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

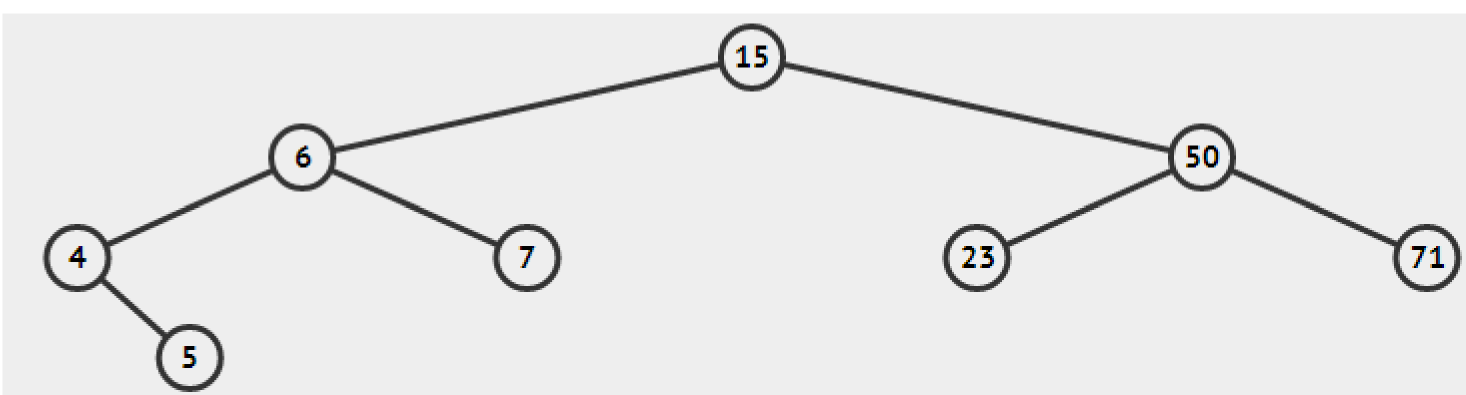
Graph Terminologies (1)

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



Note: definitions here might be a bit different from CS1231/S

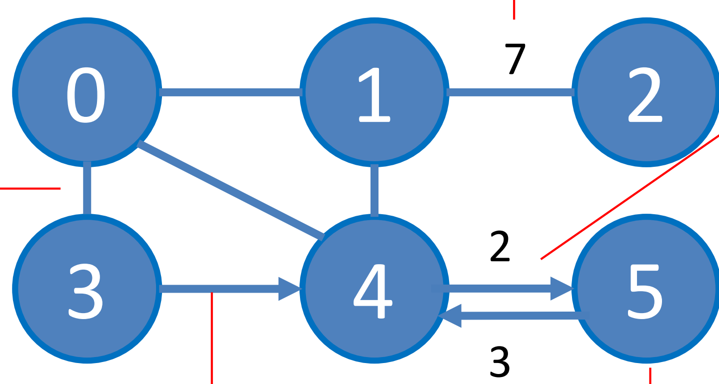
Graph is...

(Simple) graph is a set of vertices where some $[0 \; .. \; {}_{N}C_{2}]$ pairs of the vertices are connected by edges (3 types – undirected, directed, bi-directed)

- We will ignore "multi graph" where there can be more than one edge (of any edge type) between a pair of vertices

Weighted Undirected Edge

Weighted Bi-directed Edge (made of 2 directed edges going in opposite directions)



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

A graph with a combination of undirected+directed edge is called a **mixed graph**

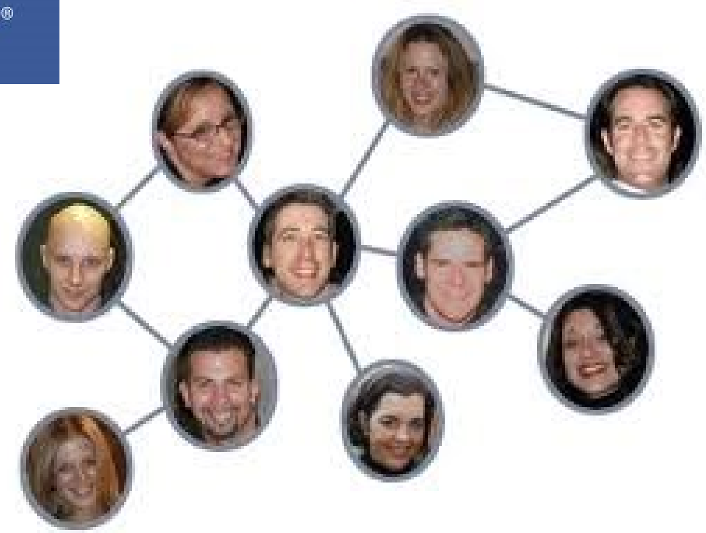
Directed Edge

A vertex, usually labeled from 0 to V-1

Social Network

facebook

twitter



LinkedIn

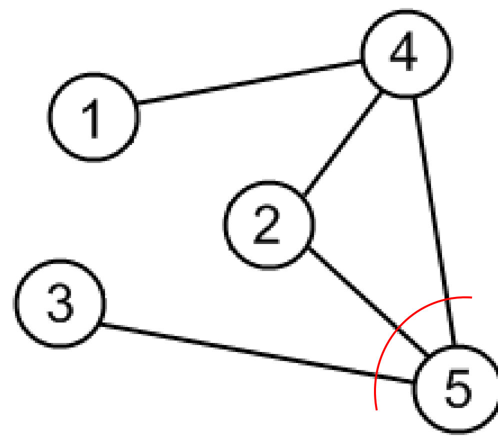
g+

Graph Terminologies (2)

More terminologies (simple graph):

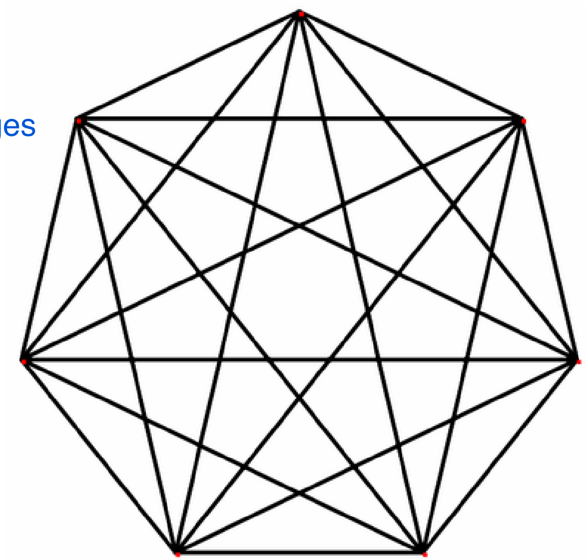
- Sparse/Dense
  - Sparse = not so many edges
  - Dense = many edges
  - No guideline for "how many"
- Complete Graph
  - Simple graph with N vertices and ${}_N C_2$ edges n(n-1)/2 num of edges
- In/Out Degree of a vertex
  - Number of in/out edges from a vertex

in undirected graph, in and out degrees are the same



Sparse Graph

In/out degree of vertex 5 = 3



Dense Graph

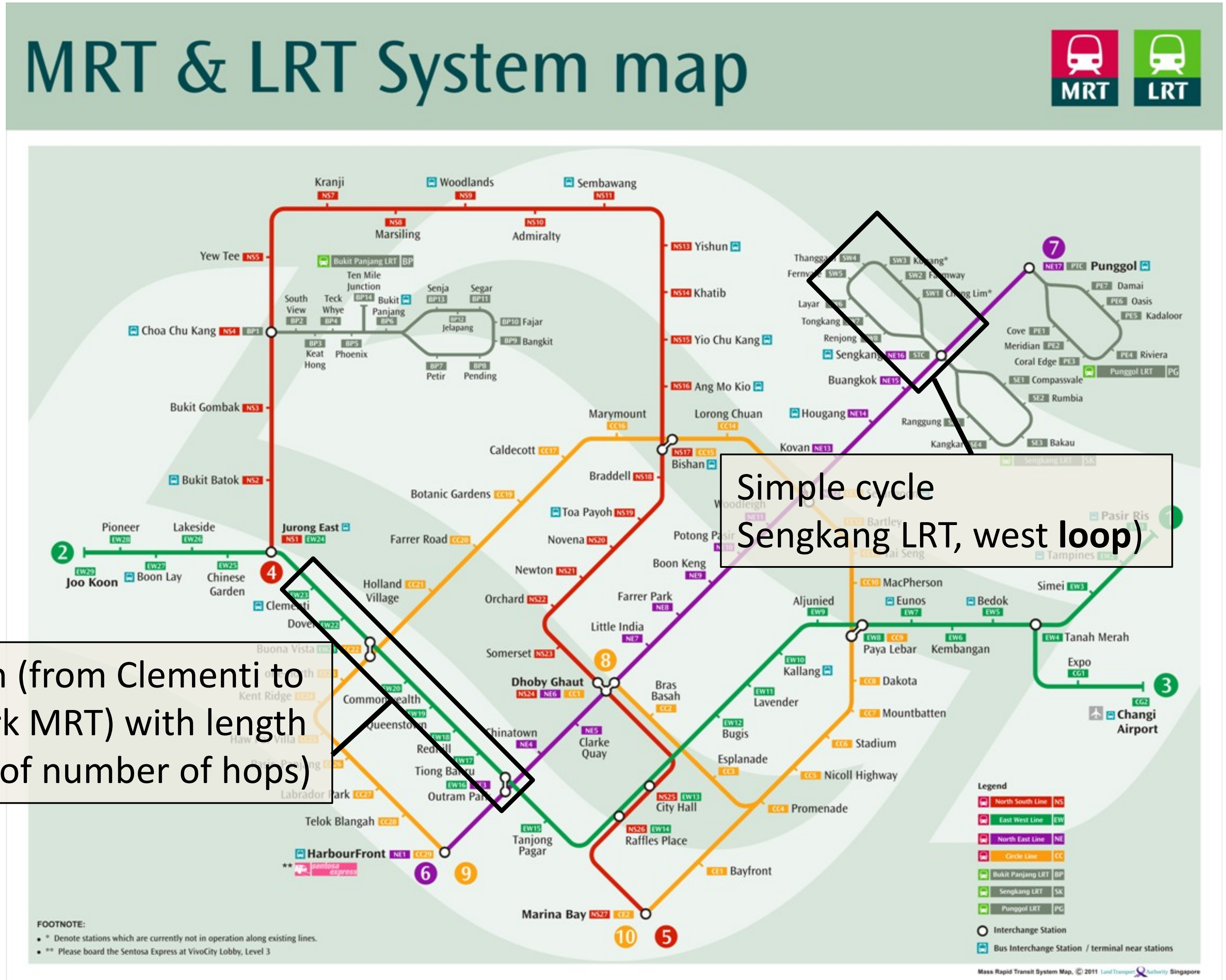
Complete Graph
7 vertices,
${}_7 C_2$ = 21 edges

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
  - A path with only 1 vertex and no edge is an empty path
- (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 and no repeated edges
  - Involves 3 or more unique vertices
- (Simple) Directed Cycle
  - Same as (Simple) Cycle with added restriction that the edges in the cycle are directed and in the same direction
  - Involves 2 or more unique vertices

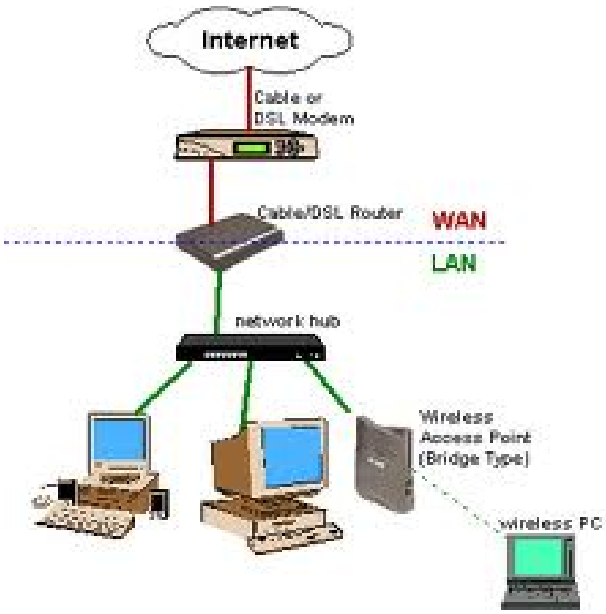
Transportation Network

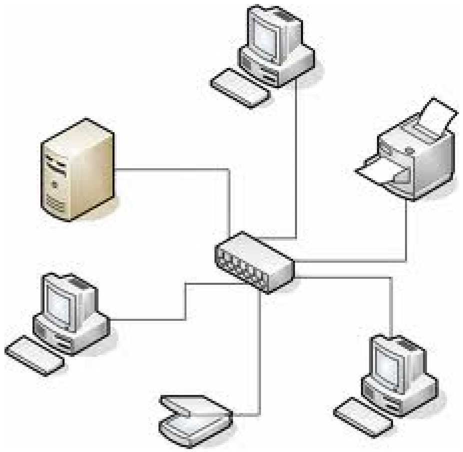


Simple cycle
Sengkang LRT, west **loop**)

Simple path (from Clementi to Outram Park MRT) with length 7 (in terms of number of hops)

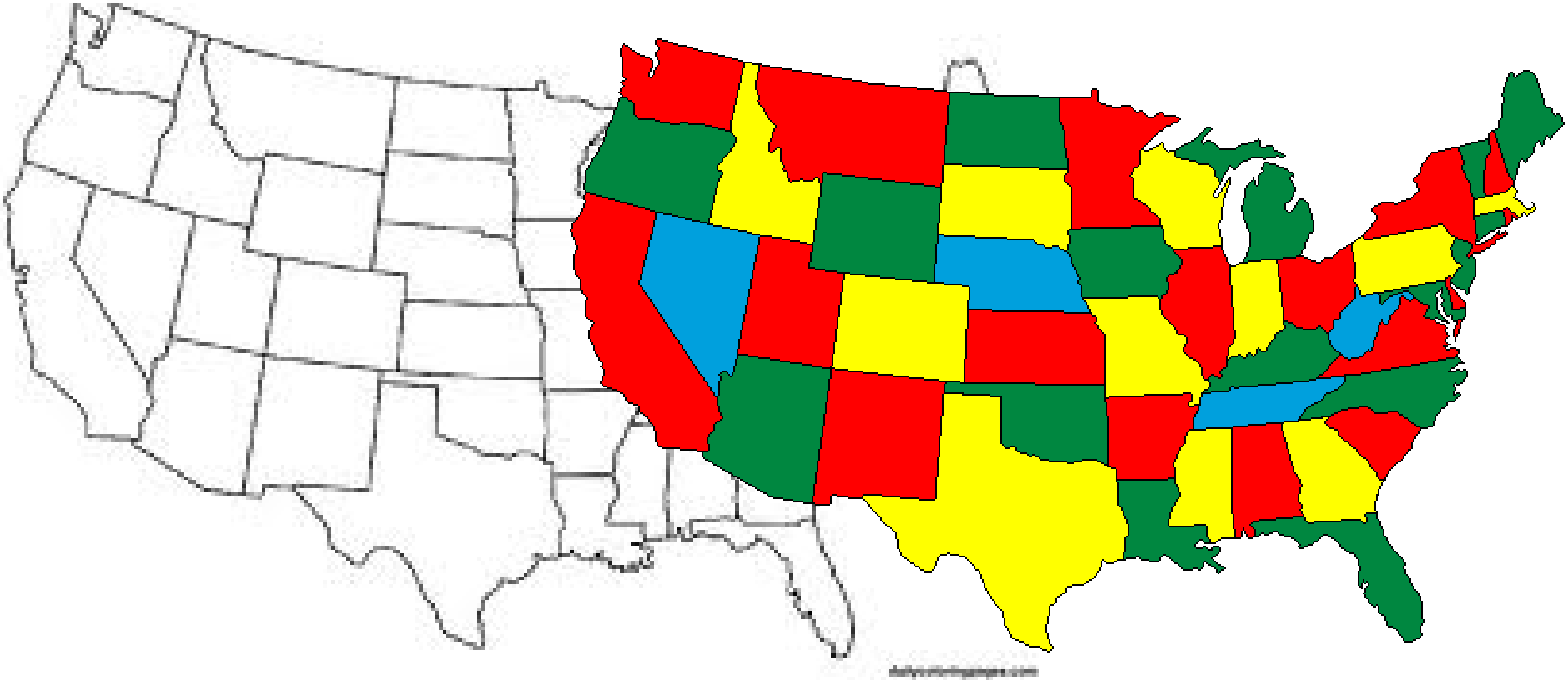
Internet / Computer Networks





Communication Network

Optimization

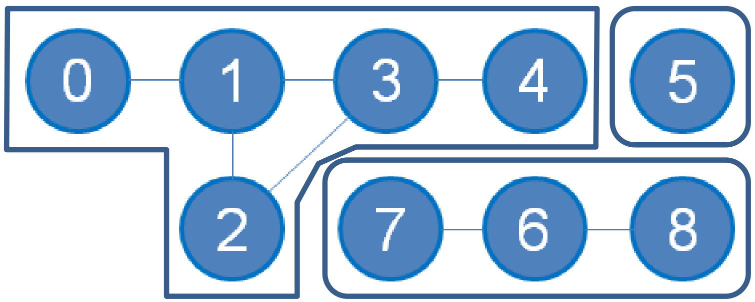


Graph Terminologies (4)

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

it maximal because we cannot add anymore vertices such that its still a component



- 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

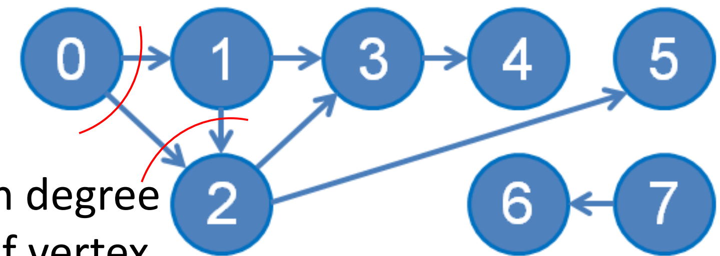
Graph Terminologies (5)

Yet More Terminologies:

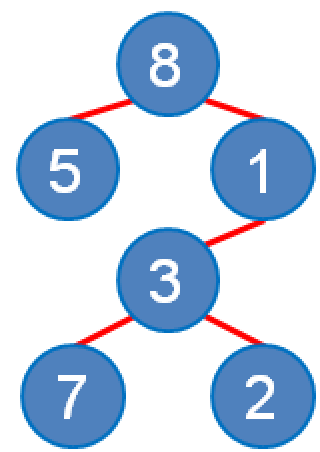
- Directed Acyclic Graph (DAG)
  - Directed graph that has no cycle
- Tree (bottom left) the min no of edges s.t. the graph is still connected
  - Connected graph, E = V - 1, one unique path between any pair of vertices if we add more edges it turns into a cycle E > V -1
- Bipartite Graph (bottom right)
  - Undirected graph where we can partition the vertices into two sets so that there are no edges between members of the same set

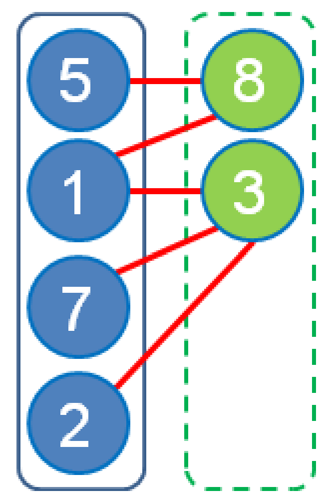
is any tree a bipartite graph? no? because they all in the same set?

Out degree of vertex 0 = 2



In degree of vertex 2 = 2





A bipartite graph has the following two properties:

1. 2-colourable: it is possible to assign a colour to every vertex in the graph such that every vertex is coloured one of two colours (say red or blue), such that no two adjacent vertices are coloured with the same colour.

2. No odd-length cycles: every cycle in the graph contains an even number of edges

a graph satisfied both properties 1 and 2 at the same time

Every subgraph H of a bipartite graph G is, itself, bipartite.

is every tree a
bipartite graph?

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

matrix of size v by v where v is the # of vertices in the graph

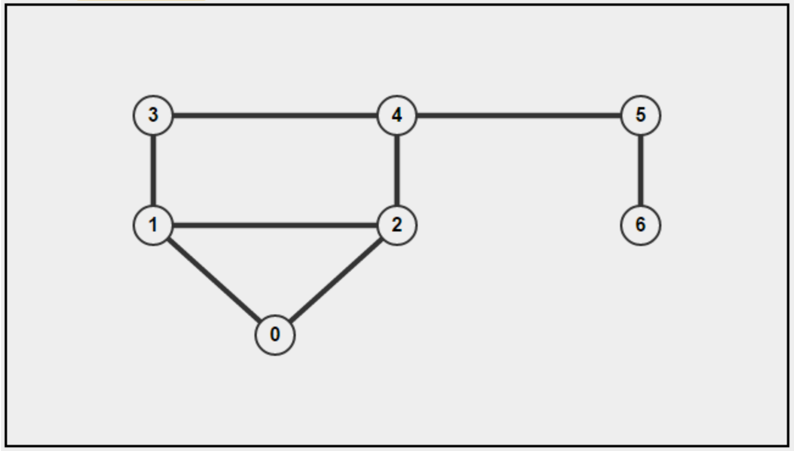
Format: a 2D array **AdjMatrix** (see an example below)

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

each entry is a list

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

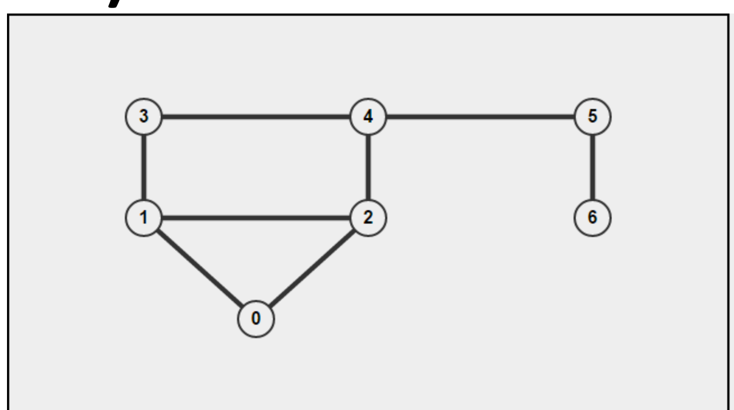
For each vertex **i**, **AdjList[i]** stores list of **i**'s neighbors

- 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)

min E: 0 -> disconnected graph
max E: vC2 = O(v^2) -> complete graph

Space Complexity: **O(V+E)** no of vertices + the no of edges each of them have

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



only stores the neighbours

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		

O(v) + O(2E) -> in undirected graph if 1 is neigh of 0, 0 is neigh of 1

Edge List

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

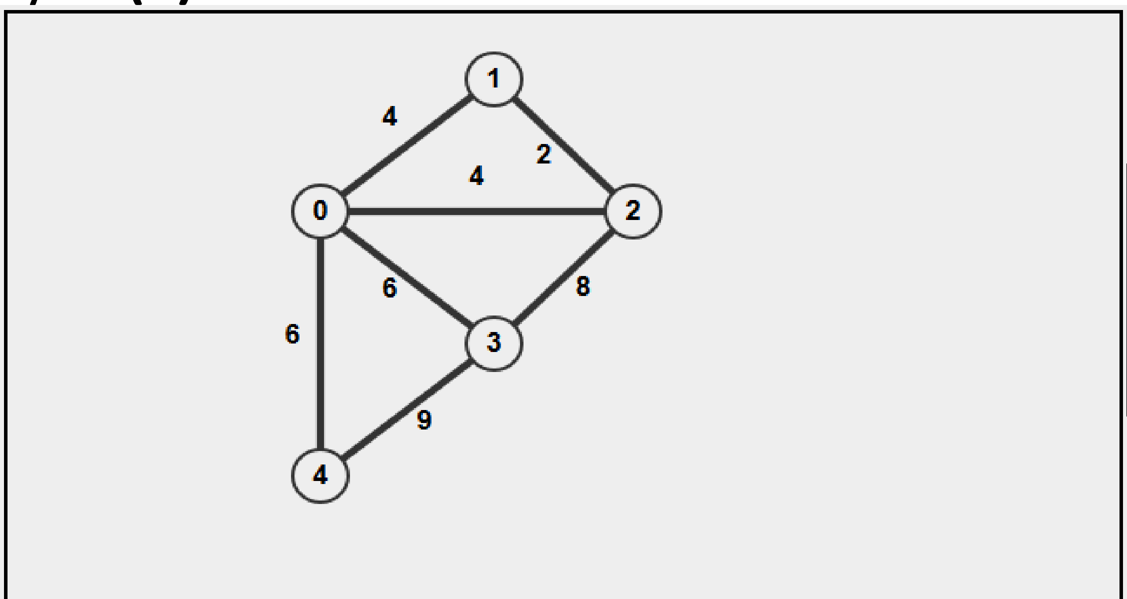
vertex u, v and weight of edge between u and v

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)**

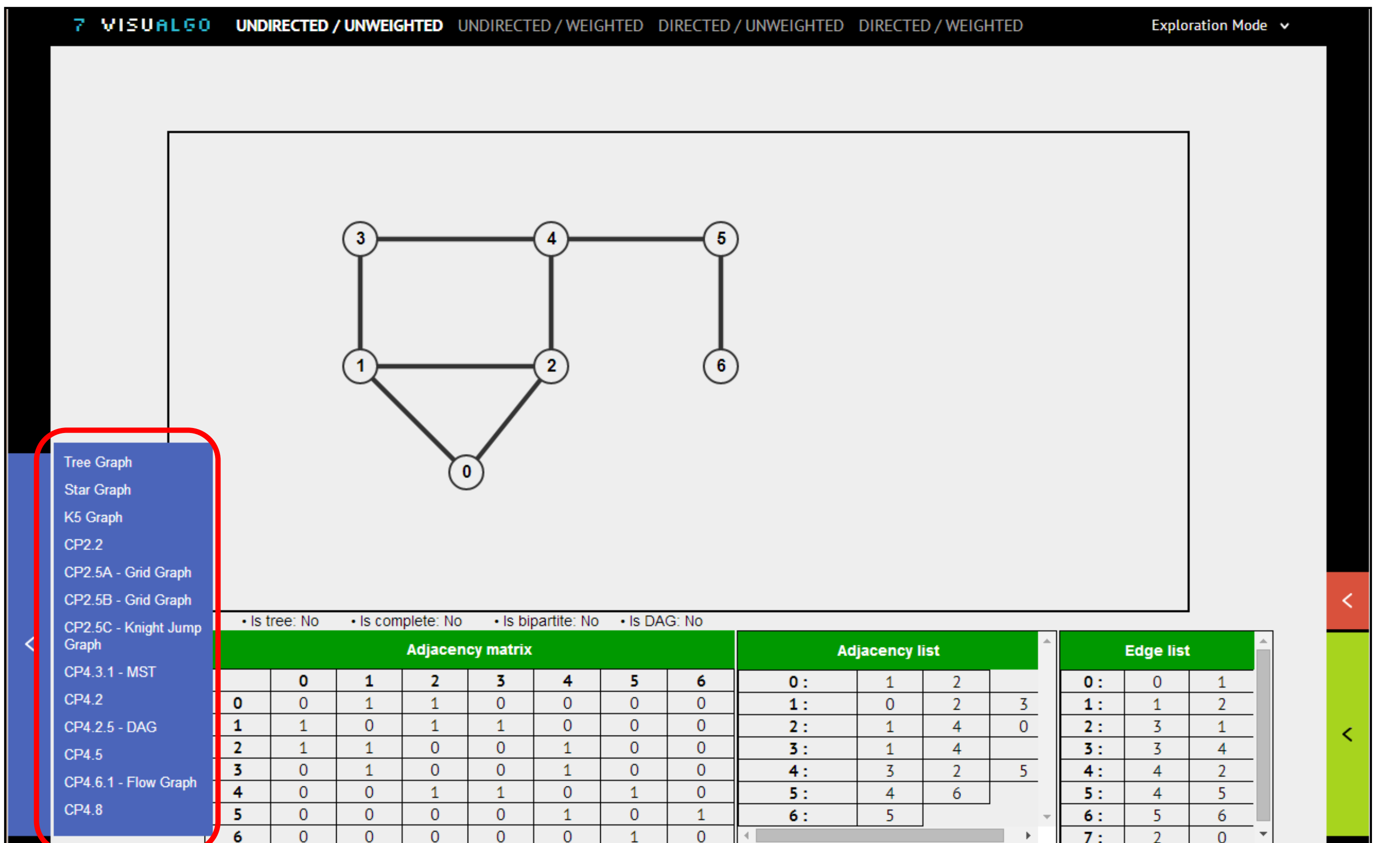
- Remember, **$E = O(V^2)$**
  in the worst case



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

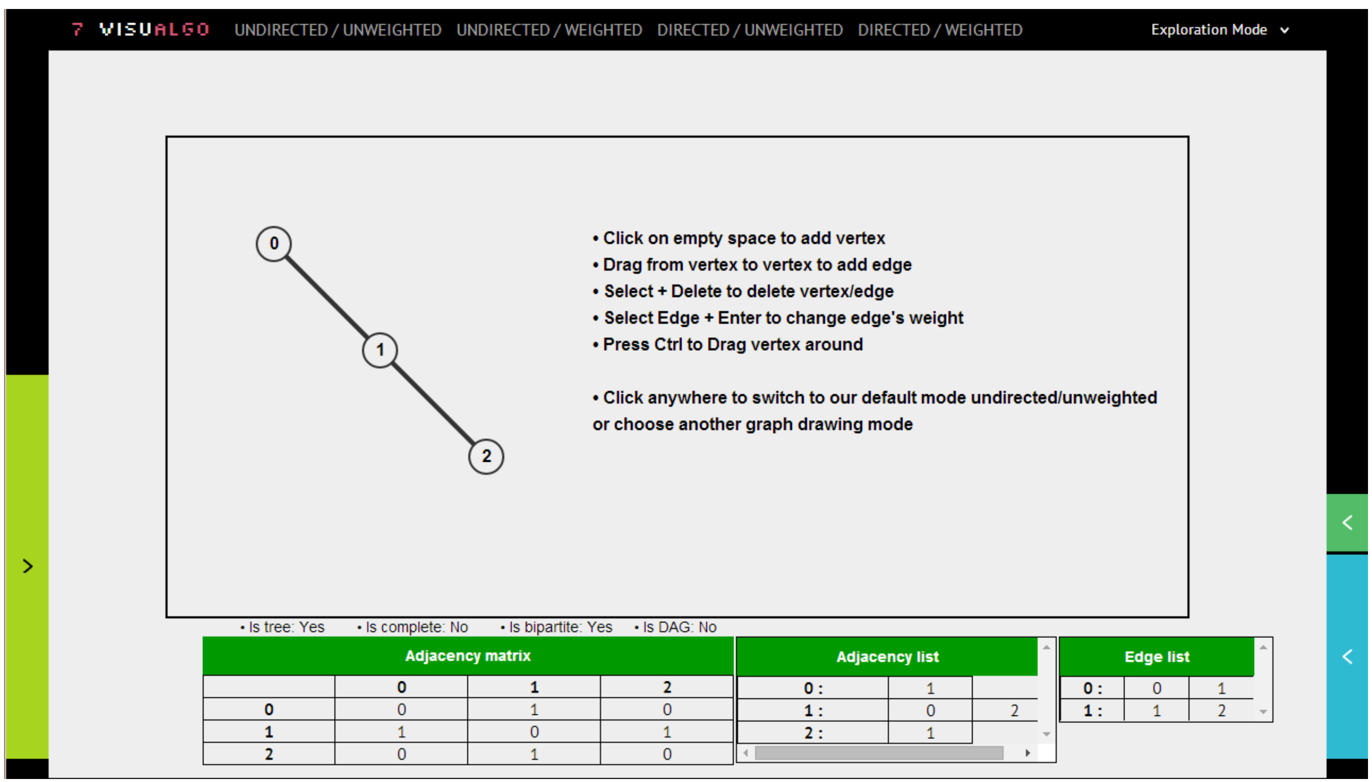
VisuAlgo Graph DS Exploration (1)

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**?

we cant because of if a graph has no edges it wont be stores in the edge list

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

O(V) for AdjMatrix: scan AdjMatrix[v][j], for all j in [0..V-1]

O(k) for AdjList, scan AdjList[v]
k is the no of neigh of vertex v (output sensitive algo)

This is impt diff between AdjMatrix and AdjList
it affects the performance of many graph algos

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

EdgeList:

O(1) -> undirected, bidirected edges may be listed once (or twice) in edge list, depending on the need

O(v^2) -> check for non zero values and if it is undirected graph, divide by 2

O(v) -> check for non zero values and if it is undirected graph, divide by 2

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

O(1) -> go row u col v see if its non zero

O(k) -> see if it contains v

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

sparse graph -> # edges <= O(v)
dense graph -> # edges = O(v^2)

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($\mathbf{V}^2$) 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