CS2040S – Data Structures and Algorithms

Lecture 15 – The Foundations ~ Graphs

chongket@comp.nus.edu.sg

NUS
National University
of Singapore

School *of* Computing

Outline of this Lecture

A.   Motivation on why you should learn graph

  –   Graph terminologies (from CS1231/CS1231S)

B.   Three Graph Data Structures

  –   Adjacency Matrix

  –   Adjacency List

  –   Edge List

  –   https://visualgo.net**/en/graphds**

C.   Some Graph Data Structure Applications

D.   This lecture is setup for the rest of the module on graph DSes and algorithms

Introductory material

Note that graph will appear from now onwards

GRAPH

Graph Terminologies (1)

Extension from what you already know: *(Binary) Tree*

- Vertex, Edge, Direction (of Edge), Weight (of Edge)

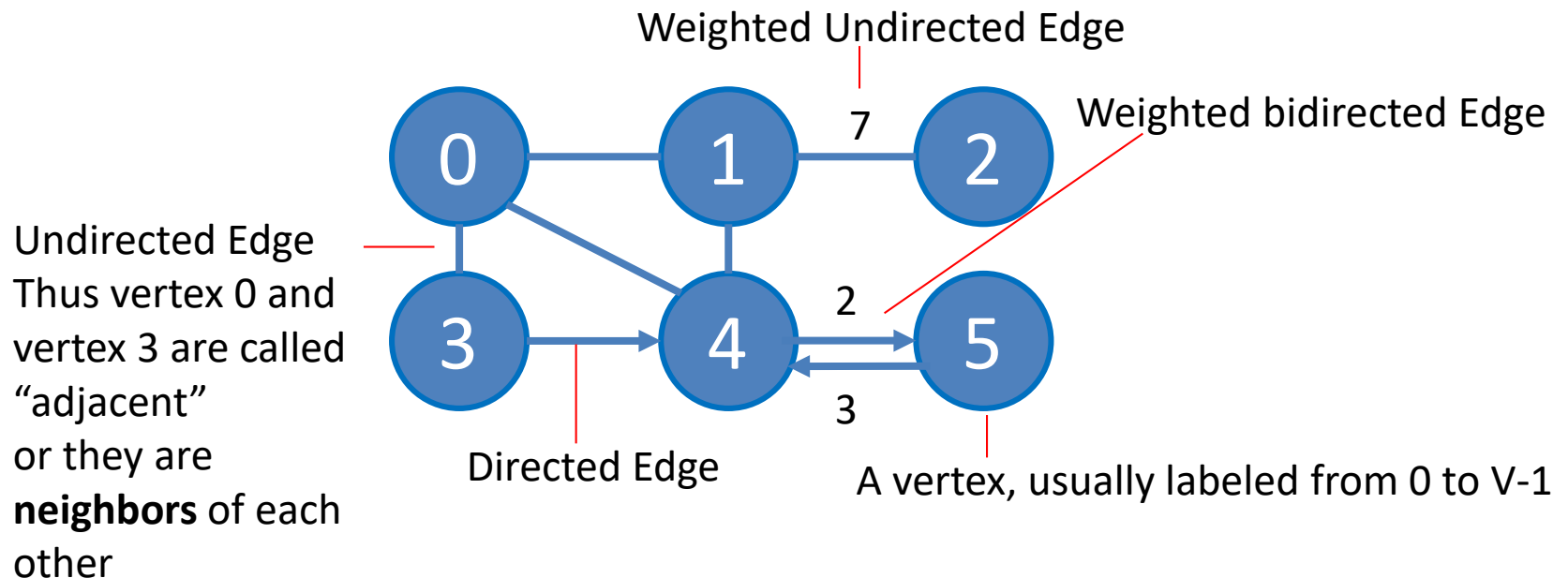But in a general graph, there is no notion of:
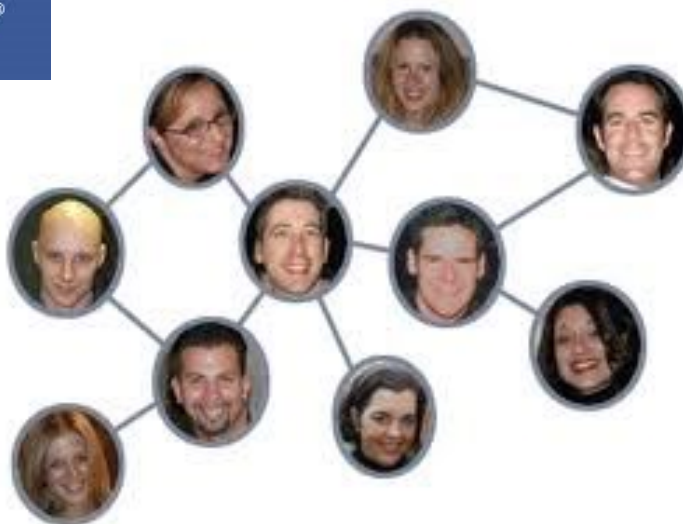
- Root
- Parent/Child
- Ancestor/Descendant

Graph is…

(Simple) graph is a set of vertices where some
[0 .. $_NC_2$] pairs of the vertices are connected by edges

- Other then bi-directed edges, we will ignore "multi graph" where there can be more than one edge between a pair of vertices

Weighted Undirected Edge

Weighted bidirected Edge

0 1 7 2

Undirected Edge
Thus vertex 0 and
vertex 3 are called
"adjacent"
or they are
neighbors of each
other

3 4 2 5

3

Directed Edge

A vertex, usually labeled from 0 to V-1

Social Network

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 $_NC_2$ 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, $_7C_2$ = 21 edges
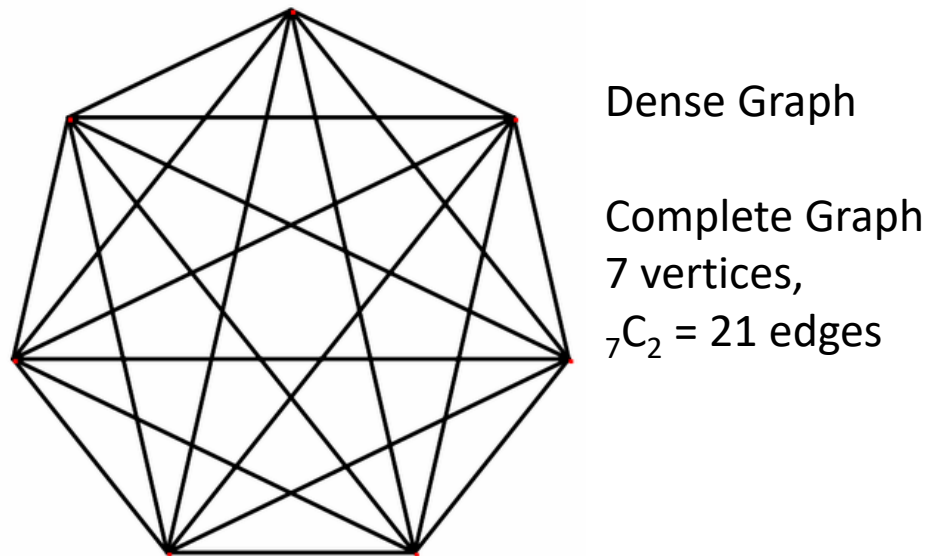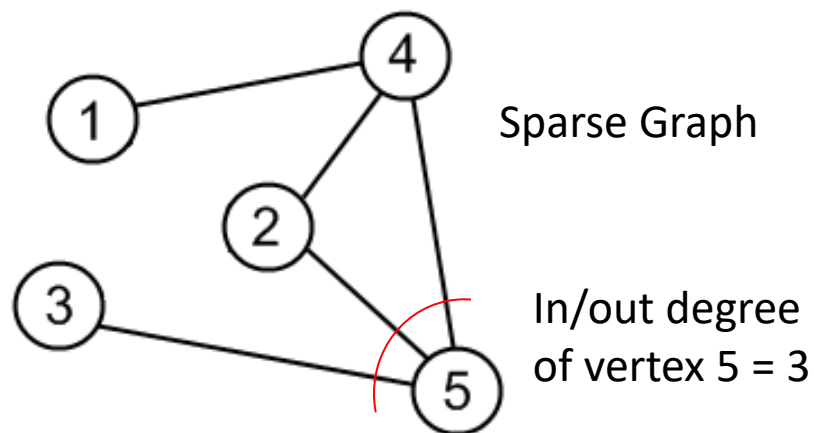
Graph Terminologies (3)

Yet more terminologies (example in the next slide):

- (Simple) Path
  - Sequence of vertices connected by a sequence of edges
  - Simple = no repeated vertex
- (Simple) Directed Path
  - Same as (Simple) Path with the added restriction that the edges in the path are directed and in the same direction
- Path Length/Cost
  - In unweighted 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 vertices except start/end vertex
- (Simple) Directed Cycle
  - Same as (Simple) Cycle with added restriction that the edges in the cycle are directed and in the same direction
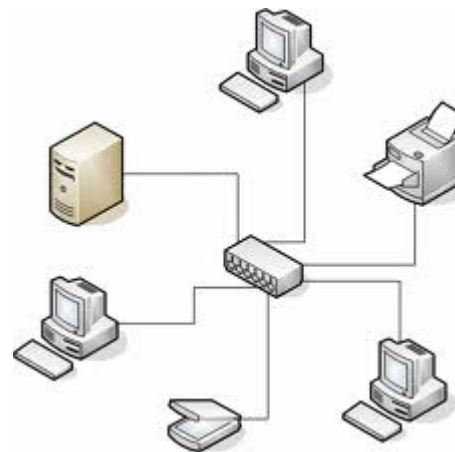
Transportation Network

Simple cycle
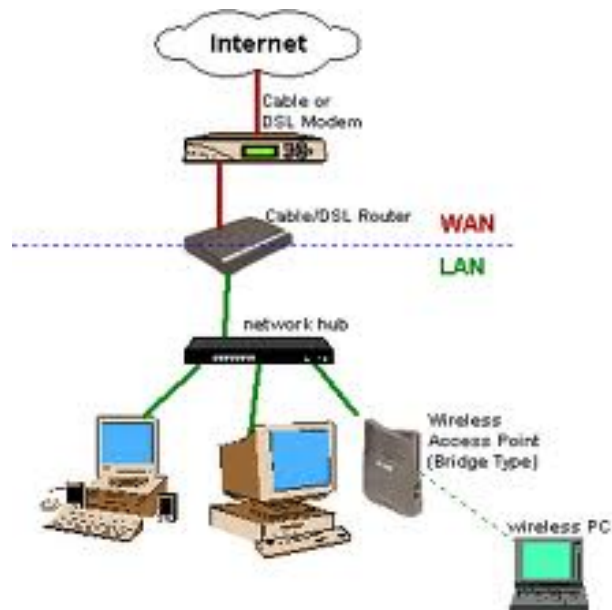Sengkang LRT, west **loop**)
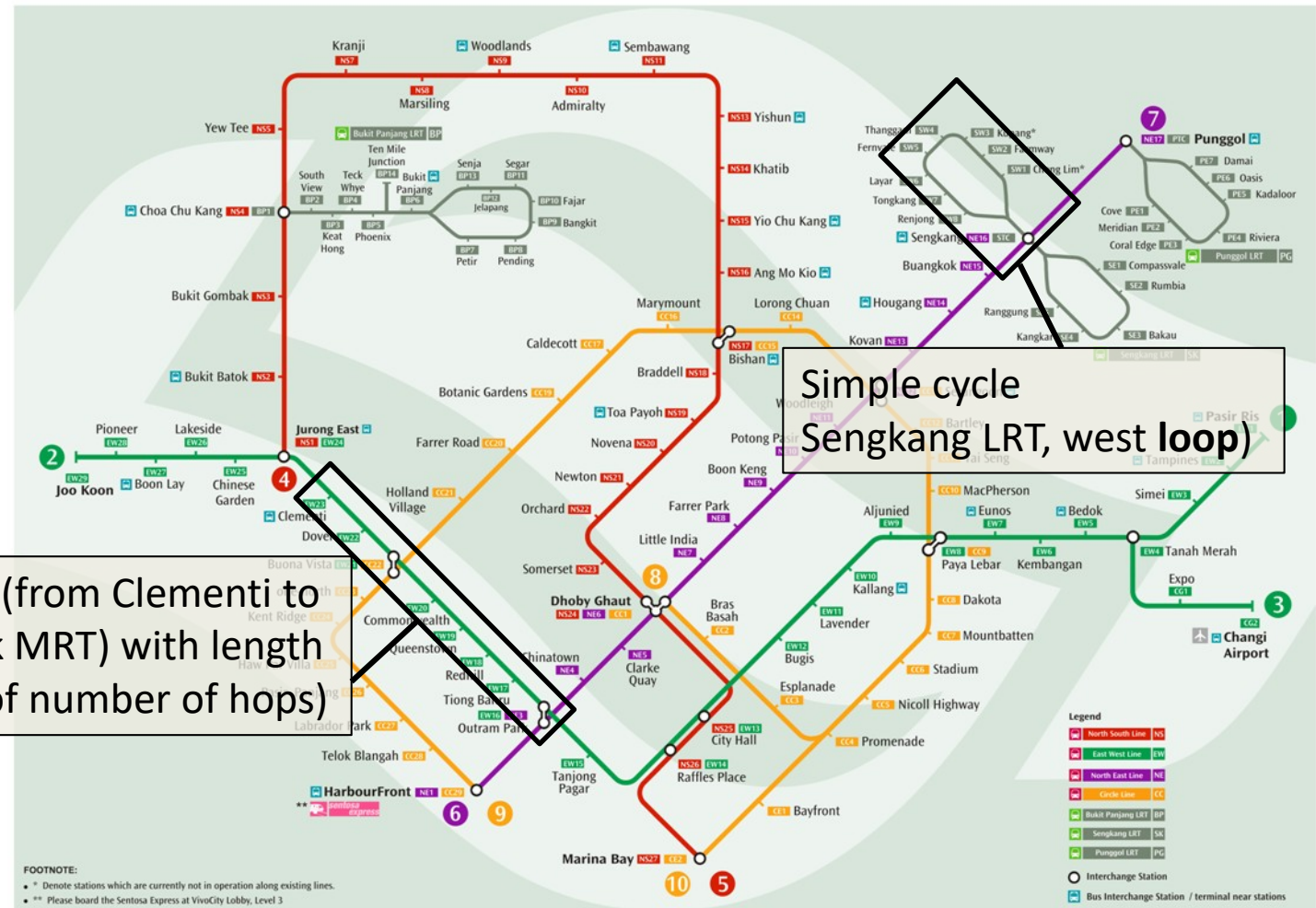
Simple path (from Clementi to Outram Park MRT) with length 7 (in terms of number of hops)
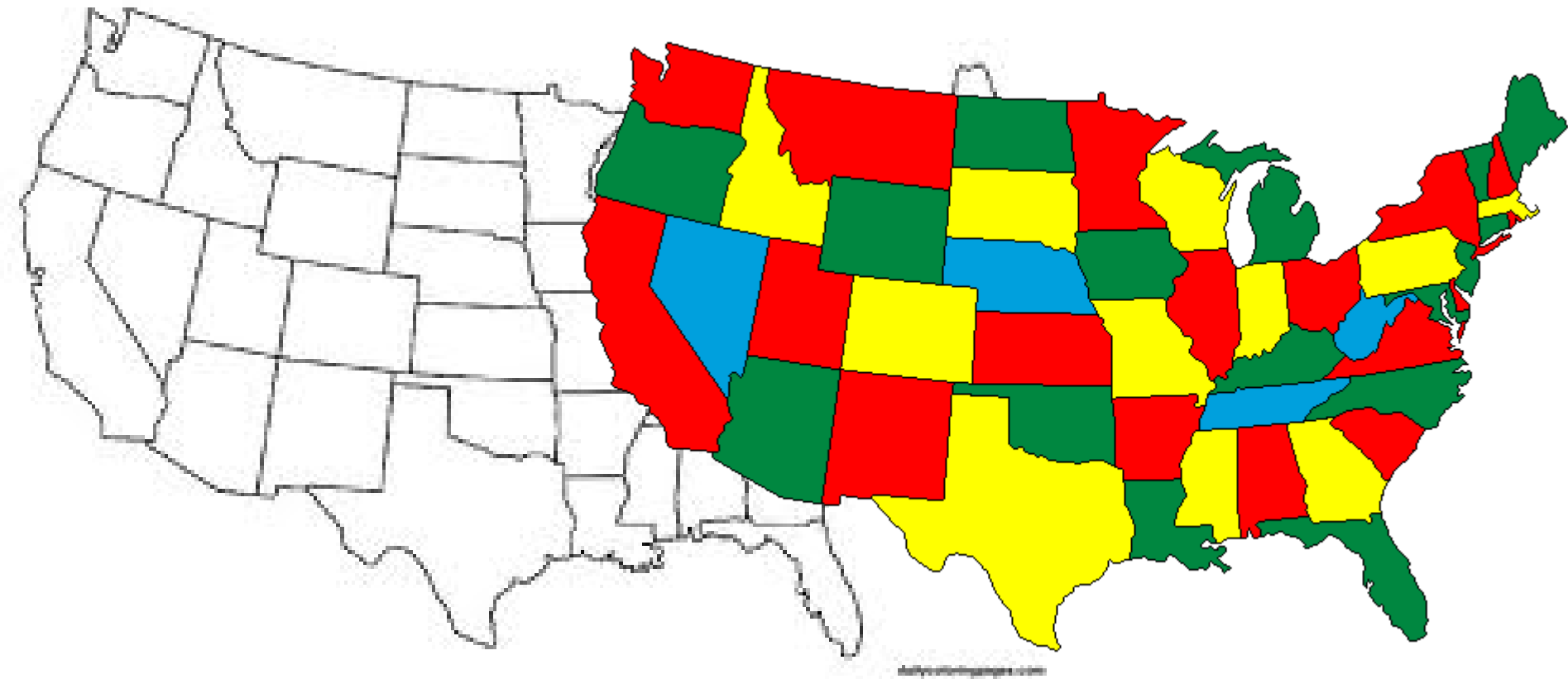
Internet / Computer Networks

Communication Network

Optimization

Graph Terminologies (4)

Yet More Terminologies:

- Component
  - A maximal group of vertices in an Undirected graph that can visit each other via some path
- Connected graph
  - Undirected graph with 1 component
- Reachable/Unreachable Vertex
  - See example
- Sub Graph
  - Subset of vertices (and their connecting edges) of the original graph
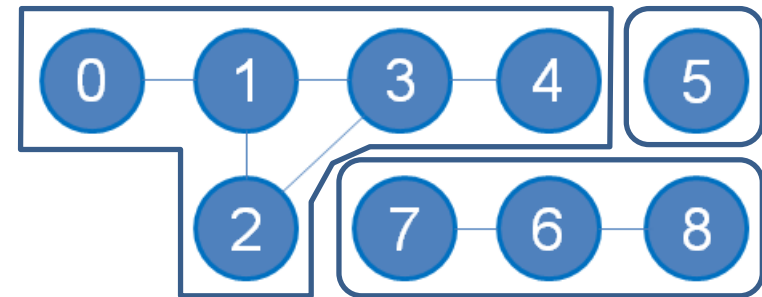
- There are 3 components in this graph
- Disconnected graph
(since it has > 1 component)
- Vertices 1-2-3-4 are reachable from vertex 0
- Vertices 5, 6-7-8 are unreachable from vertex 0
- {7-6-8 5} is a sub graph of this graph

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)
  - Undirected graph where we can partition the vertices into two sets so that there are no edges between members of the same set

Out degree of vertex 0 = 2

In degree of vertex 2 = 2
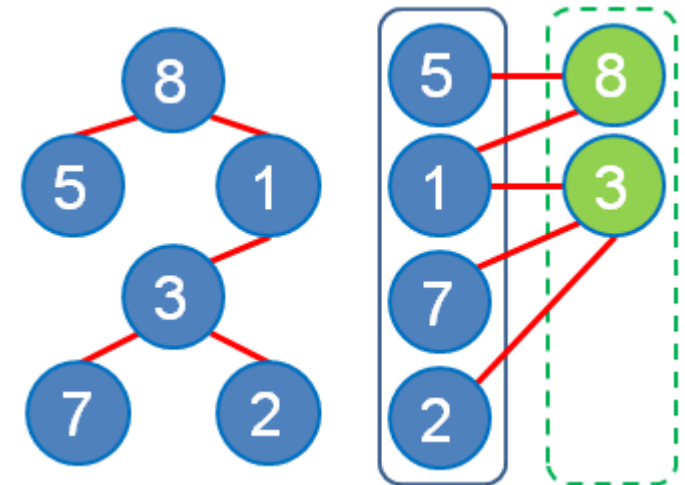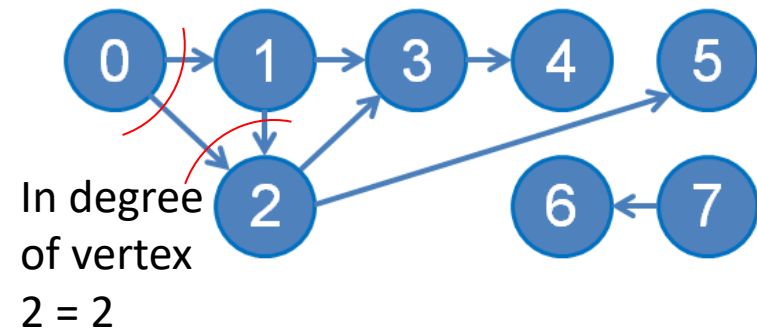
Next, we will discuss three Graph DS

https://visualgo.net**/en/graphds**

This DS will used for the rest of the module

GRAPH DATA STRUCTURES

Adjacency Matrix

Format: a 2D array **AdjMatrix** (see an 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
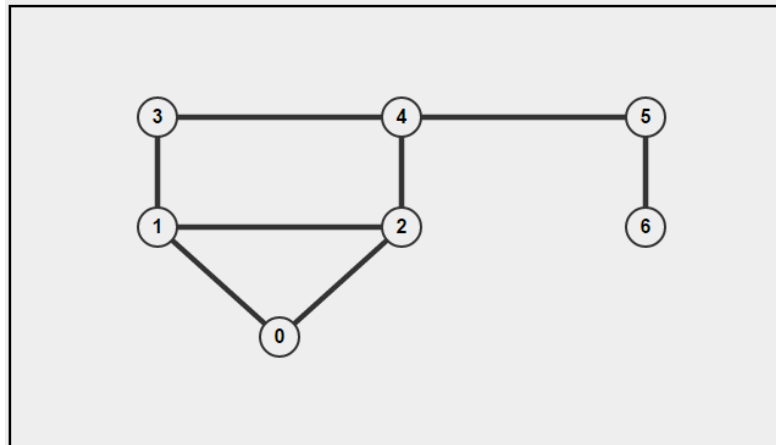
Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

Format: **AdjList** is array of **V** lists, one for each vertex

For each vertex **i**, **AdjList[i]** stores list of **i**'s neighbors

- For weighted graph, stores **pair (neighbor, weight)**
  - Note that for unweighted graph, we can also use the same strategy as the weighted version using (neighbor, weight), but the weight is set to 0 (unused) or set to 1 (to say unit weight)

Space Complexity: O(V+E)

- **E** is |E| = number of edges in G, **E = O(V²)**

- **V+E ~= max(V, E)**
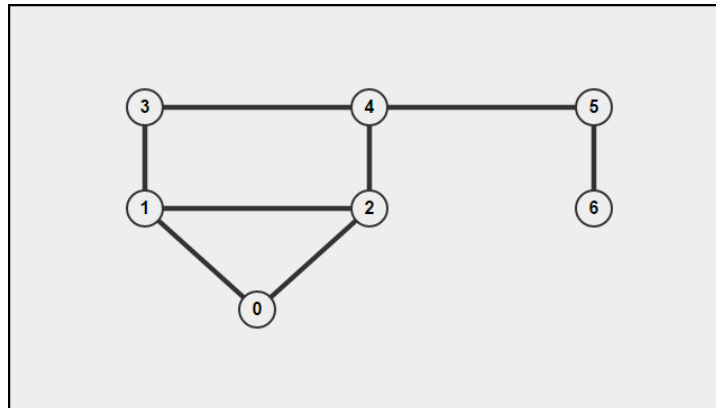
Adjacency List		
0:	1	2
1:	0	2	3
2:	0	1	4
3:	1	4
4:	2	3	5
5:	4	6
6:	5

Edge List

Format: array **EdgeList** of **E** edges

For each edge **i**, **EdgeList[i]** stores an (integer) triple {u, v, w(u, v)}

- For unweighted graph, the weight can be stored as 0 (or 1),
  or simply store an (integer) pair

Space Complexity: **O(E)**

- Remember,
  E = O(V²)
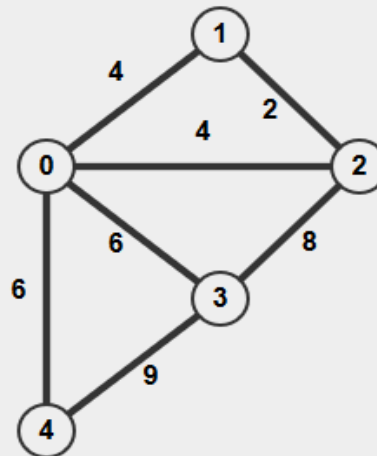
Edge List			
0:	0	1	4
1:	0	2	4
2:	0	3	6
3:	0	4	6
4:	1	2	2
5:	2	3	8
6:	3	4	9

VisuAlgo Graph DS Exploration (1)

Click each of the sample graphs one by one and verify the content of the corresponding **Adjacency Matrix**, **Adjacency List**, and **Edge List**
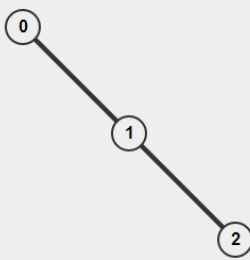
VisuAlgo Graph DS Exploration (2)

Now, use your mouse over the currently displayed graph **and start drawing some new vertices and/or edges** and see the updates in AdjMatrix/AdjList/EdgeList structures

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: With Java Collections framework

```
ArrayList < ArrayList < IntegerPair > > AdjList =
  new ArrayList < ArrayList < IntegerPair > >();
// IntegerPair is a simple integer pair class
// to store pair info, see the next slide
```

Edge List: Also with Java Collections framework

```
ArrayList < IntegerTriple > EdgeList =
  new ArrayList < IntegerTriple >();
// IntegerTriple is similar to IntegerPair
```

PS: This is *one* implementation, there are other ways

Java Implementation (2)

```
class IntegerPair implements Comparable<IntegerPair> {
  Integer _first, _second;
  public IntegerPair(Integer f, Integer s) {
    _first = f;
    _second = s;
  }
  public int compareTo(IntegerPair o) {
    if (!this.first().equals(o.first()))  // this.first() != o.first()
      return this.first() - o.first();    // is wrong!, we want to
    else                                  // compare their values,
      return this.second() - o.second();  // not their references
  }
  Integer first() { return _first; }
  Integer second() { return _second; }
}
// IntegerTriple is similar to IntegerPair, just that it has 3 fields
```

SOME GRAPH DATA STRUCTURE APPLICATIONS

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

With just graph DS, not much, but here are some:

- Counting **V** (or **|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 this every time, that is, O(**1**), *especially if the graph never changes after it is created*
  - To think about: How about **EdgeList**?

Adjacency Matrix							
	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List		
0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

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

- Enumerating neighbors of a vertex **v**

   See live lecture

Adjacency Matrix							
	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List		
0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

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

- Counting **E** (the number of edges)

See live lecture

Adjacency Matrix							
	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List		
0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

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

- Checking the existence of edge(**u**, **v**)

  See live lecture

Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List

0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

Trade-Off

Adjacency Matrix

Pros:

- Existence of edge i-j can be found in O(1)
- Good for dense graph/ Floyd Warshall's (Week 12)

Cons:

- O(V) to enumerate neighbors of a vertex
- O(V^2) space

Adjacency List

Pros:

- 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 small overhead in maintaining the list (for sparse graph)

Summary

In this lecture, we looked at:

A. Graph terminologies + why we have to learn graph
   – for Week 9-12 ....

B. How to store graph info

C. Some simple graph data structure applications