Algorithms and Data Structures

COMP261
Graph 1: data structures

Alex Potanin, Yi Mei

yi.mei@ecs.vuw.ac.nz

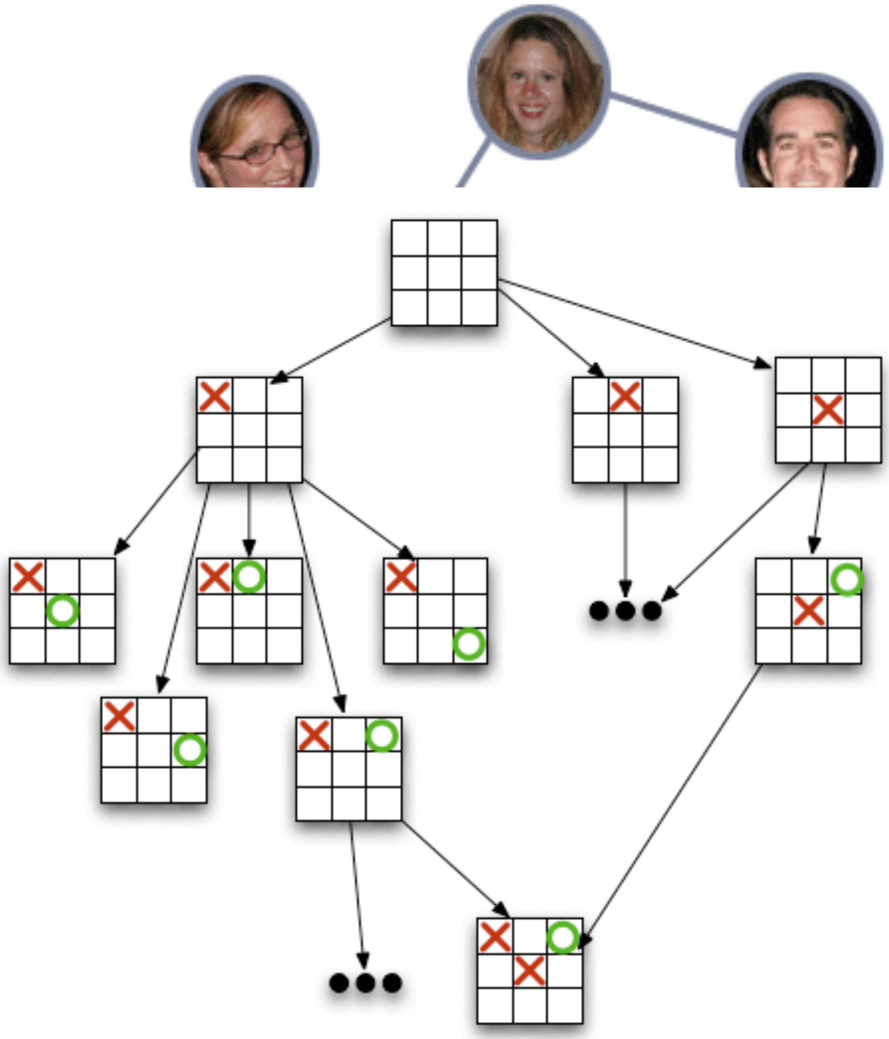
Outline

- Graph
- Adjacency matrix
- Complexity of adjacency matrix
- Adjacency list
- Complexity of adjacency list
- Improved adjacency matrix and list

Graph

- Many real-world applications
  - places with connections
    airports & flights,
    intersections & roads,
    network switches and cables
    ….
  - entities with relationships
    social networks,
    biological models
    web pages ….
  - states and actions
    games, plans, …..

- The Auckland road network in Assignment 1



Graph

- A collection of **nodes**
- A collection of **edges**
  - We only consider directed edges
  - Undirected edge can be seen as a pair of directed edges
  - (A, B) can be seen as (A -> B) and (B -> A)
- Relationship between nodes and edges
  - Nodes form edges
  - Edges connect nodes

- What **data structure** should be used to represent a graph?

Graph Data Structure

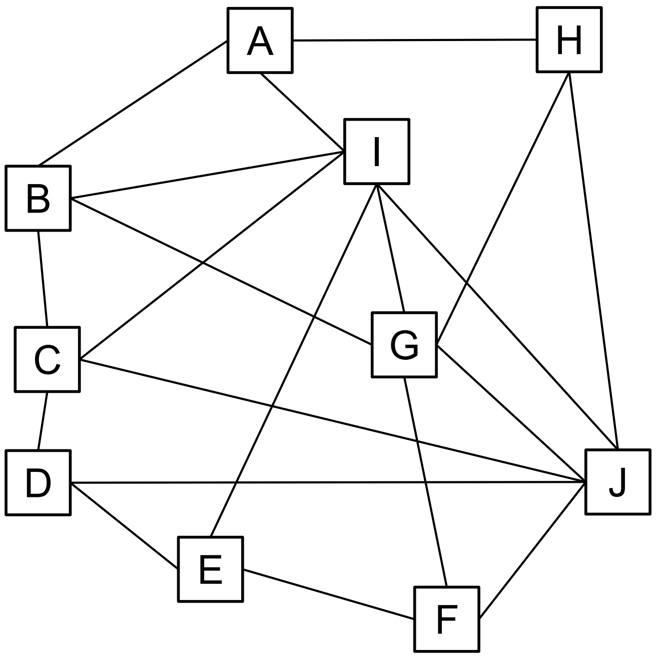
- A proper data structure should support common operators efficiently
- Consider the complexity of common operators, e.g.
  - Find all the nodes of the graph
  - Find all the edges of the graph
  - Find all outgoing edges of a node
  - Find all incoming edges of a node
  - Find all the outgoing node neighbours of a node
  - Find all the incoming node neighbours of a node
  - Find out whether two nodes are directly connected or not
  - Find the edge between two nodes
  - …

Graph Data Structure

- Two traditional data structures
  - Adjacency Matrix
  - Adjacency List

Adjacency Matrix

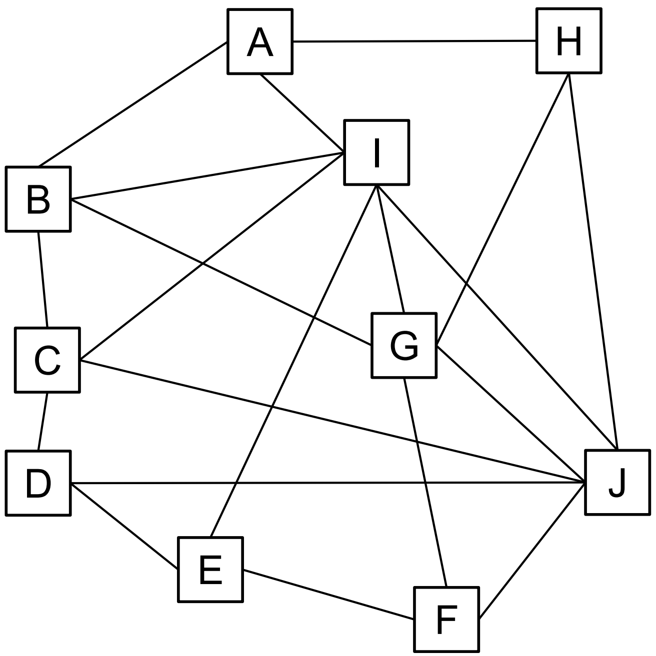
- Use a 2D matrix to represent the graph
  - Number of rows and columns = number of nodes
  - $M_{ij} = 1$ if there is an edge from node *i* to node *j*
  - $M_{ij} = 0$ (blank) otherwise
- Cannot handle weighted graph (e.g. edges have lengths)



	A	B	C	D	E	F	G	H	I	J
A		1						1	1	
B	1		1				1		1	
C		1		1					1	1
D			1		1					1
E				1		1			1	
F					1		1			1
G		1				1		1	1	1
H	1						1			1
I	1	1	1		1		1			1
J			1	1		1	1	1	1	

Adjacency Matrix

- Use a 2D matrix to represent the graph
  - Number of rows and columns = number of nodes
  - $M_{ij} = \boxed{w_{ij}}$ is the weight (e.g. length) of the directed edge from *i* to *j*
  - $M_{ij} = \infty$, or leave blank if there is no edge from *i* to *j*.
- Cannot deal with multi-graph.



	A	B	C	D	E	F	G	H	I	J
A		5						5	2	
B	5		3				7		6	
C		3		1					7	9
D			1		3					9
E				3		4			9	
F					4		5			4
G		7				5		6	3	4
H	5						6			7
I	2	6	7		9		3			6
J			9	9		4	4	7	6	

Adjacency Matrix

- Use a 2D matrix to represent the graph
  - Number of rows and columns = number of nodes
  - M_{ij} is a list of edge objects
- An edge object is unique for each edge

```
Class Edge {
    Node fromNode;
    Node toNode;
}

Edge e1 = new Edge(A, B);
Edge e2 = new Edge(A, B);

System.out.println(e1.equals(e2));
```

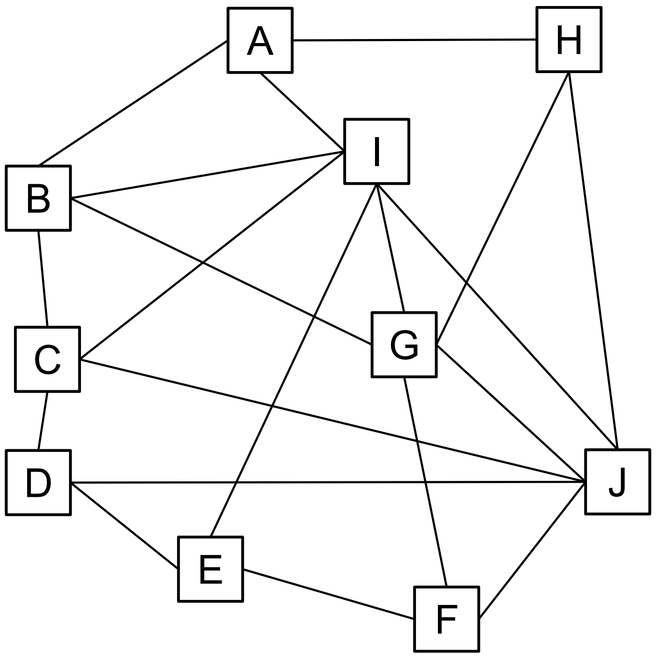
Time Complexity of Adjacency Matrix

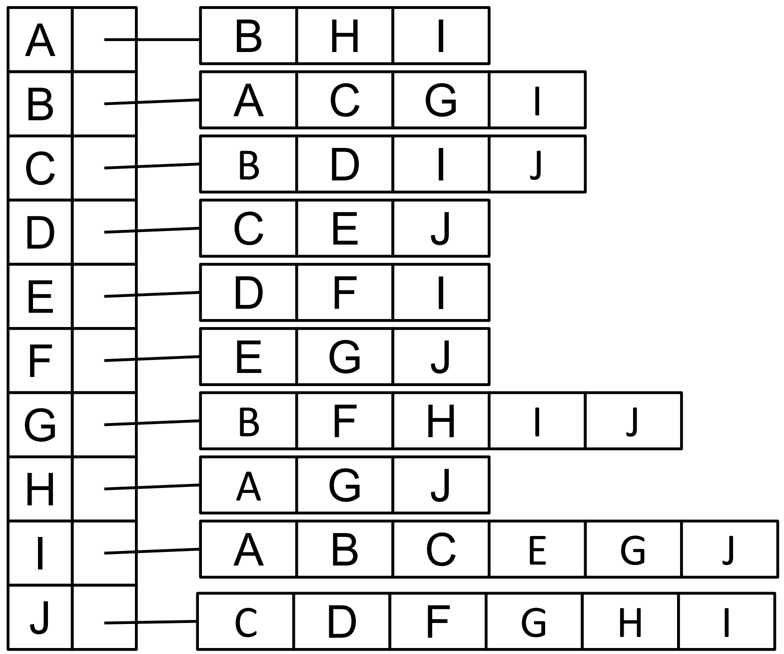
- Assume simple graph: at most one edge between each pair of nodes, with N nodes and M directed edges
- 2D array M[][], with each entry as an edge object
  - Fine all nodes
    - Enumerate all nodes: $O(N)$
  - Find all edges
    - Enumerate all node pairs: $O(N^2)$
  - Find all outgoing edges of a node
    - Enumerate the node row: $O(N)$
  - Find all incoming edges of a node
    - Enumerate the node column: $O(N)$
  - Find all outgoing node neighbours
    - Enumerate the node row: $O(N)$
  - Find all incoming node neighbours
    - Enumerate the node column: $O(N)$
  - Check if there is an edge between two nodes
    - Check the entry: $O(1)$

	A	B	C	D	E	F	G	H	I	J
A		5						5	2	
B	5		3				7		6	
C		3		1					7	9
D			1		3					9
E				3		4			9	
F					4		5			4
G		7				5		6	3	4
H	5						6			7
I	2	6	7		9		3			6
J			9	9		4	4	7	6	

Adjacency List

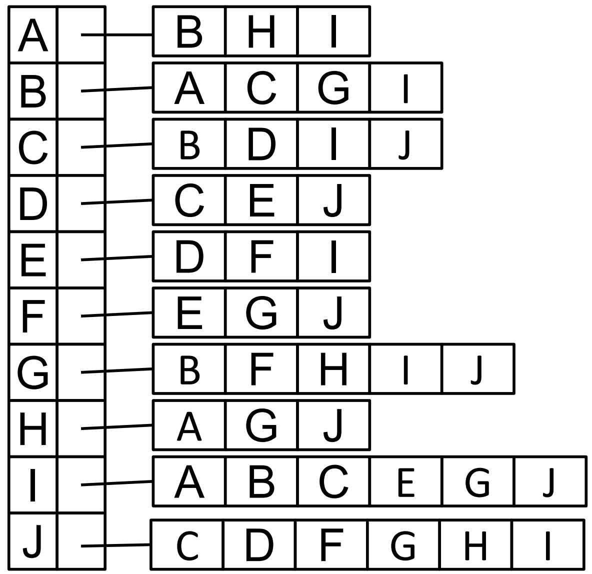
- For each node, store a list of outgoing node neighbours
- Do not need to enumerate all the nodes to find the neighbours
- Need to store a list of edge objects to store edge information, e.g. edge length





Time Complexity of Adjacency List

- Assume simple graph: at most one edge between each pair of nodes, with N nodes and M directed edges, assume $N < M$
- node.adjList() is a list of outgoing node neighbours of node *i*
  - Fine all nodes
    - Enumerate all nodes: $O(N)$
  - Find all edges
    - Enumerate all edges: $O(M)$
  - Find all outgoing edges of a node
    - Enumerate all edges to match the two nodes: $O(M)$
  - Find all incoming edges of a node
    - Enumerate all edges to match the two nodes: $O(M)$
  - Find all outgoing node neighbours
    - Enumerate all the nodes in the adjacency list: $O(N)$
  - Find all incoming node neighbours
    - Enumerate all edges to match the from node: $O(M)$
  - Check if there is an edge between two nodes
    - Enumerate all the nodes in the adjacency list: $O(N)$



Time Complexity of Adjacency List

- Not efficient in finding outgoing edges of a node
  - Need to enumerate the edge list with the two nodes to find the matching edge object
  - **Solution**: store edge objects instead of nodes in the adjacency list
  - Finding all outgoing edges of a node can be done by enumerating the adjacency list
  - Find outgoing node neighbours:
    - `node.adjList().get(i).getToNode()`

- Not efficient in finding incoming edge and node neighbours
  - Need to enumerate the edge list with the two nodes to find the matching edge object
  - **Solution**: store two adjacency lists, `outAdjList` for outgoing, `inAdjList` for incoming
  - Find incoming node neighbours:
    - `node.inAdjList().get(i).getFromNode()`

Time Complexity of Adjacency List

- Worse-case complexity of finding edge/node neighbours is $O(N)$, if the graph is fully connected.
- In practice, this complexity is much smaller
- Node degree "$deg(node)$": the number of outgoing (incoming) edges of a node
- Max degree of a graph ($\Delta = \max\{\deg(node)\}$): the maximal number of neighbours of the nodes in the graph
  - E.g.: an intersection connects at most four streets, $\Delta = 4$
- Complexity of finding all outgoing/incoming neighbours
  - $\boldsymbol{O(\Delta) \ll O(N)}$
  - Almost $O(1)$

Time Complexity Comparison

- Assume simple graph: at most one edge between each pair of nodes, with N nodes and M directed edges
- Max Degree of the graph: $\Delta_{in} = \Delta_{out} = \Delta$
  - **Adjacency matrix**: each entry stores an edge object
  - **Adjacency list**: each node has two lists, one for outgoing edge objects, and the other for incoming edge objects

	Adjacency Matrix	**Adjacency List**
Find all nodes	$O(N)$	$O(N)$
Find all edges	$O(N^2)$	$O(M)$
Find all outgoing edges of a node	$O(N)$	$O(\Delta)$
Find all incoming edges of a node	$O(N)$	$O(\Delta)$
Find all outgoing node neighbours of a node	$O(N)$	$O(\Delta)$
Find all incoming node neighbours of a node	$O(N)$	$O(\Delta)$
Check if there is an edge from u to v	$O(1)$	$O(\Delta)$

- Adjacency list has better time complexity overall

Summary

- Adjacency matrix and adjacency list are two common data structures for graph
- Different time complexities in different scenarios
- Improvements on the data structure
  - Adjacency matrix: store edge objects rather than labels
  - Adjacency list:
    - Store edge objects rather than nodes
    - Store two lists, one for outgoing and the other for incoming