CS2040 – Data Structures and Algorithms

Lecture 12 – The Foundations ~ Graphs

axgopala@comp.nus.edu.sg

National University of Singapore

School of Computing

Journey so far

- Sorting
- Lists
- HashTable
- Binary Heap
- UFDS
- Ordered Map

With Thanks to

Prof Ket Fah Chong

Prof Roger Zimmermann

What do you remember thus far? ☺

- Go to: https://menti.com (code: 29 66 92 5)

Road Ahead

- Graphs, graphs and more graphs ☺
- Lots of very cool algorithms ☺

Outline of this Lecture

A. Motivation on why you should learn graph
   - Graph terminologies

B. Three Graph Data Structures
   - Adjacency Matrix
   - Adjacency List
   - Edge List
   - [https://visualgo.net**/en/graphds**](https://visualgo.net/en/graphds)

C. Some Graph Data Structure Applications

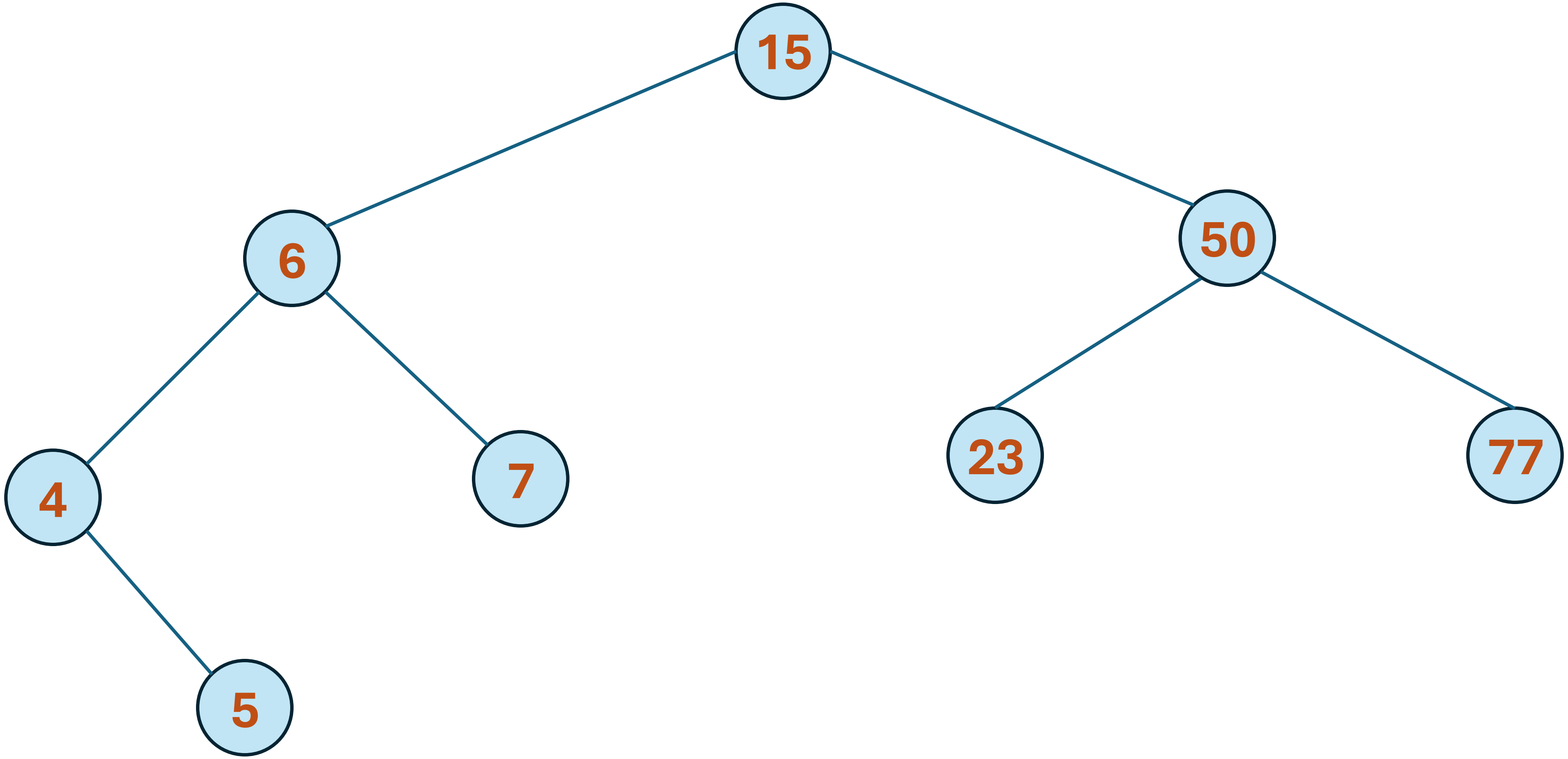
D. This lecture is setup for the rest of the module on graph DSAs

Graph Terminologies (1)

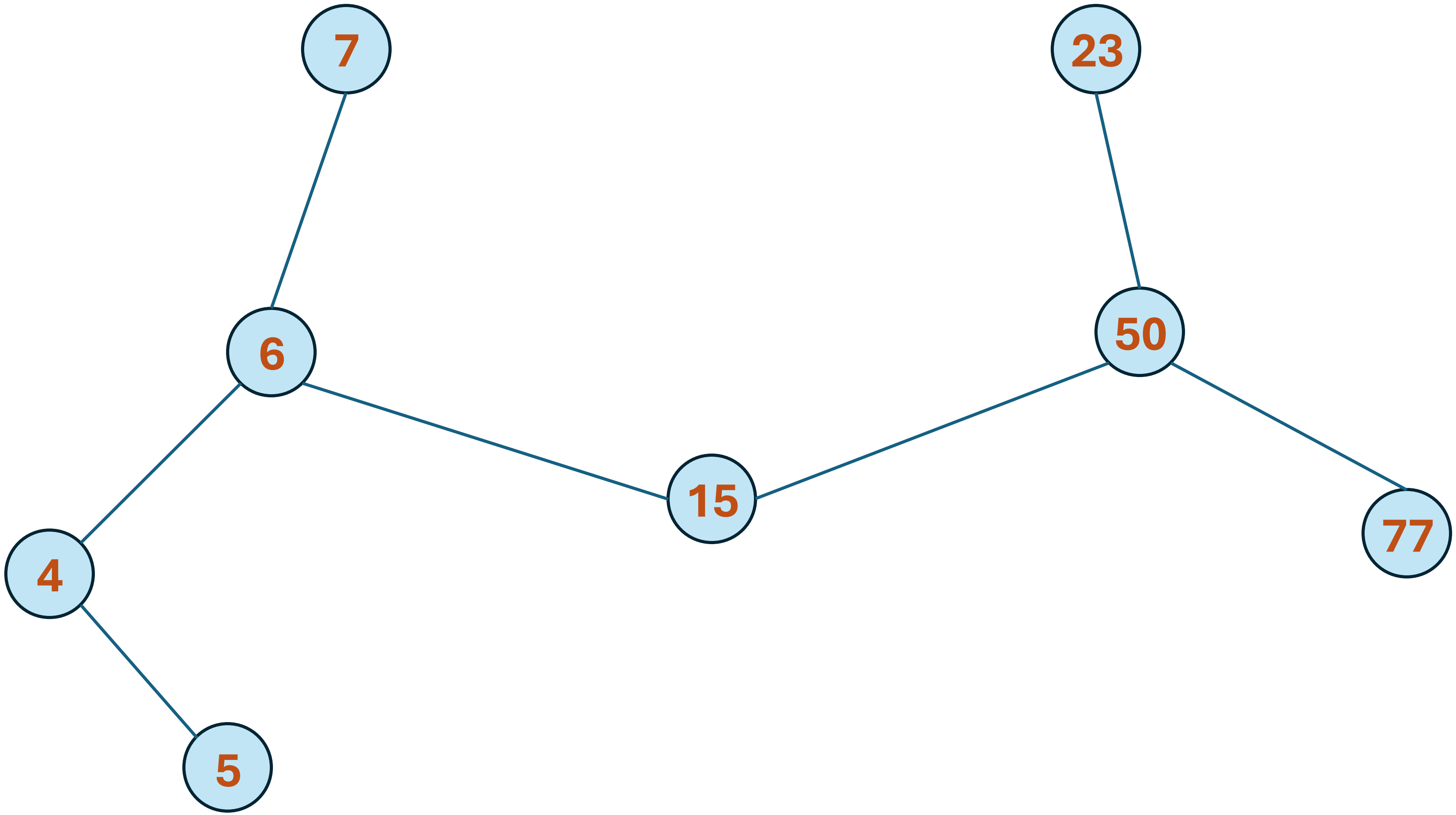
- Extension from what you already know: *(Binary) Tree*
  - Vertex, Edge, Direction (of Edge), Weight (of Edge)

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

Graph (?)



Graph (?)

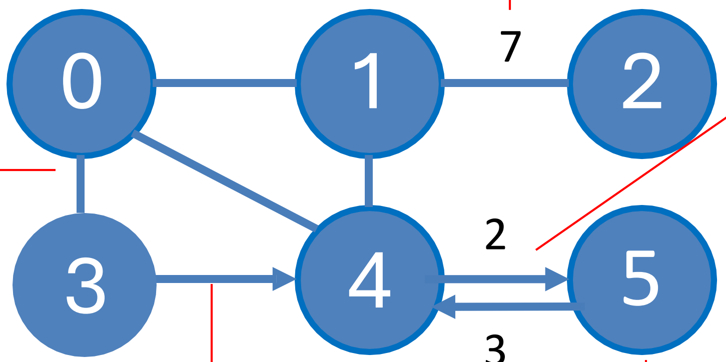


Graph is...

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

- Graph is a set of N 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 **neighbours** of each other

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

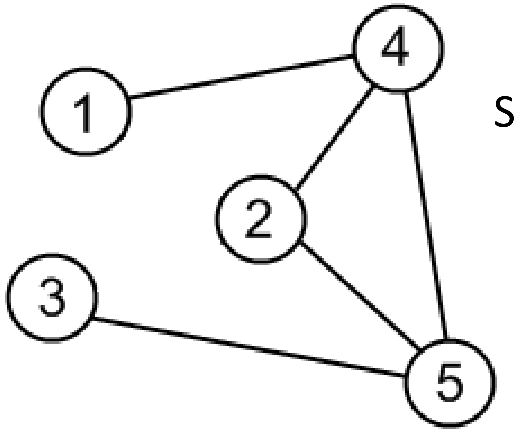
Directed Edge

A vertex, usually labeled from 0 to N-1

Example

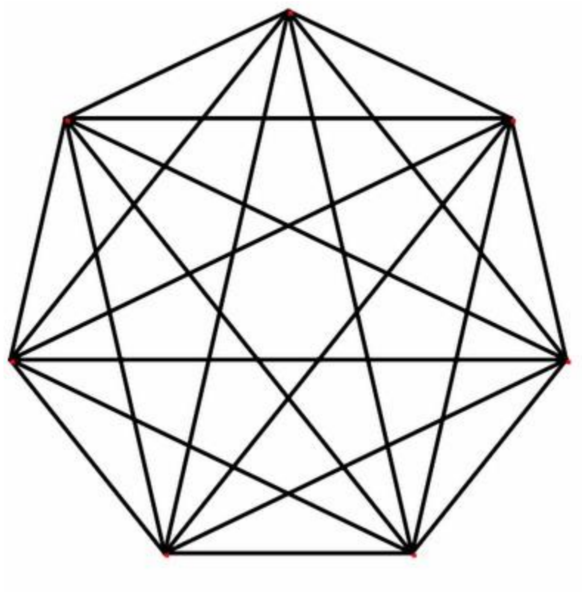
Graph Terminologies (2)

- 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
- 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,
${}_{7}C_2$ = 21 edges

Graph Terminologies (3)

- (Simple) Path
  - Sequence of vertices connected by a sequence of undirected edges
  - Simple = no repeated vertex
  - A path with only 1 vertex and no edge is an empty path

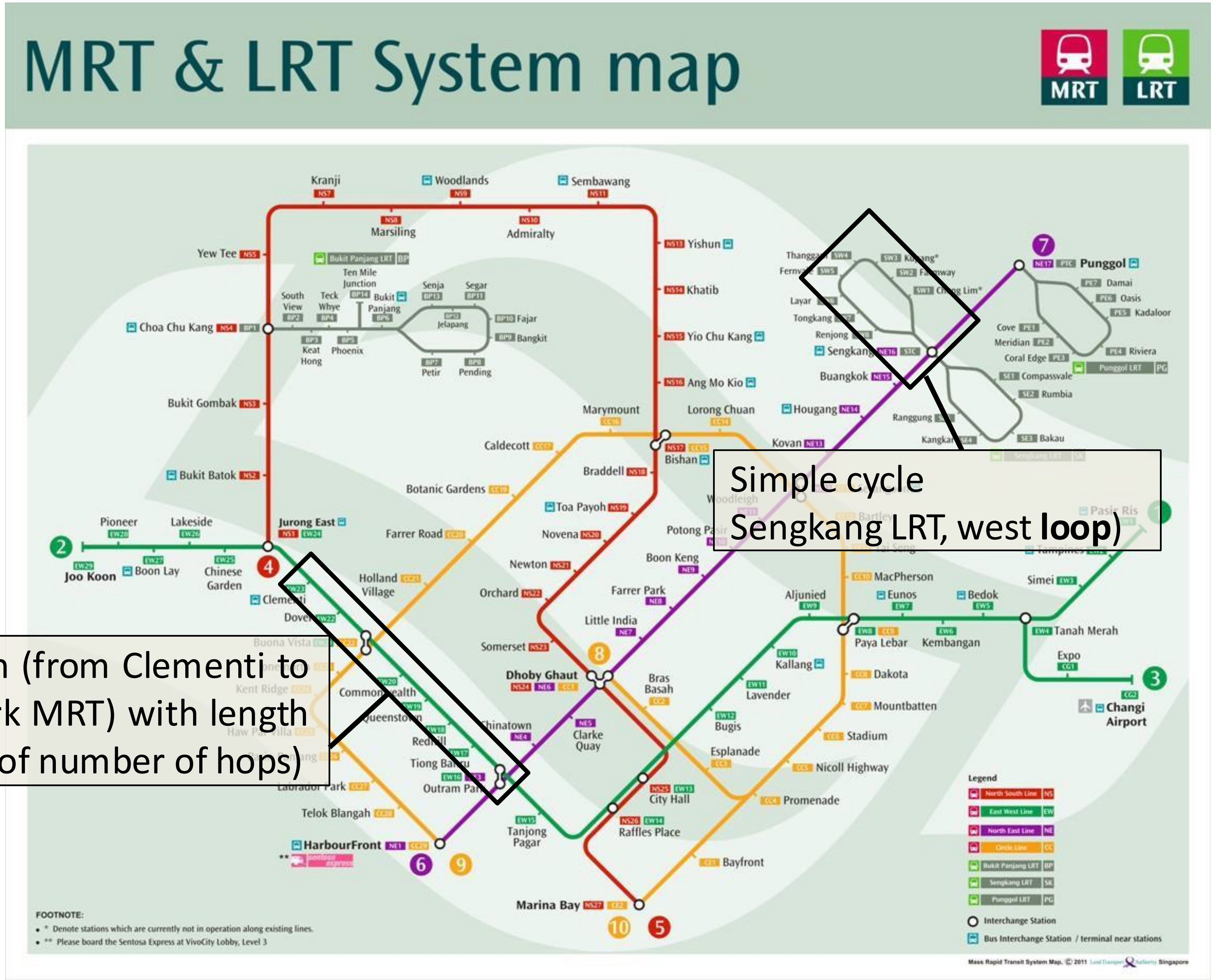
- (Simple) Directed Path
  - Same as (Simple) Path with the added restriction that the edges in the path are directed and in the same direction

- Path Length/Cost
  - In unweighted graph, usually number of edges in the path
  - In weighted graph, usually sum of edge weight in the path

Graph Terminologies (4)

- (Simple) Cycle
  - Path that starts and ends with the same vertex and with no repeated vertices except start/end vertex and no repeated edges
  - <u>Involves 3 or more unique vertices</u>
- (Simple) Directed Cycle
  - Same as (Simple) Cycle with the added restriction that the edges in the cycle are directed and in the same direction
  - <u>Involves 2 or more unique vertices</u>

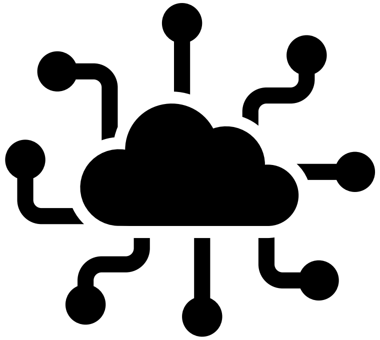
Transportation Network

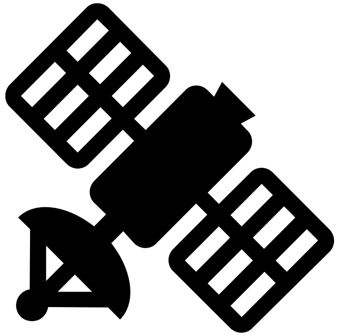


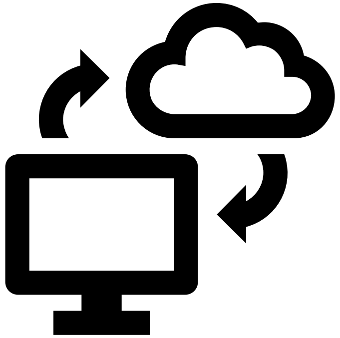
Simple cycle
Sengkang LRT, west **loop**)

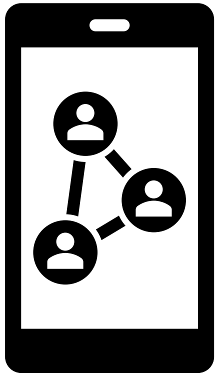
Simple path (from Clementi to Outram Park MRT) with length 7 (in terms of number of hops)

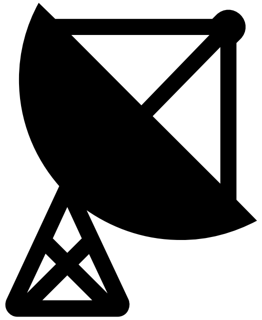
Internet/Computer/Communication Networks









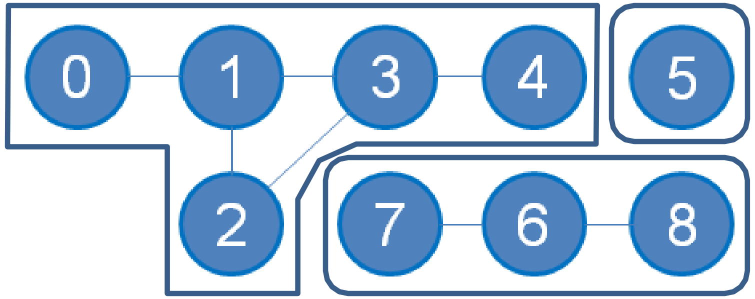


Graph Terminologies (5)

- 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

3 components in this graph

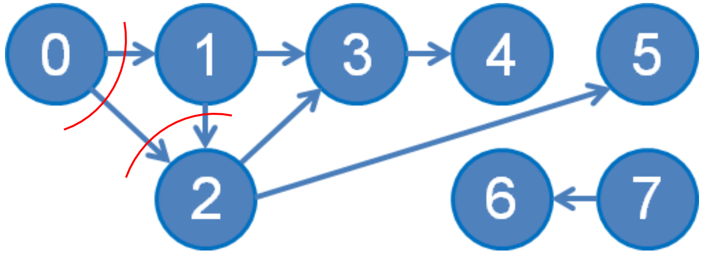
- Disconnected graph (since it has > 1 component)
- Vertices 1-2-3-4 are reachable from vertex 0
- Vertices 5, 6-7-8 are unreachable from vertex 0
- {7-6-8 5} is a sub graph of this graph



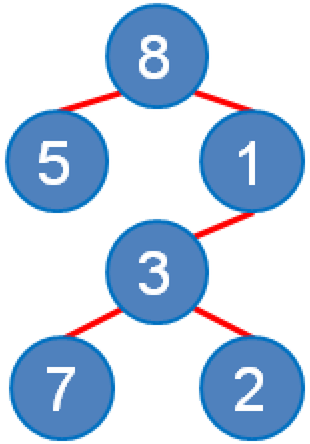
Graph Terminologies (6)

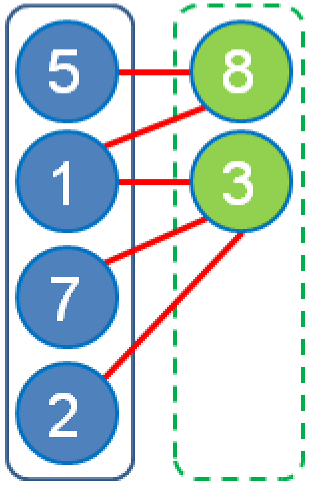
- Directed Acyclic Graph (DAG)
  - **Directed** graph that has no cycle
- Tree (bottom left)
  - Connected graph – one unique path between any pair of vertices
- Bipartite Graph (bottom right)
  - **Undirected** graph where we can partition the vertices into two sets so that there are no edges between members of the same set

Out degree of vertex 0 = 2



In degree of vertex 2 = 2





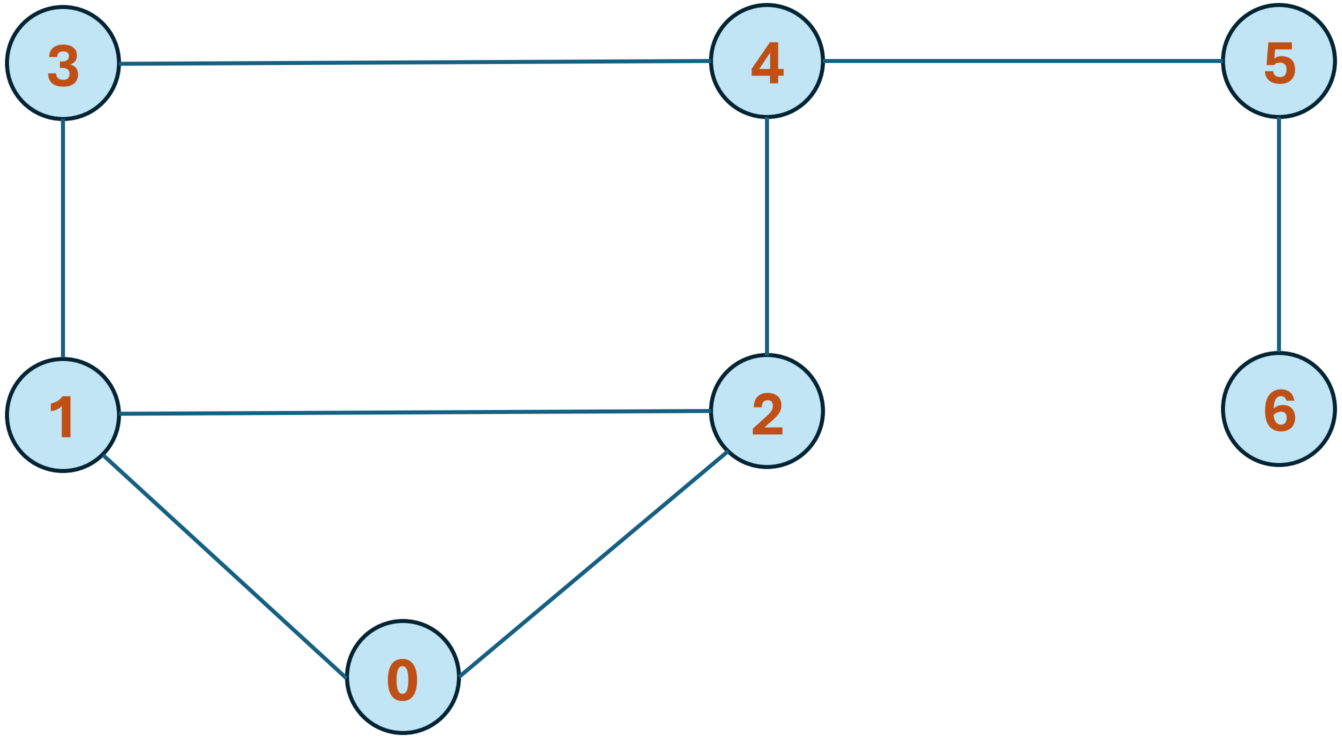
Graph Data Structures

https://visualgo.net/en/graphds

Adjacency Matrix

- A 2D array (AdjMatrix)
- AdjMatrix[i][j] = 1, if there exist an edge i ➔ j in G, otherwise 0
- For weighted graph, AdjMatrix[i][j] contains the weight of edge i ➔ j, not just binary values {1, 0}
- Space Complexity (V = number of vertices in G)
  - **$O(V^2)$**

Example of Adjacency Matrix

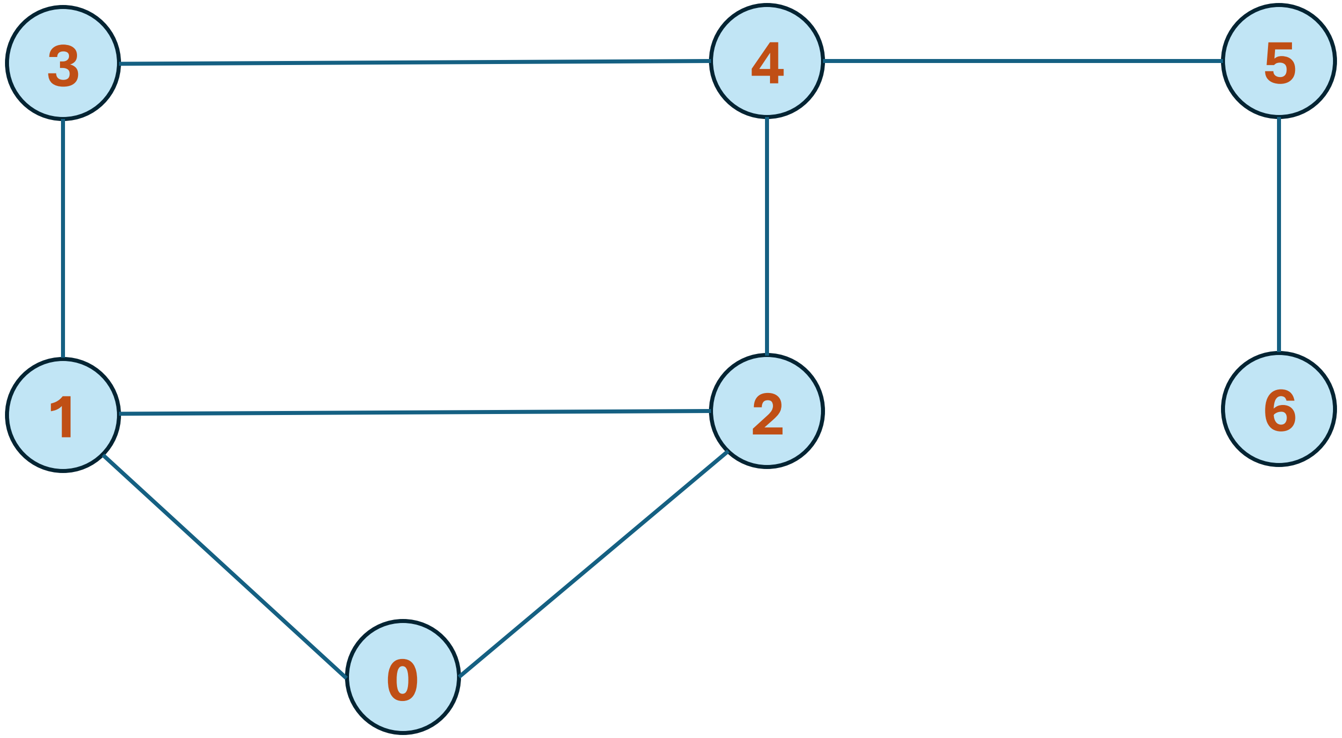


Adjacency Matrix							
	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

- An array of V lists (AdjList)
  - One element for each vertex
- For each vertex i, AdjList[i] = list of i's neighbours
- For weighted graph, stores pair (neighbour, weight)
  - Can use same strategy for unweighted graph
  - Connected to neighbour? Set weight to 1 (unit weight), 0 otherwise
- Space Complexity (E = number of edges in G)
  - **$O(V + E)$** ($E = O(V^2)$)

Example of Adjacency List

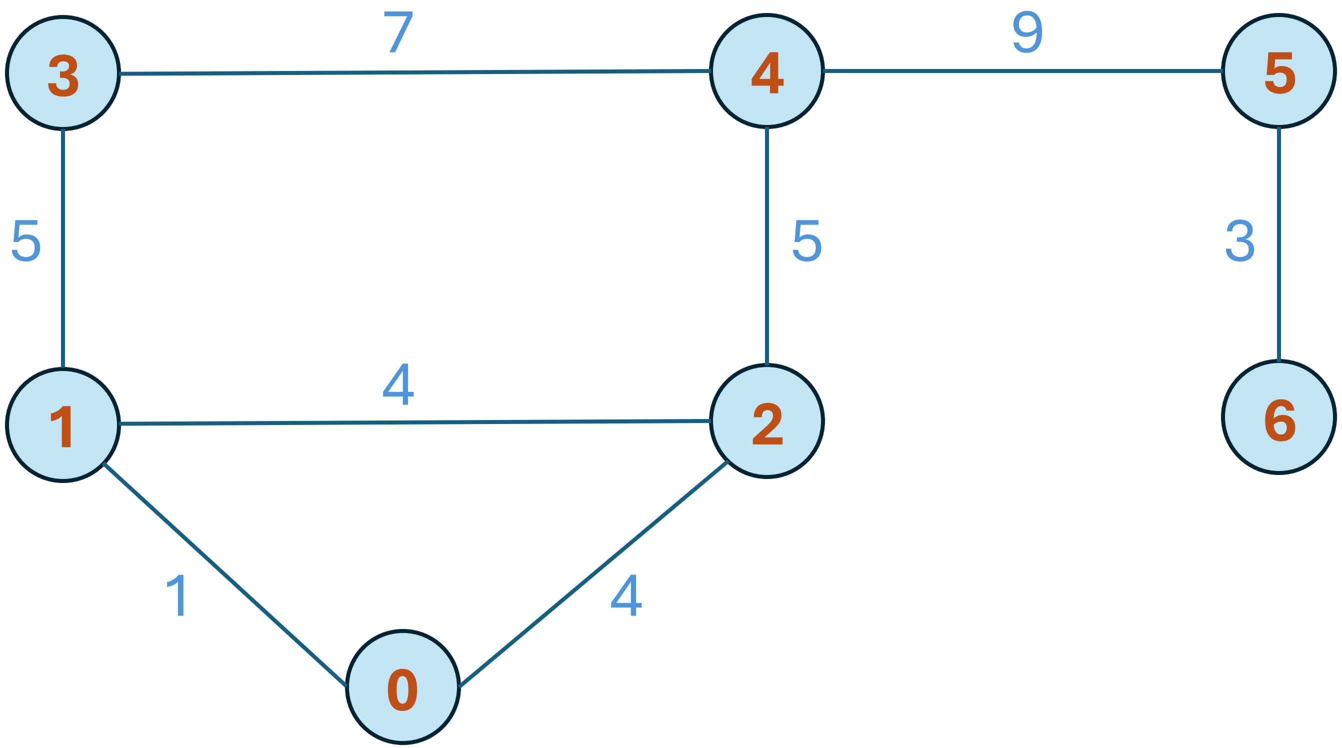


Adjacency List			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List

- An array of E edges (EdgeList)
  - One element for each edge
- For each edge i, EdgeList[i] = 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)**

Example of Edge List



Edge List			
0:	0	1	1
1:	0	2	4
2:	1	2	4
3:	1	3	5
4:	2	4	5
5:	3	4	7
6:	4	5	9
7:	5	6	3

Java Implementation (1)

Adjacency Matrix: Simple built-in 2D array

```
int 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 ☺

Graph Data Structures

What can we do with them?

Some basic calculations for now, but more later ☺

Counting Vertices

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
1:	0	2	4
2:	1	2	4
3:	1	3	5
4:	2	4	5
5:	3	4	7
6:	4	5	9
7:	5	6	3

Counting Vertices

- 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?

Enumerating Neighbours

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

O(V)

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(k)

Edge List

0:	0	1	1
1:	0	2	4
2:	1	2	4
3:	1	3	5
4:	2	4	5
5:	3	4	7
6:	4	5	9
7:	5	6	3

Enumerating Neighbours

- O(V) for AdjMatrix: scan AdjMatrix[v][j], $\forall j \in [0..V\text{-}1]$
- O(k) for AdjList: scan AdjList[v]
  - k is the number of neighbours of vertex v (output-sensitive algorithm)
- This is an important difference between AdjMatrix versus AdjList
  - It affects the performance of many graph algorithms. Remember this!
- *Usually,* the neighbours are listed in increasing vertex number
- Again, what about EdgeList?

Counting Edges

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

$O(V^2)$

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		

$O(V)$

Edge List			
0:	0	1	1
1:	0	2	4
2:	1	2	4
3:	1	3	5
4:	2	4	5
5:	3	4	7
6:	4	5	9
7:	5	6	3

$O(1)$

Counting Edges

- O(1) for EdgeList
  - Undirected/Bidirected edges may be listed once (or twice) in EdgeList, depending on the need

- $O(V^2)$ for AdjMatrix: count non-zero entries in AdjMatrix

- O(V) 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 this every time, i.e. O(1), *especially if the graph never changes after it is created*

Existence of Edges

- Given vertices u and v, does edge(u,v) exist?
- O(1) for AdjMatrix: see if AdjMatrix[u][v] is non zero
- O(k) for AdjList: see if AdjList[u] contains v
- How about EdgeList?

Trade-Off

Adjacency Matrix

- Pros
  - Existence of edge i – j can be found in O(1)
  - Good for dense graph/ Floyd Warshall's
- Cons
  - O(V) to enumerate neighbours of a vertex
  - $O(V^2)$ space

Adjacency List

- Pros
  - O(k) to enumerate k neighbours 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)

Live Quiz ☺

- On Zoom
- And a question asking your opinion

Feedback

https://forms.office.com/r/jcLS2bxjth