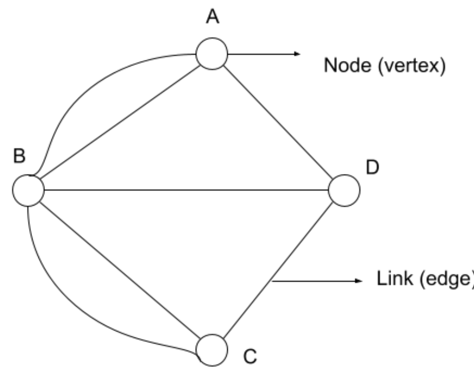
Topic 9 - Graphs

Graphs: Introduction

- Classifying graphs:
    - Undirected: no arrows
    - Directed: arrows
    - Weighted: number
    - Unweighted no numbers
- Graph Topologies
    - Bus
    - Ring
    - Tree
    - Manhattan
    - Mesh



Graphs: Representations

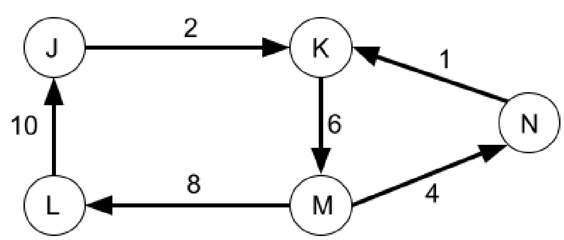
Data organization

Edge List

The edge list representation stores every link in the graph as a triplet; starting node, ending node, and the weight of the link:

```
E={(J,K,2),(K,M,6),(L,J,10),
    (M,L,8),(M,N,4),(N,K,1)}
```

Space Complexity: $\Theta(E)$ - considering only the edge list (E:number of **edges**)



Adjacency Matrix

We use ∞ to signal that there is no link. Sometimes we use 0, -1 or any unused number

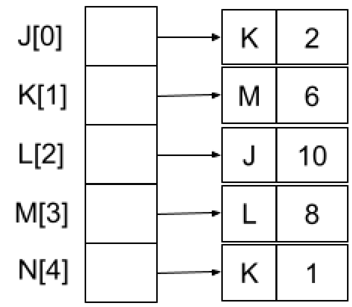
Space complexity: $\Theta(V^2)$ (V: number of **vertices**)

$$
\begin{array}{c} \\ J \\ K \\ L \\ M \\ N \end{array}
\begin{array}{c} \begin{array}{ccccc} J & K & L & M & N \end{array} \\ \begin{pmatrix} \infty & 2 & \infty & \infty & \infty \\ \infty & \infty & \infty & 6 & \infty \\ 10 & \infty & \infty & \infty & \infty \\ \infty & \infty & 8 & \infty & 4 \\ \infty & 1 & \infty & \infty & \infty \end{pmatrix} \end{array}
$$

Adjacency List

Every position in the array represents one node of the graph

`Space complexity: Θ(V+E) (V: number of` **`vertices`**`; E: number of` **`edges`**`)`



Graph Operations / Data Manipulation

Graph Construction

`Graph(V,E)`	make a new graph with given set of vertices and edges collections(constructor)
`addVertex(v)`	add vertex (node) `v` to the graph
`addEdge(e)`	add edge `e` (link to the graph)

Graph Modification

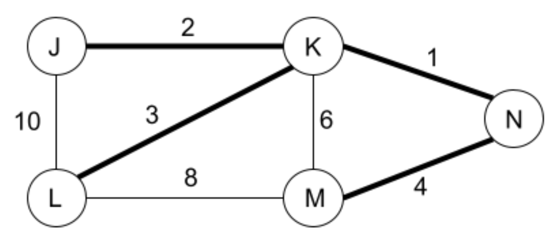
`removeVertex(v)`	remove vertex (node) `v` from the graph
`removeEdge(e)`	remove edge (link) `e` from graph

Graph Query

`vertices(G)`	return the collection of vertices of `G`
`from(e)`	return the source vertex of edge `e`
`to(e)`	return the destination vertex of edge `e`
`neighbors(v)`	return the collection of nodes directly connected to `v`
`edges(G)`	return the collection of edges of `G`
`weight(e)`	return the weight of edge `e`

Minimum spanning tree (MST)

- The minimum spanning tree of a graph G is the tree that includes **all the nodes** of G using the subset of edges with **minimum total weight.**
- V **vertices** and (V-1) **edges**
- **Algorithms:**
  a. Prim's Algorithm
  b. Kruskal's Algorithm



- Steps for Prim:
  a. Start building the tree with any graph node

   b. while the tree is not complete: select the lowest-cost link connecting any node in the tree to any node not yet in the tree
- Steps for Kruskal
   a. Start building the tree with all the graph's nodes, disconnected from each other
   b. while the tree is not complete: select the lowest-cost link that connects two different trees and the node at its extreme to be part of the MST

Prim's algorithm

```
function PRIM_MST
     // Initialise the spanning tree T with one vertex from the
Graph
     vs = vertices(G)
     T = new Graph (FIRST(vs),{})
     // While there are still vertices to add
     while ( |T| < |G| )
          // find the set of links L from node in
          // tree to node not in tree
          L={ e| e ∈ edges(G) ^ FROM(e) ∈ T ^ TO(e) ∈ G }
          // find, in L, the minimum weight edge
          newE =min_{e∈L} weight(e)
          // add that edge and vertex to spanning tree
          addVertex(T,TO(e))
          addEdge(T,newE)
     end while
end function
```

L={ e| e ∈ edges(G) ^ FROM(e) ∈ T ^ TO(e) ∈ G }

newE $=min_{e \in L}$ weight(e)

In Pseudocode:

```
w=∞
for (e ∈ edges(G) ^ FROM(e) ∈ T ^ TO(e) ∈ G )
     if weight (e)<w then
          w = weight
          newE=e
          newVertex = TO(e)
     end if
end for
```

Kruskal algorithm

```
function KRUSKAL_MST(G)
      // Initialisation of Tree (spanning Tree) and F (Disjoint Set)
      vs = vertices (G)
      T = new Graph(vs,{})
      for v ∈ vs  do
            MAKE-SET(F,v)
      // Initialisation of L edges sorted by weight
      L = sort(edges(G))
      for e ∈ L do
            // Adding edges to T and updating F
            if FIND(F,FROM(e) != FIND(F,TO(e))
                  addEdge(T,e)
                  UNION(F,FROM(e), TO(e))
end function
```

Dijkstra's algorithm

1. Initialise routing table
2. Initialise set of unexplored nodes
3. **while** (U has elements)

```
      Select node u ∈ U with minimum distance to source node
      for each neighbor n of u in U
            calculate new distance d = dist(A,u)+weight (u,n)
            if (d< dist(A,n)in routing table, update table)
      end for
      remove u from U
```