Admin

- Assign 1 due tomorrow 12noon

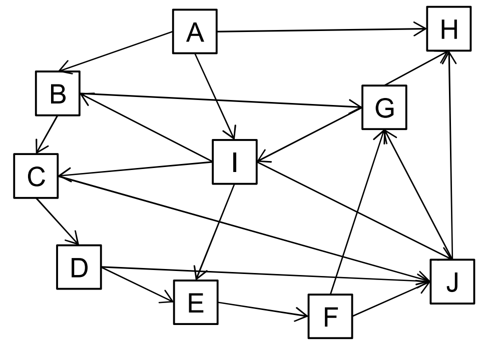
- All classes and interfaces in Parser.java
- Do not use packages
- Test it using all our testing classes: ParserTester1&2, ExecuteTester
- Report.txt for any partially completed stages

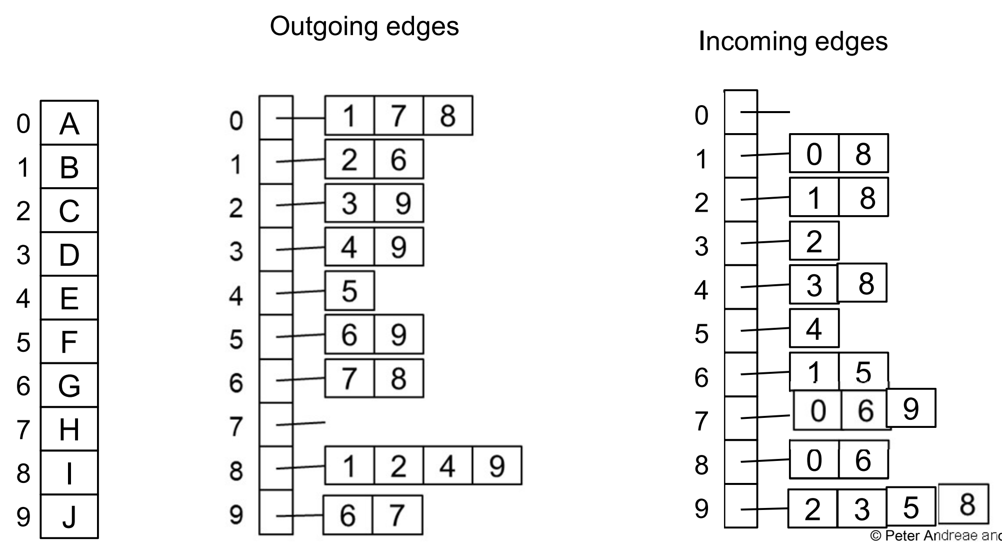
- I hate Plagiarism
  - Do not copy and paste code
  - Do not give your code to anybody

Adjacency List, Directed Graph

Two lists:

- out edges
- in edges





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;
```

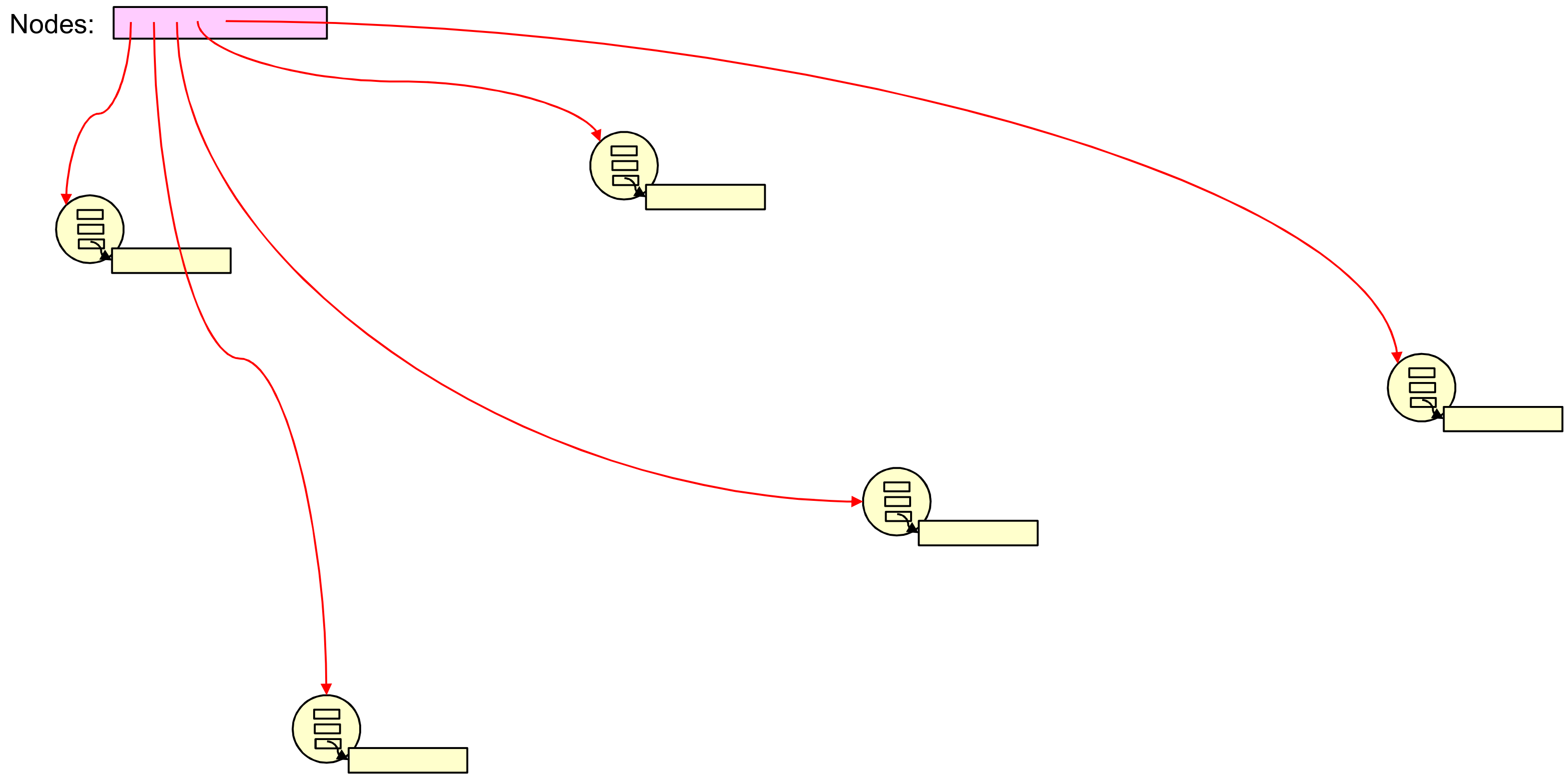
or two if directed graph:

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

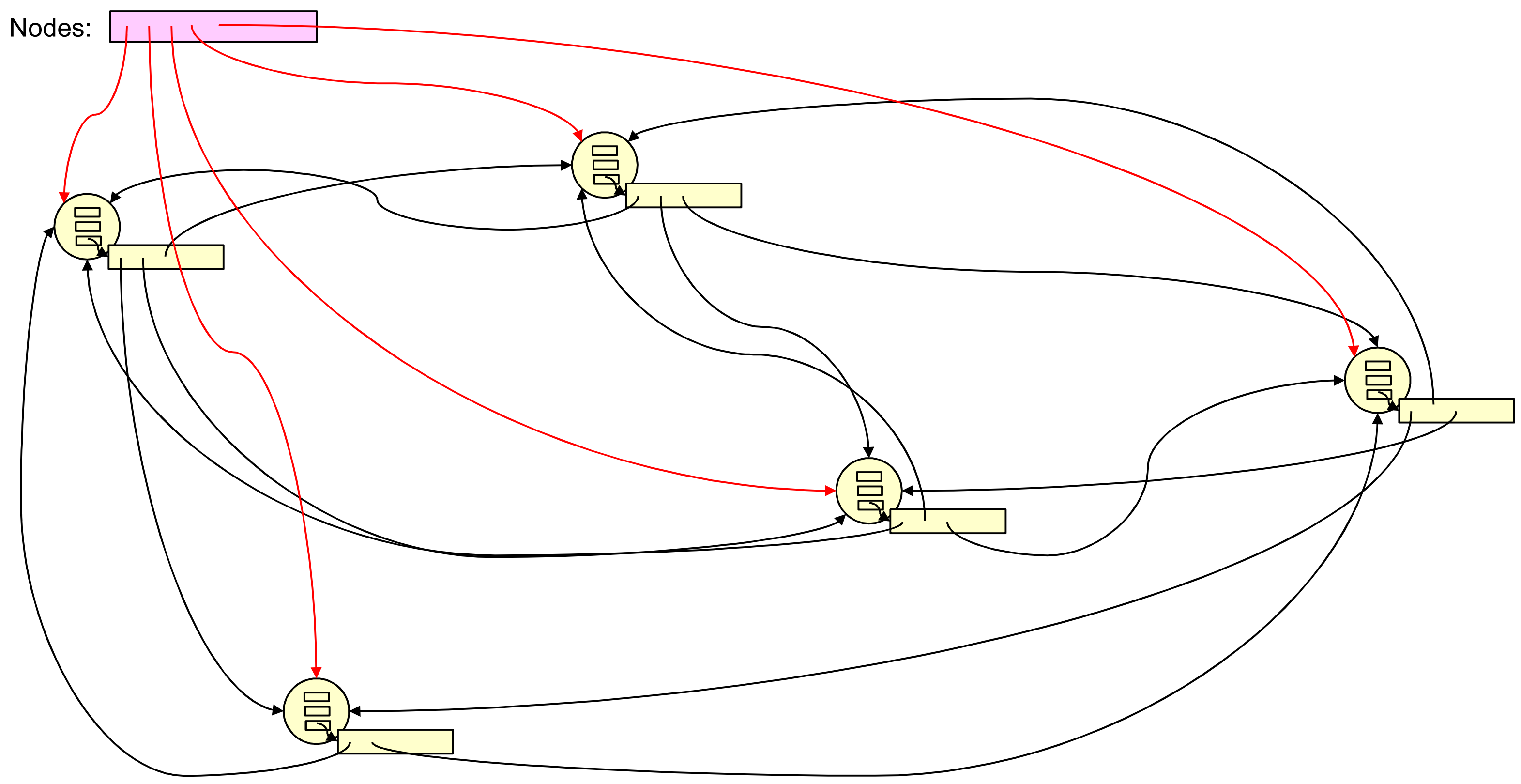
- Edges contain two Nodes

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

A Linked Graph Structure.



A Linked Graph Structure.

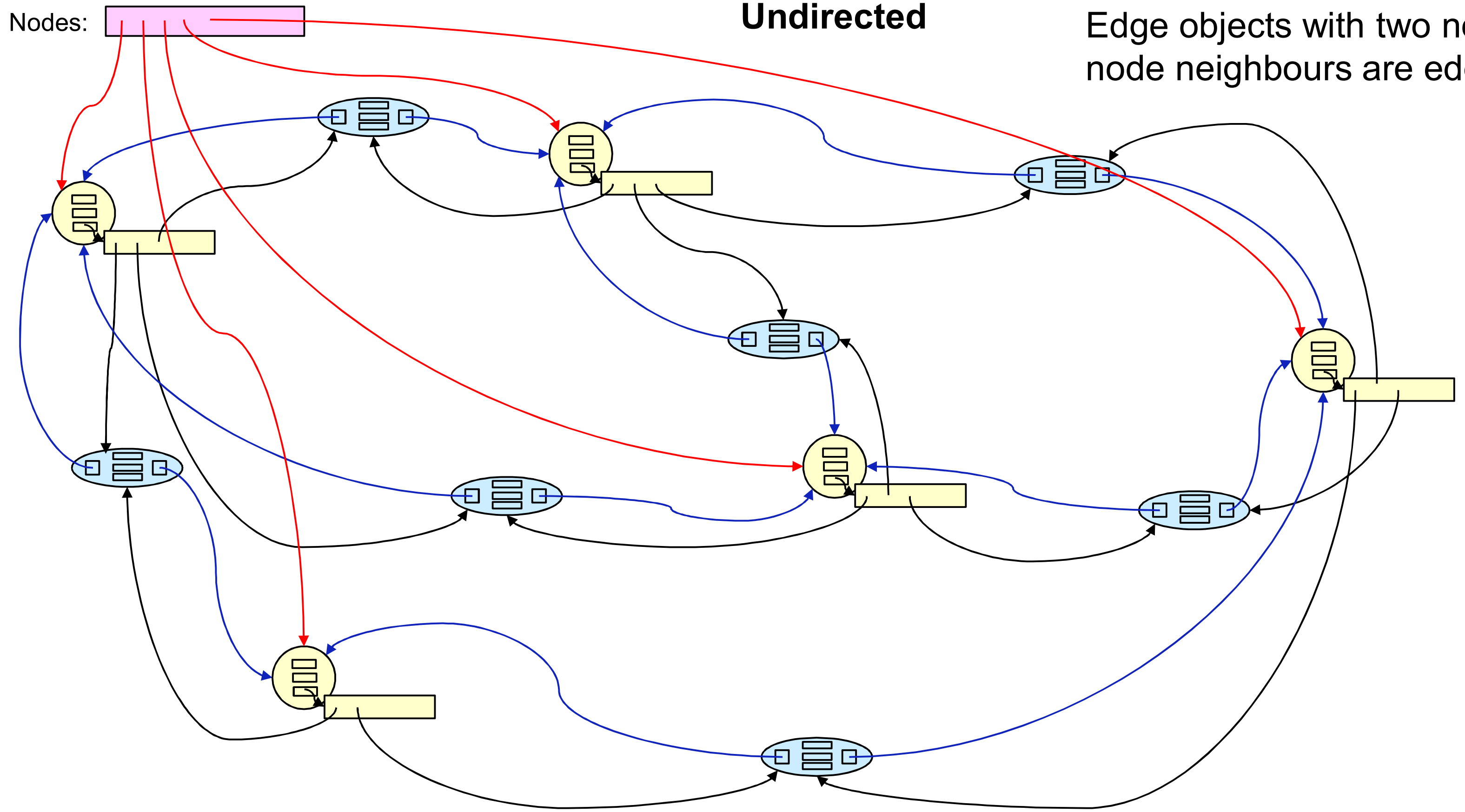


No information about the edges: neighbours are the nodes

A Linked Graph Structure.

Undirected

Edge objects with two nodes, node neighbours are edges

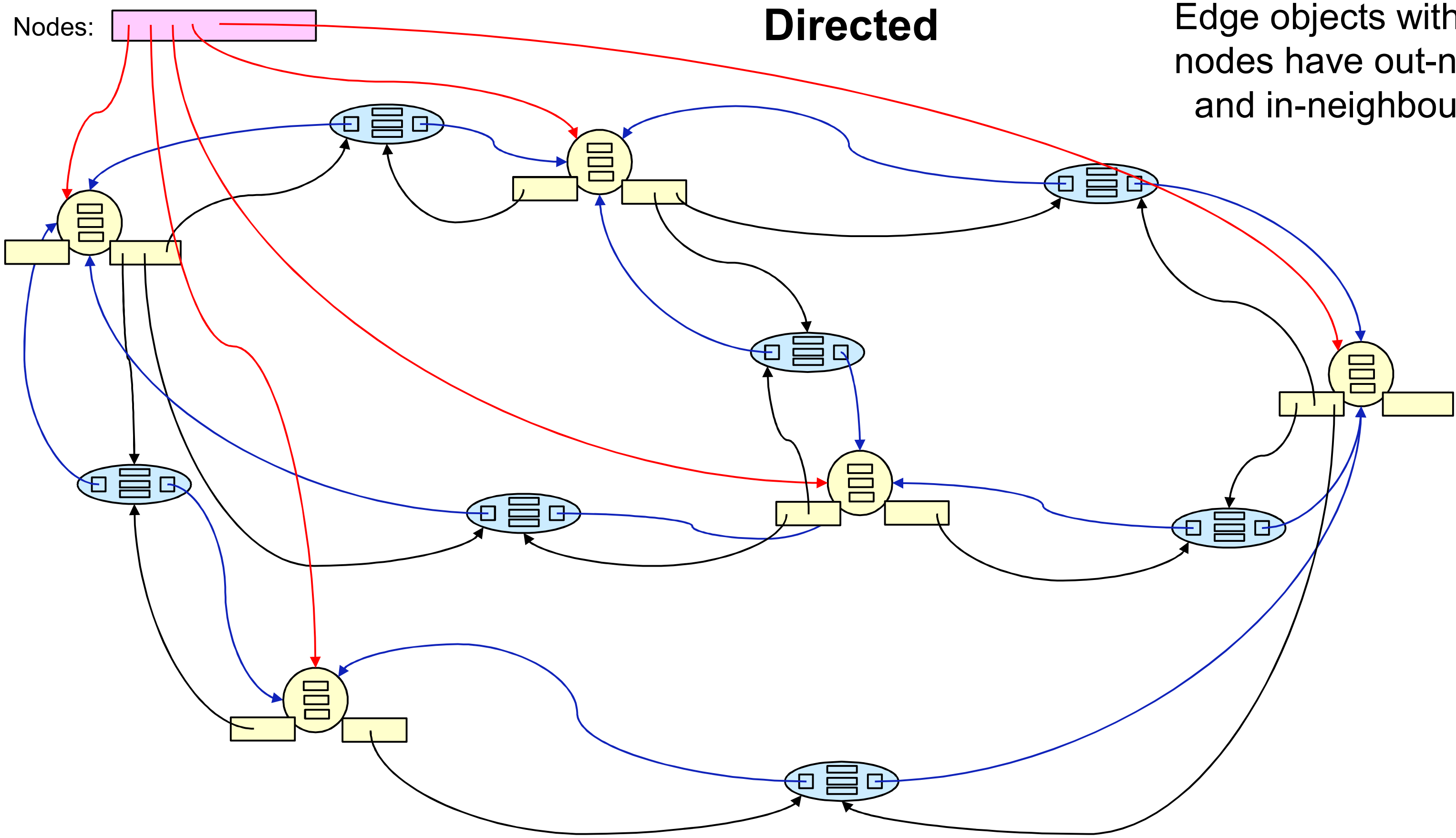


A Linked Graph Structure.

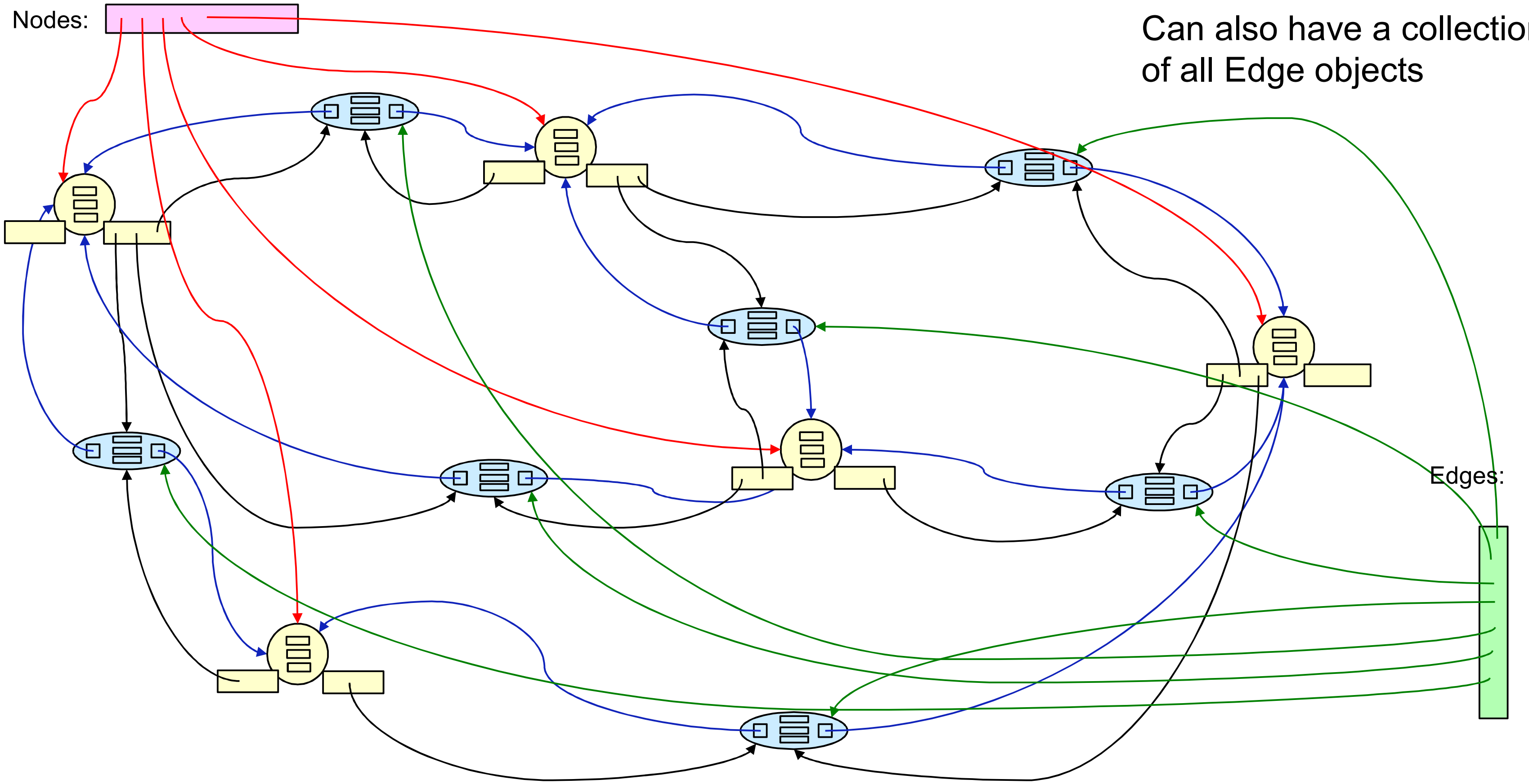
Nodes:

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

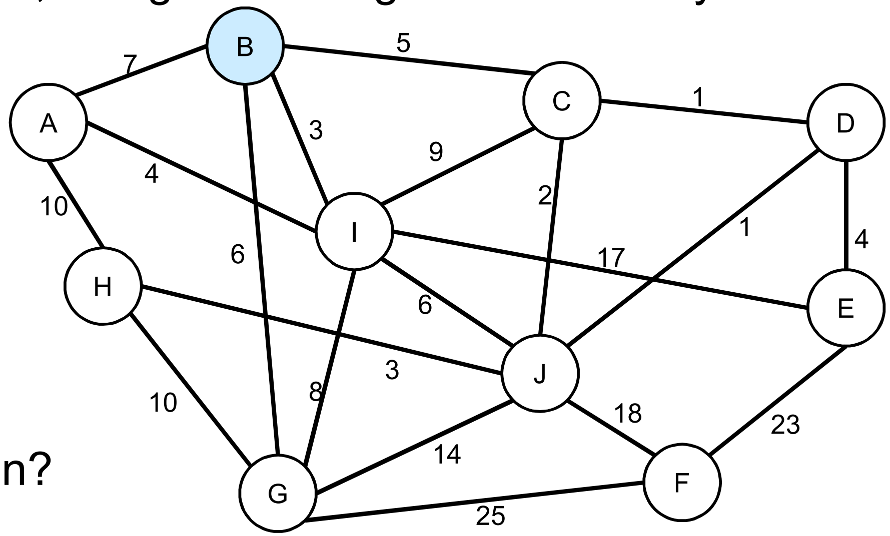
Wellington Public Transport Map

- 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"

Graph Algorithms.

- Many graph problems require searching through the graph, following edges.
- Simplest: search a graph from a node, doing something to each node you reach.



- Key issue:
  - Must keep track of the nodes you have visited, so you don't visit them again.
- Key question: what order to search in?
  - Depth first search
  - Breadth first search
  - Priority first search (search the most promising options)

Basic Graph Traversal Algorithm 1: Recursive DFS

```
TraverseGraph(node):
   if  node is not visited:
      visit the node
      process the node
      for each neighbour of node:
         if  neighbour is not visited:
            TraverseGraph(neighbour)
```

- Recording visited:
  - mark the node [not a good option]
  - keep a Set of visited nodes.

- Works on undirected graphs and on directed graphs.

Basic Graph Traversal Algorithm 2: Iterative

```
TraverseGraph(startNode):
   fringe ← Collection of nodes          Stack, Queue,
   put startNode on the fringe.
   while fringe is not empty:
      node ← remove from fringe
      if node is not visited:
         visit node
         process node
         for each neighbour of node:
            if neighbour is not visited:
               add neighbour to fringe
```

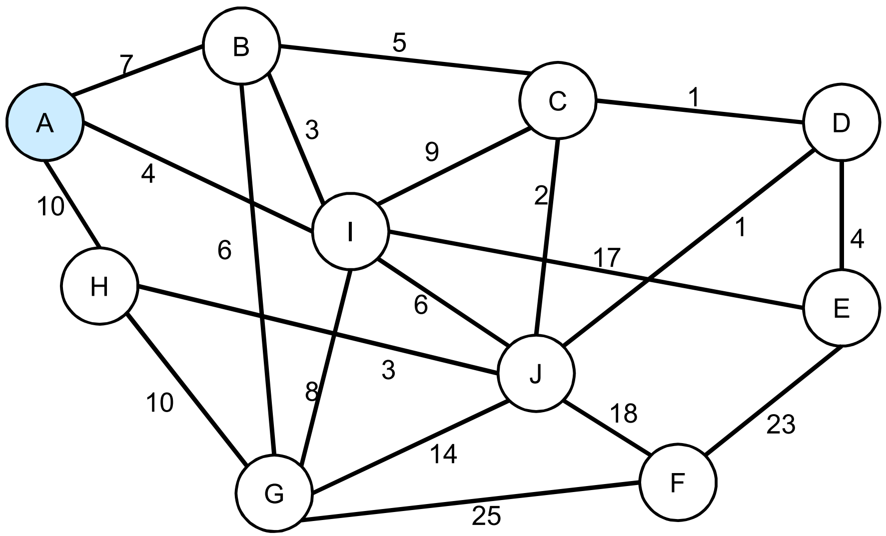
- Fringe is the collection of nodes that have been "seen" but not yet processed
- Stack/Queue determines the order: DFS or BFS

(Java code for the pseudocode algorithm)

```
public void traverseGraph(Node start){
    Set<Node> visited = new HashSet<Node>();
    Queue<Node> fringe = new ArrayDeque<Node>();    (or Stack, or PriorityQueue)
    fringe.offer(start);
    while  (!fringe.isEmpty()){
        Node node = fringe.poll();
        if (!visited.contains(node)) {
            visited.add(node);
            process(node);
            for (Node neighbour : node.getNeighbours()){
                if  (!visited.contains(neighbour)){
                    fringe.offer(neighbour);
                }
            }
        }
    }
}
```

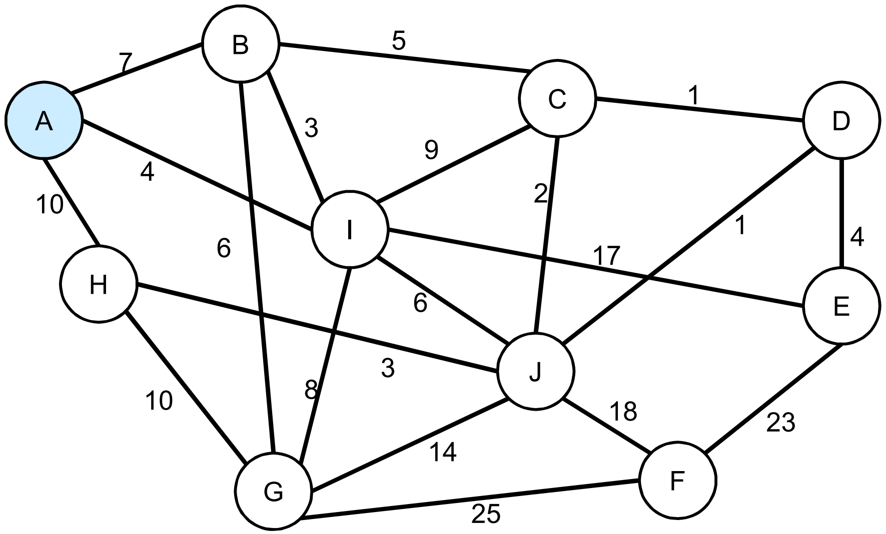
Iterative Traversal: Stack

Visited:



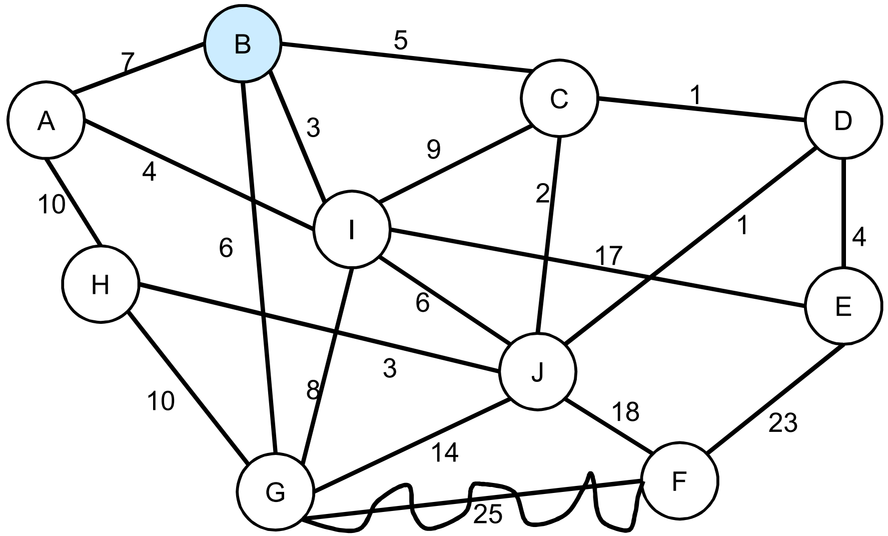
Iterative Traversal: Queue

Visited:



Finding a path:

- Suppose we want to find a path from start to a goal?
- Assume graph is of physical places,
  - each node has a location.
  - each edge has the actual path length
- Which order should we choose?
  - DFS?
  - BFS?
  - ??



Iterative traversal: finding a path: version 1

FindPath(start, goal):

```
fringe ←  PriorityQueue of nodes
put start on the fringe.
while  fringe is not empty:
    node ← remove from fringe
    if  node is not visited:
        visit node
        if node=goal:
            return the path to node
        for each neighbour of node:
            if  neighbour is not visited:
                add neighbour to fringe
```

Ordered by shortest straight-line distance from node to goal
= estimate of how much further to go.

Always removes the node on the fringe closest to the goal

How?

Problems:

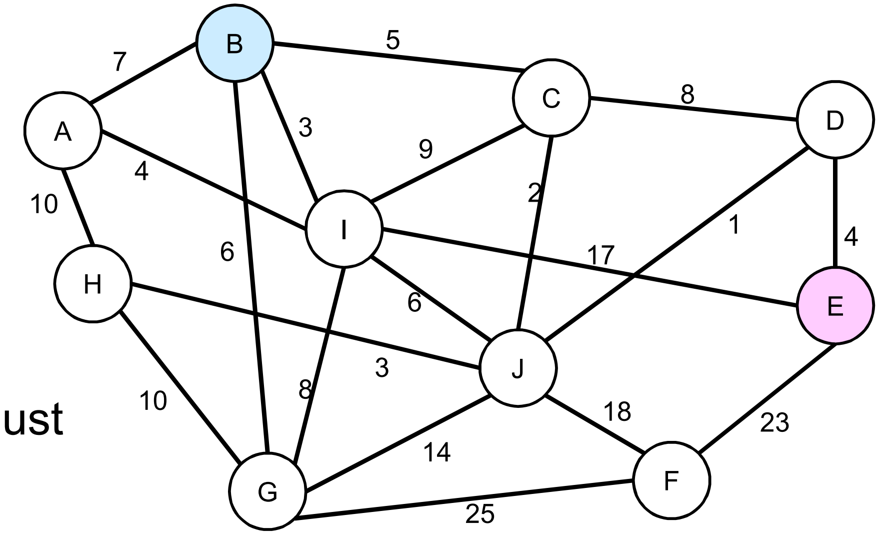
- Will it find the shortest path?
- How do we return the path?

Iterative search, keeping track of the path

- When we visit a node, we need to record how we got to it ("backpointers")

- Use a Map from node to previous node
- But how do we know where we came from when we take the node off the fringe?

- The fringe needs to contain more than just the node:
  - the node,
  - the node we came from,
  - …. the edge we came along
  - …. other information to help decide



Iterative traversal: finding a path: Storing paths.

FindPath(start, goal):

```
fringe ←  PriorityQueue of ⟨node, prev, edge…⟩        Ordered by shortest node-goal distance .
backpointers ← Map of nodes to previous node          or Map of nodes to edges
put ⟨start,null,null⟩ on the fringe.
while fringe is not empty:
   ⟨node, prev, edge…⟩ ← remove from fringe
   if node is not visited:
      visit node
      put ⟨node, prev⟩ into backpointers
      if node=goal:
         return backpointers                 Can reconstruct path to goal from the backpointers
      for each edge out of node to a neighbour:
         if neighbour is not visited:
            add ⟨neighbour, node, edge…⟩ to fringe
```

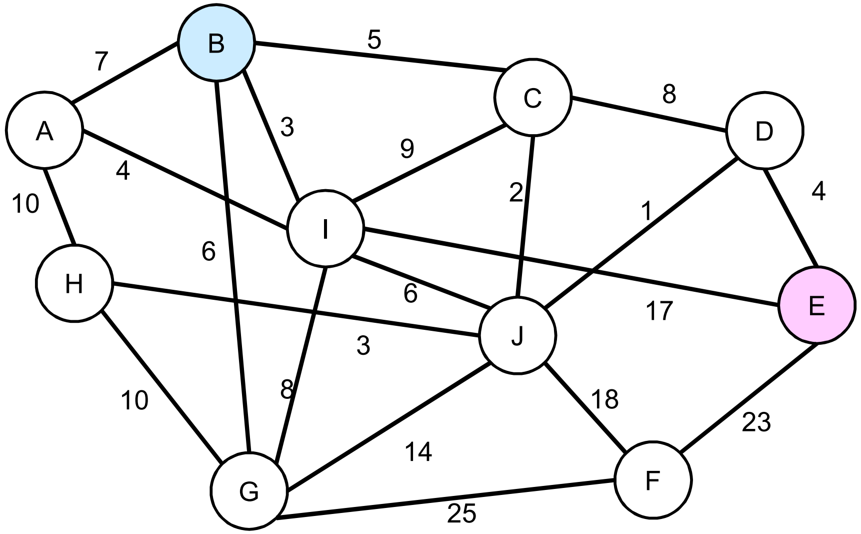
Ordered by shortest node-goal distance .
or Map of nodes to edges
Can reconstruct path to goal from the backpointers

Problems:

Will it find the shortest path?

If edges are directed, and contain the from-node and to-node, then we may only need to put the edge on the fringe!

Paths from BackPointers



- Backpointers:

```
ReconstructPath(start, goal, backpointers)
    path ← List of nodes
    add goal to path
    node ← goal
    while node ≠ start
        node ← backpointers.get(node)
        add node to path
    reverse path
```

Map:node→prev

```
ReconstructPath(start, goal, backpointers)
    path ← List of edges
    node ← goal
    do
        edge ← backpointers.get(node)
        add edge to path
        node ← edge.from
    until node = start
```

Map:node→edge