CS2010 – Data Structures and Algorithms II

Lecture 12 – Finding Your Way
from Any Point to Another Point (Part III)

stevenhalim@gmail.com

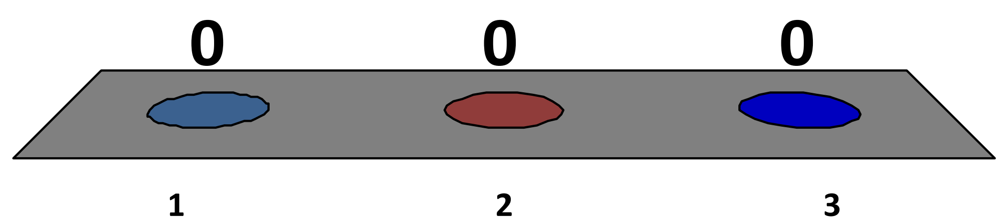
National University of Singapore
School of Computing

Outline

- What are we going to learn in this lecture?
  - (We will finish off DP TSP discussion first
    - Especially the discussion of *bitmask* data structure
  - Quick Review: The **Single-Source** Shortest Paths Problem
  - Introducing: The **All-Pairs** Shortest Paths Problem
    - With some motivating examples
  - Floyd Warshall's **Dynamic Programming** algorithm
    - The short code first ☺
    - Then the DP formulation (long one)
  - Some Interesting Variants of Floyd Warshall's
  - (If still have time, Quiz 2 review, full details on Week12)

The SSSP problem is about…

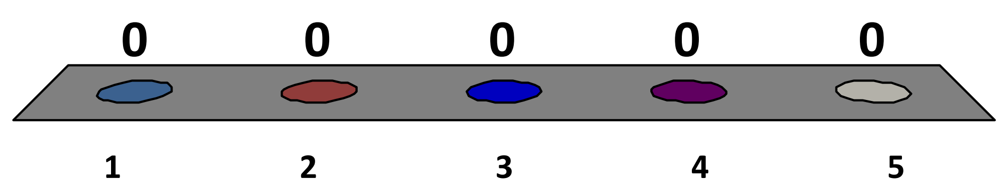
1. Finding the shortest path between a pair of vertices in the graph
2. Finding the shortest paths between any pair of vertices
3. Finding the shortest paths between one vertex to the other vertices in the graph



0 of 120

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

1. DFS
2. BFS
3. Original Dijkstra’s
4. Modified Dijkstra’s
5. Bellman Ford’s



0 of 120

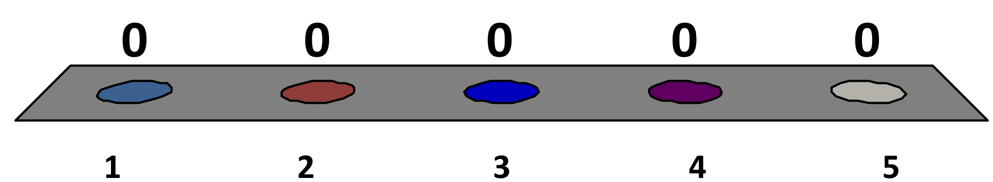
Let’s move on to the next topic

ALL-PAIRS SHORTEST PATHS

What is your knowledge level about APSP now?

(each clicker can select up to 3 times)

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 used it for PS4 :O
4. I used it for PS7, eh? :O:O
5. I know how Floyd Warshall's algorithm works, not just how to code that four lines…

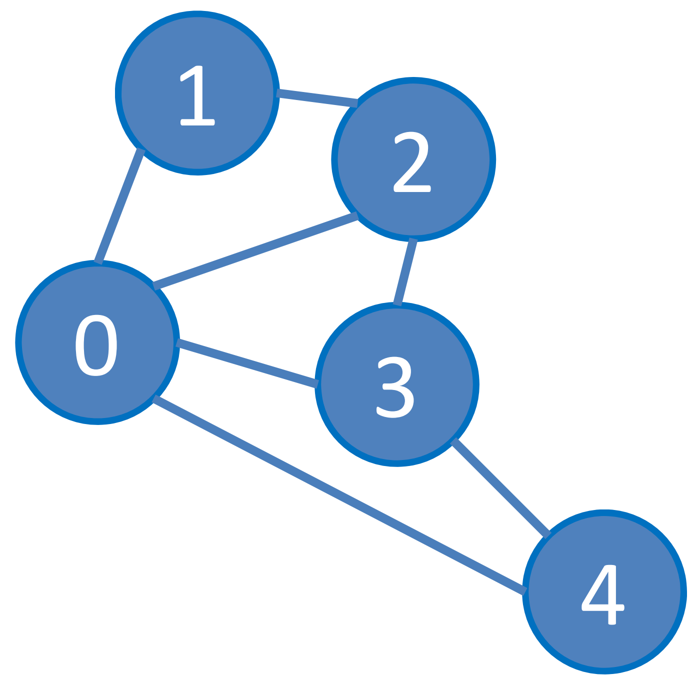


0 of 120

Motivating Problem 1

Diameter of a Graph

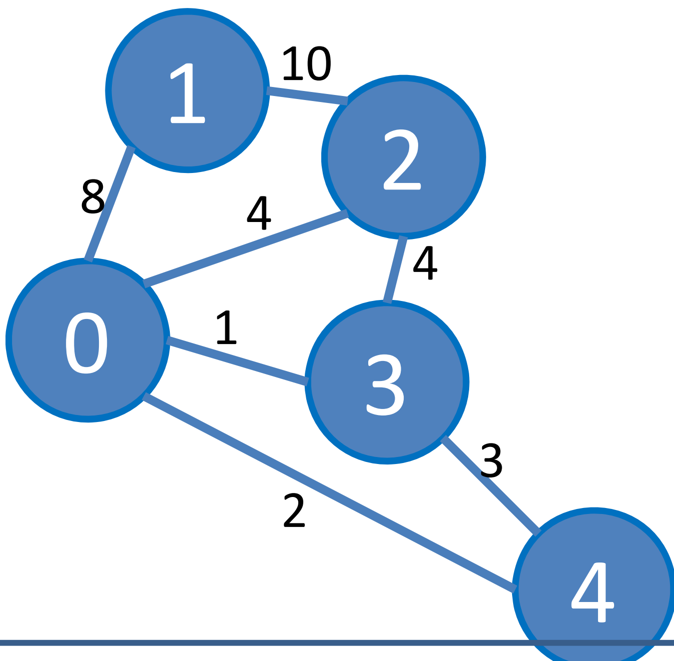
- The diameter of a graph is defined as the **greatest** *shortest path distance* between any pair of vertices
- For example, the diameter of this graph is **2**
  - The paths with length equal to diameter are:
    - 1-0-3 (or the reverse path)
    - 1-2-3 (or the reverse path)
    - 1-0-4 (or the reverse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



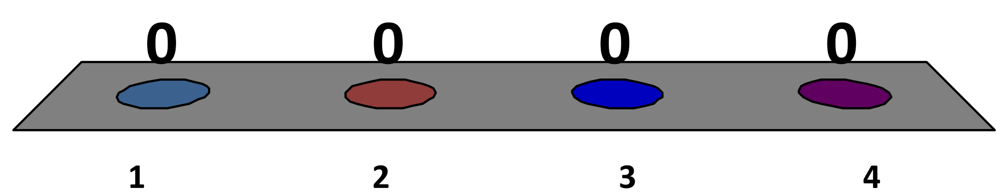
What is the diameter of this graph?

(you will need some time to calculate this)

1. 8, path = ____
2. 10, path = ____
3. 12, path = ____
4. I do not know ☹…



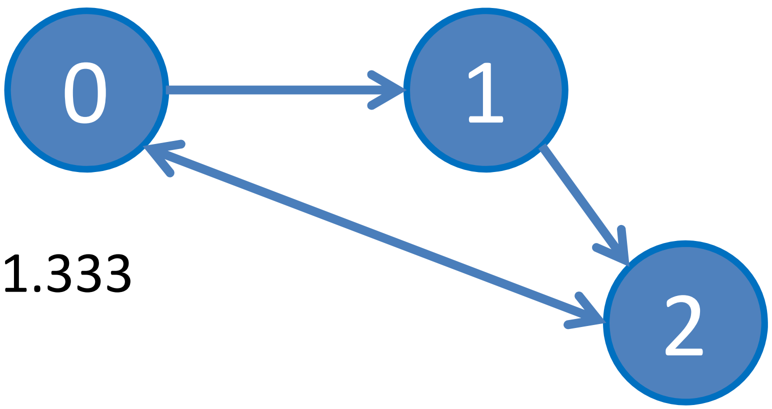
0 of 120



Motivating Problem 2

Analyzing the average number of clicks to browse the WWW

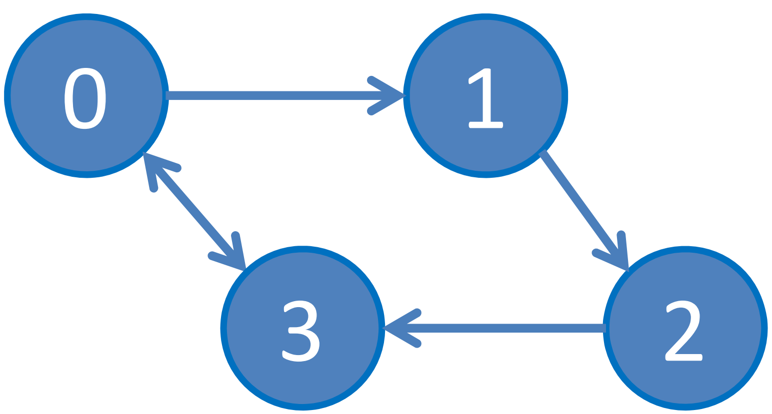
- In year 2000, only 19 clicks are necessary to move from any page on the WWW to any other page :O
  - That is, if the pages on the web are viewed as vertices in a graph, then the average path length between **arbitrary pairs of vertices** in the graph is 19
  - For example, the average path length between 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



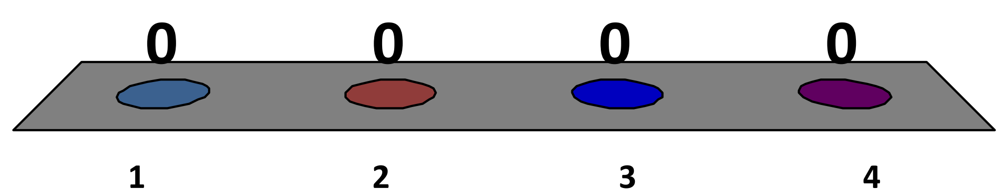
What is the average path length of this graph?

(you will need some time to calculate this)

1. 22/10 = 2.200
2. 22/12 = 1.833
3. 23/12 = 1.917
4. I do not know ☹ …



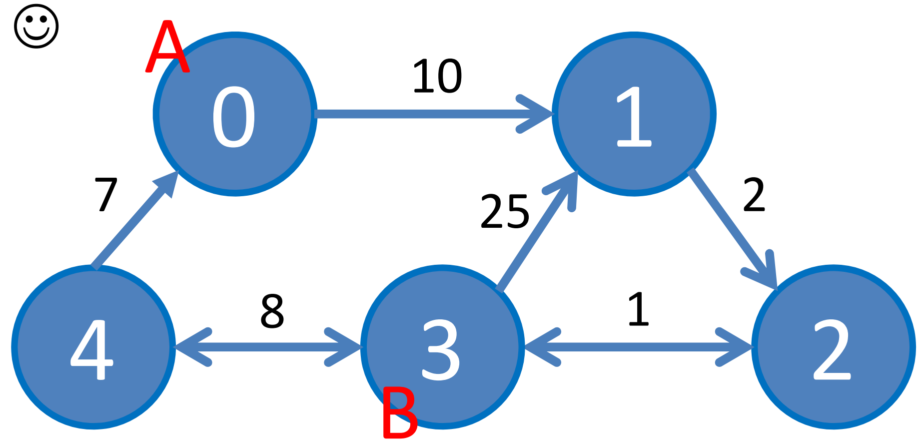
0 of 120



Motivating Problem 3

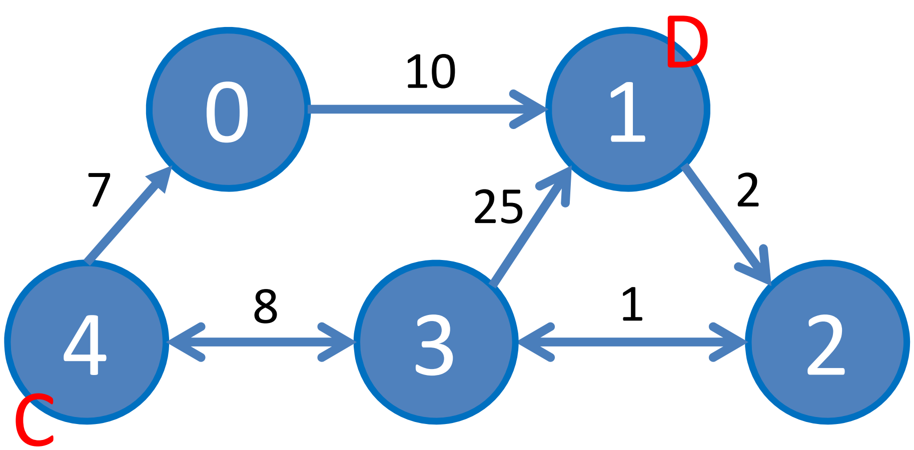
Finding the best meeting point

- 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 who are currently in two different vertices (**<u>lots of</u>** such queries)
  - 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 ☺

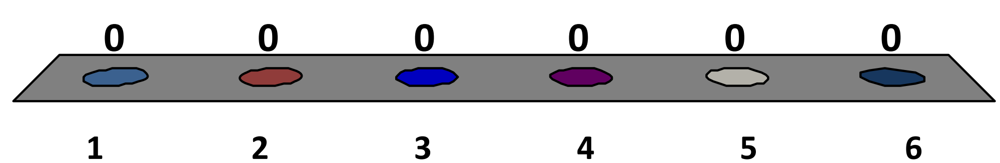


What is the best meeting point for C and D?
(you will need some time to calculate this)

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
6. I do not know ☹ …



0 of 120



All-Pairs Shortest Paths

- Problem definition:
  - Find shortest paths between **any pair** of vertices in the graph
- Several solutions from what we have known earlier:
  - On unweighted graph
    - Call BFS V times, once from each vertex
      - Time complexity: $O(V * (V + E))$ = $\mathbf{O(V^3)}$ 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: $O(V * (V + E) * \log V)$ = $\mathbf{O(V^3 \log V)}$ if $E = O(V^2)$
    - Call Bellman Ford's V times, once from each vertex
      - Time complexity: $O(V * VE)$ = $\mathbf{O(V^4)}$ 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) → <u>O(1)</u>**
  - 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++) // 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]);
```

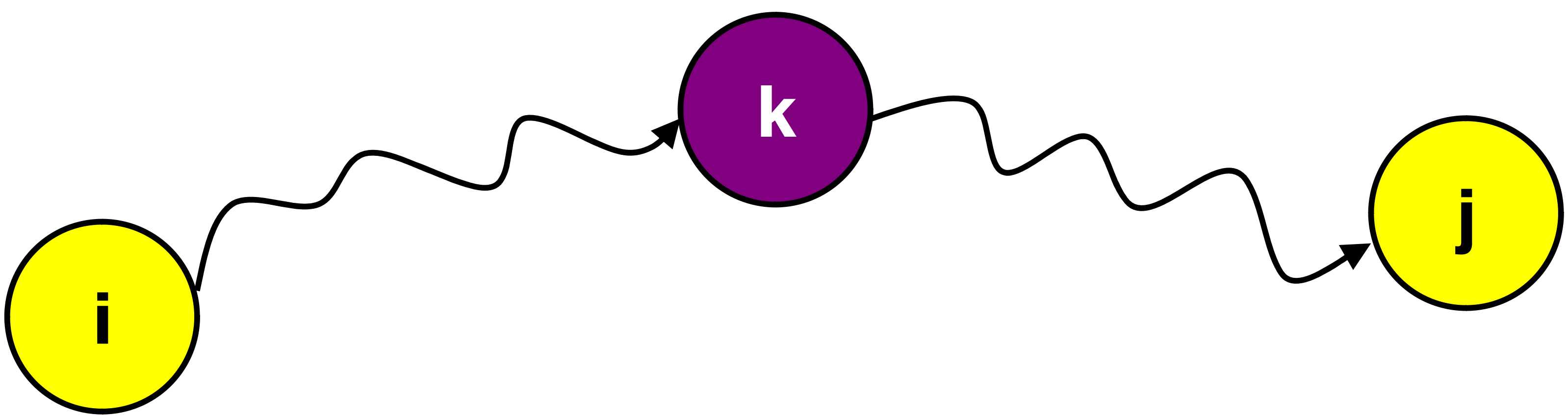
- $O(V^3)$ since we have three nested loops!
  - Apparently, if we only given a short amount of time, we can only solve the APSP problem for small graph, as none of the APSP solution shown in previous slide runs better than $O(V^3)$

Preprocessing + (Lots of) Queries

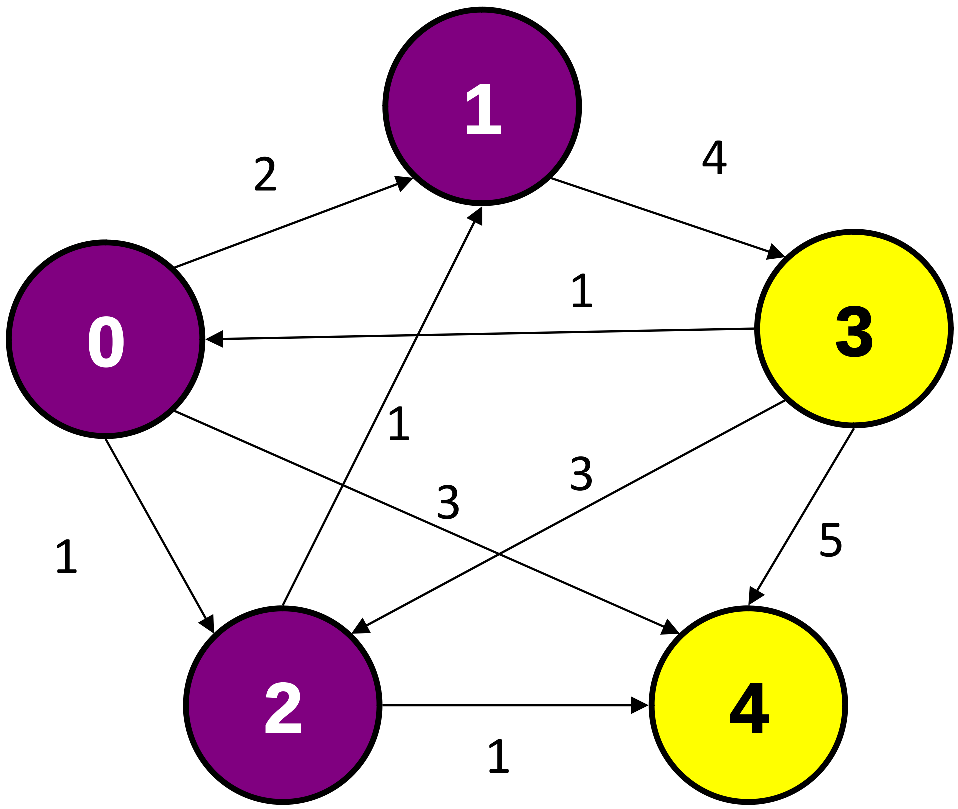
- This is another problem solving technique
- Preprocess the data once (can be a costly operation)
- But then future queries can be (much) faster
  by working on the processed data
- Example with APSP problem:
  - Once we have pre-processed the APSP information with $O(V^3)$ Floyd Warshall's algorithm…
  - Future queries that asks "what is the shortest path weight between vertex i and j" can now be answered in $O(1)$…

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** by one until k = V - 1



Floyd Warshall's – Basic Idea (2)



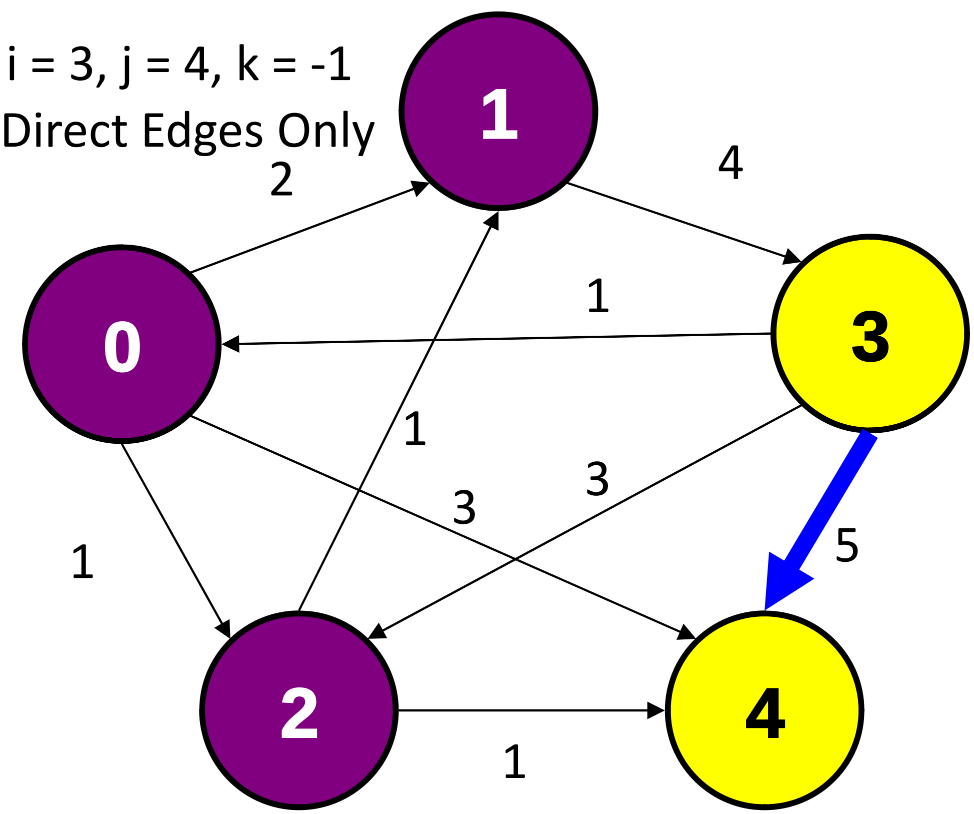
sp(3, 4, 4) = ?

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)

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

Direct Edges Only



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

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

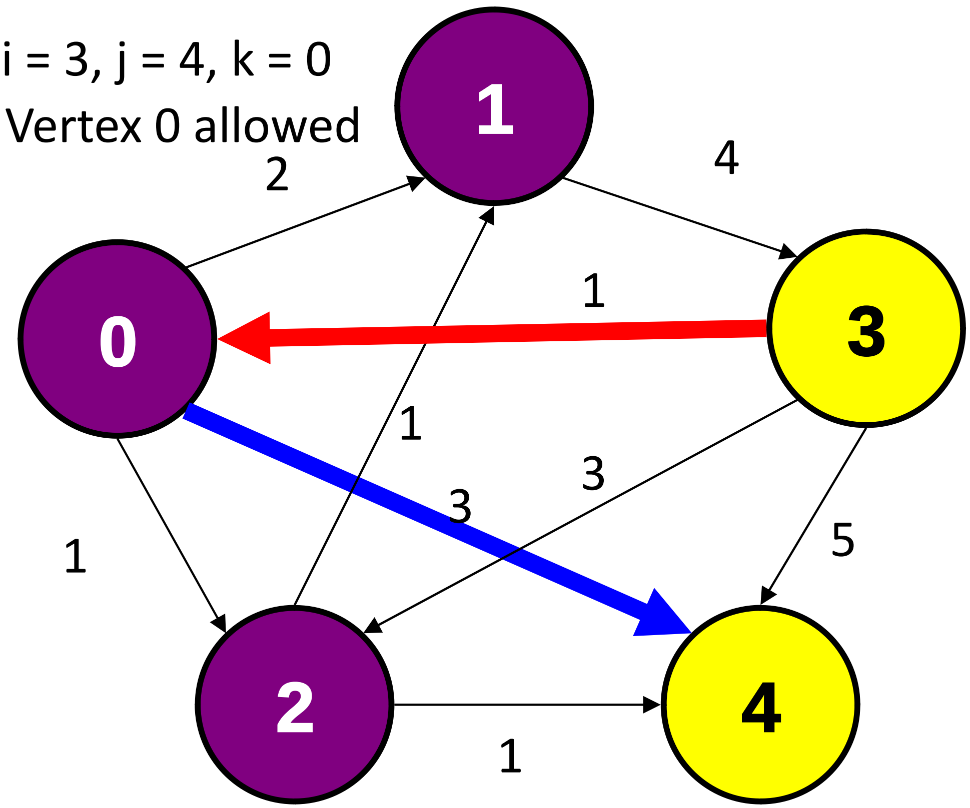
We will monitor these two values

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	0	**5**
4	∞	∞	∞	∞	0

Floyd Warshall's – Basic Idea (4)

i = 3, j = 4, k = 0
Vertex 0 allowed



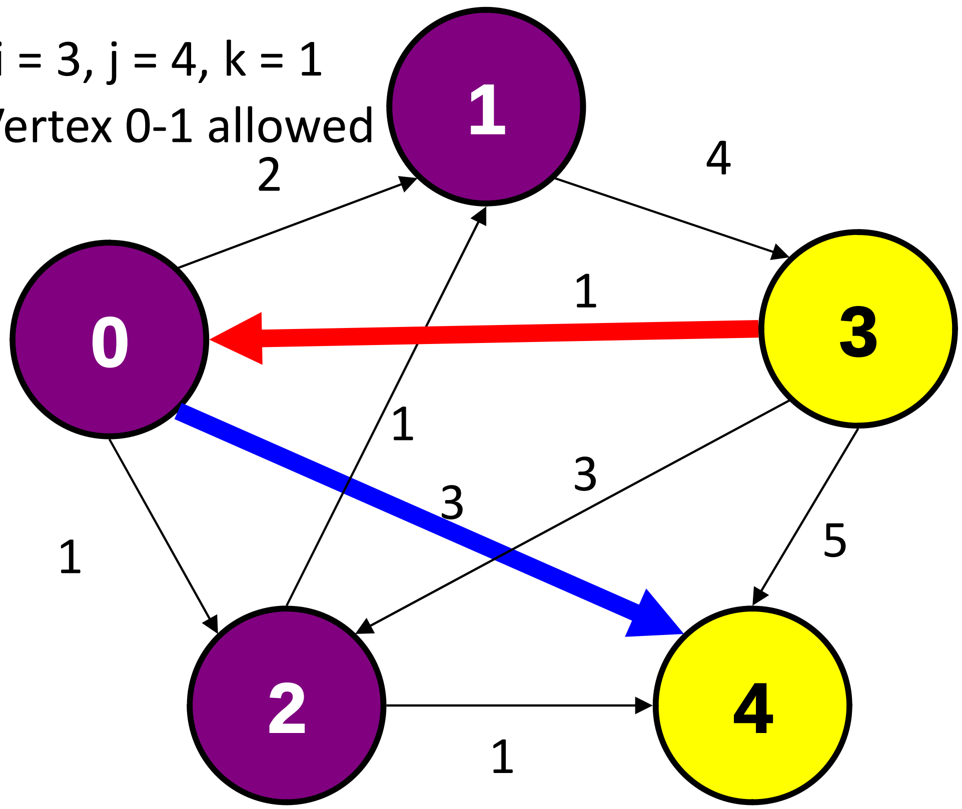
sp(3, 2, 0) = **2** sp(2, 4, 0) = **1** sp(3, 4, 0) = **4**

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)

i = 3, j = 4, k = 1
Vertex 0-1 allowed

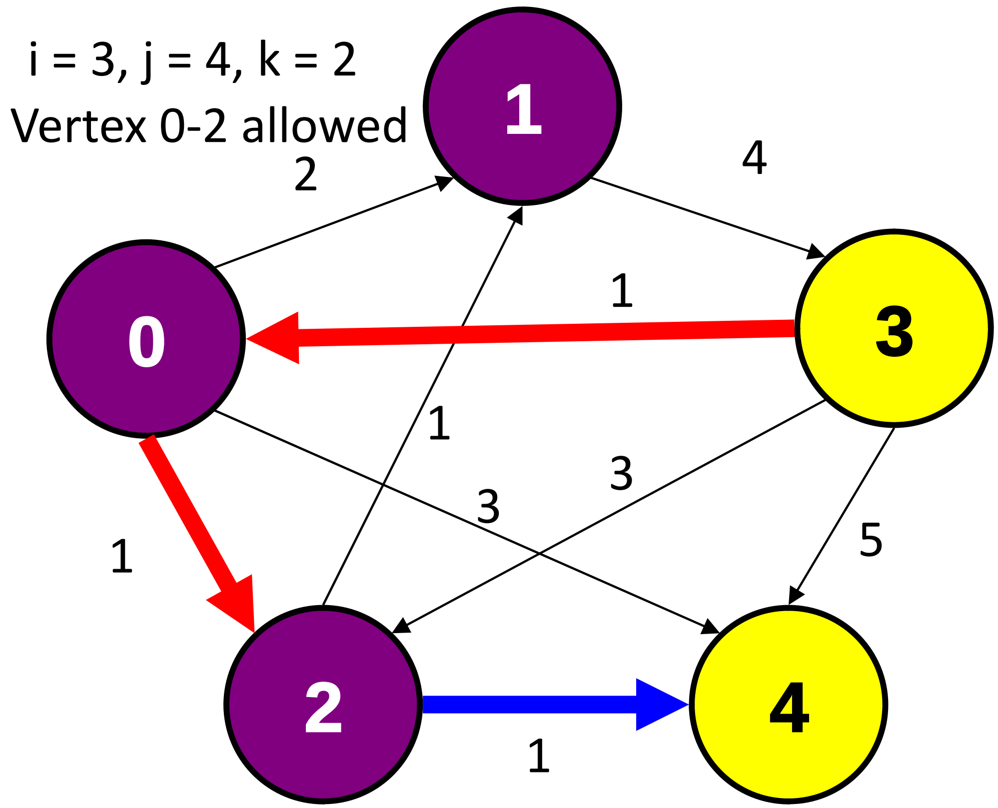


sp(3, 2, 1) = **2** sp(2, 4, 1) = **1** sp(3, 4, 1) = **4**

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)

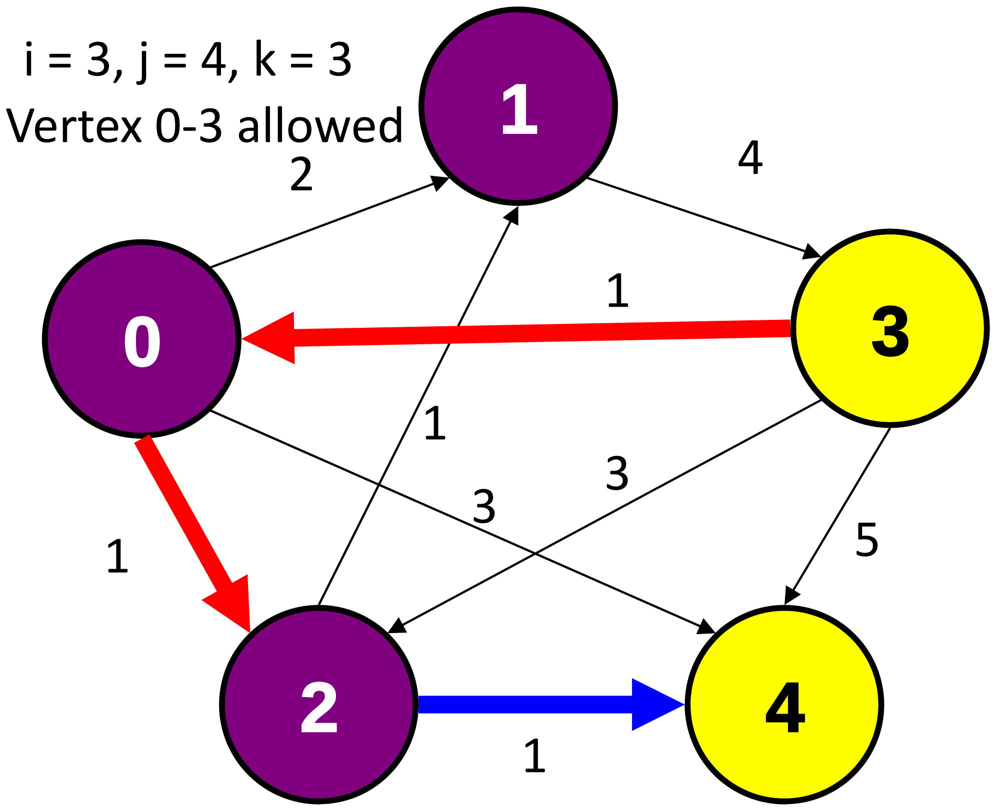


sp(3, 2, 2) = **2** sp(2, 4, 2) = **1** sp(3, 4, 2) = **3**

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)

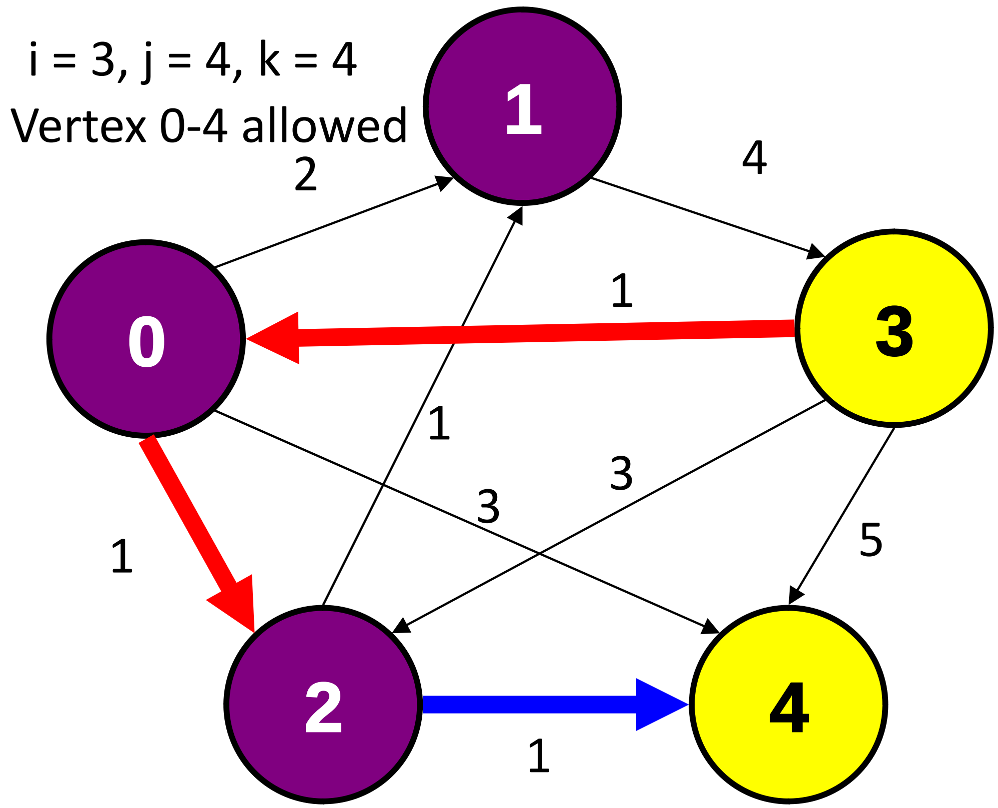


sp(3, 2, 2) = **2** sp(2, 4, 2) = **1** sp(3, 4, 2) = **3**

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)



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

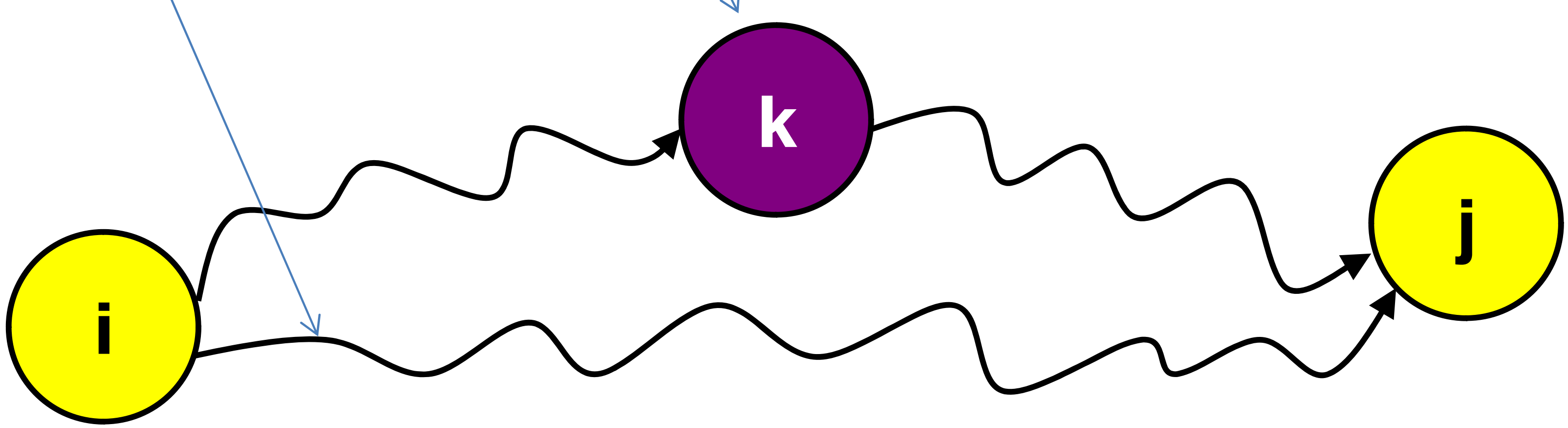
$D_{i,j}^{-1}$: Edge weight of the original graph

$D_{i,j}^{k}$: 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,** row by row, left to right
  - The topological order is obtained via 3 nested loops: **k → i → j**

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

k = 1

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

k = 2

Floyd Warshall's – DP (3)

The Near Final Code

```
// the "memory unfriendly" version, O(V^3) space complexity
int[][][] D3 = new int[V+1][V][V]; // 3D matrix
for (k = 0; k <= V; k++) // initialization phase
  for (i = 0; i < V; i++) {
    Arrays.fill(D3[k][i], 1000000000); // cannot use Collections.nCopies
    D3[k][i][i] = 0;
  }

for (i = 0; i < E; i++) { // direct edges
  u = sc.nextInt(); v = sc.nextInt(); w = sc.nextInt();
  D3[0][u][v] = w; // directed weighted edge
}

// main loop, O(V^3): this three nested loops are the "topological order"
for (k = 0; k < V; k++) // be careful, put k first
  for (i = 0; i < V; i++) // before i
    for (j = 0; j < V; j++) // and then j
      D3[k+1][i][j] = Math.min(D3[k][i][j], // note, I shift index k by +1
                               D3[k][i][k] + D3[k][k][j]);
```

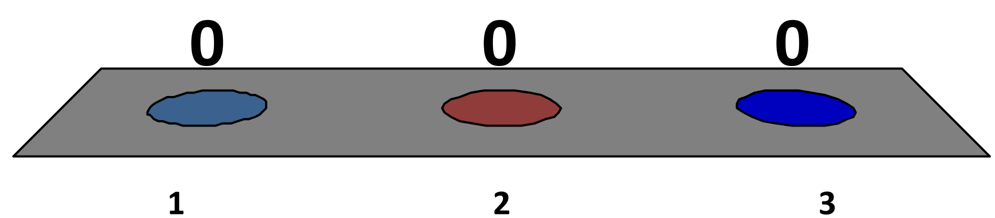
Floyd Warshall's – DP (4)

The Final Code, drop dimension 'k'

```
int[][] D = new int[V][V]; // 2D adjacency matrix
for (i = 0; i < V; i++) { // initialization phase
  Arrays.fill(D[i], 1000000000); // cannot use nCopies
  D[i][i] = 0;
}
for (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): the "topological order"
for (k = 0; k < V; k++) // be careful, put k first
  for (i = 0; i < V; i++) // before i
    for (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. 50-50
3. I understand both the code and the DP formulation ☺



0 of 120

5 minutes break, and then…

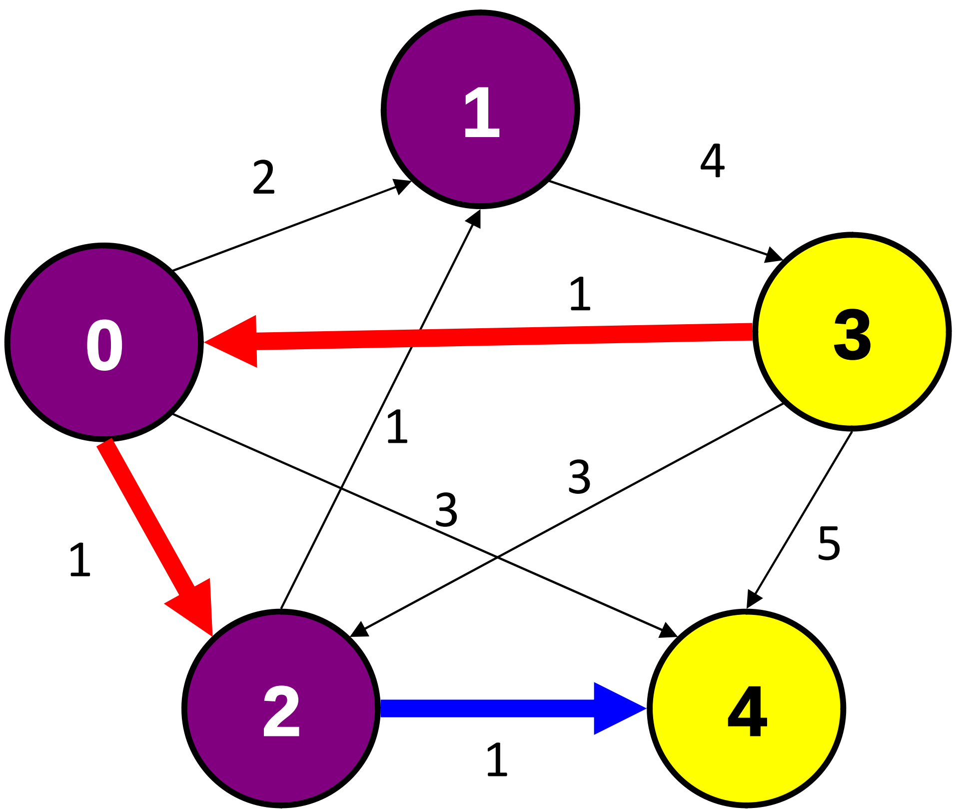
VARIANTS OF FLOYD WARSHALL’S

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 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	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

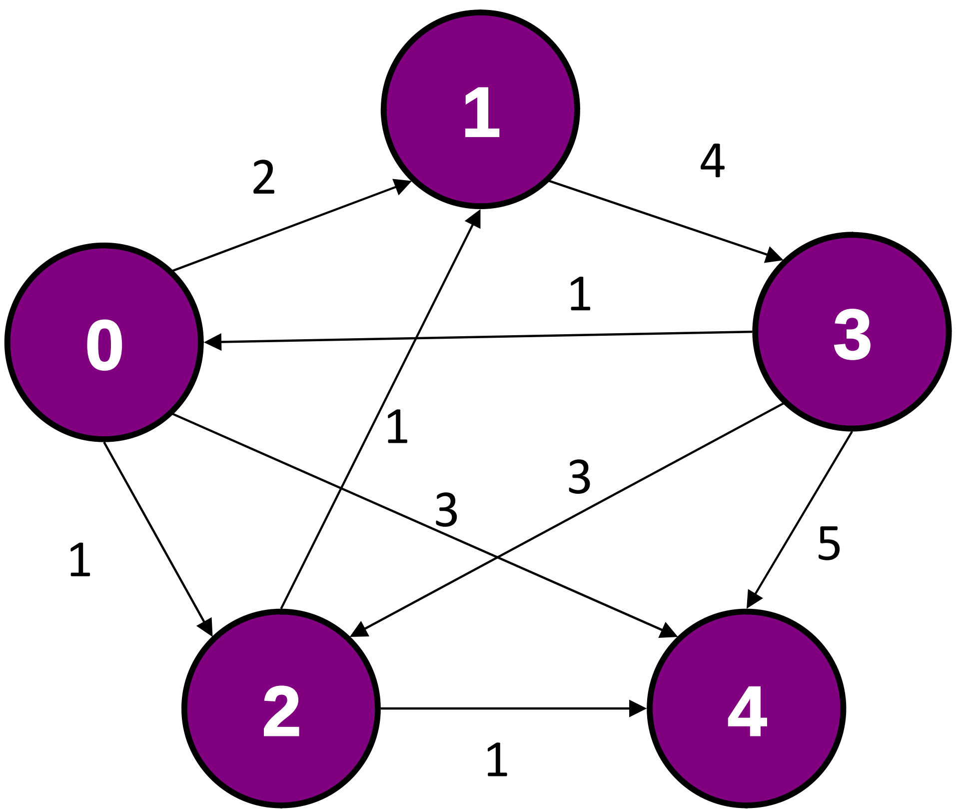
Variant 2 – Transitive Closure (1)

- 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]); // faster
```

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

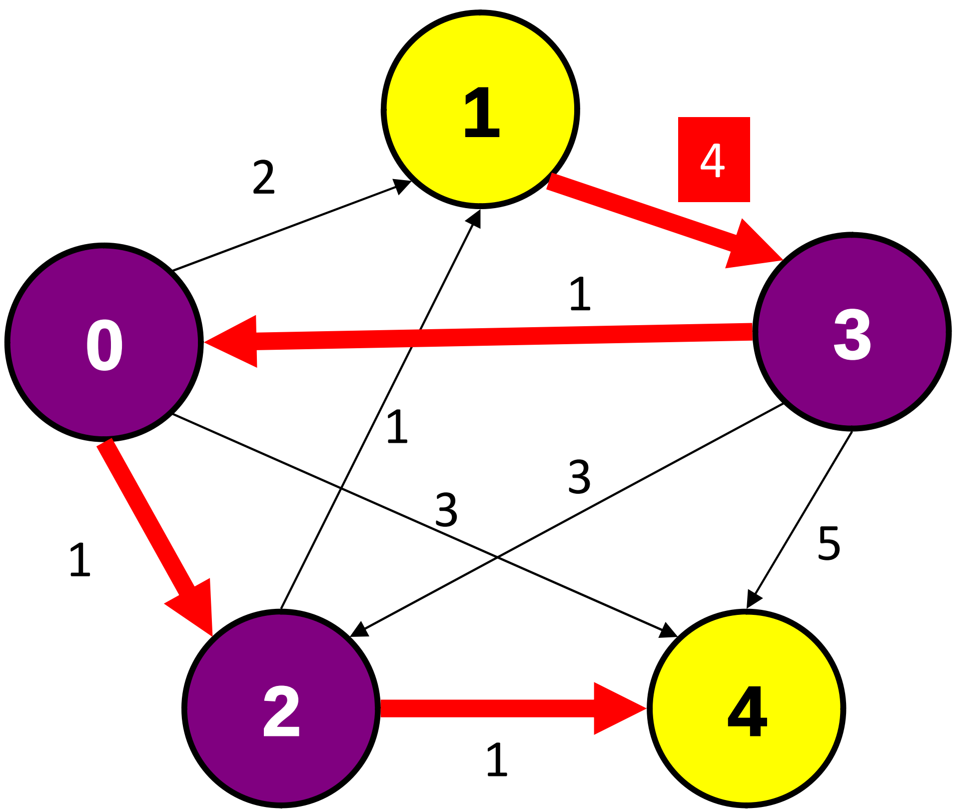
Variant 3 – Minimax/Maximin (1)

- 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, 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]));
```

Variant 3 – Minimax/Maximin (2)

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

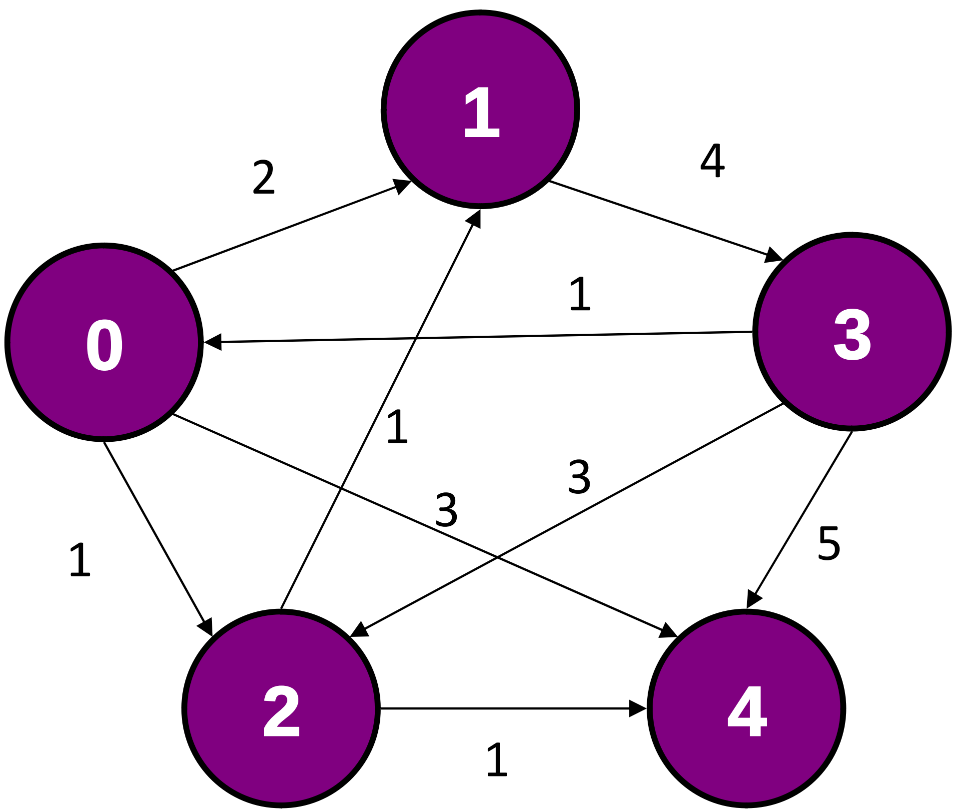


D,init	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0

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 – Detecting Any/-ve Cycle

- Set the main diagonal of D to ∞
- Run Floyd Warshall's
- Recheck the main diagonal



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

- Let's see: FloydWarshallDemo.java
- Let's see how easy to change the basic form of Floyd Warshall's algorithm to its variants
- Note: The given Java file is not written in OOP fashion
  – Please re-factor it if you need to use it for other projects!

Summary

- In this lecture, we have seen:
  - Introduction to the APSP problem (yes, outside CS2010 syllabus)
  - Introduction to the Floyd Warshall's DP algorithm
  - Introduction to 4 variants of Floyd Warshall's
  - Simple Java implementations
- Floyd Warshall's is a DP algorithm
  - But many just view this as "another graph algorithm"
- For the "last" lecture next week…
  - (if we have not complete Quiz 2 review, we will do this next week)
  - A "mystery lecture"
  - Review of the third part of CS2010: DAG/Algorithms on DAG