WIA 1002 DATA STRUCTURE

SEM 2, SESSION 2024/205

NURUL JAPAR
nuruljapar@um.edu.my

HOO WAI LAM
wlhoo@um.edu.my

Graph

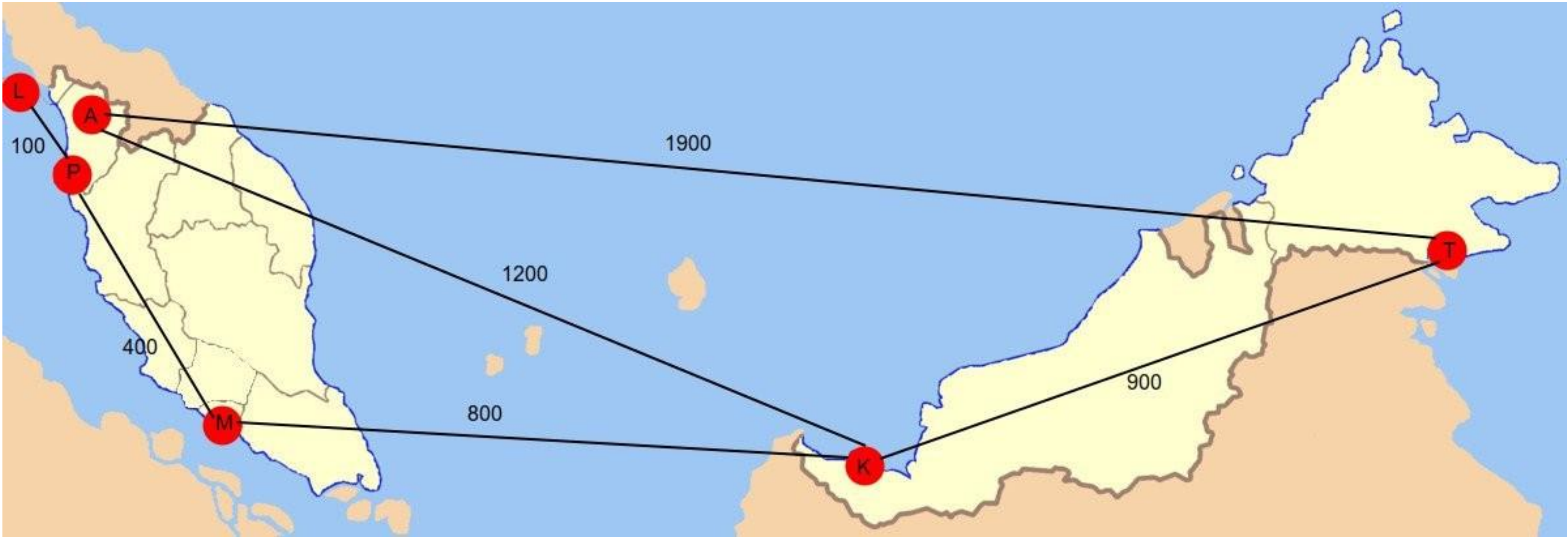
- Concept
- Modelling Graph
- Implementation
- Graph Traversal

CONCEPT

Graph

- A concept in mathematics, also a data structure.
- A set of vertices, V and edges, G=(V,E).
- In graph-like problems, these components have natural correspondences to problem elements
  - » Entities are nodes and interactions between entities are edges
  - » Many complex systems are graph-like.

Example – Flights between cities



Graph

- 2 vertices are “adjacent” to each other if they share the same edge.
- If, from vertex p, after travel along 1 or more edges, we eventually reach vertex q, we say there is a “path” from p to q.

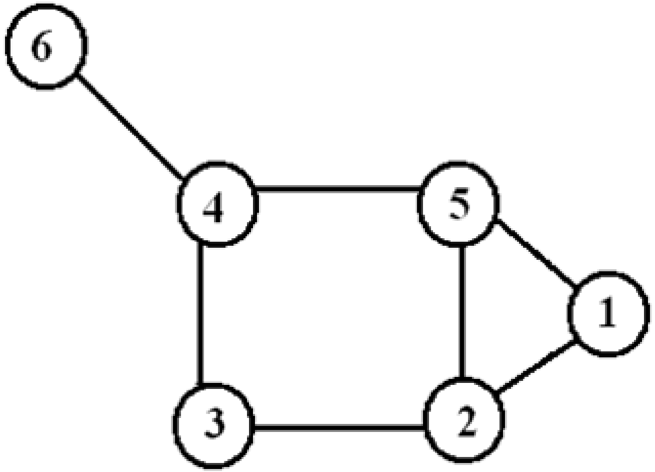
Graph

- Can be directed or undirected.
- Can be unweighted or weighted
  - » Each edge in a weighted graph carries a value – weight of the edge

MODELLING GRAPH

Modelling Graph

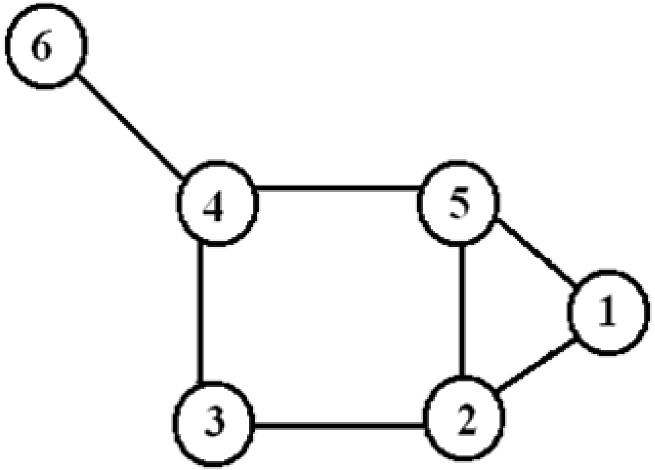
- There are many ways to model graphs in mathematics, among all: adjacency matrix and adjacency list



Modelling Graph

$$
\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 1 & 0 \\
3 & 0 & 1 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

adjacency matrix



graph

Node list	
1	2 5
2	1 3 5
3	2 4
4	3 5 6
5	1 2 4
6	4

adjacency list

Representing Vertices

- Vertices can be represented with Array, ArrayList or Linked-list.
- Array implementation is easy but less flexible.

Using ArrayList:

```
// Class City has created before this line
ArrayList<City> vertices = new ArrayList<City>();

vertices.add("Seattle");
vertices.add("San Francisco");
vertices.add("Los Angeles");
… …
```

Representing Edges : Matrix

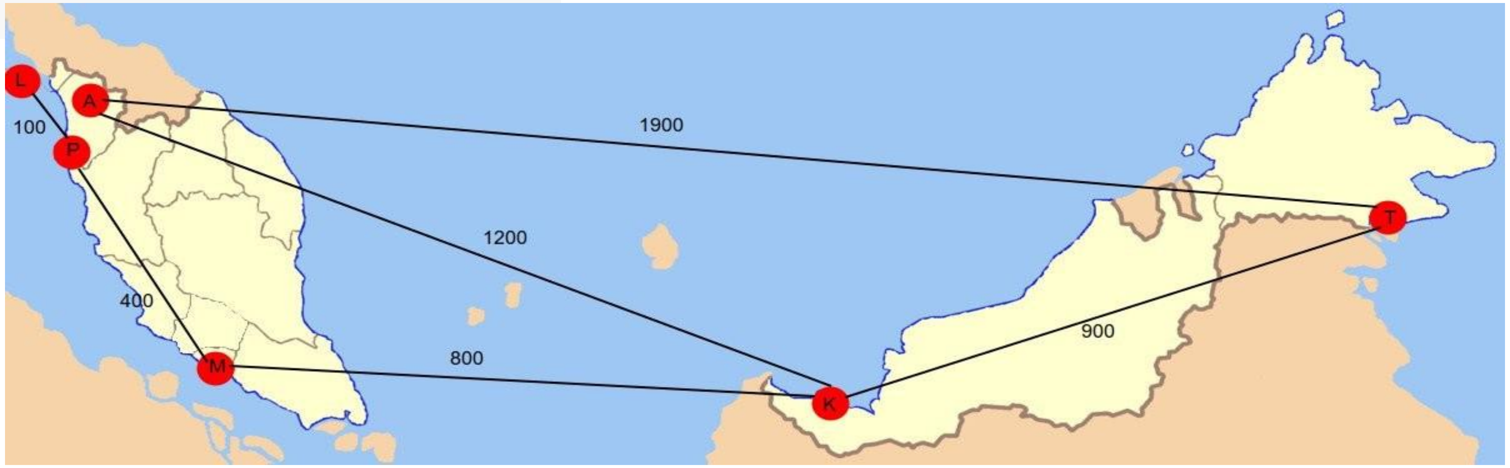
- Can be implemented with adjacency matrix or adjacency list.
- For adjacency matrix implementation, a 2D array with value 1 and 0 is used to show the presence of an edge.
- If the graph has n vertices, the size of the matrix is nXn.

Representing Edges : Matrix

- E.g: edge[2][5] =1 and edge[2][6] = 0 means vertices[2] is adjacent to vertices[5] but not vertices[6].
- For weighted graphs, the values are replaced with the weight of the edges

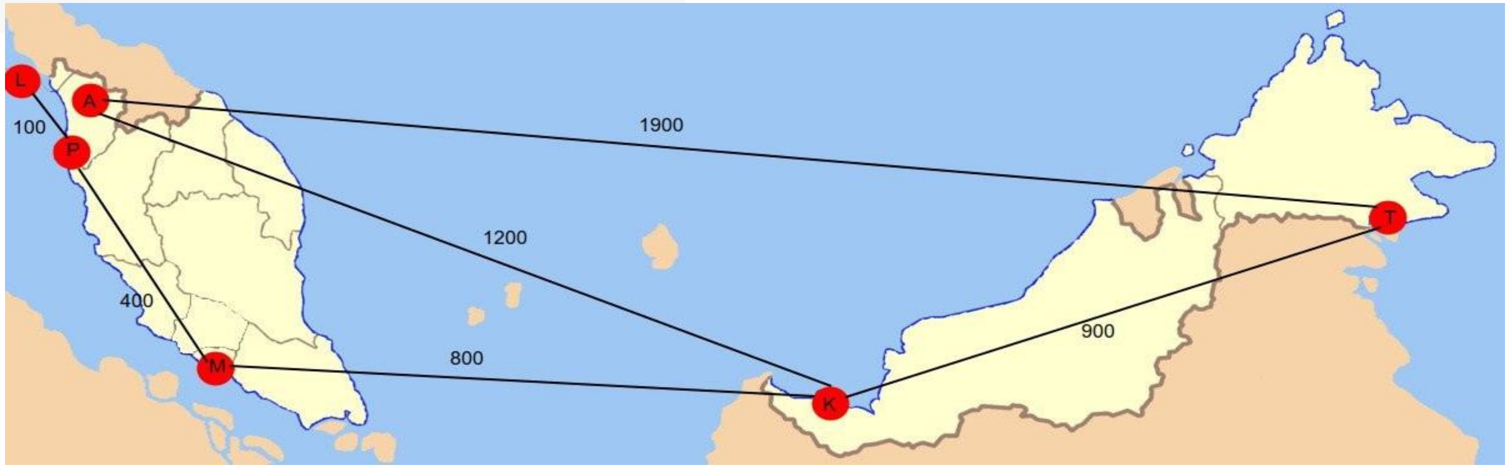
Representing Edges : Matrix

```
int[][] adjacencyMatrix = {
    { 0, 1, 0, 0, 0, 1}, // Alor Setar
    { 1, 0, 0, 1, 0, 1}, // Kuching
    { 0, 0, 0, 0, 1, 0}, // Langkawi
    { 0, 1, 0, 0, 1, 0}, // Melaka
    { 0, 0, 1, 1, 0, 0}, // Penang
    { 1, 1, 0, 0, 0, 0}  // Tawau
}
```



Representing Edges : Matrix (Directed and Weighted)

```
int[][] adjacencyMatrix = {
    {    0, 1200, 0,   0,   0,   0}, // Alor Setar
    {    0,    0, 0, 800,   0, 900}, // Kuching
    {    0,    0, 0,   0, 100,   0}, // Langkawi
    {    0,    0, 0,   0, 400,   0}, // Melaka
    {    0,    0, 0,   0,   0,   0}, // Penang
    { 1900,    0, 0,   0,   0,   0}  // Tawau
}
```



Representing Edges : Matrix

- Adjacency matrix is fast and easy to implement, but it needs large space to hold the matrix if *n* is large. (imagine a graph that represents “close contacts” for covid-19)

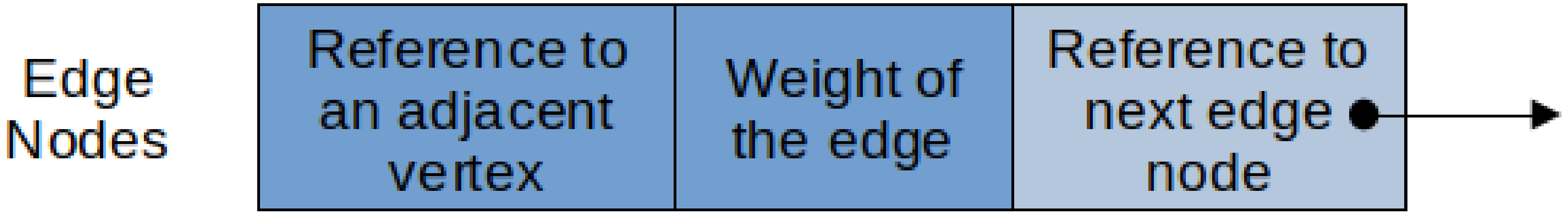
- If the number of edges is also large, we have a dense matrix, and it is justifiable.

Representing Edges : Matrix

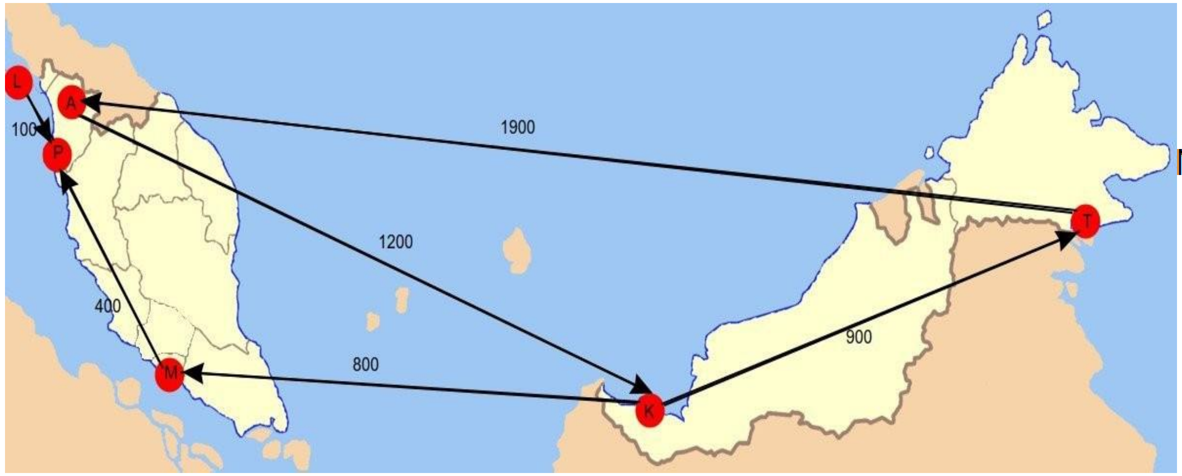
- If only a few edges, we have sparse matrix (many elements with value 0), and it is a waste of memory space. Should consider linked-list

Representing Edges : Linked-list

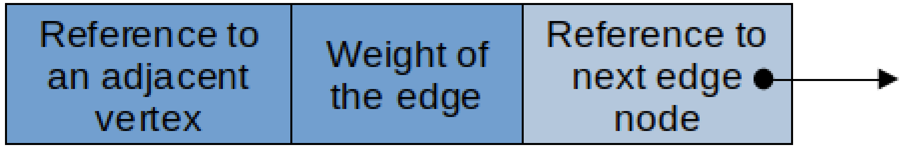
- One linked-list for each vertex.
- Each node in the linked list contains a reference to an adjacency vertex.
- Additional entry for weight if it is a weighted graph.

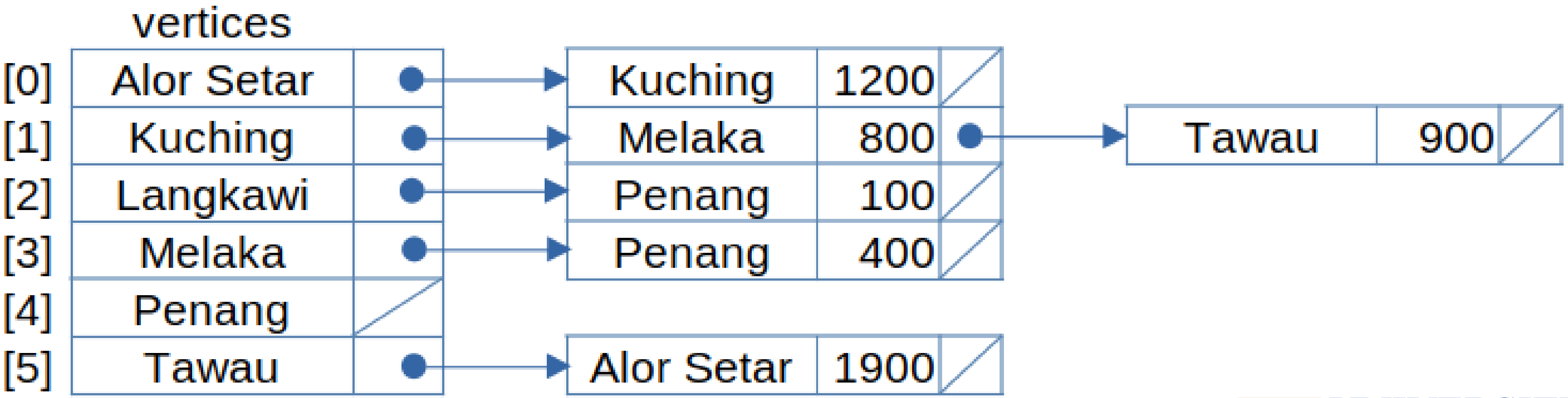


Representing Edges : Linked-list with Array



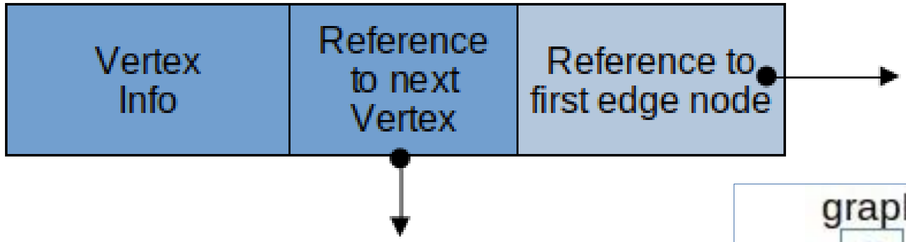
Edge Nodes



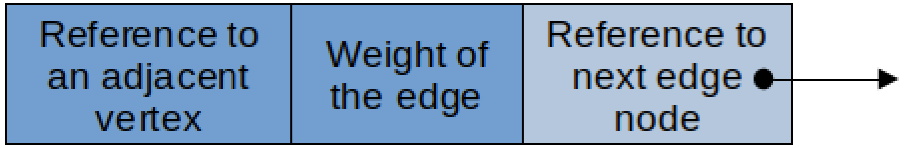


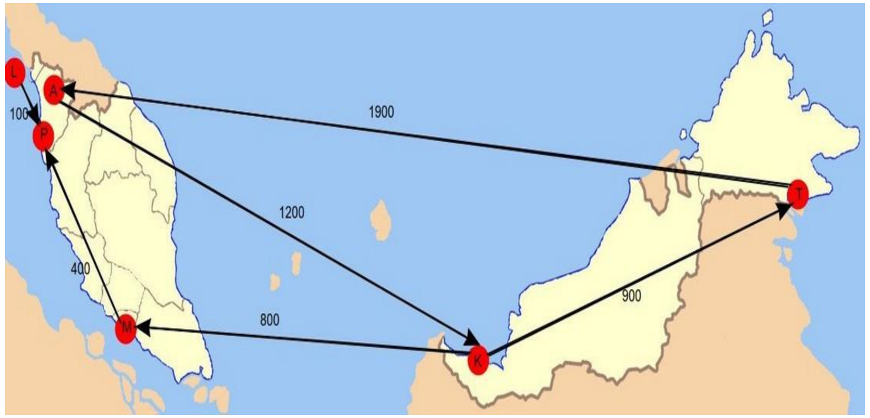
Representing Edges : Linked-List (second way of implementation)

Vertex



Edge Nodes





IMPLEMENTATION

Implementing Graphs - Vertex

```
class Vertex<T extends Comparable<T>, N extends Comparable <N>> {
   T vertexInfo;
   int indeg;
   int outdeg;
   Vertex<T,N> nextVertex;
   Edge<T,N> firstEdge;

   public Vertex() {
      vertexInfo=null;
      indeg=0;
      outdeg=0;
      nextVertex = null;
      firstEdge = null;
   }

   public Vertex(T vInfo, Vertex<T,N> next) {
      vertexInfo = vInfo;
      indeg=0;
      outdeg=0;
      nextVertex = next;
      firstEdge = null;
   }
}
```

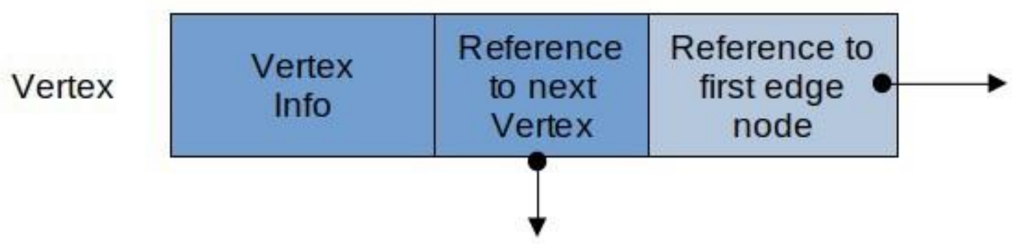
Implementing Graphs - Vertex

```
class Vertex<T extends Comparable<T>, N extends Comparable <N>> {
    T vertexInfo;
    int indeg;
    int outdeg;
    Vertex<T,N> nextVertex;
    Edge<T,N> firstEdge;

    public Vertex() {
        vertexInfo=null;
        indeg=0;
        outdeg=0;
        nextVertex = null;
        firstEdge = null;
    }

    public Vertex(T vInfo, Vertex<T,N> next) {
        vertexInfo = vInfo;
        indeg=0;
        outdeg=0;
        nextVertex = next;
        firstEdge = null;
    }
}
```

Vertex info

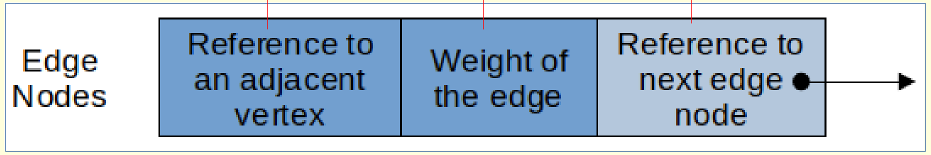
Reference to next vertex

Reference to first edge node



Implementing Graphs – Weighted Edge

```java
class Edge<T extends Comparable<T>, N extends Comparable <N>> {
    Vertex<T,N> toVertex;
    N weight;
    Edge<T,N> nextEdge;

    public Edge()  {
        toVertex = null;
        weight = null;
        nextEdge = null;
    }

    public Edge(Vertex<T,N> destination, N w, Edge<T,N> a)   {
        toVertex = destination;
        weight = w;
        nextEdge = a;
    }
}
```



Implementing Graphs – Weighted Graph

```
class Graph<T extends Comparable<T>, N extends Comparable <N>> {
    Vertex<T,N> head;
    int size;

    public Graph() {
        head=null;
        size=0;
    }
```

Get number of vertices

```
public int getSize()    {
    return this.size;
}
```

hasVertex – is this vertex in graph?

```
public boolean hasVertex(T v) {
   if (head==null)
      return false;
   Vertex<T,N> temp = head;
   while (temp!=null)    {
      if ( temp.vertexInfo.compareTo( v ) == 0 )
         return true;
      temp=temp.nextVertex;
   }
   return false;
}
```

Compare: to determine whether it is the vertex we are looking for

get inDeg of a Vertex

```
public int getIndeg(T v)  {
    if (hasVertex(v)==true) {
        Vertex<T,N> temp = head;
        while (temp!=null) {
            if ( temp.vertexInfo.compareTo( v ) == 0 )
                return temp.indeg;
            temp=temp.nextVertex;
        }
    }
    return -1;
}
```

Return -1 if cannot find

get inDeg of a Vertex

```
public int getIndeg(T v)  {
    if (hasVertex(v)==true) {
        Vertex<T,N> temp = head;
        while (temp!=null) {
            if ( temp.vertexInfo.compareTo( v ) == 0 )
                return temp.indeg;
            temp=temp.nextVertex;
        }
    }
    return -1;
}
```

Return -1 if cannot find

Get outDeg?
Try to code it!

Add Vertex

```
public boolean addVertex(T v) {
   if (hasVertex(v)==false)   {
      Vertex<T,N> temp=head;
      Vertex<T,N> newVertex = new Vertex<>(v, null);
      if (head==null)
         head=newVertex;
      else {
         Vertex<T,N> previous=head;;
         while (temp!=null)  {
            previous=temp;
            temp=temp.nextVertex;
         }
         previous.nextVertex=newVertex;
      }
      size++;
      return true;
   }
   else
      return false;
}
```

Add Vertex

```
public boolean addVertex(T v) {
   if (hasVertex(v)==false)    {
      Vertex<T,N> temp=head;
      Vertex<T,N> newVertex = new Vertex<>(v, null);
      if (head==null)
         head=newVertex;
      else {
         Vertex<T,N> previous=head;;
         while (temp!=null)   {
            previous=temp;
            temp=temp.nextVertex;
         }
         previous.nextVertex=newVertex;
      }
      size++;
      return true;
   }
   else
      return false;
}
```

The vertex is not in the graph

Graph is empty. Point head to this vertex

Use previous to move to the last vertex

Add the vertex as last in the list

Vertex is already in the graph

Find the index of the vertex

```
public int getIndex(T v) {
    Vertex<T,N> temp = head;
    int pos=0;
    while (temp!=null)   {
        if ( temp.vertexInfo.compareTo( v ) == 0 )
            return pos;
        temp=temp.nextVertex;
        pos+=1;
    }
    return -1;
}
```

Loop to find the vertex

Vertex is found

Move temp to next vertex

Return all the vertex info: ArrayList

Return an ArrayList that stores T

```
public ArrayList<T> getAllVertexObjects() {
    ArrayList<T> list = new ArrayList<>();
    Vertex<T,N> temp = head;
    while (temp!=null)    {
        list.add(temp.vertexInfo);
        temp=temp.nextVertex;
    }
    return list;
}
```

Use “add” method of ArrayList to add each vertex info

Get vertex info: Specific index/position

If the position is not valid

```
public T getVertex(int pos) {
    if (pos>size-1 || pos<0)
        return null;
    Vertex<T,N> temp = head;
    for (int i=0; i<pos; i++)
        temp=temp.nextVertex;
    return temp.vertexInfo;
}
```

Check Edge

Graph is empty

No such vertices

Search for the edge in valid condition

```
public boolean hasEdge(T source, T destination) {
    if (head==null)
        return false;
    if (!hasVertex(source) || !hasVertex(destination))
        return false;
    Vertex<T,N> sourceVertex = head;
    while (sourceVertex!=null) {
        if ( sourceVertex.vertexInfo.compareTo( source ) == 0 )   {
            // Reached source vertex, look for destination now
            Edge<T,N> currentEdge = sourceVertex.firstEdge;
            while (currentEdge != null) {
                if (currentEdge.toVertex.vertexInfo.compareTo(destination)==0)
                // destination vertex found
                    return true;
                currentEdge=currentEdge.nextEdge;
            }
        }
        sourceVertex=sourceVertex.nextVertex;
    }
    return false;
}
```

Find no such edge in previous loop

Check Edge

Source vertex found. Create an edge reference here and look for destination vertex in the second while loop

```
public boolean hasEdge(T source, T destination) {
    if (head==null)
        return false;
    if (!hasVertex(source) || !hasVertex(destination))
        return false;
    Vertex<T,N> sourceVertex = head;
    while (sourceVertex!=null) {
        if ( sourceVertex.vertexInfo.compareTo( source ) == 0 )   {
            // Reached source vertex, look for destination now
            Edge<T,N> currentEdge = sourceVertex.firstEdge;
            while (currentEdge != null) {
                if (currentEdge.toVertex.vertexInfo.compareTo(destination)==0)
                // destination vertex found
                    return true;
                currentEdge=currentEdge.nextEdge;
            }
        }
        sourceVertex=sourceVertex.nextVertex;
    }
    return false;
}
```

If the source vertex is not found, go to next iteration of outer while loop

Add a new edge + a weight

```
public boolean addEdge(T source, T destination, N w)   {
   if (head==null)
      return false;
   if (!hasVertex(source) || !hasVertex(destination))
      return false;
   Vertex<T,N> sourceVertex = head;
   while (sourceVertex!=null) {
      if ( sourceVertex.vertexInfo.compareTo( source ) == 0 )   {
         // Reached source vertex, look for destination now
         Vertex<T,N> destinationVertex = head;
         while (destinationVertex!=null)  {
            if ( destinationVertex.vertexInfo.compareTo( destination ) == 0 )   {
               // Reached destination vertex, add edge here
               Edge<T,N> currentEdge = sourceVertex.firstEdge;
               Edge<T,N> newEdge = new Edge<>(destinationVertex, w, currentEdge);
               sourceVertex.firstEdge=newEdge;
               sourceVertex.outdeg++;
               destinationVertex.indeg++;
               return true;
            }
            destinationVertex=destinationVertex.nextVertex;
         }
      }
      sourceVertex=sourceVertex.nextVertex;
   }
   return false;
}
```

UNIVERSITI MALAYA

Add a new edge + a weight

```
public boolean addEdge(T source, T destination, N w)   {
   if (head==null)
      return false;
   if (!hasVertex(source) || !hasVertex(destination))
      return false;
   Vertex<T,N> sourceVertex = head;
   while (sourceVertex!=null) {
      if ( sourceVertex.vertexInfo.compareTo( source ) == 0 )   {
         // Reached source vertex, look for destination now
         Vertex<T,N> destinationVertex = head;
         while (destinationVertex!=null)  {
            if ( destinationVertex.vertexInfo.compareTo( destination ) == 0 )   {
               // Reached destination vertex, add edge here
               Edge<T,N> currentEdge = sourceVertex.firstEdge;
               Edge<T,N> newEdge = new Edge<>(destinationVertex, w, currentEdge);
               sourceVertex.firstEdge=newEdge;
               sourceVertex.outdeg++;
               destinationVertex.indeg++;
               return true;
            }
            destinationVertex=destinationVertex.nextVertex;
         }
      }
      sourceVertex=sourceVertex.nextVertex;
   }
   return false;
}
```

Only this part is different from “hasEdge”.

This block loop to find destination vertex in the nested while

Add a new edge + a weight

Create an edge pointer, and point to edges list which the source vertex is pointing to

```
// Reached source vertex, look for destination now
Vertex<T,N> destinationVertex = head;
while (destinationVertex!=null)  {
   if ( destinationVertex.vertexInfo.compareTo( destination ) == 0 )   {
      // Reached destination vertex, add edge here
      Edge<T,N> currentEdge = sourceVertex.firstEdge;
      Edge<T,N> newEdge = new Edge<>(destinationVertex, w, currentEdge);
      sourceVertex.firstEdge=newEdge;
      sourceVertex.outdeg++;
      destinationVertex.indeg++;
      return true;
   }
   destinationVertex=destinationVertex.nextVertex;
}
```

Add 1 to in and out degree

Let the source vertex point to the new edge object

Create the edge. Let the "ref to next edge" point to the edges list

Edge Nodes	Reference to an adjacent vertex	Weight of the edge	Reference to next edge node

Retrieve the weight of an Edge

```
public N getEdgeWeight(T source, T destination) {
   N notFound=null;
   if (head==null)
      return notFound;
   if (!hasVertex(source) || !hasVertex(destination))
      return notFound;
   Vertex<T,N> sourceVertex = head;
   while (sourceVertex!=null) {
      if ( sourceVertex.vertexInfo.compareTo( source ) == 0 )   {
         // Reached source vertex, look for destination now
         Edge<T,N> currentEdge = sourceVertex.firstEdge;
         while (currentEdge != null) {
            if (currentEdge.toVertex.vertexInfo.compareTo(destination)==0)
            // destination vertex found
               return currentEdge.weight;
            currentEdge=currentEdge.nextEdge;
         }
      }
      sourceVertex=sourceVertex.nextVertex;
   }
   return notFound;
}
```

Retrieve the weight of an Edge

Quite similar to hasEdge

```
public N getEdgeWeight(T source, T destination) {
    N notFound=null;
    if (head==null)
        return notFound;
    if (!hasVertex(source) || !hasVertex(destination))
        return notFound;
    Vertex<T,N> sourceVertex = head;
    while (sourceVertex!=null) {
        if ( sourceVertex.vertexInfo.compareTo( source ) == 0 )   {
            // Reached source vertex, look for destination now
            Edge<T,N> currentEdge = sourceVertex.firstEdge;
            while (currentEdge != null) {
                if (currentEdge.toVertex.vertexInfo.compareTo(destination)==0)
                // destination vertex found
                    return currentEdge.weight;
                currentEdge=currentEdge.nextEdge;
            }
        }
        sourceVertex=sourceVertex.nextVertex;
    }
    return notFound;
}
```

Edge found, return weight

Return all the neighbours to an ArrayList

```
public ArrayList<T> getNeighbours (T v)  {
   if (!hasVertex(v))
      return null;
   ArrayList<T> list = new ArrayList<T>();
   Vertex<T,N> temp = head;
   while (temp!=null)    {
      if ( temp.vertexInfo.compareTo( v ) == 0 )    {
         // Reached vertex, look for destination now
         Edge<T,N> currentEdge = temp.firstEdge;
         while (currentEdge != null) {
            list.add(currentEdge.toVertex.vertexInfo);
            currentEdge=currentEdge.nextEdge;
         }
      }
      temp=temp.nextVertex;
   }
   return list;
}
```

UNIVERSITI MALAYA

Return all the neighbours to an ArrayList

Outer while: loop to find the vertex, and create a ref to edge if found

```
public ArrayList<T> getNeighbours (T v)  {
    if (!hasVertex(v))
        return null;
    ArrayList<T> list = new ArrayList<T>();
    Vertex<T,N> temp = head;
    while (temp!=null)    {
        if ( temp.vertexInfo.compareTo( v ) == 0 )    {
            // Reached vertex, look for destination now
            Edge<T,N> currentEdge = temp.firstEdge;
            while (currentEdge != null) {
                list.add(currentEdge.toVertex.vertexInfo);
                currentEdge=currentEdge.nextEdge;
            }
        }
        temp=temp.nextVertex;
    }
    return list;
}
```

Nested while: read edges and add to ArrayList

UNIVERSITI MALAYA

Print graph

```
public void printEdges()    {
   Vertex<T,N> temp=head;
   while (temp!=null) {
      System.out.print("# " + temp.vertexInfo + " : " );
      Edge<T,N> currentEdge = temp.firstEdge;
      while (currentEdge != null) {
         System.out.print("[" + temp.vertexInfo + ","
            + currentEdge.toVertex.vertexInfo +"] " );
         currentEdge=currentEdge.nextEdge;
      }
      System.out.println();
      temp=temp.nextVertex;
   }
}
```

Print graph

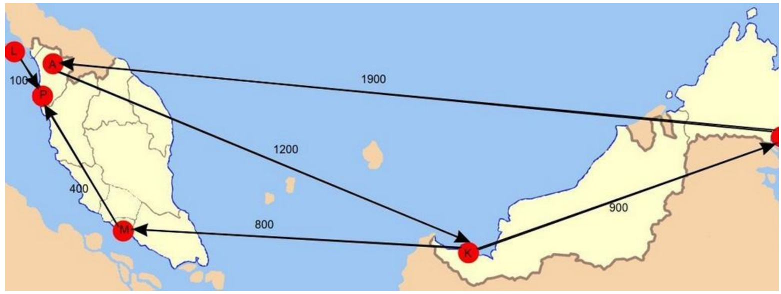
Print a vertex

Print edges in a nested loop

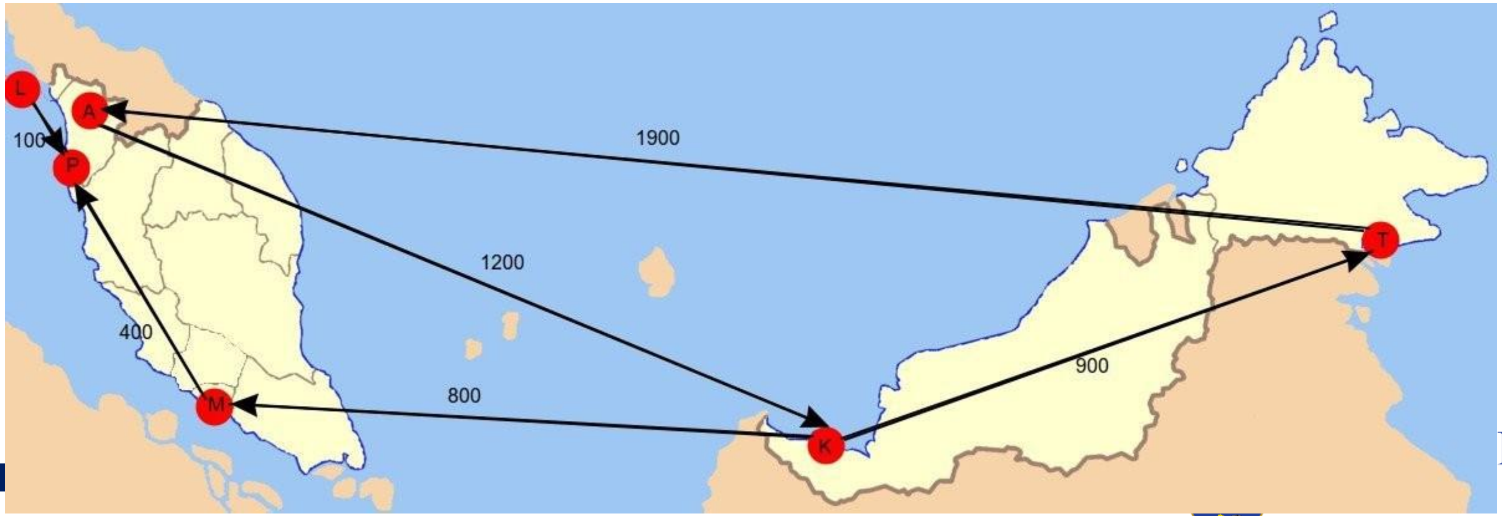
```
public void printEdges()    {
    Vertex<T,N> temp=head;
    while (temp!=null) {
        System.out.print("# " + temp.vertexInfo + " : " );
        Edge<T,N> currentEdge = temp.firstEdge;
        while (currentEdge != null) {
            System.out.print("[" + temp.vertexInfo + ","
                + currentEdge.toVertex.vertexInfo +"] " );
            currentEdge=currentEdge.nextEdge;
        }
        System.out.println();
        temp=temp.nextVertex;
    }
}
```

Test Program

```
public class TestWeightedGraph {
  public static void main(String[] args) {
     WeightedGraph<String, Integer> graph1 = new WeightedGraph<>();
     String[] cities = {"Alor Setar", "Kuching", "Langkawi", "Melaka", "Penang", "Tawau"};
     for (String i : cities)
        graph1.addVertex(i);

     System.out.println("The number of vertices in graph1: " + graph1.getSize());

     System.out.println("Cities and their vertices ");
     for (int i = 0; i<=graph1.getSize()-1; i++)
        System.out.print( i + ": " + graph1.getVertex(i) + "\t");
     System.out.println();

     System.out.println("Is Melaka in the Graph? " + graph1.hasVertex("Melaka"));
     System.out.println("Is Ipoh in the Graph? " + graph1.hasVertex("Ipoh"));
     System.out.println();

     System.out.println("Kuching at index:  " + graph1.getIndex("Kuching"));
     System.out.println("Ipoh at index:  " + graph1.getIndex("Ipoh"));
     System.out.println();

     System.out.println("add edge Kuching - Melaka: " + graph1.addEdge("Kuching", "Melaka", 800) );
     System.out.println("add edge Langkawi - Penang : " + graph1.addEdge("Langkawi", "Penang", 100) );
     System.out.println("add edge Melaka - Penang : " + graph1.addEdge("Melaka", "Penang", 400) );
     System.out.println("add edge Alor Setar - Kuching : " + graph1.addEdge("Alor Setar", "Kuching", 1200) );
     System.out.println("add edge Tawau - Alor Setar : " + graph1.addEdge("Tawau", "Alor Setar", 1900) );
     System.out.println("add edge Kuching - Tawau : " + graph1.addEdge("Kuching", "Tawau", 900) );
     System.out.println("add edge Langkawi - Ipoh : " + graph1.addEdge("Langkawi", "Ipoh", 200) );
     System.out.println();
```

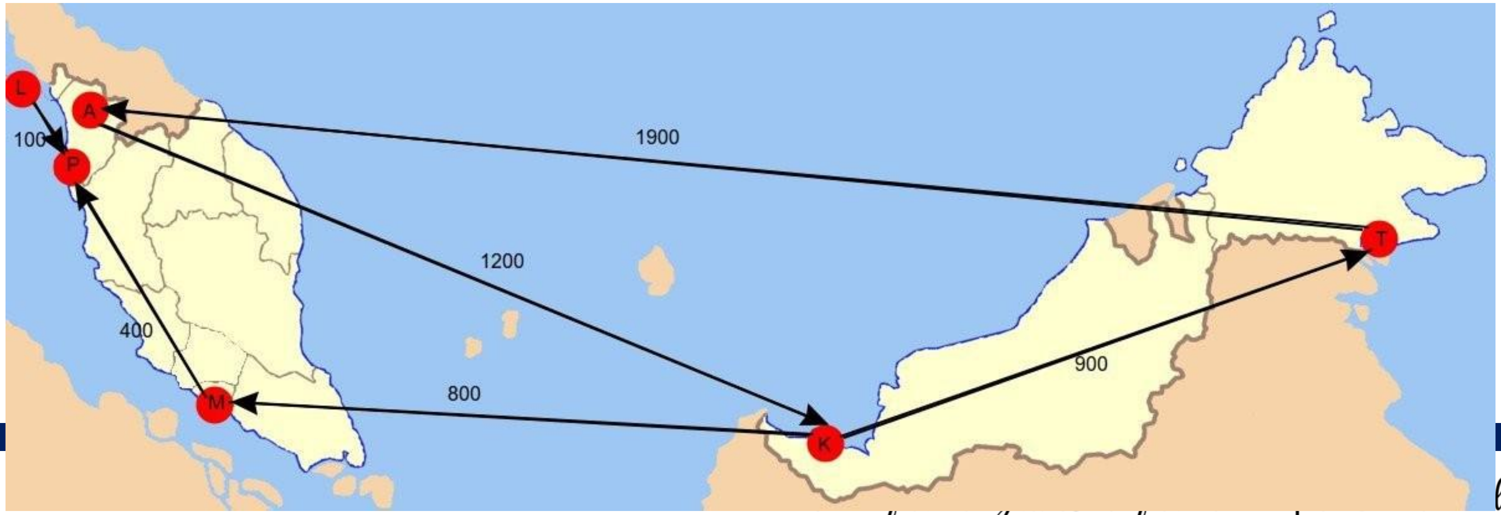


```java
    System.out.println("has edge from Kuching to Melaka?  " + graph1.hasEdge("Kuching", "Melaka") );
    System.out.println("has edge from Melaka to Langkawi?  " + graph1.hasEdge("Melaka", "Kuching") );
    System.out.println("has edge from Ipoh to Langkawi?  " + graph1.hasEdge("Ipoh", "Langkawi") );
    System.out.println();

    System.out.println("weight of edge from Kuching to Melaka?  " + graph1.getEdgeWeight("Kuching", "Melaka") );
    System.out.println("weight of edge from Tawau to Alor Setar?  " + graph1.getEdgeWeight("Tawau", "Alor Setar") );
    System.out.println("weight of edge from Semporna to Ipoh?  " + graph1.getEdgeWeight("Semporna", "Ipoh") );
    System.out.println();

    System.out.println("In and out degree for Kuching is " + graph1.getIndeg("Kuching") + " and " + graph1.getOutdeg("Kuching") );
    System.out.println("In and out degree for Penang is " + graph1.getIndeg("Penang") + " and " + graph1.getOutdeg("Penang") );
    System.out.println("In and out degree for Ipoh is " + graph1.getIndeg("Ipoh") + " and " + graph1.getOutdeg("Ipoh") );
    System.out.println();

    System.out.println("Neighbours of Kuching : " + graph1.getNeighbours("Kuching"));
    System.out.println("\nPrint Edges : " );
    graph1.printEdges();

  }
}
```



```
The number of vertices in graph1: 6
Cities and their vertices
0: Alor Setar   1: Kuching  2: Langkawi 3: Melaka   4: Penang   5: Tawau
Is Melaka in the Graph? true
Is Ipoh in the Graph? false

Kuching at index:  1
Ipoh at index:  -1

add edge Kuching - Melaka: true
add edge Langkawi - Penang : true
add edge Melaka - Penang : true
add edge Alor Setar - Kuching : true
add edge Tawau - Alor Setar : true
add edge Kuching - Tawau : true
add edge Langkawi - Ipoh : false
```

```
has edge from Kuching to Melaka?  true
has edge from Melaka to Langkawi?  false
has edge from Ipoh to Langkawi?  false

weight of edge from Kuching to Melaka?  800
weight of edge from Tawau to Alor Setar?  1900
weight of edge from Semporna to Ipoh?  null

In and out degree for Kuching is 1 and 2
In and out degree for Penang is 2 and 0
In and out degree for Ipoh is -1 and -1

Neighbours of Kuching : [Tawau, Melaka]

Print Edges :
# Alor Setar : [Alor Setar,Kuching]
# Kuching : [Kuching,Tawau] [Kuching,Melaka]
# Langkawi : [Langkawi,Penang]
# Melaka : [Melaka,Penang]
# Penang :
# Tawau : [Tawau,Alor Setar]
```



GRAPH TRAVERSAL

Graph Traversals

- Also called graph search.
- The process of visiting (checking and/or updating) each vertex in a graph
- Depth-first search and breadth-first search
- Both traversals result in a spanning tree, which can be modeled using a class.

Depth-First Search

- The search can start at any vertex.
- Algorithm:
  1. Start by putting any one of the graph's vertices on top of a **stack**.
  2. Take the top item of the stack and add it to the visited list.
  3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
  4. Keep repeating steps 2 and 3 until the stack is empty.

Applications of the DFS

- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected.
- Otherwise, the graph is not connected.
- Detecting whether there is a path between two vertices.
- Finding a path between two vertices.
- Detecting whether there is a cycle in the graph.

Breadth-First Search

- With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root from left to right, and so on.

Breadth-First Search

- Algorithm:

1. Start by putting any one of the graph's vertices at the back of a **queue**.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

Applications of the BFS

- Quite similar to DFS, but:
  - BFS able to find the path with smallest edges count (not weight/cost/distance) between 2 vertices.
  - It is easy to use BFS to check whether a graph is bipartite. A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set.
  - BFS is inefficient in terms of memory consumption, compared to DFS.

Q&A