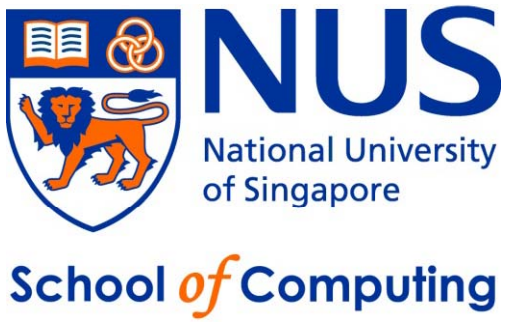
CS2020 – Data Structures and Algorithms Accelerated

Lecture 13 – Introduction to Graphs

stevenhalim@gmail.com



Outline

- What are you going to learn in this lecture?
  - Quick review of 1st half + Outline of the 2nd half of CS2020
  - Motivation on why you should learn graph
  - Two graph data structures

Stuffs that you have learned so far…

- Basically, the entire? CS1020 module + a bit of CS2010 (BST, Heap, Hashing):
  - Introduction (1 lecture)
  - Divide & Conquer technique (1 lecture)
  - Tree Data Structure: Binary Search Tree, Balanced BST: AVL/etc, Heap (3 lectures)
  - Sorting: Slow/Merge/Quick/Linear time Sort (3 lectures)
  - Skip List: Linked List (1 lecture)
  - Hashing (3 lectures)
  - Java skills: OO design, Stacks/Queues (scattered)

Stuffs in the official syllabus that are not yet covered…

- Compared with existing/previous/future courses
  - Last year's CS1102:
    - Graph: AdjMatrix/List/DFS/BFS/Toposort (only as last lecture ☹)
  - Last year's CS1102S:
    - As above, plus: Max Flow (brief); **MST** (Prim's); Halting problem; **Dijkstra's**; **DP** (Fibo; O-BST); **Disjoint Sets**
  - Official plan for CS2010/CS2020:
    - Graph data structure
    - Graph search/traversal
    - Minimum Spanning Tree
    - Shortest Paths
    - Algorithm analysis: space-time trade-off
    - (Recursive) Tree and Graph algorithms
    - Data structure design, implementation, application

2nd half of CS2020 for this Semester

- My plan for the 2nd half of CS2020 (all graph):
  - Graph data structure → today
  - Graph search/traversal → this coming Friday
  - Minimum Spanning Tree → next Tuesday
  - Shortest Paths → next Fri, next-next (next^2) Tue, next^3 Fri
  - Quiz2 (those in red are included) → next^2 Fri
  - Algorithm analysis: space-time trade-off → DP, next^3 Tue
  - (Recursive) Tree and Graph algorithms → next^4 Tue & Fri
  - Data structure design, implementation, application (all)
  - Competitive Programming (2nd last lecture)
  - Mystery lecture (still mysterious for me too)

The Spirit of CS2020 – 2nd half…

- Given (mostly) well-known^ (graph) problems, we want to use *the most appropriate* data structures and algorithms for the 'best known' performance…
  - Or at least reasonable performance
- In the 2nd half of CS2020, you will see:
  - (Hopefully) motivating graph problems and solutions
  - Various trade-off questions
  - Alternatives / special cases of the problem
  - Practical Java implementations
    - Your 2nd lecturer is a competitive programmer, after all…

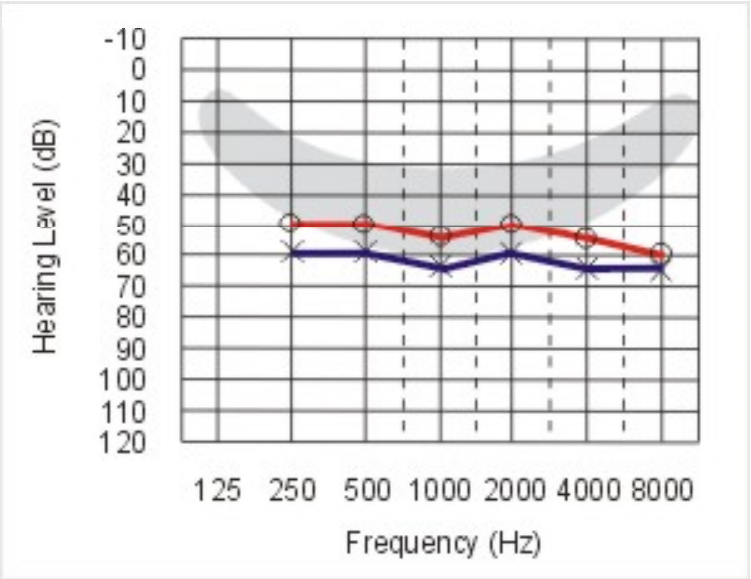
Class Format (2nd half)

More or less the same as the 1st half

- 2 hours Tuesday Lectures @ COM1/204 (SR2):
  - High speed download…
- 2 hours Wed/Thu Discussion Groups (with your DG leader):
  - Recite previous lecture materials; usually with additional insights
  - Discuss relevant problems
- 2 hours Friday Lectures @ LT15:
  - Yet another high speed download…
- 1 hour Friday Recitations:
  - Revisions, clarifications, motivational talks

About My Lecture Style

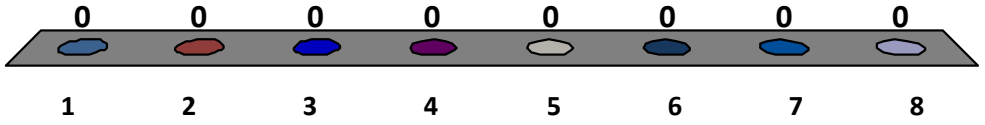
- Mostly **interactive**
  - "Quick Challenges" will appear throughout the lecture
    - You can discuss the answers with another students near you
      - Note: I will **not** update my lecture notes to include the solutions
  - Most questions will be MCQ, use your clicker!
- If you need clarifications during lecture…
  - Wait until lecture breaks
    - Reason: hearing issue… ☹
  - Or post questions in the IVLE discussion forum



INTRODUCING GRAPH

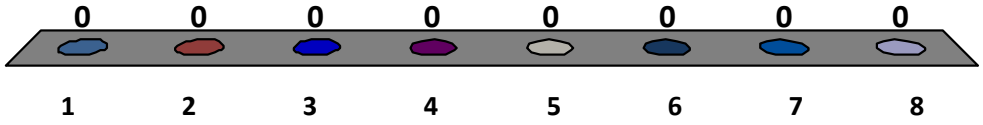
Select graph terminologies that you already know… (can select up to 8/clicker)

1. Adjacency Matrix/List
2. DFS/BFS
3. Topological Sort
4. MST/Prim's
5. MST/Kruskal's
6. SSSP/Bellman Ford's
7. SSSP/Dijkstra's
8. APSP/Floyd Warshall's



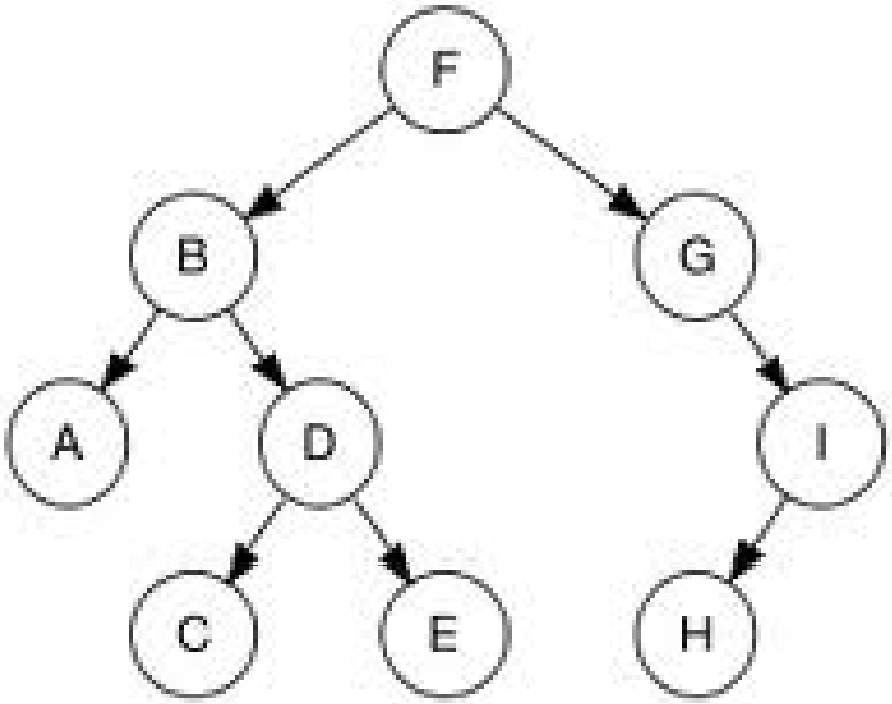
Select DS/algorithms that you have already **implement**… (can select up to 8/clicker)

1. Adjacency Matrix/List
2. DFS/BFS
3. Topological Sort
4. MST/Prim's
5. MST/Kruskal's
6. SSSP/Bellman Ford's
7. SSSP/Dijkstra's
8. APSP/Floyd Warshall's



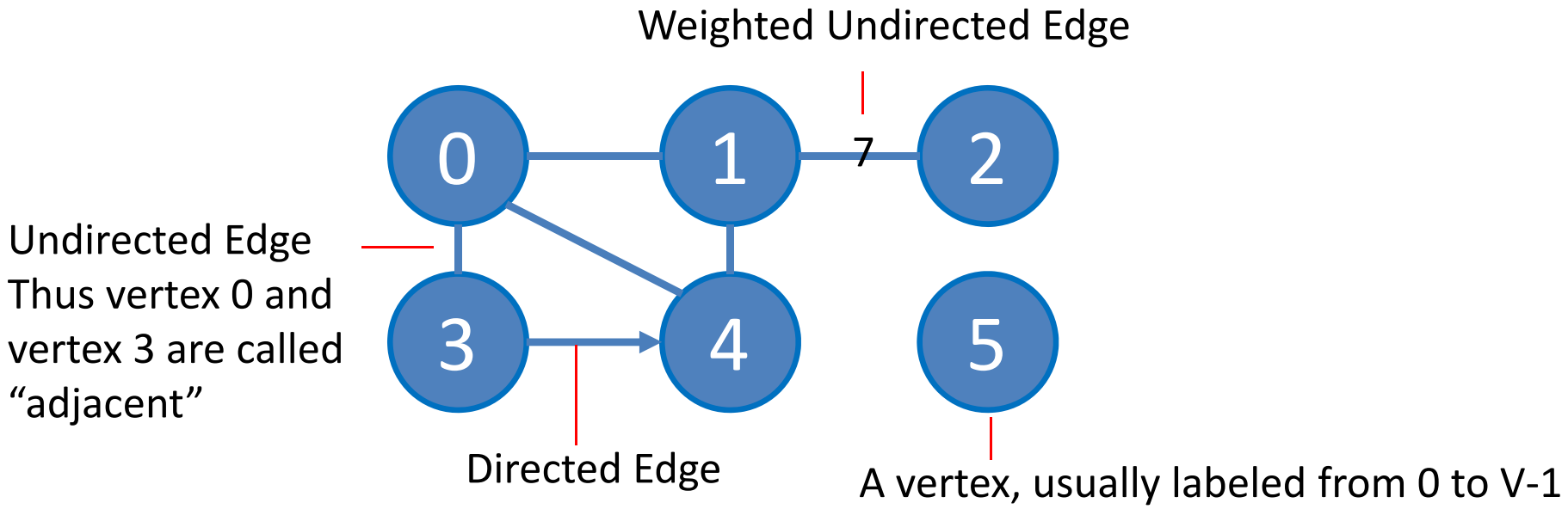
Graph Terminologies (1)

- Extension from what you already know: *(Binary) Tree*
  - Vertex/Node
  - Edge
  - Direction (of Edge)
  - Weight
- But in general graph, there is no notion of:
  - Root
  - Parent/Child
  - Ancestor/Descendant



Graph is…

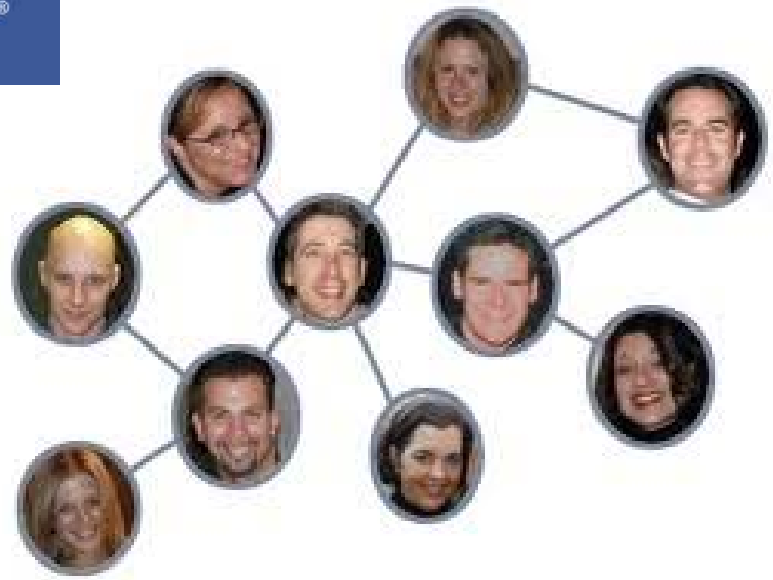
- (Simple) graph is a set of vertices where some [0 .. $_{N}C_{2}$] pairs of the vertices are connected by edges
  - We will ignore "multi graph" for CS2020 where there can be more than one edge between a pair of vertices



Social Network

facebook

twitter

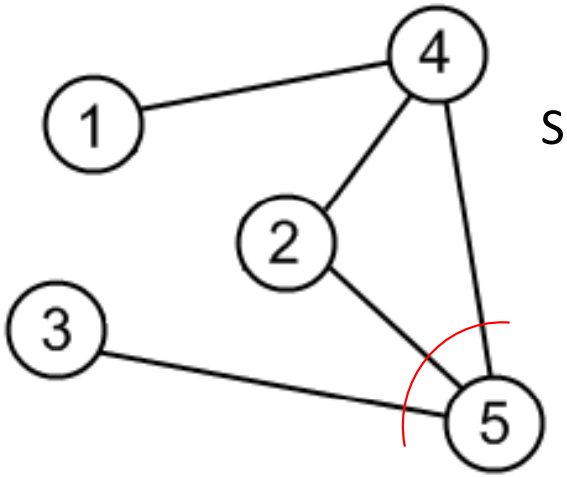


LinkedIn

friendster

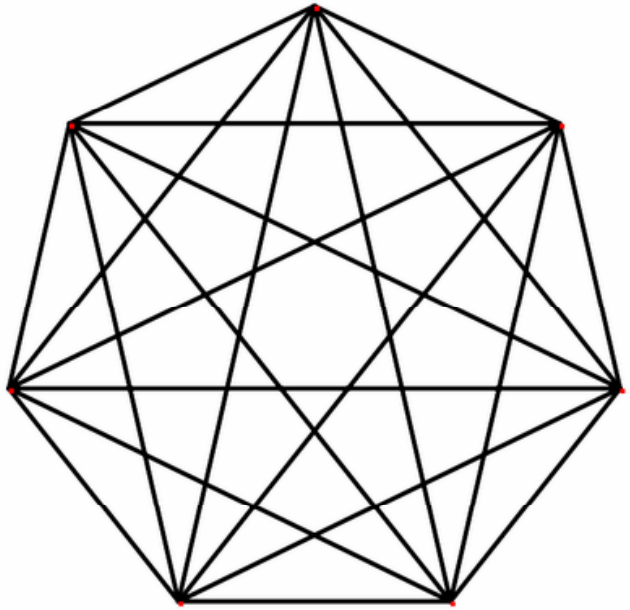
Graph Terminologies (2)

- More terminologies (simple graph):
  - Sparse/Dense
    - Sparse = not so many edges
    - Dense = many edges
    - No guideline for “how many”
  - Complete Graph
    - Simple graph with N vertices and NC2 edges
  - In/Out Degree of a vertex
    - Number of in/out edges from a vertex



Sparse Graph

In/out degree of vertex 5 = 3



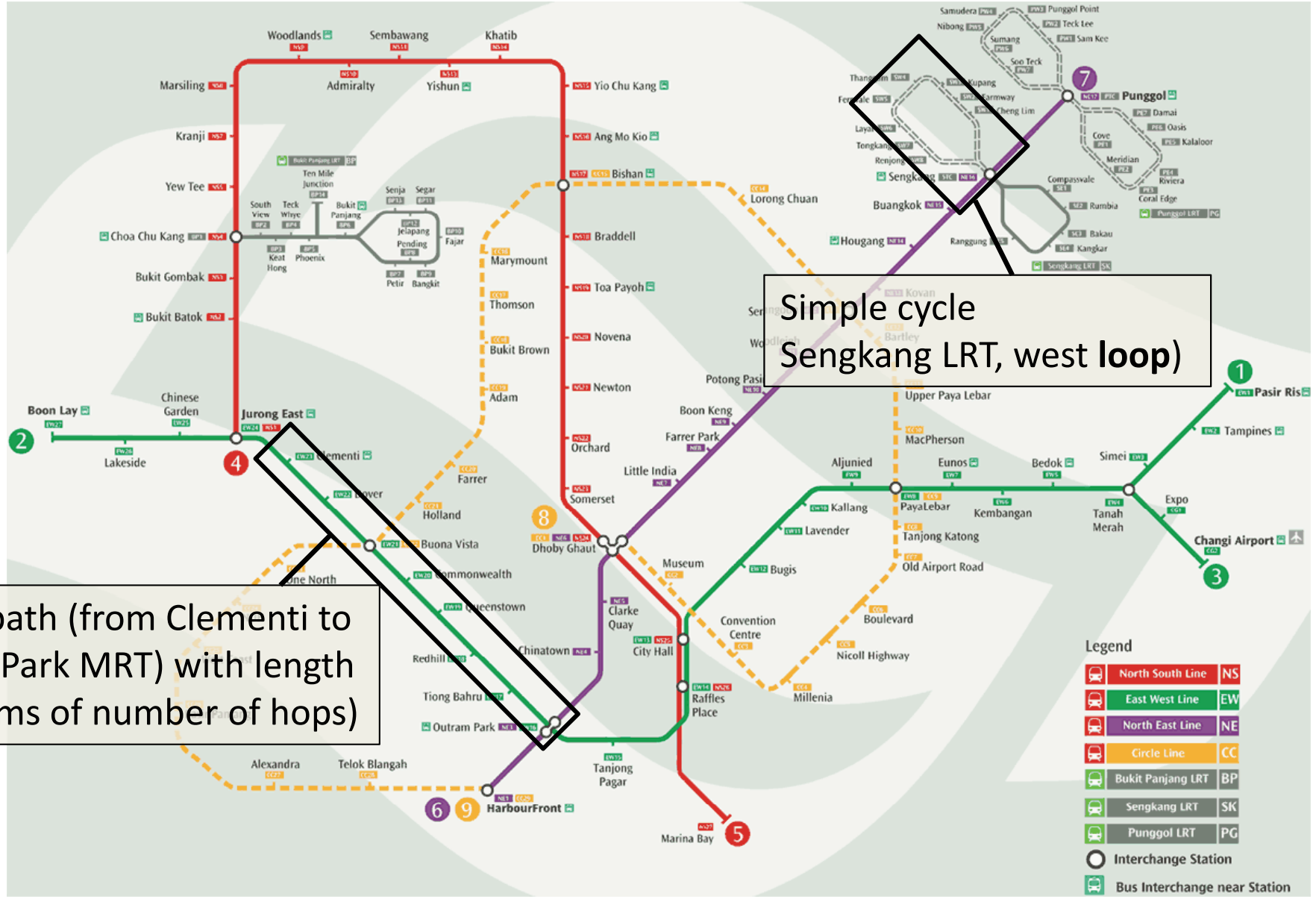
Dense Graph

Complete Graph
7 vertices,
7C2 = 21 edges

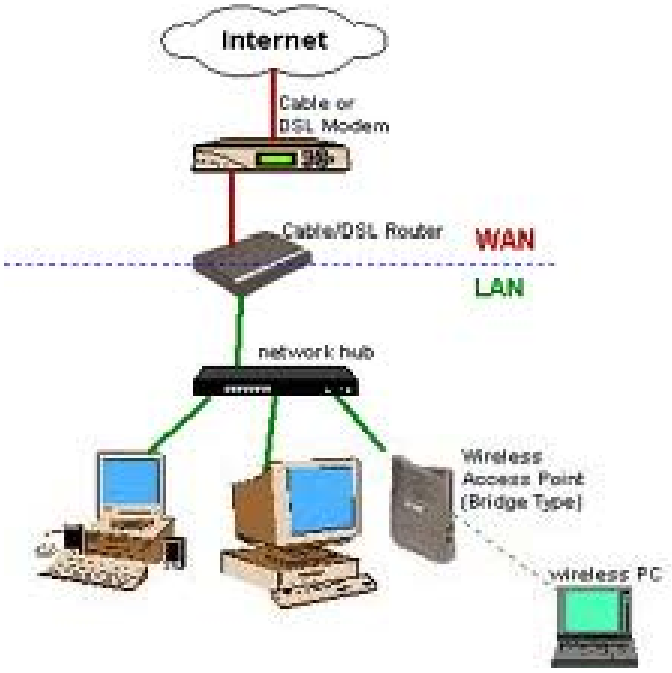
Graph Terminologies (3)

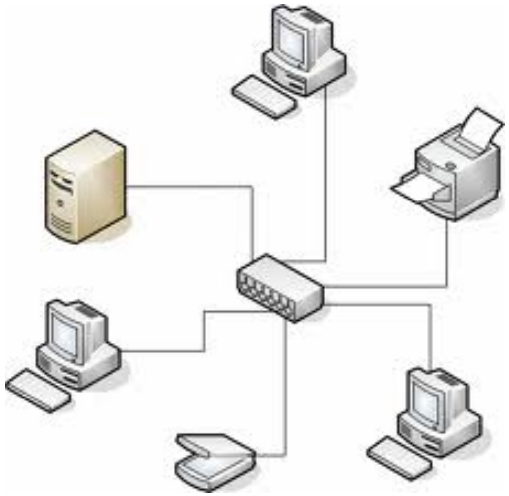
- Yet more terminologies (example in the next slide):
  - (Simple) Path
    - Sequence of vertices adjacent to each other
    - Simple = no repeated vertex
  - Path Length/Cost
    - In undirected graph, usually number of edges in the path
    - In weighted graph, usually sum of edge weight in the path
  - (Simple) Cycle
    - Path that starts and ends with the same vertex
    - With no repeated vertex except start/end

Transportation Network



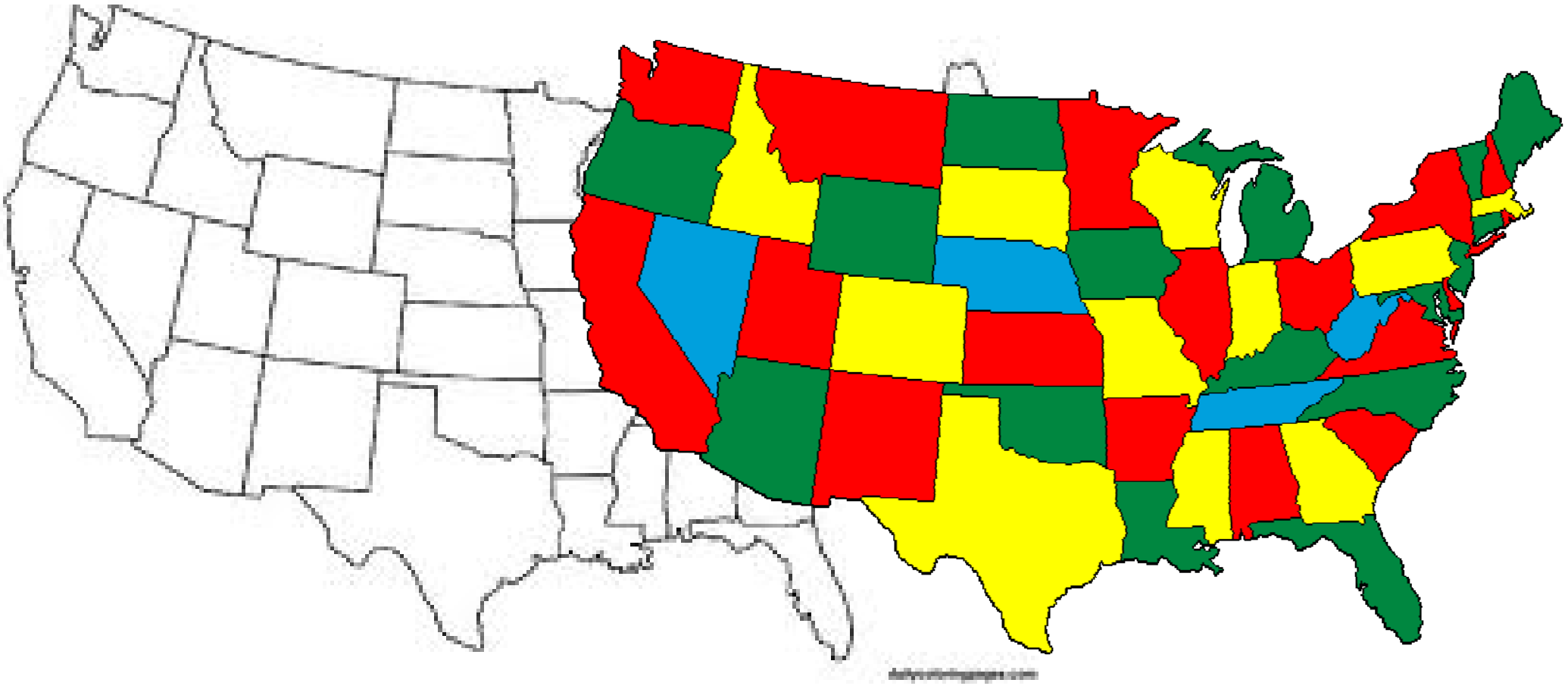
Internet / Computer Networks





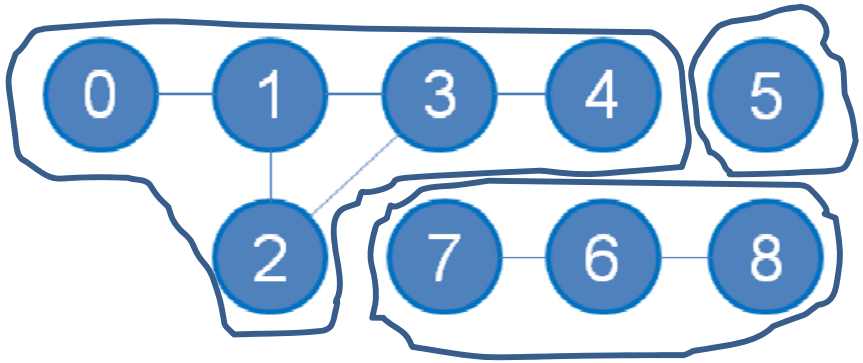
Communication Network

Optimization



Graph Terminologies (4)

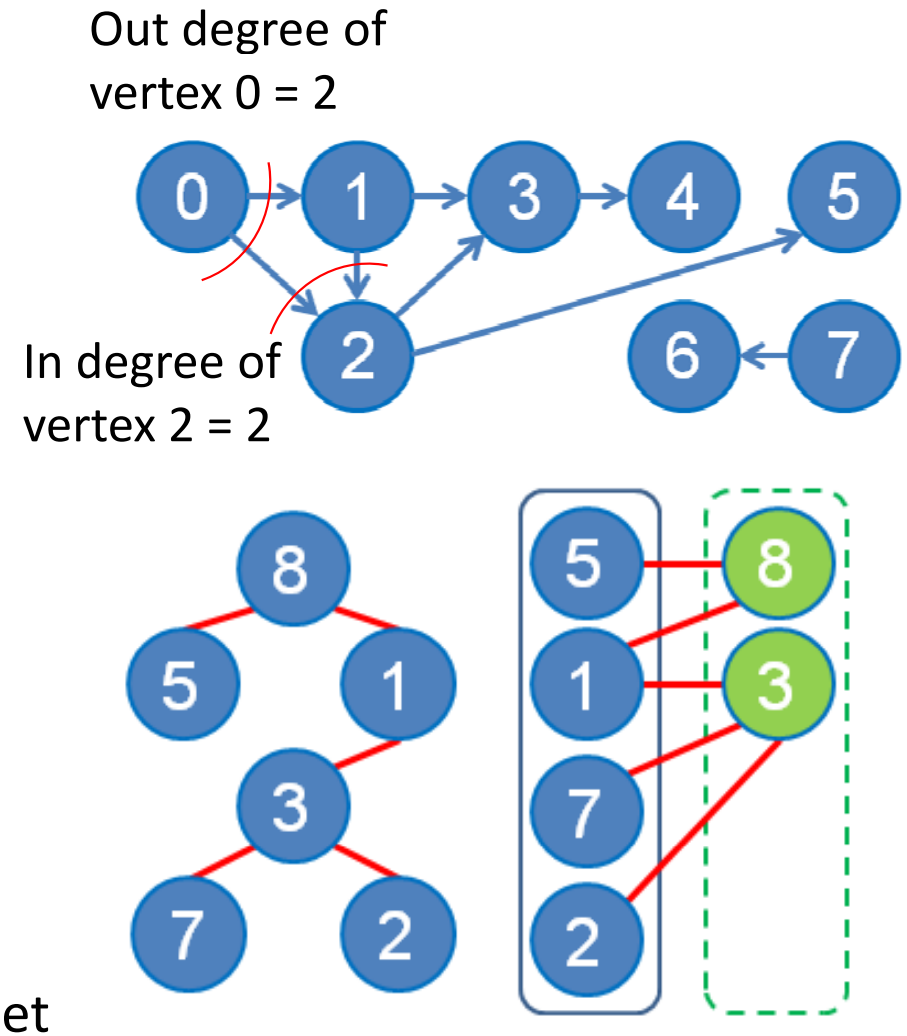
- Yet More Terminologies:
  - Component (of undirected graph)
    - A group of vertices that can visit each other via some path
  - Connected graph
    - Graph with only 1 component
  - Isolated/Reachable Vertex
    - See example
  - Sub Graph
    - Subset of vertices (and their edges) of the original graph



- There are 3 components in this graph
- Disconnected graph
(since it has > 1 component)
- Vertex 1-2-3-4 are reachable from vertex 0
- Vertex is isolated/unreachable from vertex 0
- {7-6-8} is a sub graph of this graph

Graph Terminologies (5)

- Yet More Terminologies:
  - Directed Acyclic Graph (DAG)
    - Directed graph that has no cycle
  - Tree (bottom left)
    - Connected graph, E = V - 1, one unique path between any pair of vertices
  - Bipartite Graph (bottom right)
    - If we can partition the vertices into two sets so that there is no edge between members of the same set

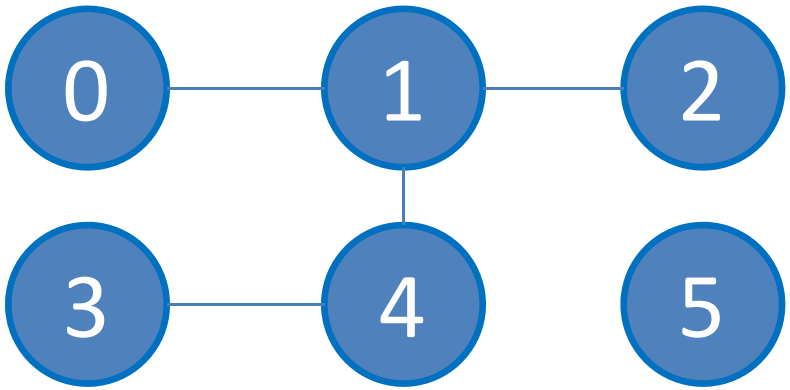


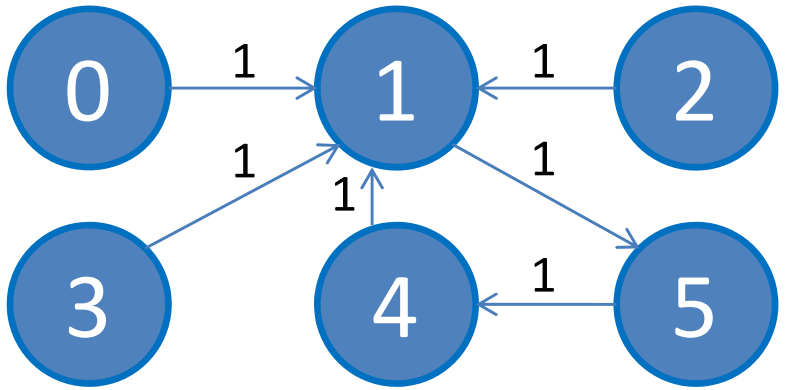
Quick Challenge (1)

- Find several real-life graphs around you (in this room) or around your life!
  - State what are the vertices, the edges
  - Find simple and meaningful graph problems (if any)
- Simple example:
  - Vertices: NUS modules
  - Edges: Module pre-requisites (it is a DAG!!)
  - Graph problem: I have taken a set of modules, can I take a certain future module given this pre-requisites DAG?

Quick Challenge (2)

- Elaborate the properties of these two graphs
  - How many V? E? Components? Is it connected? Is it weighted? Is it directed? Does it have cycle? Is it a tree? Is it bipartite? Etc…





Note: we skip "Strongly Connected Component" in CS2020

10 minutes break

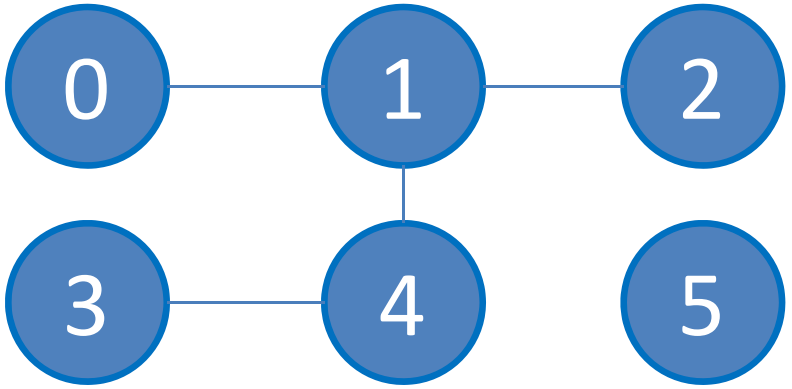
Next, we will discuss two Graph DS
One more will be discussed next week (MST lecture)

GRAPH DATA STRUCTURES

Storing Graph Information

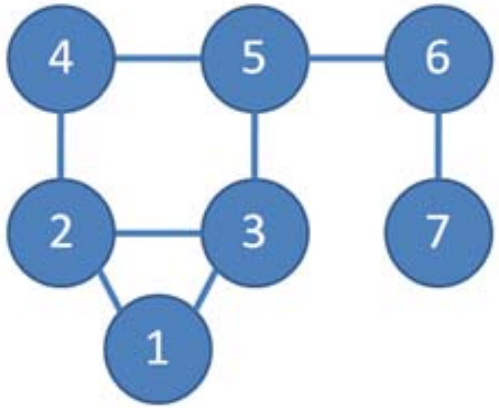
Can we store only vertex information?

1. YES, vertex information is enough to rebuild the graph
2. NO, we should store edge/connectivity information!



Adjacency Matrix

- Format: a 2D array **AdjMatrix** (see example below)
- Cell **AdjMatrix**[i][j] contains value 1 if there exist an edge i->j in G, otherwise **AdjMatrix**[i][j] contains 0
  - For weighted graph, **AdjMatrix**[i][j] contains the weight of edge i->j, not just binary values {1, 0}.
- **Space Complexity**: $O(V^2)$
  - V is |V| = number of vertices in G

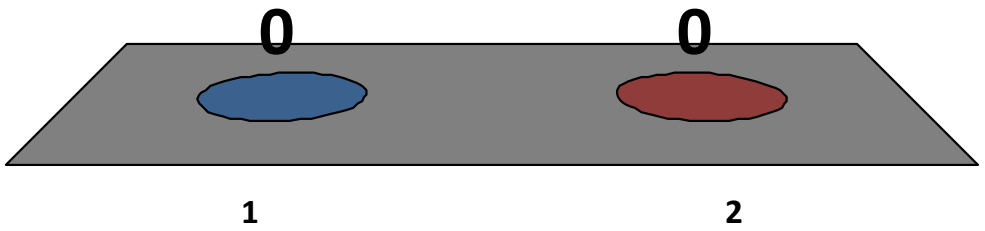


	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

To think about: If I have a graph with V = 100000 vertices, can I use **Adjacency Matrix**?

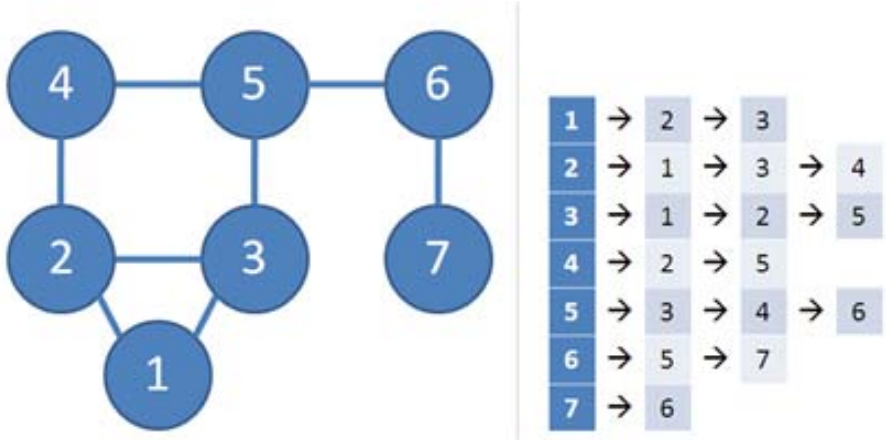
1. Yes, what is the problem?
2. No, because

_____



Adjacency List

- Format: array **AdjList** of V lists, 1 for each vertex in V (see example below)
- For each vertex i, **AdjList[i]** stores list of i's neighbors
  - For weighted graph, stores **pairs (neighbor, weight)**
    - Note that unweighted graph can also use the same strategy as the weighted version: (neighbor, weight = 0 or 1)
- **Space Complexity**: O(V + E)
  - E is |E| = number of edges in G, $E = O(V^2)$
  - V + E ~= max(V, E)



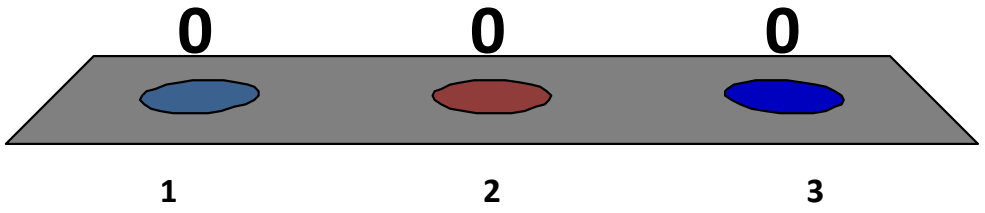
To think about: If I have a graph with V = 100000 vertices, can I use **Adjacency List**?

1. Yes, or course
2. No, because

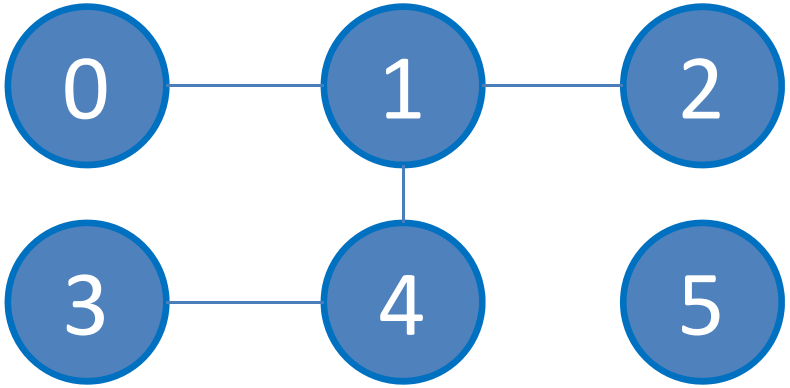
   _____

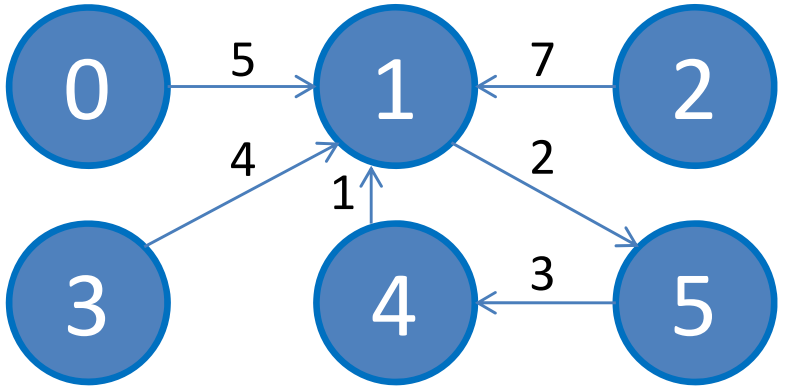
3. Depends, because

   _____



Quick Challenge (3)





	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

0	→			
1	→			
2	→			
3	→			
4	→			
5	→			

Java Implementation (1)

- Adjacency Matrix
  - Simple built-in 2D array

```
int i, V = NUM_V; // NUM_V has been set before
int[][] AdjMatrix = new int[V][V];
```

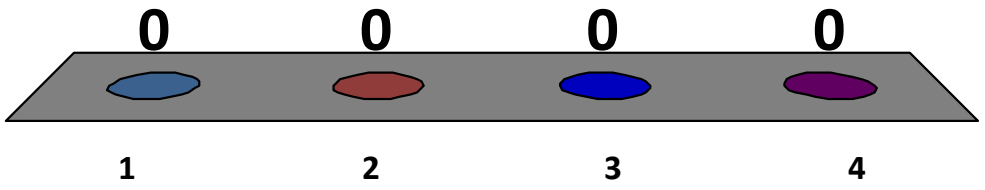
- Adjacency List
  - Use Java Collections framework

```
Vector < Vector < ii > > AdjList =
  new Vector < Vector < ii > >();
// ii is a simple integer pair class
// to store pair info
```

- PS: This is *my* implementation, there are other ways

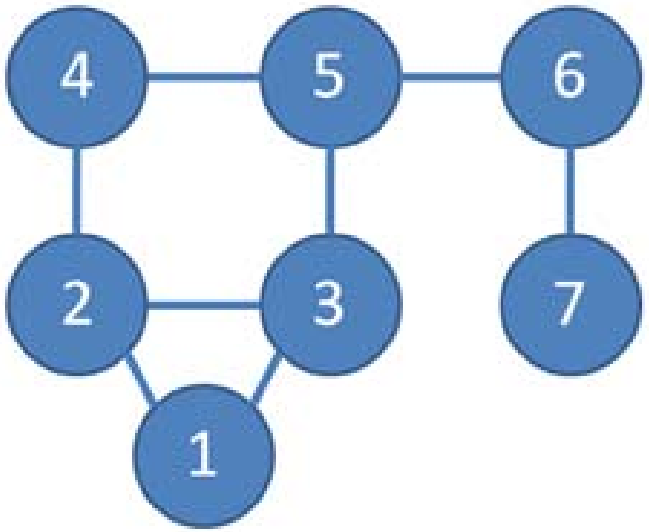
Trade Off (1), after knowing Adjacency Matrix and List, which one should be our **default choice**?

1. Adjacency Matrix
2. Adjacency List
3. Use BOTH at the same time
4. Depends on what you want to do with the graph…



So, what can we do so far? (1)

- With just graph DS, not much that we can do…
- But here are some:
  - Counting V (the number of vertices)
    - Very trivial for both AdjMatrix and AdjList: **V = number of rows!**
    - Sometimes this number is stored in separate variable so that we do not have to re-compute every time



A

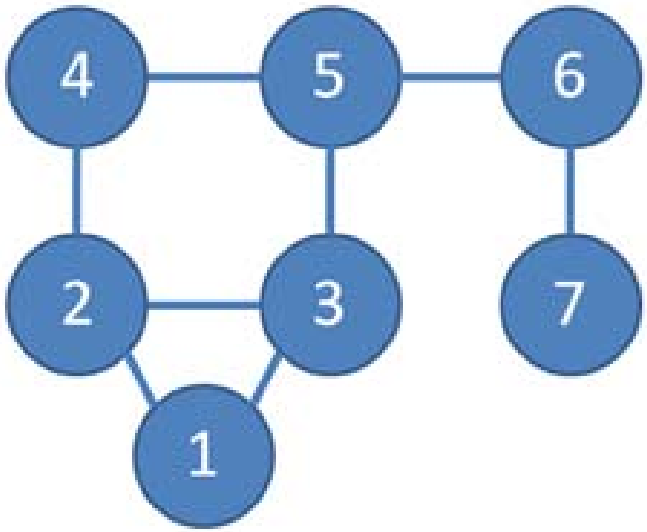
	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1	→	2	→	3			
2	→	1	→	3	→	4	
3	→	1	→	2	→	5	
4	→	2	→	5			
5	→	3	→	4	→	6	
6	→	5	→	7			
7	→	6					

So, what can we do so far? (2)

- Enumerating neighbors of a vertex v
  - O(V) for AdjMatrix: **scan AdjMatrix[v][j], for all j in [0 .. V-1]**
  - O(k) for AdjList, **scan AdjList[v]**
    - Where k is the number of neighbors of vertex v



A

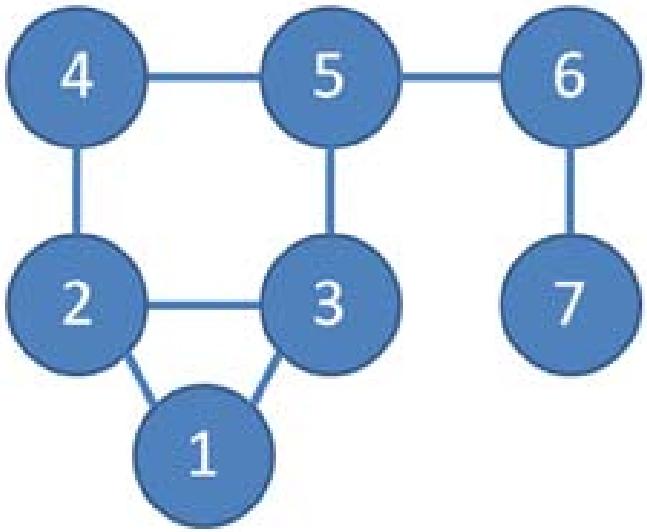
	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1	→	2	→	3			
2	→	1	→	3	→	4	
3	→	1	→	2	→	5	
4	→	2	→	5			
5	→	3	→	4	→	6	
6	→	5	→	7			
7	→	6					

So, what can we do so far? (3)

- Counting E (the number of edges)
  - O(V^2) for AdjMatrix: **count non zero entries in AdjMatrix**
  - O(V + E) for AdjList: **sum the length of all V lists**
  - Sometimes this number is stored in separate variable so that we do not have to re-compute every time



A

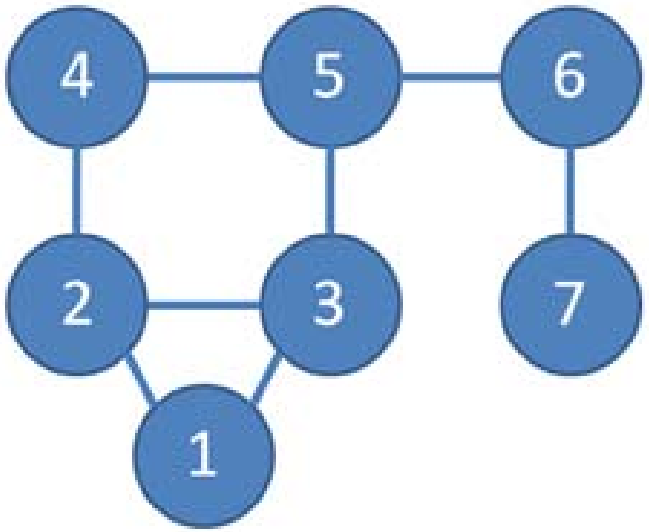
	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1	→	2	→	3			
2	→	1	→	3	→	4	
3	→	1	→	2	→	5	
4	→	2	→	5			
5	→	3	→	4	→	6	
6	→	5	→	7			
7	→	6					

So, what can we do so far? (4)

- Checking the existence of edge(u, v)
  - O(1) for AdjMatrix: **see if AdjMatrix[u][v] is non zero**
  - O(k) for AdjList: **see if AdjList[u] contains v**
- There are few others, but let's reserve them as PS6 ☺



A

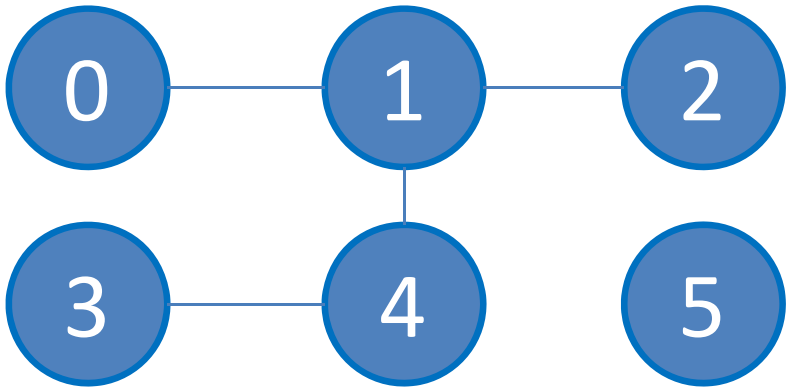
	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

B

1	→	2	→	3			
2	→	1	→	3	→	4	
3	→	1	→	2	→	5	
4	→	2	→	5			
5	→	3	→	4	→	6	
6	→	5	→	7			
7	→	6					

Java Implementation (2)

- Implementation of what we have discussed so far (in adjacency list mode only, try adjacency matrix by yourself):
  - Counting V
  - Enumerating neighbors of a vertex v
  - Counting E
  - Checking the existence of edge(u, v)



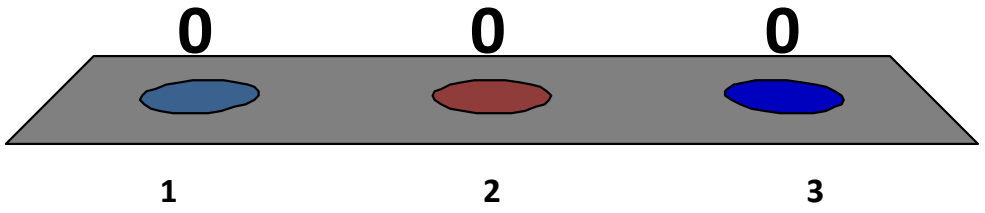
Trade-Off (2)

- Adjacency Matrix:
  - Pro:
    - Existence of edge i-j can be found in O(1)
    - Good for dense graph/ Floyd Warshall's*
  - Cons:
    - O(V) to enumerate neighbors of a vertex
    - $O(V^2)$ space
- Adjacency List:
  - Pro:
    - O(k) to enumerate k neighbors of a vertex
    - Good for sparse graph/ Dijkstra's*/DFS/BFS, O(V + E) space
  - Cons:
    - O(k) to check the existence of edge i-j
    - A little bit overhead in maintaining the list (for sparse graph)

Which One To Use? (1)

V = 10000, E = 10000

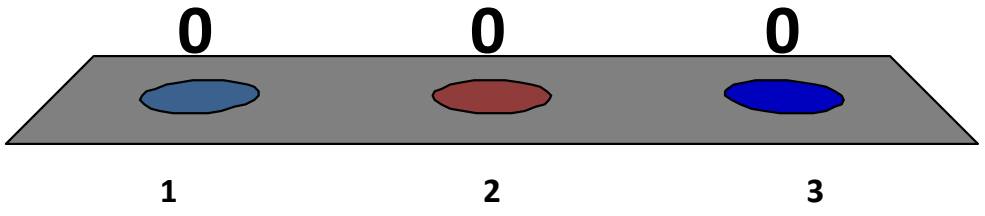
1. Adjacency Matrix
2. Adjacency List
3. This is a trick question, the answer must be something else, which is _____



Which One To Use? (2)

V = 100, existence of edge(u, v) frequently asked

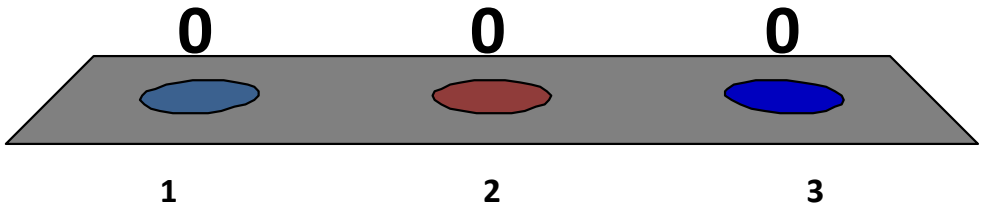
1. Adjacency Matrix
2. Adjacency List
3. This is a trick question, the answer must be something else, which is _____



Which One To Use? (3)

V = 200, E = 19900, neighbors frequently enumerated

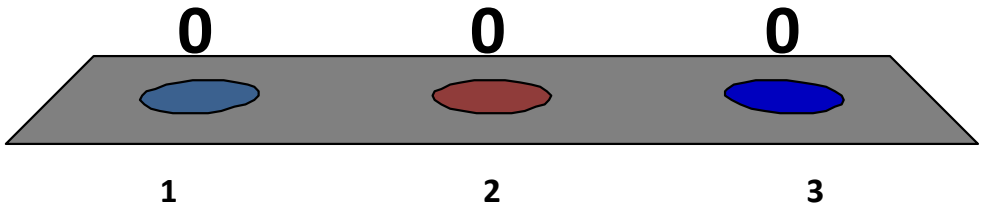
1. Adjacency Matrix
2. Adjacency List
3. This is a trick question, the answer must be something else, which is _____



Which One To Use? (4)

V = 200, E = 19900, sort the edges based on weight

1. Adjacency Matrix
2. Adjacency List
3. This is a trick question, the answer must be something else, which is _____



PS6 Preview

- Two Programming Tasks
  - Task 1: Hashing, Longest Common Substring
  - Task 2: Simple Graph DS Manipulation
  - Due next Wed, 16 March 2011, 13:59

Summary

- In this lecture we looked at:
  - Graph terminologies + why we have to learn graph
  - How to store graph information in computer memory
  - Some very very simple applications
- This Thursday's discussion group
  - For last week's Hash Tables
- See you again on Friday
  - Lecture on "Graph Traversal"