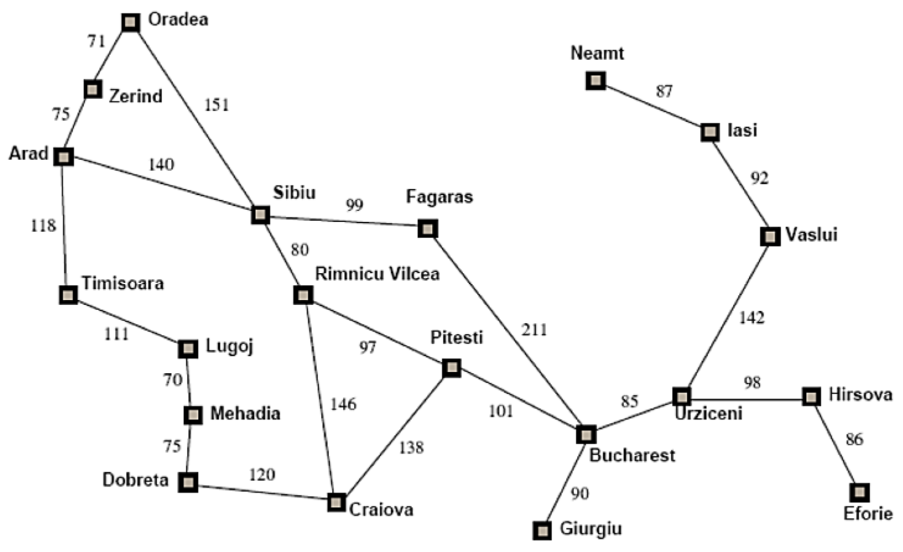
Practical 2(A): Problem Solving and Uninformed Search

Instructor: Shailesh B. Pandey (Everest Engineering College) Course: Intelligent System (BEIT)
Course webpage: sites.google.com/view/eec-is

1. Overview

You are on holiday in Romania, currently in Arad. Flight leaves back home tomorrow from Bucharest. Being a computer engineer, you decide to write a program to find the shortest path from Arad to Bucharest. You try three strategies: (a) BFS, (b) DFS, and (c) best-first search using the map given below. To help with the search you also have the straight-line distance from every city in the map to Bucharest. As you would expect, straight line distance from Bucharest to Bucharest is zero.



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Your task in this lab is to use the code in Java I have provided to solve this problem of finding the path using uninformed search.

2. Graph Representation for the Romania's Map

```
public class Graph {

    // Each node maps to a list of all his neighbours
    HashMap<Node, LinkedList<Node>> adjacencyMap;
    boolean directed;   // For undirected graph this is set to False
}

public class Node {
    int nodeID;             // An integer identifier of a node
    String name;            // This will store the name of the city
    boolean visited;        // True if we have already been to this city
}
```

Task A:

a. Add a method `insertEdge(Node source, Node target)` in the `Graph` class which inserts the `target` node to the adjacency map of `source` node. For our case the graph is undirected, so we insert the `source` node in the adjacency map of the target node as well.
b. Add constructor, getter and setter methods to initialize the variables and return the values.
c. Create a graph based on the Romania map:
    - Create nodes to represent the cities
    - Create edges between cities if there is a road connecting them
d. Add a method `printEdges()` in the `Graph` class which prints the edges for a node.

After you have created the required methods, you should be able to create nodes with the statements:

```
Node n1 = new Node(1, "Arad");
Node n2 = new Node(2, "Zerind");
```

And you create edges using the Graph object:

```
Graph graph = new Graph(false);  // False signifying the graph is undirected
graph.insertEdge(n1,n2);         // Add an edge between n1 to n2 and n2 to n1
```

Hint: The `insertEdge()` method adds n2 to the `adjacentMap` of n1. When the graph is undirected, it also adds n1 to the `adjacencyMap` of n2.

3. Searching the graph

Now you are going to implement Breadth First Search (BFS) and Depth First Search (DFS) in the graph you have created in Task A.

Task B:

a. Implement DFS
   The simplest way to do DFS is using recursive calls. There is no need for an explicit stack data structure. If you perform DFS using loops, you will have to maintain a stack of nodes to be visited.

```
DFS(Node v):
Visit(v)
for(each node w adjacent to v)
      if (w has not yet been visited)
            DFS(w)
```

   Hint: Once you visit a node you set the variable visited as true. All you are doing is recursively calling the same function for all the nodes in the adjacency map.

b. Implement BFS
   This requires you to maintain a Queue data structure. One way to do this is:
   `LinkedList<Node> queue = new LinkedList<Node>();`

   The algorithm is as follows:

```
BFS(Node v):
```

```
Create a queue q
enqueue(v)
while(q is not empty) {
        dequeue(v)
        if(v has not been visited)
                Visit(v)
        for(each node w adjacent to v)
                if(w has not been visited AND not queued)
                        enqueue(w)
}
```

c. Run BFS and DFS using three different start nodes and present the results.

Practical 2(B): Problem Solving and Informed Search

1. Graph Representation for the Romania's Map

You will have to make some additions to the Node class you wrote for BFS and DFS assignment. Best-first search requires that each node has a heuristic (h) score associated with it. This is just the straight-line distance values shown in the figure above.

```java
public class Graph {

    // Each node maps to a list of all his neighbours
    HashMap<Node, LinkedList<Node>> adjacencyMap;
    boolean directed;   // For undirected graph this is set to False
}
public class Node implements Comparable<Node> {
    int nodeID;          // An integer identifier of a node
    String name;        // This will store the name of the city
    boolean visited;    // This will be True if we have already been to this city
    int hscore;         // hscore is the straight line distance heuristic
    Node parent = null; // Parent node of this node
}
```

You may have noticed that the Node class implements the Comparable interface. We need this because we implement Priority Queue. The queue is ordered in descending order of h(n) i.e. node with the smallest h(n) is at the front of the queue and the largest at the end of the queue. In order to do that we implement the `compareTo(Node)` method.

Task A:

a. Implement the compareTo(Node n) method.

2. Best-First Search Algorithm

```
Best-FirstSearch(Node start, Node destination):
        closed-list = []
        open-list = [start]
        while(open-list is not empty) {
                dequeue(n) from open-list
                if(n is destination)
                    return n
                for(each node w adjacent to n) {
                    if(w is not in open-list and closed-list) {
                        set n as w's parent
                        add w to open-list
                    }
                }
                remove n from open-list
                add n to closed-list
        }
```

Task B:

a. Implement the above algorithm.
b. When the algorithm encounters the destination node, it returns the node to the calling location. Implement a `printPath()` method that prints the best path by following the parent chain. Basically, you start with a node n and then print n->parent, then n->parent->parent and so on until you get null. You should have done this in your data structure course.
c. Run the algorithm for three different start nodes and show the results.