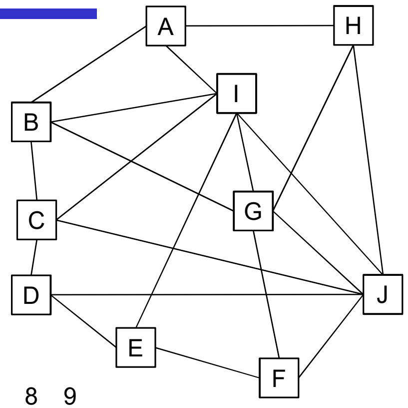
Adjacency Matrix



- Use integers 0..n-1 to represent nodes
- Use an array to represent info about nodes

```
private Node[] nodes;
```

0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J

- Use a 2D matrix to represent the graph

```
private Edge[][] edges;
```

  - 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

	0	1	2	3	4	5	6	7	8	9
0		5						5	2	
1	5		3				7		6	
2		3		1					7	9
3			1		3					9
4				3		4			9	
5					4		5			4
6		7				5		6	3	4
7	5						6			7
8	2	6	7		9		3			6
9			9	9		4	4	7	6	

Edge object

- What about edges with labels (lengths/weights/capacities/etc)?
- Cannot deal with multi-graphs.

Adjacency List

- Use integers 0..n-1 to represent nodes,
  and array to represent info about nodes:

  ```
  private Node[] nodes;
  ```

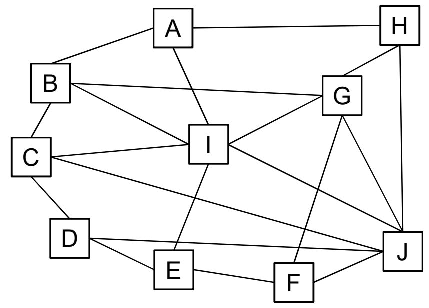
- Use an array of arrays/lists to represent the graph

  ```
  private int[][] neighbours;    or
  private List<Integer>[] neighbours;
  ```

- What about edge information?

  Lists could store edge objects containing
  - nodes at each end
  - length/capacity/labels on edges

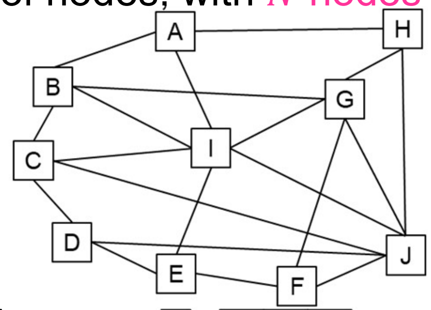
  ```
  private List<Edge>[] edges;
  ```



0	A	0	1 7 8
1	B	1	0 2 6 8
2	C	2	1 3 8 9
3	D	3	2 4 9
4	E	4	3 5 8
5	F	5	4 6 9
6	G	6	1 5 7 8 9
7	H	7	0 6 9
8	I	8	0 1 2 4 6 9
9	J	9	2 3 5 6 7 8

Time Complexity of Adjacency List,

- Assume simple graph: at most one edge between each pair of nodes, with N nodes and E directed edges, assume $N < E < 2N^2$
- Row i: a list of outgoing **node neighbours** of node *i*
  - Find all nodes
  - Find all edges
  - Find all edges of a node
  - Find all node neighbours
  - Check if there is an edge between two nodes



0	A	0	1, 7, 8
1	B	1	0, 2, 6, 8
2	C	2	1, 3, 8, 9
3	D	3	2, 4, 9
4	E	4	3, 5, 8
5	F	5	4, 6, 9
6	G	6	1, 5, 7, 8, 9
7	H	7	0, 6, 9
8	I	8	0, 1, 2, 4, 6, 9
9	J	9	2, 3, 5, 6, 7, 8

Adjacency List, Directed Graph

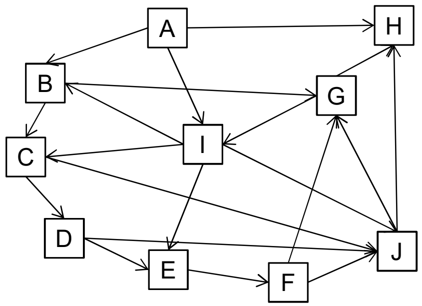
Same data structure

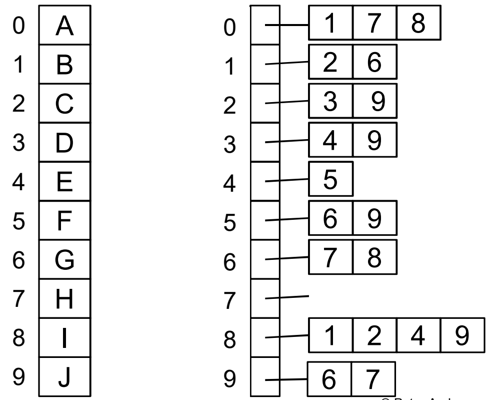
- Use integers 0..n-1 to represent nodes, and array to represent info about nodes:

```
private Node[] nodes;
```

- Use an array of arrays/lists to represent the graph

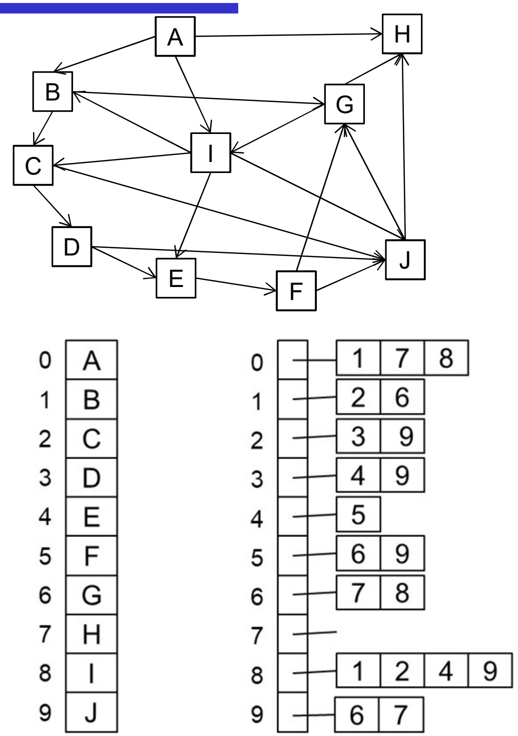
```
private int[][] outNeighbours;    or
private List<Integer>[] outNeighbours;
private List<Edge>[] outEdges;
```





Time Complexity of Adjacency List, Directed

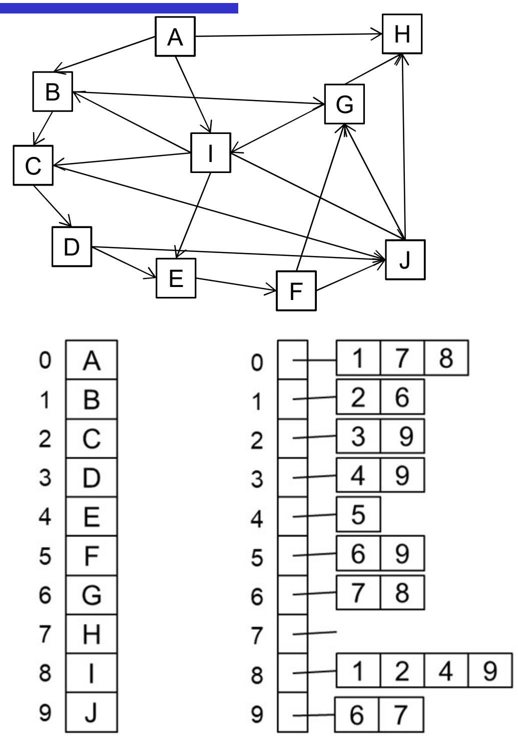
- Assume simple graph: at most one edge between each pair of nodes, with N nodes and E directed edges, assume $N < E < 2N^2$
- If graph has a maximum in-degree and/or out-degree: Δ_{in}, Δ_{out}, $\Delta = \max(\Delta_{in}, \Delta_{out})$
  - (maximum number of neighbours)
  - Find all nodes
  - Find all edges
  - Find all outgoing edges of a node
  - Find all incoming edges of a node
  - Find all outgoing node neighbours
  - Find all incoming node neighbours
  - Check if there is an edge between two nodes



i th list has the outgoing **neighbours** of node *i*

Time Complexity of Adjacency List, Directed

- Assume simple graph: at most one edge between each pair of nodes, with N nodes and E directed edges, assume $N < E < 2N^2$
- If graph has a maximum in-degree and/or out-degree: Δ_{in}, Δ_{out}, $\Delta = \max(\Delta_{in}, \Delta_{out})$
  - (maximum number of neighbours)
  - Find all nodes   O(N)
  - Find all edges   O(E)
  - Find all outgoing edges of a node   O(Δ)
  - Find all incoming edges of a node   O(E)
  - Find all outgoing node neighbours O(Δ)
  - Find all incoming node neighbours O(E)
  - Check if there is an edge between two nodes O(E)



i th list has the outgoing **neighbours** of node *i*

Adjacency List for Directed Graph

- Not efficient in finding incoming edges or neighbours of a node
  - Solution: store two adjacency lists

```
private List<Edge>[] outEdges;
private List<Edge>[] inEdges;
```

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 E directed edges, max degree of graph: $\Delta_{in} = \Delta_{out} = \Delta$
  - Adjacency matrix: each entry stores an edge object
  - Adjacency list: each node has list of edge objects or two lists, (outgoing and incoming) for directed graph

	Adjacency Matrix	Adjacency List	Edge List
Find all nodes	$O(N)$	$O(N)$	$O(\mathrm{E})$
Find all edges	$O(N^2)$	$O(\mathrm{E})$	$O(\mathrm{E})$
Find all outgoing edges of a node	$O(N)$	$O(\Delta)$	$O(\mathrm{E})$
Find all incoming edges of a node	$O(N)$	$O(\Delta)$	$O(\mathrm{E})$
Find all outgoing node neighbours of a node	$O(N)$	$O(\Delta)$	$O(\mathrm{E})$
Find all incoming node neighbours of a node	$O(N)$	$O(\Delta)$	$O(\mathrm{E})$
Check if there is an edge from u to v	$O(1)$	$O(\Delta)$	$O(\mathrm{E})$
Get next shortest edge	$O(N^2)$	$O(\mathrm{E})$	$O(\log(\mathrm{E}))$

- Adjacency list has better time unless checking edge from u to v is important.

Edge List:

- Array of Edges

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
to	0	0	0	1	1	2	2	2	3	3	4	5	5	5	6	7	7	8	9	9
from	3	4	7	2	0	6	4	5	9	3	1	7	0	5	4	8	1	2	0	5
length	25	31	19	82	43	74	86	21	10	33	17	66	47	65	53	68	46	22	3	92

- Slow for almost everything,
  except finding the next shortest edge:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
to	9	3	4	0	2	8	0	0	3	1	7	5	6	5	5	7	2	1	2	9
from	0	9	1	7	5	2	3	4	3	0	1	0	4	5	7	8	6	2	4	5
length	3	10	17	19	21	22	25	31	33	43	46	47	53	65	66	68	74	82	86	92

Object Oriented representation

- Forget the arrays.
- Don't use integers to represent nodes.
- Graph has a Collection of Nodes:

```
private Collection<Node> allNodes;
```

And maybe a Collection of Edges:

```
private Collection<Edge> allEdges;
```

Graph could contain a HashMap from Pairs of Nodes to Edges:

```
HashMap<Pair<Node,Node>,Edge> allEdges;
```

- Big linked structure of Objects
- Collections may be Lists or Sets
- Nodes contain collection of Edges

```
private Collection<Edge> edges;
```

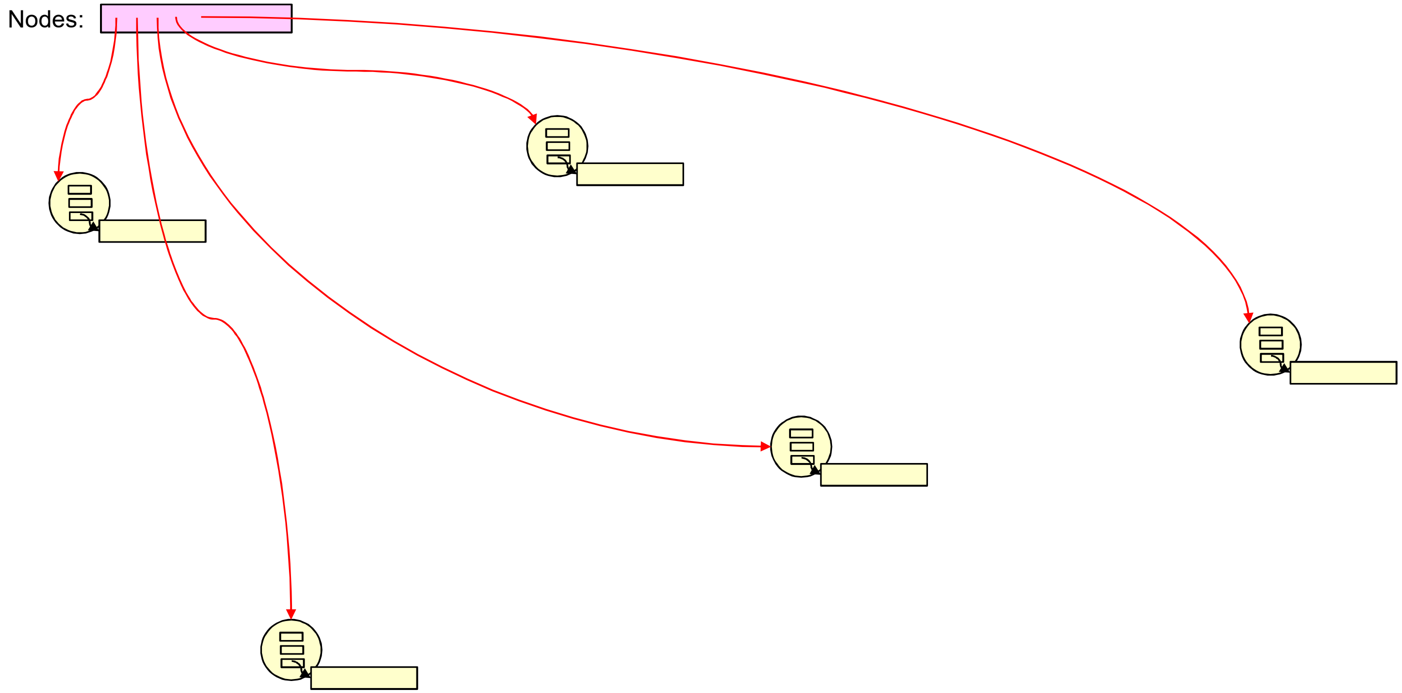
or two if directed graph:

```
private Collection<Edge> outgoing;
private Collection<Edge> incoming;
```

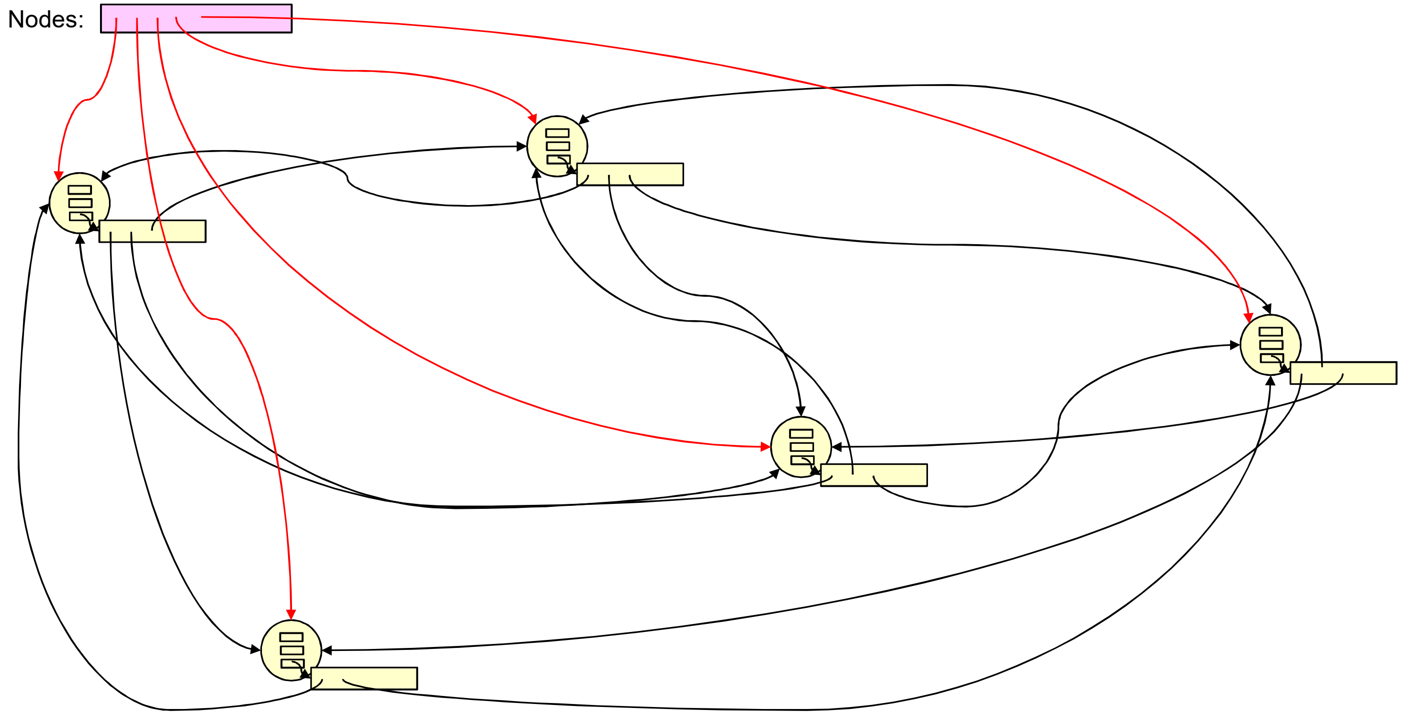
- Edges contain two Nodes

```
private Node from;
private Node to;
```

A Linked Graph Structure.



A Linked Graph Structure.

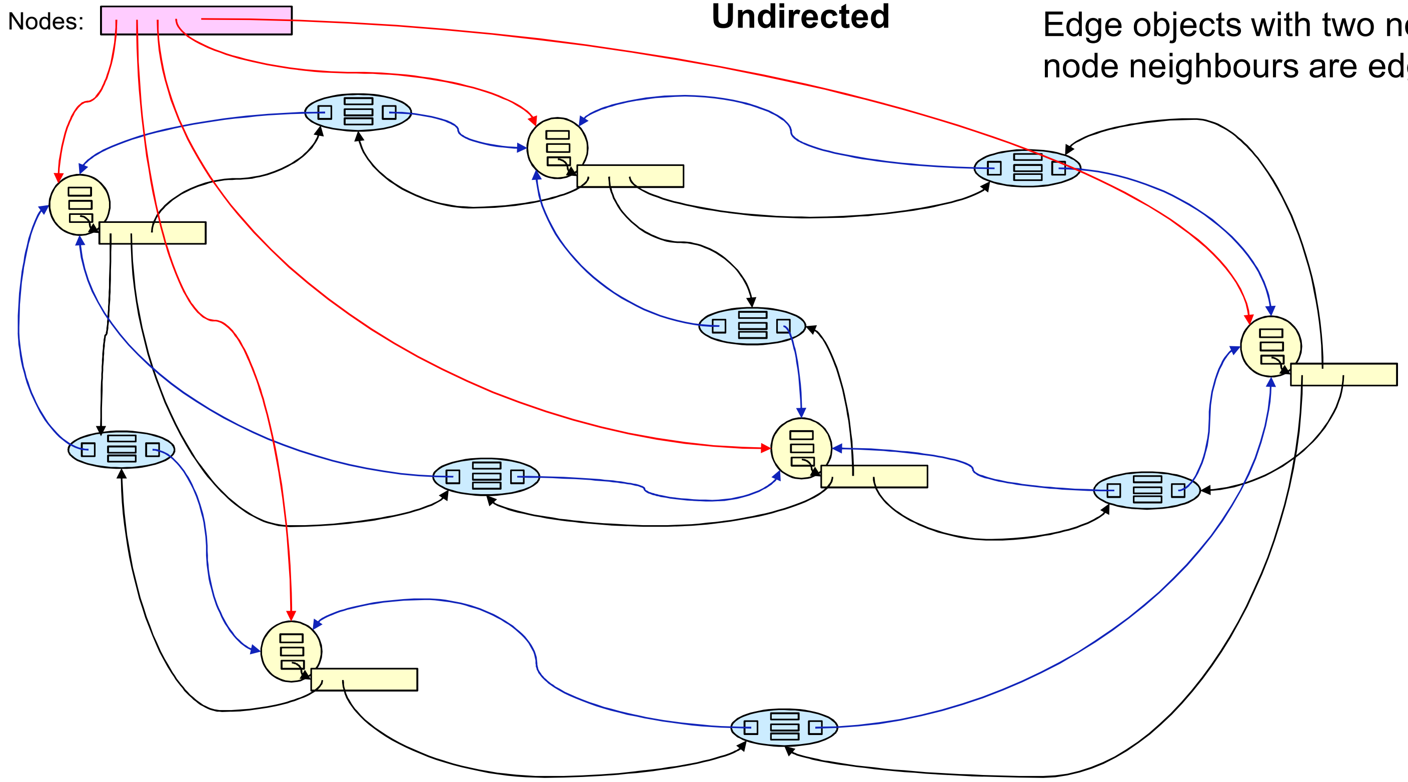


No information about the edges: neighbours are the nodes

A Linked Graph Structure.

Undirected

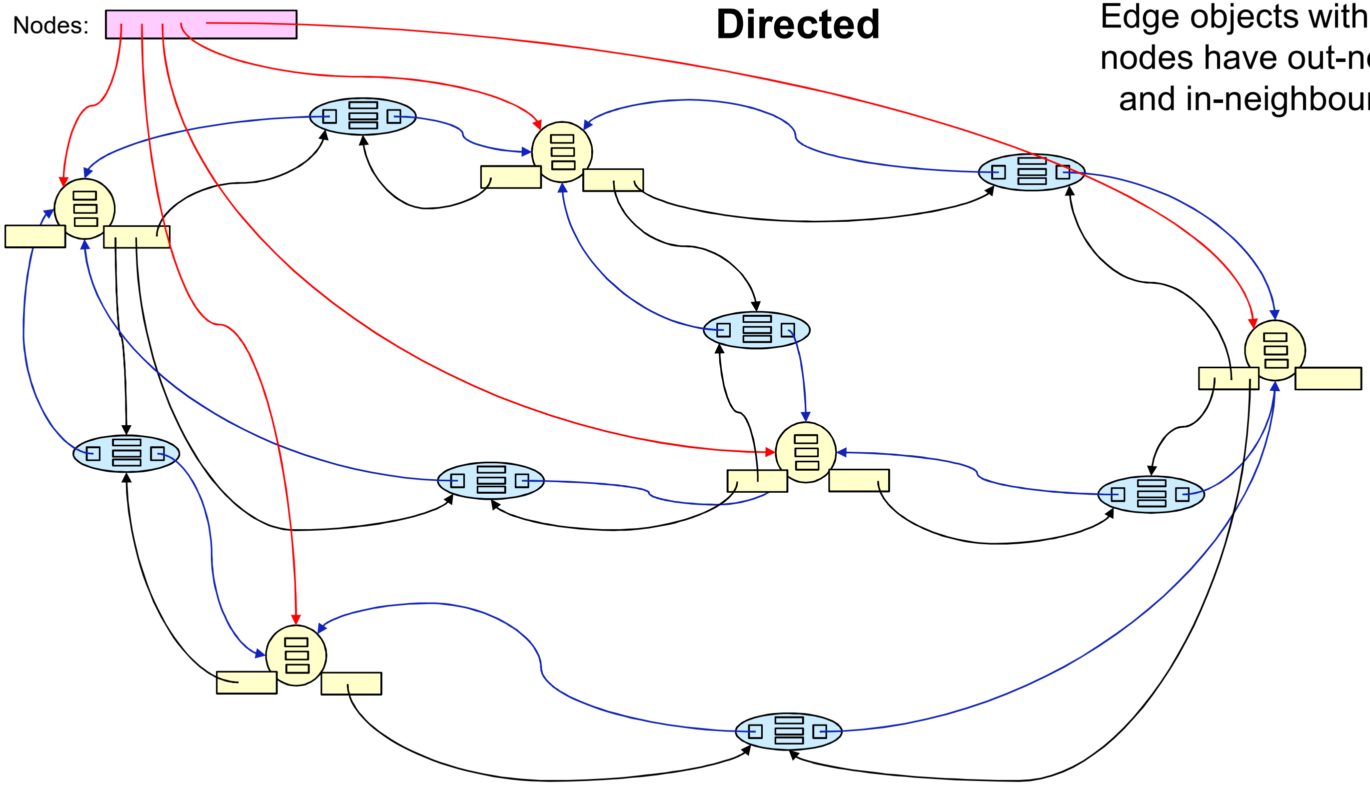
Edge objects with two nodes, node neighbours are edges



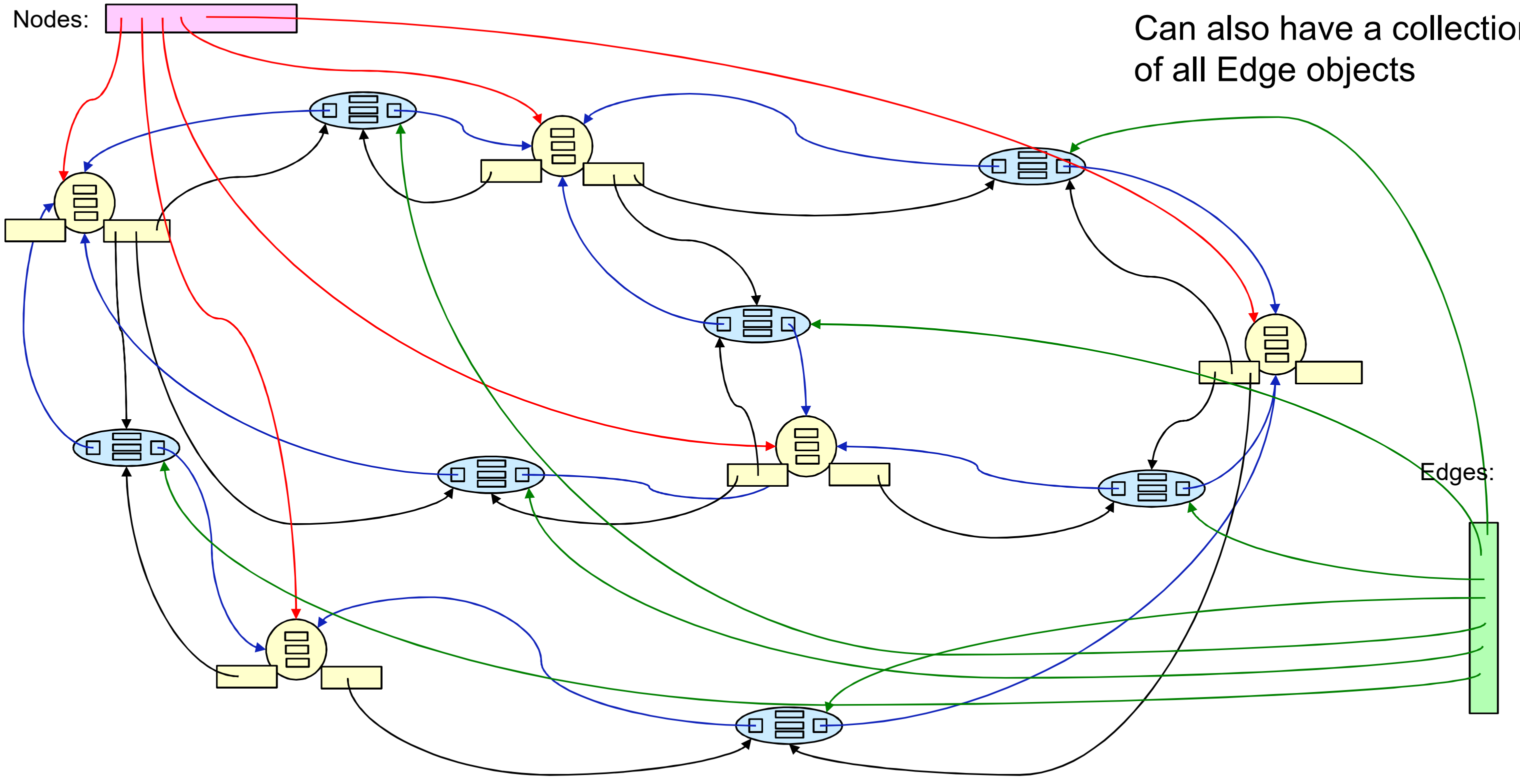
A Linked Graph Structure.

Directed

Edge objects with two nodes, nodes have out-neighbours and in-neighbours



A Linked Graph Structure.



Can also have a collection of all Edge objects

Wellington Public Transport Map

- Complex Graph structure
  - directed graph
  - multi-graph
  - lots of information on nodes and edges
  - multiple tasks.
  - Additional structure (“lines"), kinds of edges.

- Assignment:
  - build the graph structure edges and neighbours
  - Find shortest paths
  - Find strongly connected subgraphs
  - Find "articulation points"