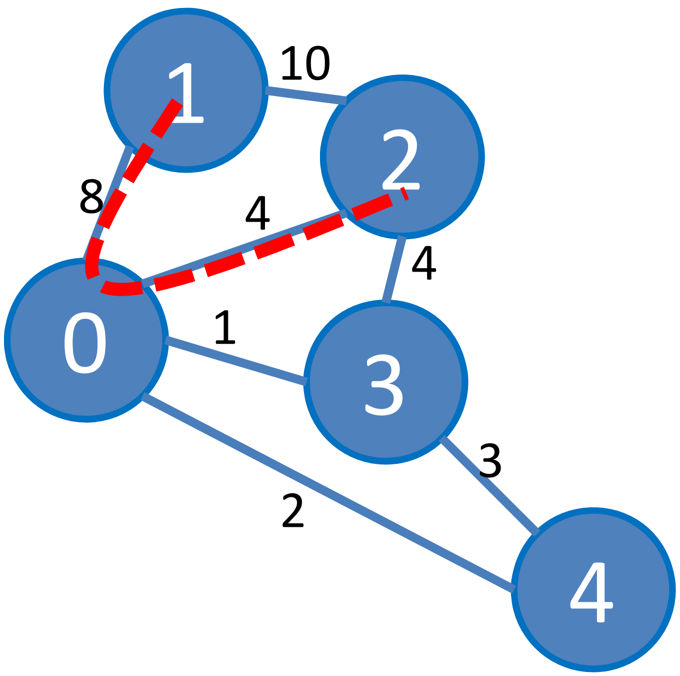
- The diameter of a graph is defined as the **greatest** *shortest path* distance between any pair of vertices
- For example, the diameter of this graph is **2**
  - Paths with length equal to diameter are:
    - 1-0-3 (or the reverse path)
    - 1-2-3 (or the reverse path)
    - 1-0-4 (or the reverse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



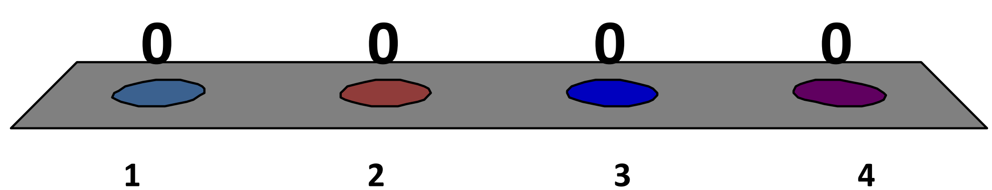
What is the diameter of this graph?

(you will need some time to calculate this)

1. 8, path = ____
2. 10, path = ____
3. 12, path = ____
4. 14, path = ____



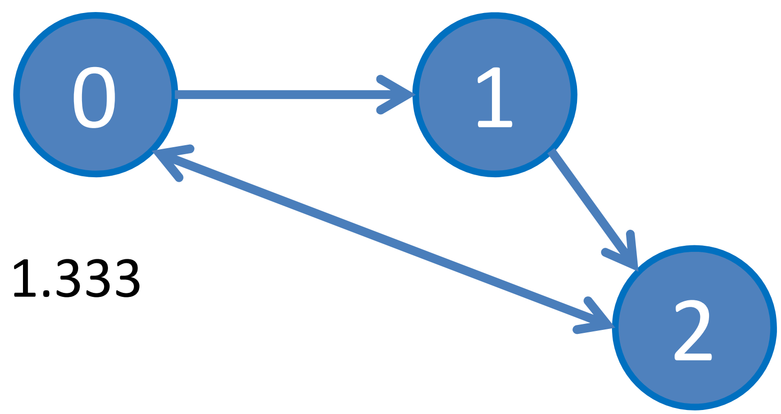
0 of 5



Motivating Problem 2

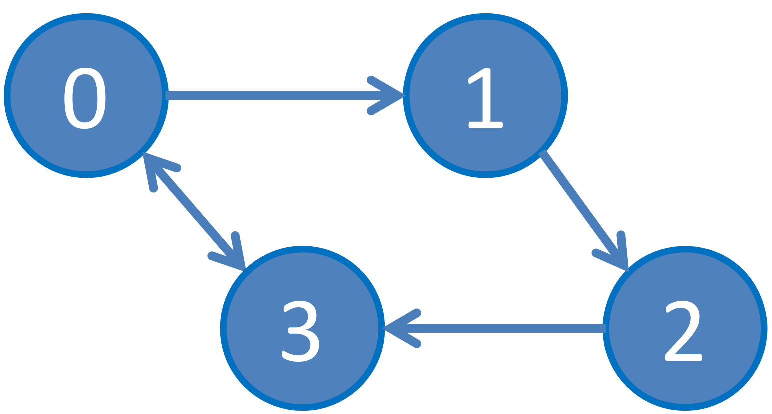
Analyzing the average number of clicks to browse WWW

- In year 2000, only 19 clicks are necessary to move from any page on the WWW to any other page :O
  - That is, if the pages on the web are viewed as vertices in a graph, then the average path length between **arbitrary pairs of vertices** in the graph is 19
  - For example, the average path length between arbitrary pair of vertices in this graph below is:
    - 0→1 = 1; 0→2 = 1
    - 1→0 = 2; 1→2 = 1
    - 2→0 = 1; 2→1 = 2
    - Average = (1+1+2+1+1+2) / 6 = 8 / 6 = 1.333

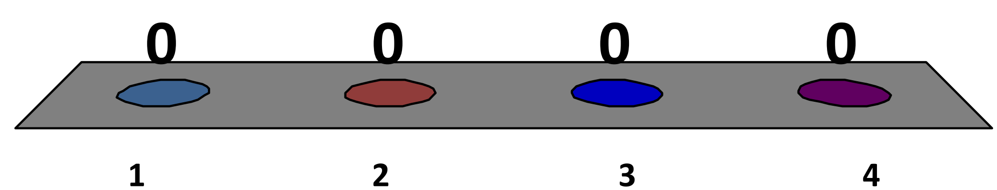


What is the average path length of this graph?
(you will need some time to calculate this)

1. 22/10 = 2.200
2. 22/12 = 1.833
3. 23/12 = 1.917
4. 24/12 = 2.000



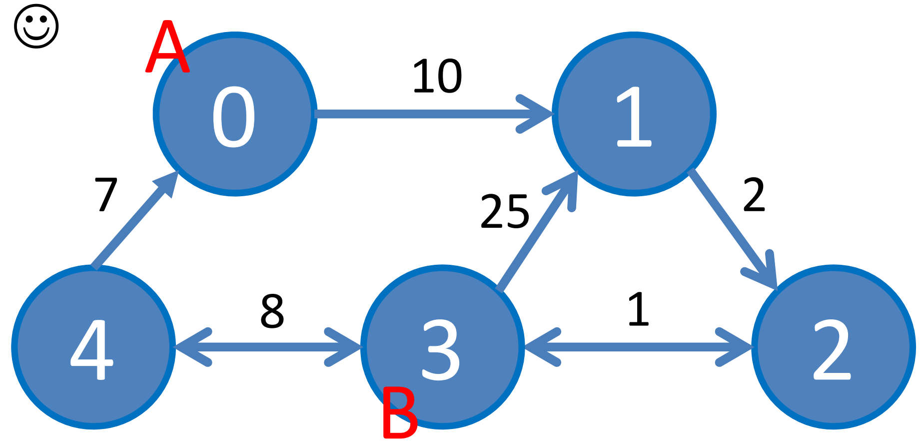
0 of 5



Motivating Problem 3

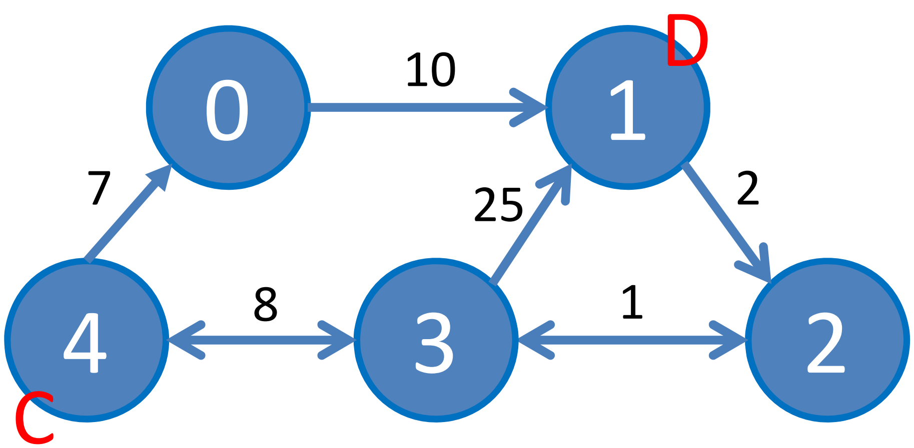
Finding the best meeting point

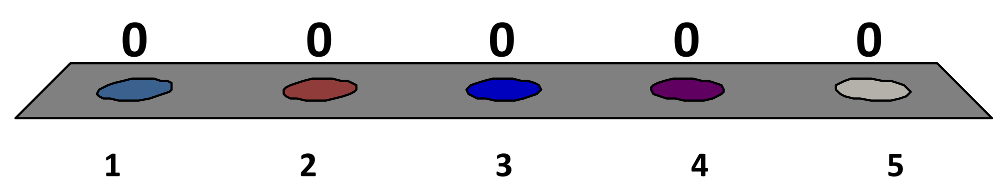
- Given a weighted graph that model a city and the travelling time between various places in that city
  - Find the best meeting point for two persons (there are **lots of** queries), one is currently in A and the other is in B
  - For example, the best meeting point between two persons currently in A = 0 and B = 3 is at vertex 2
    - B just need 1 unit of time to walk from 3→2 and then wait for A
    - A needs 12 units of time to walk from 0→2
    - After 12 units of time, they meet ☺



What is the best meeting point for C and D?
(you will need some time to calculate this)

1. Vertex 0, ___ units of time
2. Vertex 1, ___ units of time
3. Vertex 2, ___ units of time
4. Vertex 3, ___ units of time
5. Vertex 4, ___ units of time





0 of 5

All-Pairs Shortest Paths

- Problem definition:
  - Find shortest paths between any pair of vertices in the graph
- Several solutions from what we know earlier:
  - 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 Dijkstra's V times, once from each vertex
      - Time complexity: O(V * (V + E) * log V) = **$O(V^3 \log V)$** if E = $O(V^2)$
    - Call Bellman Ford's V times, once from each vertex
      - Time complexity: O(V * VE) = **$O(V^4)$** if E = $O(V^2)$

Floyd Warshall's – Sneak Preview

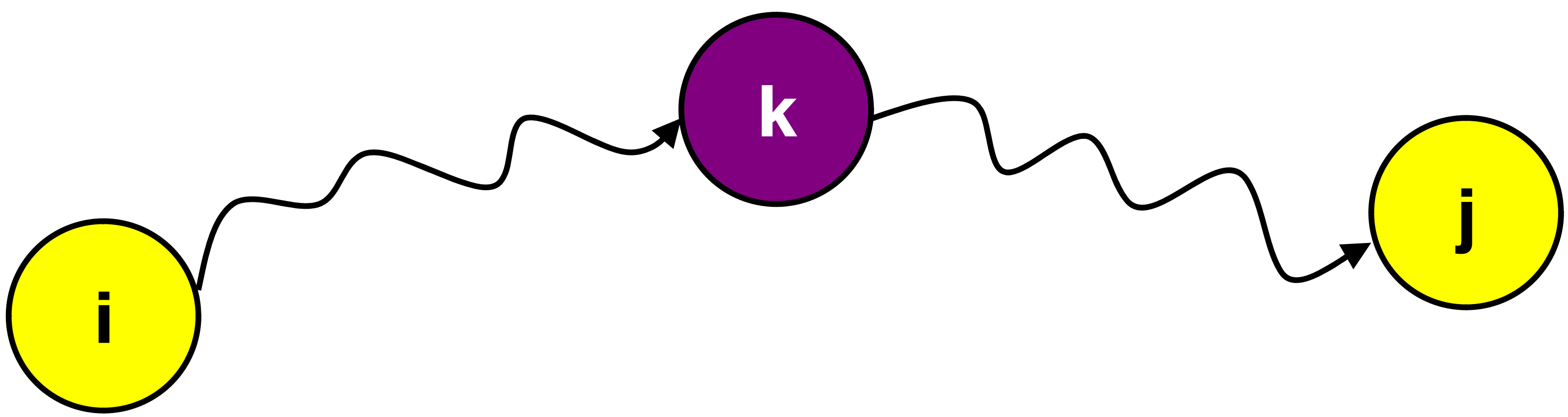
- We use an Adjacency Matrix: `D[|V|][|V|]`
  - Originally D[i][j] contains the weight of **edge(i, j)**
  - After Floyd Warshall's stop, it contains the weight of **path(i, j)**
  - It is usually a nice algorithm for the **pre-processing** part ☺

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

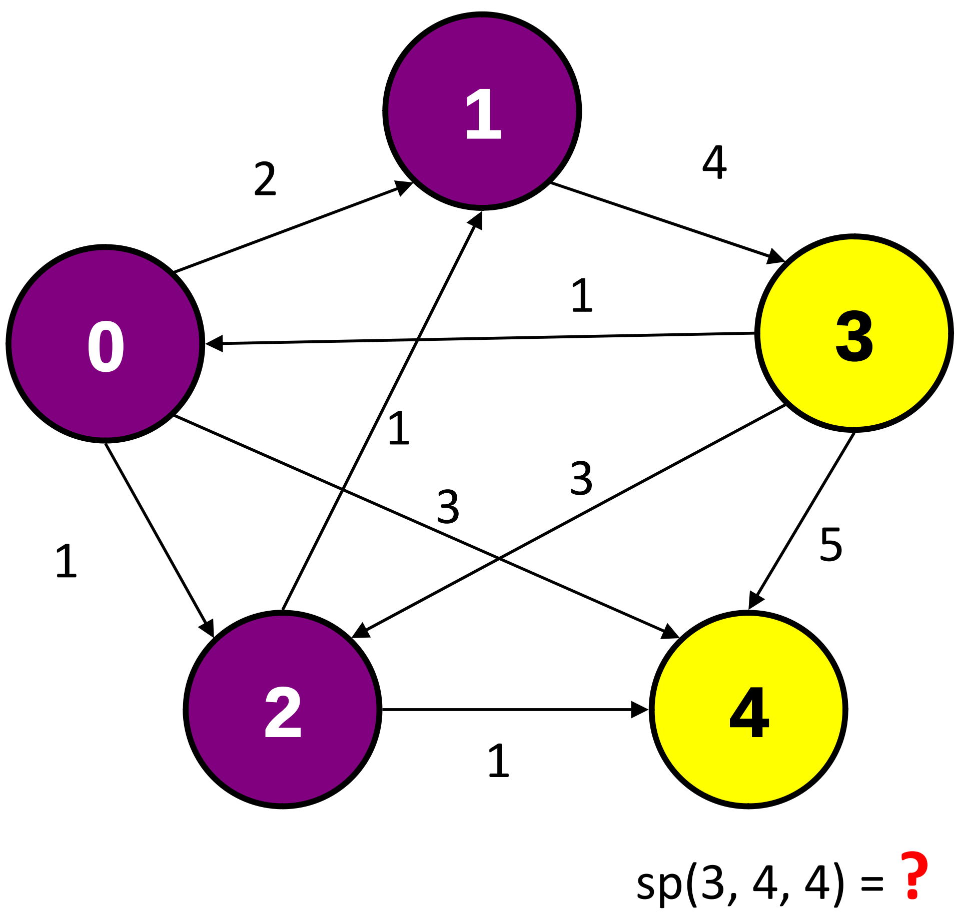
- $O(V^3)$ since we have three nested loops!
  - Apparently, if we only given a short amount of time, we can only solve the APSP problem for small graph, as none of the APSP solution shown here runs better than $O(V^3)$…

Floyd Warshall's – Basic Idea (1)

- 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** until k = V - 1



Floyd Warshall's – Basic Idea (2)

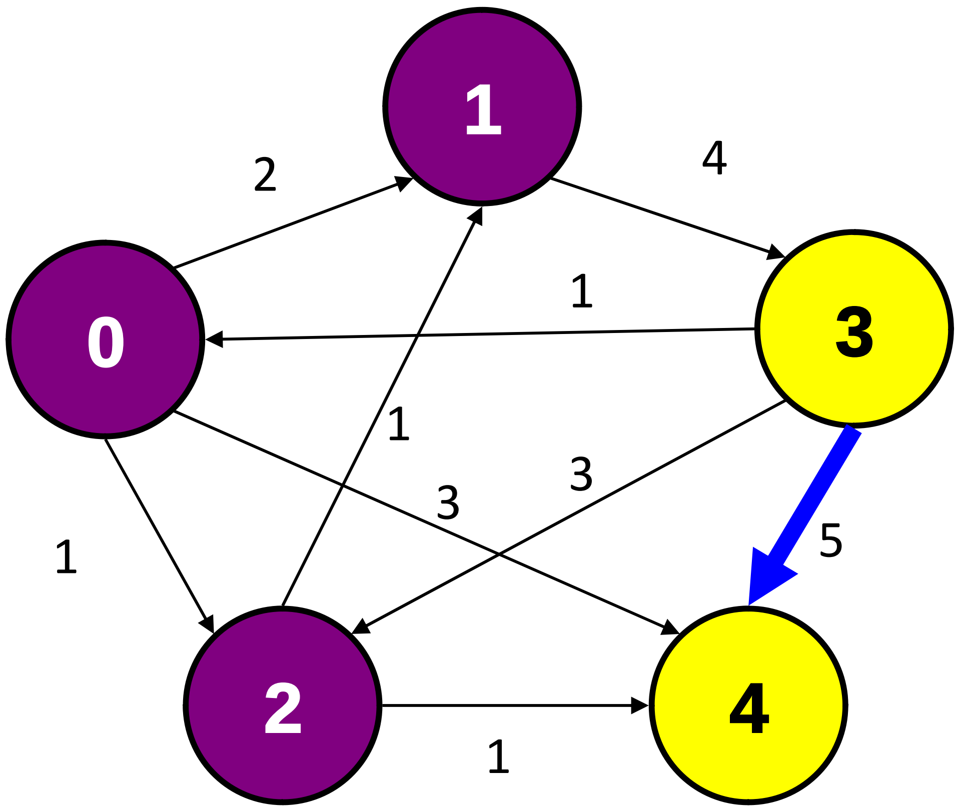


Suppose we want to know the shortest path between vertex 3 and 4, using any intermediate vertices from k = [0 .. 4], i.e. sp(3, 4, 4)

Floyd Warshall's – Basic Idea (3)

Direct Edges Only

i = 3, j = 4, k = -1



sp(3, 2, -1) = **3** sp(2, 4, -1) = **1**

sp(3, 4, -1) = **5**

We will monitor these two values

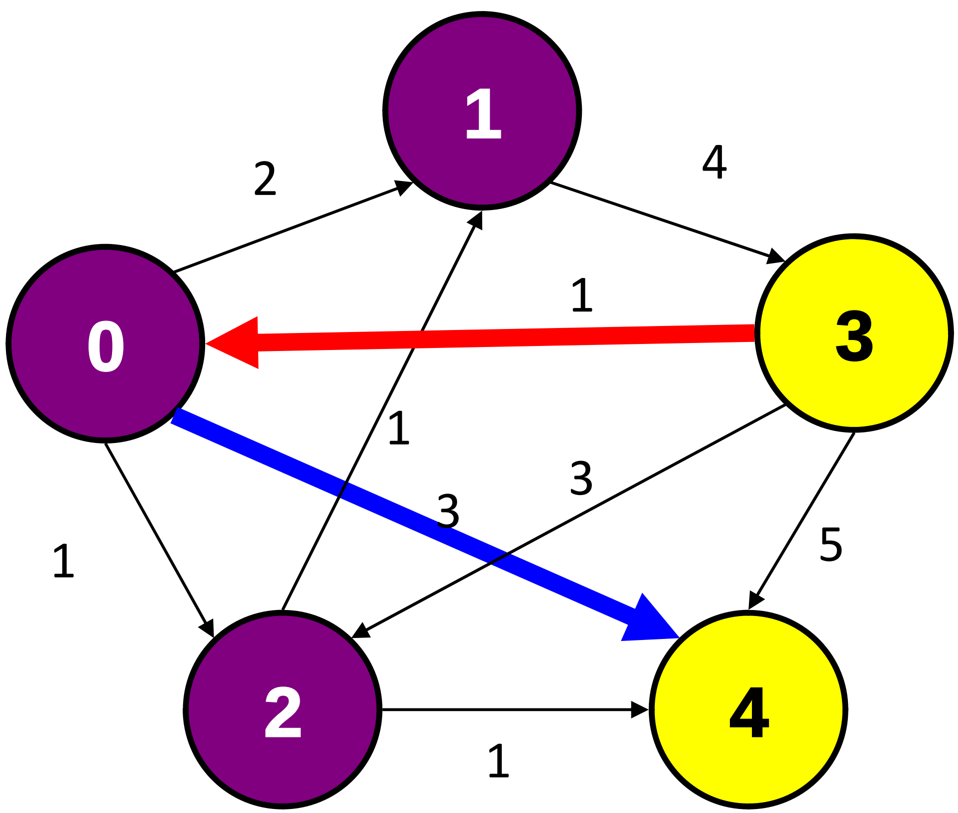
The current content of Adjacency Matrix D at **k = -1**

k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	3	3	0	**5**
4	∞	∞	∞	∞	0

Floyd Warshall's – Basic Idea (4)

Vertex 0 is allowed

i = 3, j = 4, k = 0



sp(3, 2, 0) = **2** sp(2, 4, 0) = **1** sp(3, 4, 0) = **4**

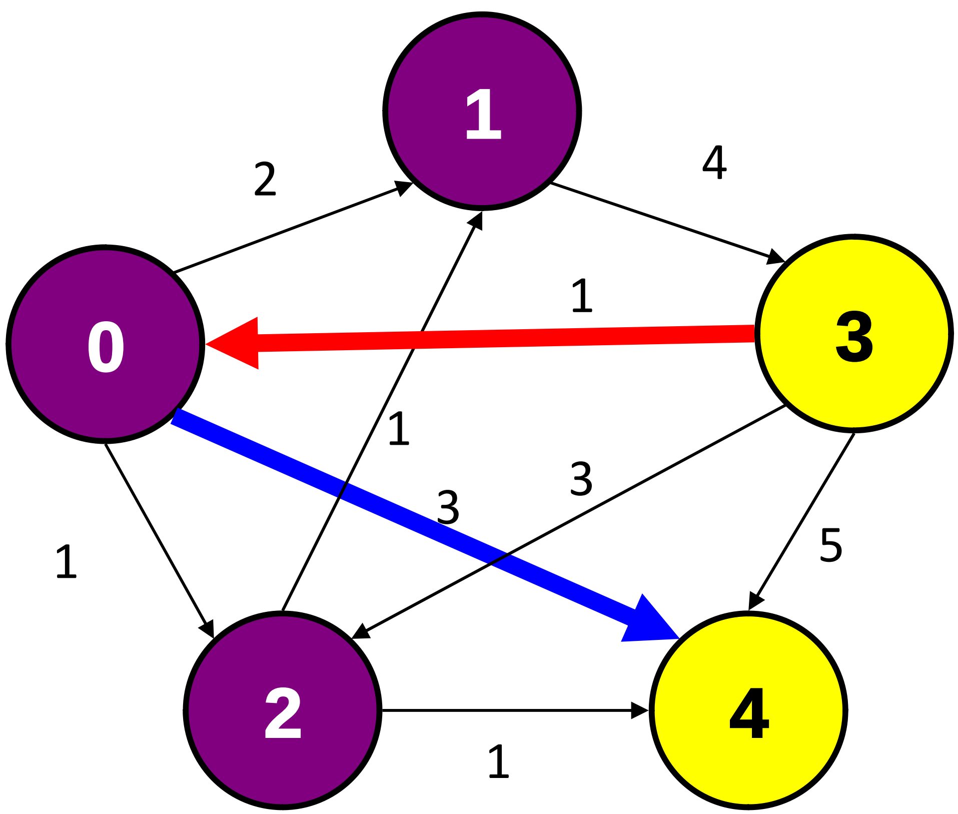
The current content of Adjacency Matrix D at **k = 0**

k = 0	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

Floyd Warshall's – Basic Idea (4)

Vertices 0-1 are allowed

i = 3, j = 4, k = 1



sp(3, 2, 1) = **2** sp(2, 4, 1) = **1** sp(3, 4, 1) = **4**

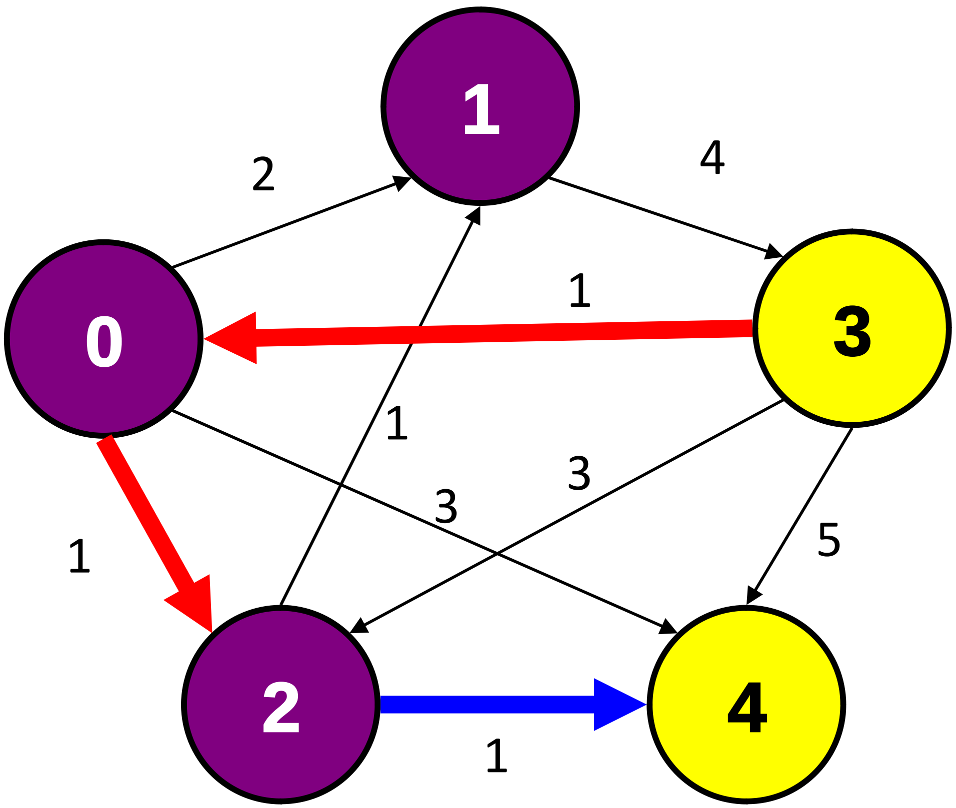
The current content of Adjacency Matrix D at **k = 1**

k = 1	0	1	2	3	4
0	0	2	1	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	**4**
4	∞	∞	∞	∞	0

Floyd Warshall's – Basic Idea (5)

Vertices 0-2 are allowed

i = 3, j = 4, k = 2



sp(3, 2, 2) = **2** sp(2, 4, 2) = **1** sp(3, 4, 2) = **3**

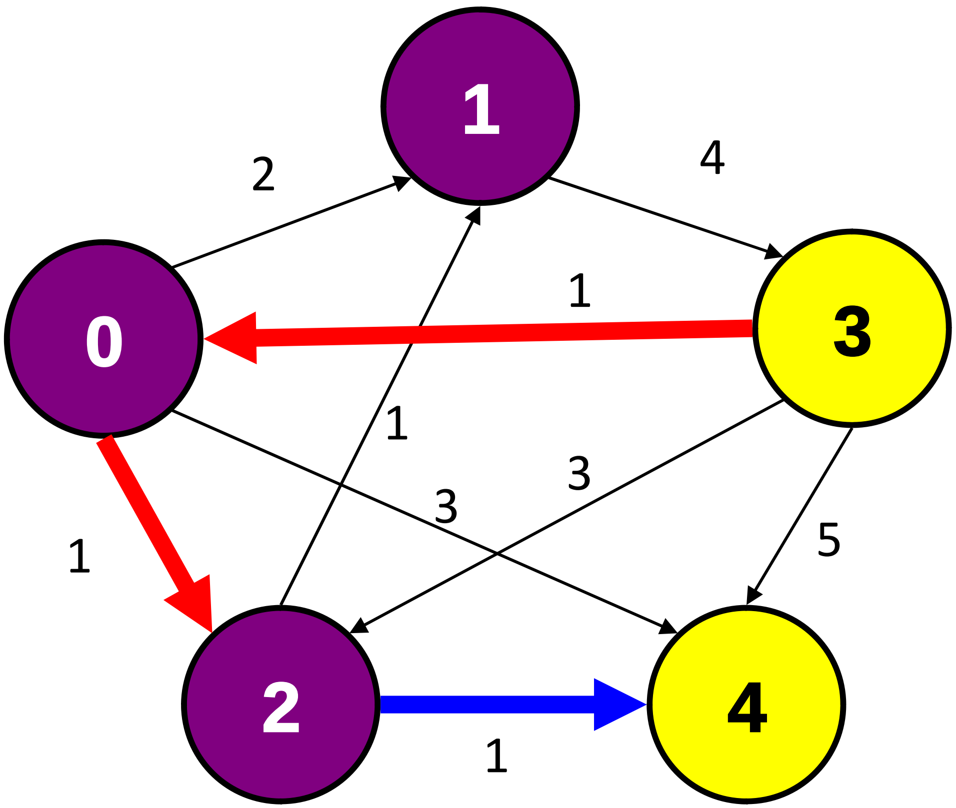
The current content of Adjacency Matrix D at **k = 2**

k = 2	0	1	2	3	4
0	0	2	1	6	**2**
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	**3**
4	∞	∞	∞	∞	0

Floyd Warshall's – Basic Idea (6)

Vertices 0-3 are allowed

i = 3, j = 4, k = 3



sp(3, 2, 2) = **2** sp(2, 4, 2) = **1** sp(3, 4, 2) = **3**

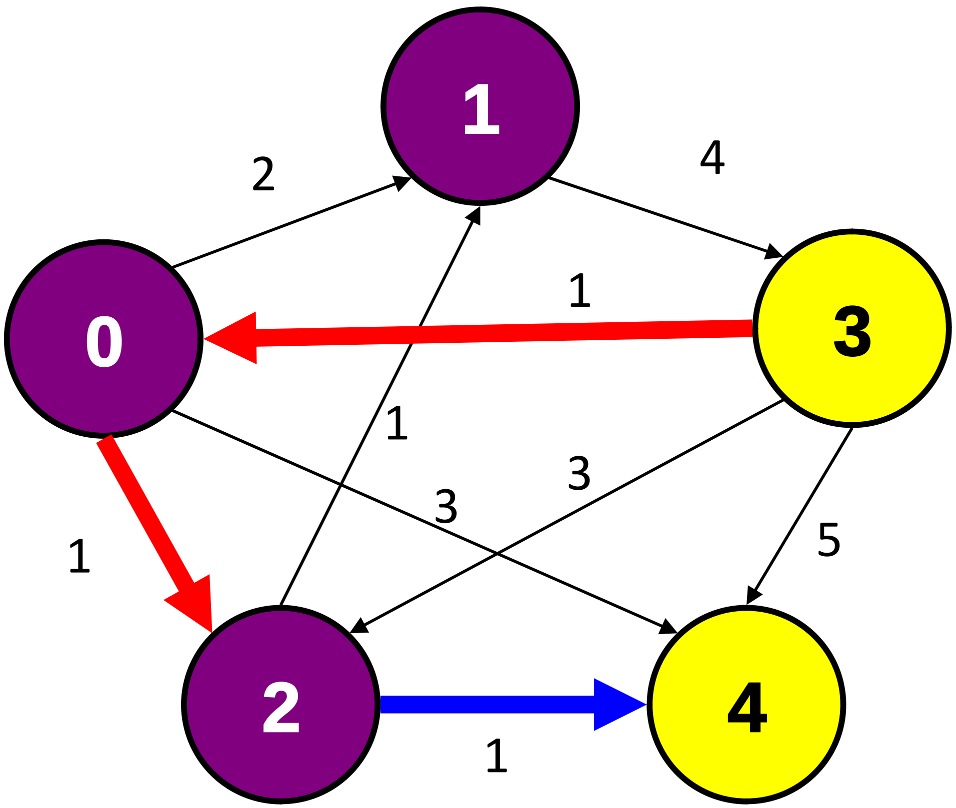
The current content of Adjacency Matrix D at **k = 3**

k = 3	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

Floyd Warshall's – Basic Idea (7)

Vertices 0-3 are allowed

i = 3, j = 4, k = 3



sp(3, 2, 2) = **2** sp(2, 4, 2) = **1** sp(3, 4, 2) = **3**

The current content of Adjacency Matrix D at **k = 4**

k = 4	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

Floyd Warshall's – DP (1)

Recursive Solution / Optimal Sub structure

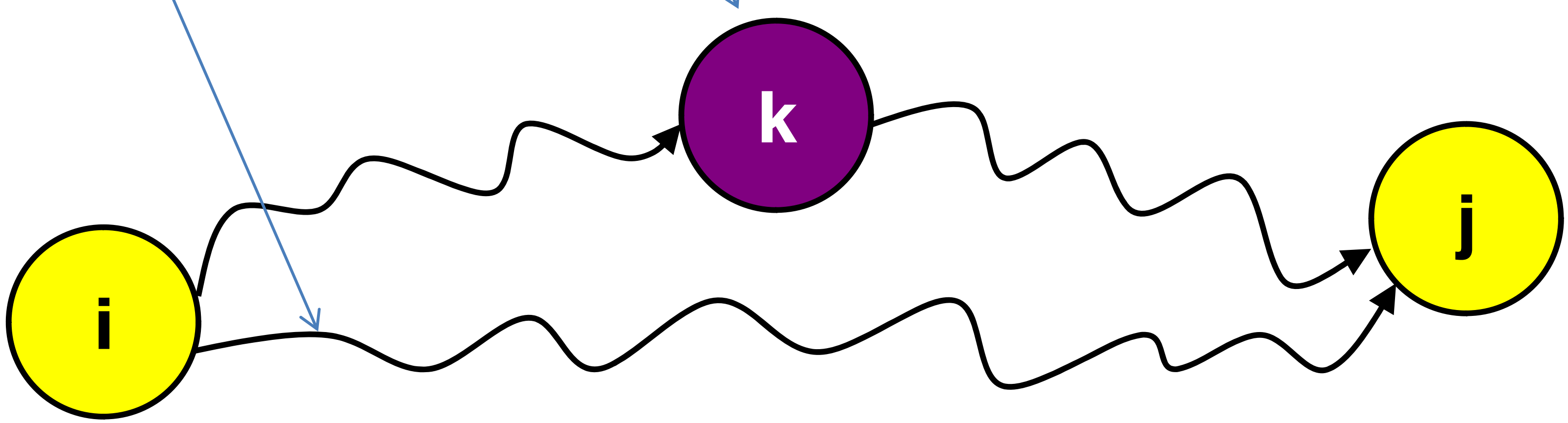
$D_{i,j}^{-1}$: Edge weight of the original graph

$D_{i,j}^{k}$: Shortest distance from i to j involving [0 .. k] only as intermediate vertices

$$D_{i,j}^{k} = \begin{cases} w_{i,j} & \text{for k = -1} \\ \min(D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1}) & \text{for k} \geq 0 \end{cases}$$

Not using vertex k

Using vertex k



Floyd Warshall's – DP (2)

Overlapping Sub problems

- Avoiding re-computation: To fill out an entry in the table **k**, we make use of entries in table **k - 1**

k **j**

k = 1	**0**	**1**	**2**	**3**	**4**
0	0	2	1	6	**3**
1	∞	0	∞	4	∞
k **2**	∞	1	0	5	1
i **3**	1	3	2	0	**4**
4	∞	∞	∞	∞	0

k = 1

j

k = 2	**0**	**1**	**2**	**3**	**4**
0	0	2	1	6	**2**
1	∞	0	∞	4	∞
2	∞	1	0	5	1
i **3**	1	3	2	0	**3**
4	∞	∞	∞	∞	0

k = 2

Floyd Warshall's – DP (3)

The Final Code

```
int[][] D = new int[V][V]; // 2D adjacency matrix
for (int i = 0; i < V; i++) { // initialization phase
  Arrays.fill(D[i], 1000000000); // cannot use nCopies
  D[i][i] = 0;
}
for (int i = 0; i < E; i++) { // direct edges
  u = sc.nextInt(); v = sc.nextInt(); w = sc.nextInt();
  D[u][v] = w; // directed weighted edge
}
// main loop, O(V^3)
for (int k = 0; k < V; k++) // be careful, put k first
  for (int i = 0; i < V; i++) // before i
    for (int j = 0; j < V; j++) // and then j
      D[i][j] = Math.min(D[i][j], D[i][k] + D[k][j]);
```

Floyd Warshall's algorithm…

1. Code looks easy, but I still do not understand the DP formulation
2. I understand both the code and the DP formulation ☺

0 of 5

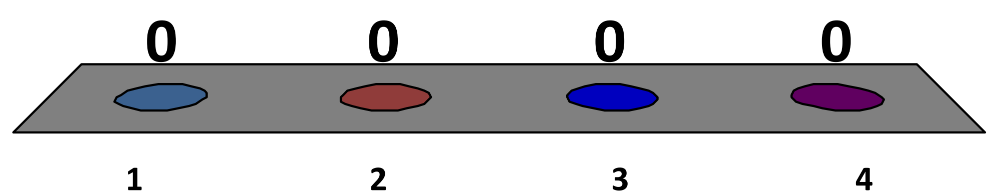
10 minutes break, and then…

VARIANTS OF FLOYD WARSHALL’S

Only for those who already know Floyd Warshall's algorithm before

you can select up to 4 times

1. I have used it to compute the actual all-pairs shortest path, not just the shortest path length
2. I have used it for transitive closure
3. I have used it for minimax/ maximin (Quiz 2 ☺)
4. I have used it to compute "safest path"



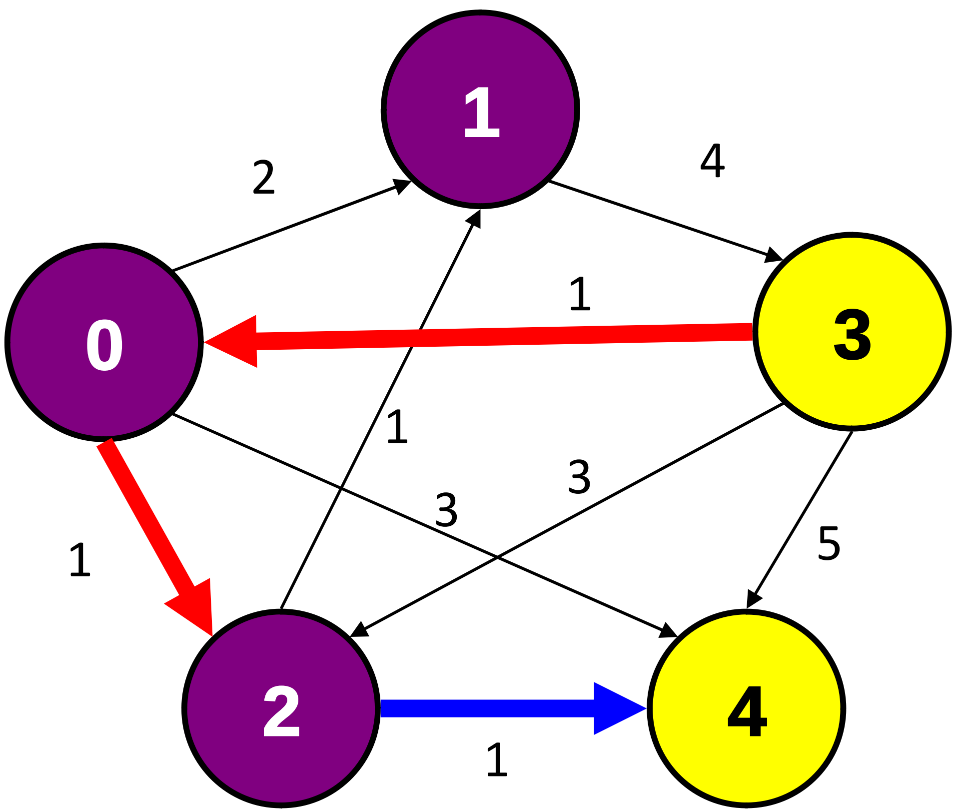
0 of 5

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 last vertex before **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**
  - 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 last vertex before **j** in the shortest path

Variant 1 – Print the Actual SP (2)

- The two matrices, **D** and **p**
  - Shortest path from 3 ~→ 4
  - 3→0→2→4



D	**0**	**1**	**2**	**3**	**4**
0	0	2	1	6	3
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

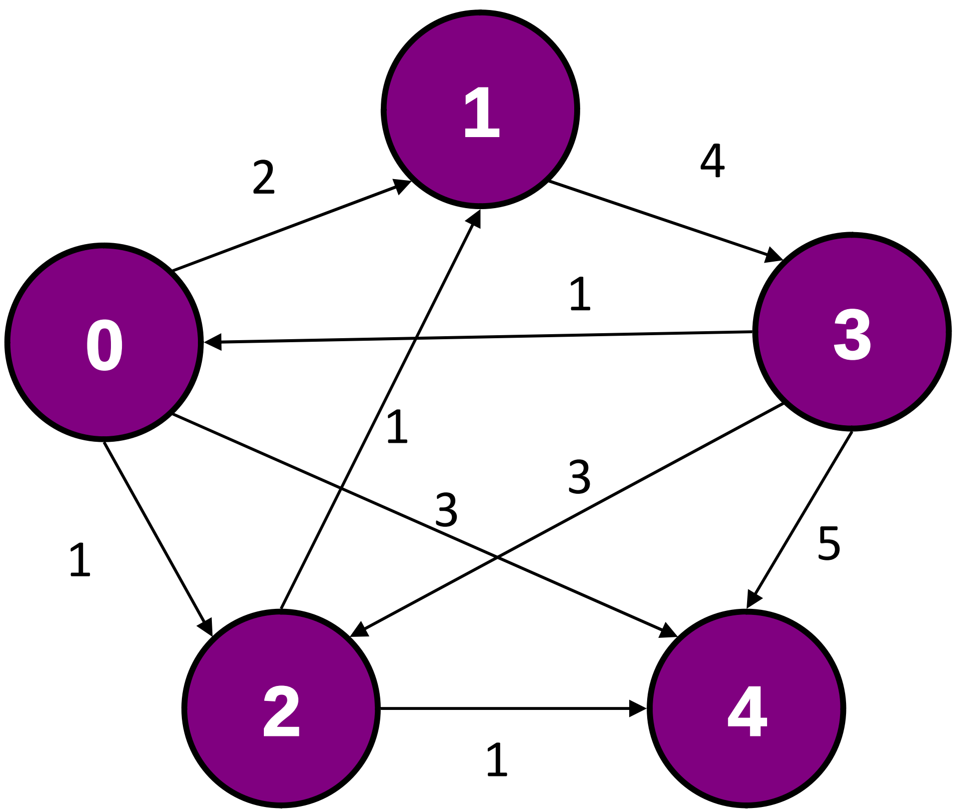
Variant 2 – Transitive Closure (1)

- Stephen Warshall actually invented this algorithm 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
  - In the main loop of 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]);
```

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

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

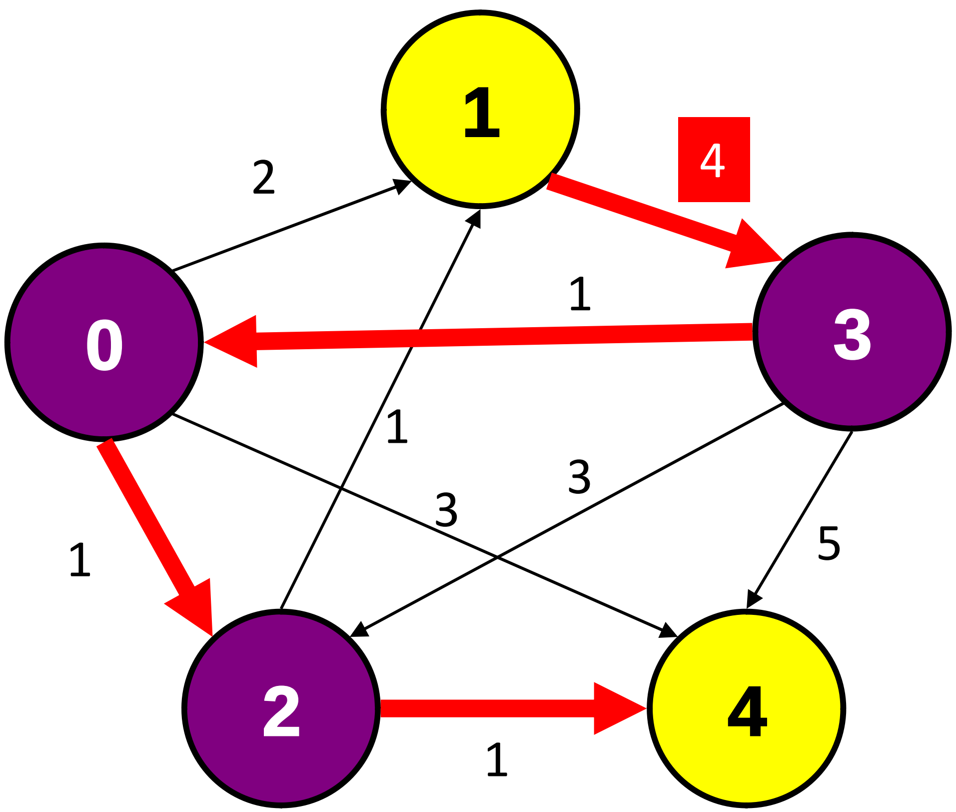
Variant 3 – Minimax/Maximin (1)

- The minimax problem is a problem of finding the minimum of maximum edge weight along all possible paths 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 one with the minimum max-edge-weight
- Solution: again, 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]));
```

Variant 3 – Minimax/Maximin (2)

- The minimax 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

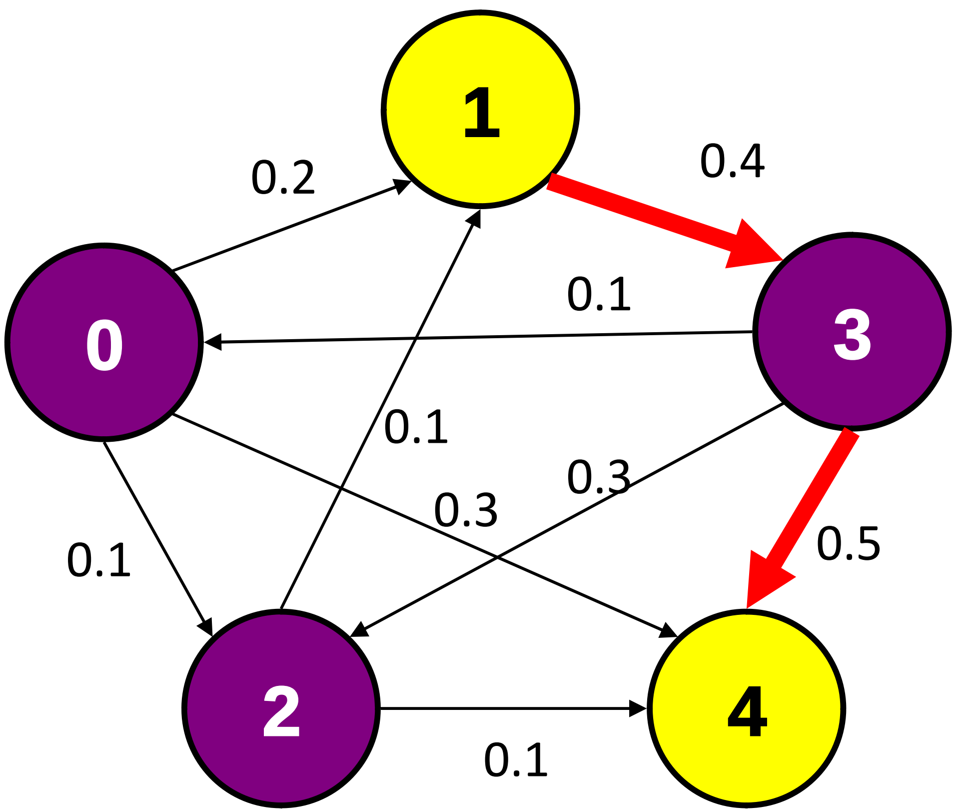
Variant 4 – Safest Paths (1)

- Given a directed graph where the edge weights represent the survival probabilities of passing through that edge, your task is to compute the safest path between two vertices
  - i.e. the path that maximizes the *product of probabilities* along the path
- Solution: again, modification of Floyd Warshall's

```
// Initially, D[i][i] = 1.0
// D[i][j] = weight(i, j), 0.0 otherwise
// the three nested loops as per normal
D[i][j] = Math.max(D[i][j], D[i][k] * D[k][j]);
```

Variant 4 – Safest Paths (2)

- The safest path from 1 to 4 is 0.20, via this path
  - 1→3→4



D,init	0	1	2	3	4
0	1.00	0.20	0.10	0.00	0.30
1	0.00	1.00	0.00	0.40	0.00
2	0.00	0.10	1.00	0.00	0.10
3	0.10	0.00	0.30	1.00	0.50
4	0.00	0.00	0.00	0.00	1.00

D,final	0	1	2	3	4
0	1.00	0.20	0.10	0.08	0.30
1	0.04	1.00	0.12	0.40	0.20
2	0.00	0.10	1.00	0.04	0.10
3	0.10	0.03	0.30	1.00	0.50
4	0.00	0.00	0.00	0.00	1.00

Java Implementations

- Let’s see: FloydWarshallDemo.java
- Let’s see how easy to change the basic form of Floyd Warshall’s algorithm to its variants

Summary

- In this lecture, we have seen:
  - Introduction to the APSP problem
  - Introduction to the Floyd Warshall's DP algorithm
  - Introduction to 4 variants of Floyd Warshall's
  - Simple Java implementations
- This lecture is again not yet DP-heavy
  - Floyd Warshall's is a DP algorithm,
    but many just view this as "another graph algorithm"
- Next week is the (pure) DP week… get ready ☺