Graph Algorithms

Eric Roberts
CS 106B
February 25, 2015

Outline

1. A review the **`graphtypes.h`** and **`graph.h`** interfaces
2. Depth-first and breadth-first search
3. Dijkstra’s shortest-path algorithm
4. Kruskal’s minimum-spanning-tree algorithm

The **Node** and **Arc** Structures

```
struct Node;      /* Forward references to these two types so  */
struct Arc;       /* that the C++ compiler can recognize them. */

/*
 * Type: Node
 * -----------
 * This type represents an individual node and consists of the
 * name of the node and the set of arcs from this node.
 */

struct Node {
   string name;
   Set<Arc *> arcs;
};

/*
 * Type: Arc
 * ----------
 * This type represents an individual arc and consists of pointers
 * to the endpoints, along with the cost of traversing the arc.
 */

struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
```

Entries in the `graph.h` Interface

```
template <typename NodeType,typename ArcType>
class Graph {
public:

   Graph();
   ~Graph();

   void clear();

   NodeType *addNode(string name);
   NodeType *addNode(NodeType *node);

   ArcType *addArc(string s1, string s2);
   ArcType *addArc(NodeType *n1, NodeType *n2);
   ArcType *addArc(ArcType *arc);

   bool isConnected(NodeType *n1, NodeType *n2);
   bool isConnected(string s1, string s2);

   NodeType *getNode(string name);

   Set<NodeType *> & getNodeSet();
   Set<ArcType *> & getArcSet();
   Set<ArcType *> & getArcSet(NodeType *node);

};
```

Depth-First Search

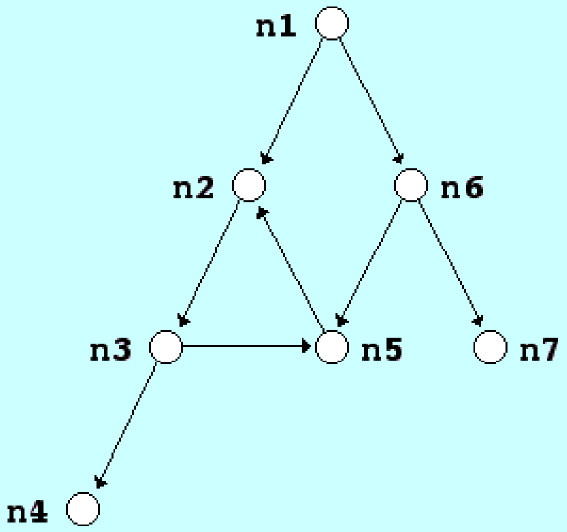
- The traversal strategy of ***depth-first search*** (or ***DFS*** for short) recursively processes the graph, following each branch, visiting nodes as it goes, until every node is visited.
- The depth-first search algorithm requires some structure to keep track of nodes that have already been visited. Common strategies are to include a **`visited`** flag in each node or to pass a set of visited nodes, as shown in the following code:

```
void depthFirstSearch(Node *start) {
   Set<Node *> visited;
   visitUsingDFS(start, visited);
}

void visitUsingDFS(Node *start, Set<Node *> & visited) {
   if (visited.contains(start)) return;
   visit(start);
   visited.add(start);
   for (Arc *ap : start->arcs) {
      visitUsingDFS(ap->finish, visited);
   }
}
```

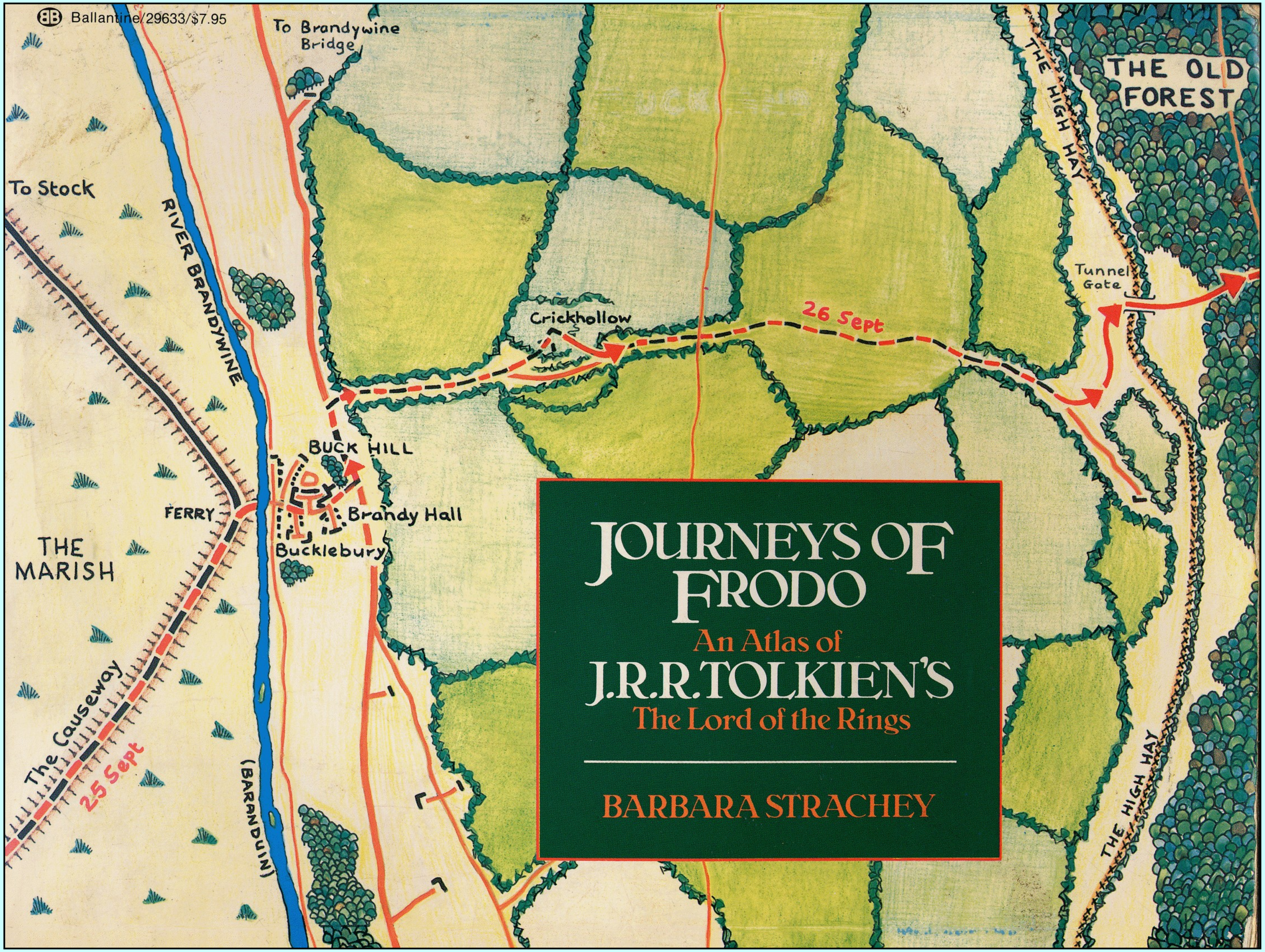
Breadth-First Search

- The traversal strategy of breadth-first search (which you used on Assignment #2) proceeds outward from the starting node, visiting the start node, then all nodes one hop away, and so on.
- For example, consider the graph:

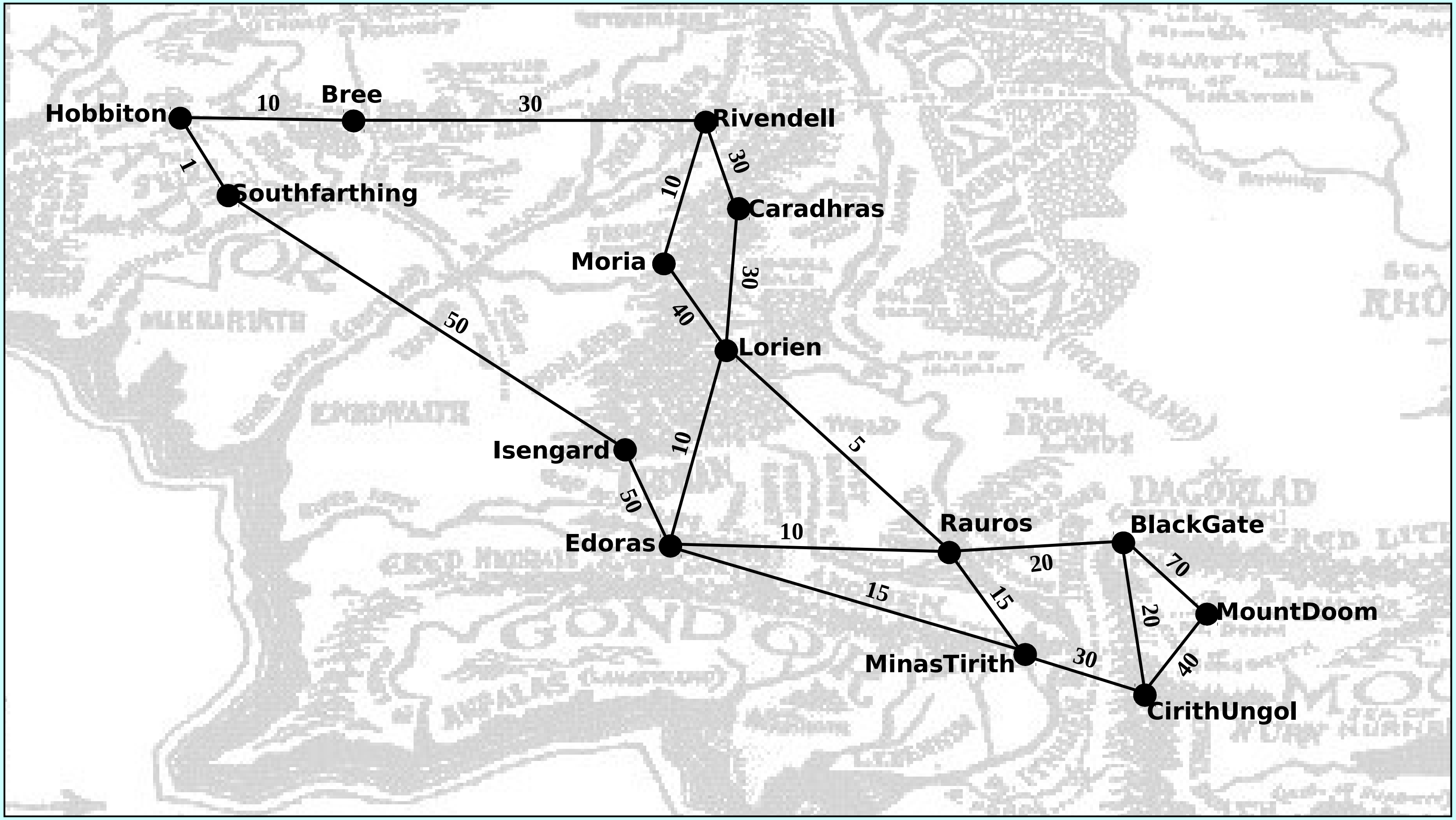


- Breadth-first search begins at the start node (**n1**), then does the one-hops (**n2** and **n6**), then the two hops (**n3**, **n5**, and **n7**) and finally the three hops (**n4**).

Frodo’s Journey

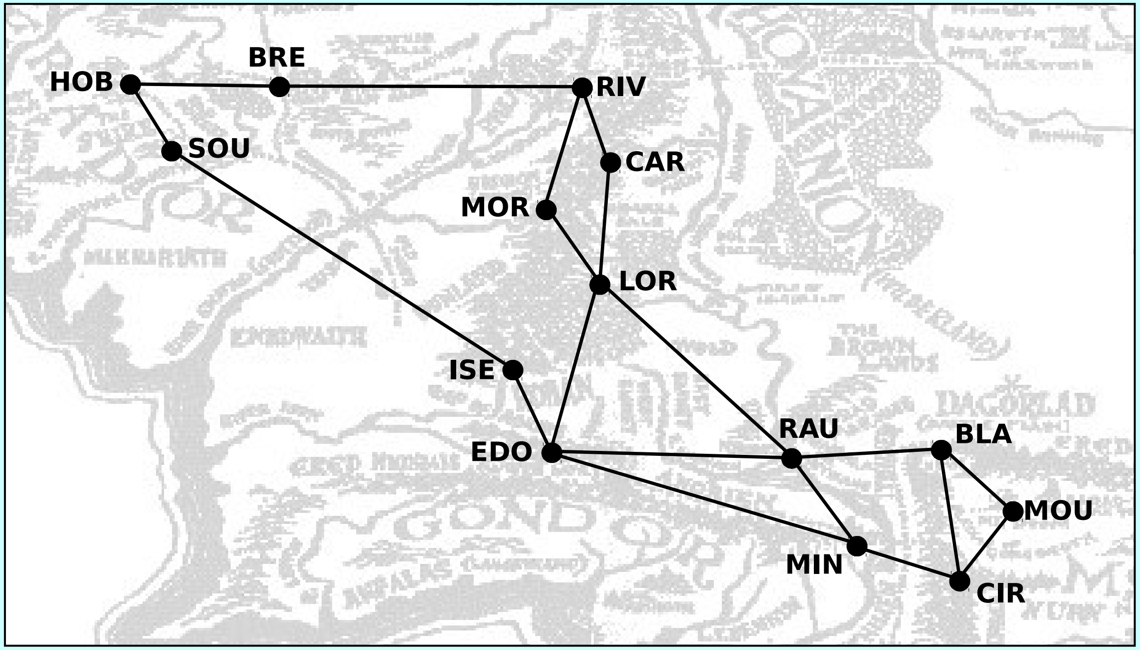


The Middle Earth Graph



Exercise: Depth-First Search

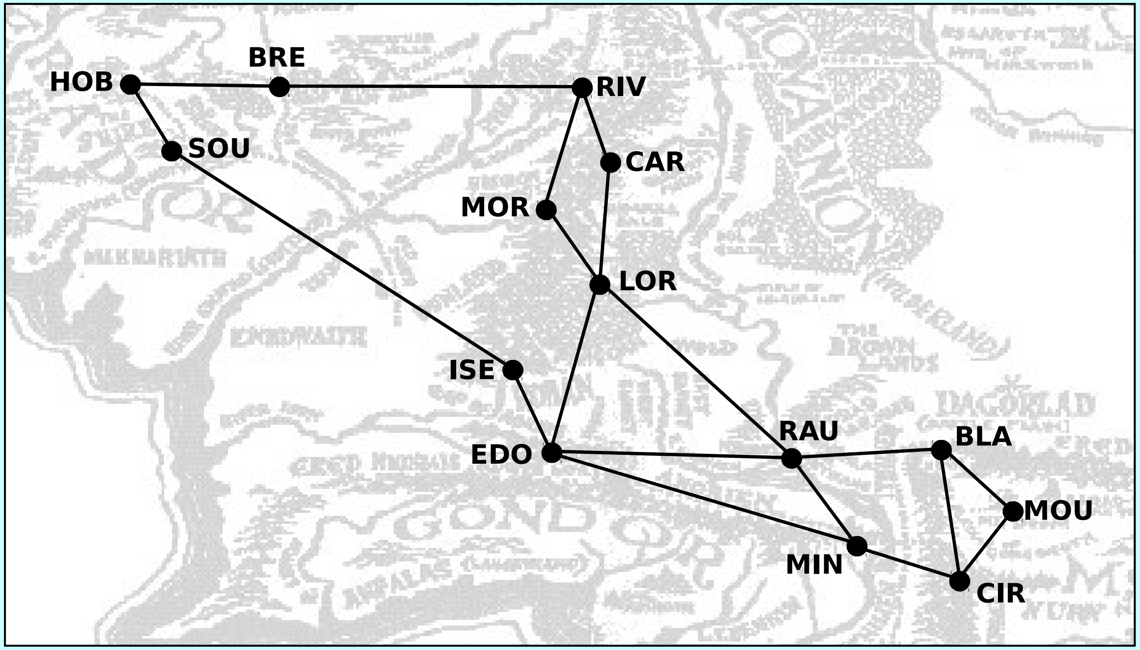
Construct a depth-first search starting from Hobbiton (**HOB**):



Visiting node **HOB**
Visiting node **BRE**
Visiting node **RIV**
Visiting node **CAR**
Visiting node **LOR**
Visiting node **EDO**
Visiting node **ISE**
Visiting node **SOU**
Visiting node **MIN**
Visiting node **CIR**
Visiting node **BLA**
Visiting node **MOU**
Visiting node **RAU**
Visiting node **MOR**

Exercise: Breadth-First Search

Construct a breadth-first search starting from Isengard (**ISE**):



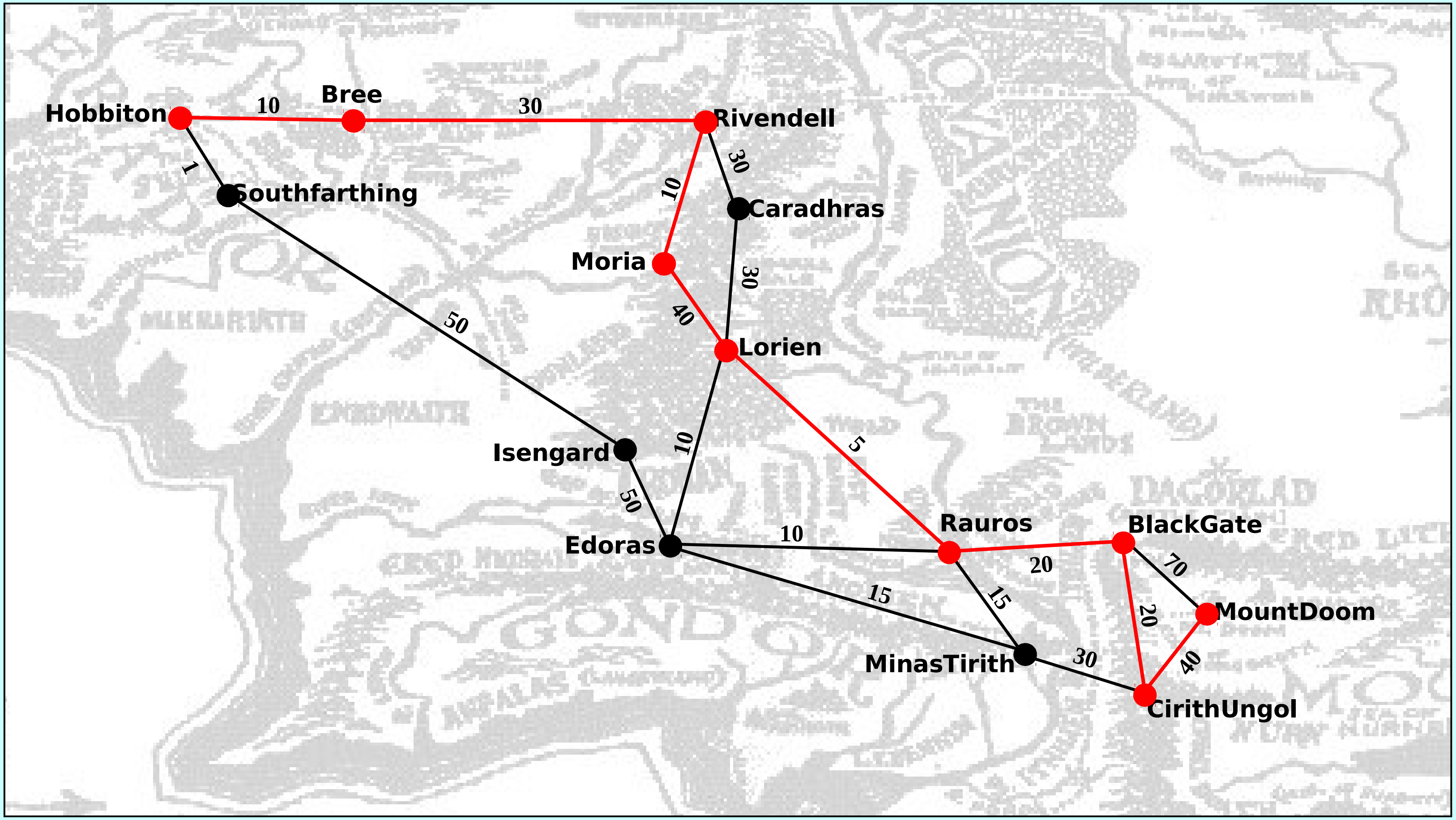
Visiting node **ISE**
Visiting node **EDO**
Visiting node **SOU**
Visiting node **LOR**
Visiting node **MIN**
Visiting node **RAU**
Visiting node **HOB**
Visiting node **CAR**
Visiting node **MOR**
Visiting node **CIR**
Visiting node **BLA**
Visiting node **BRE**
Visiting node **RIV**
Visiting node **MOU**

Queue: **~~ISE~~ ~~EDO~~ ~~SOU~~ ~~LOR~~ ~~MIN~~ ~~RAU~~ ~~HOB~~ ~~CAR~~ ~~MOR~~ ~~CIR~~ ~~BLA~~ ~~BRE~~ ~~RIV~~ ~~MOU~~**

Dijkstra's Algorithm

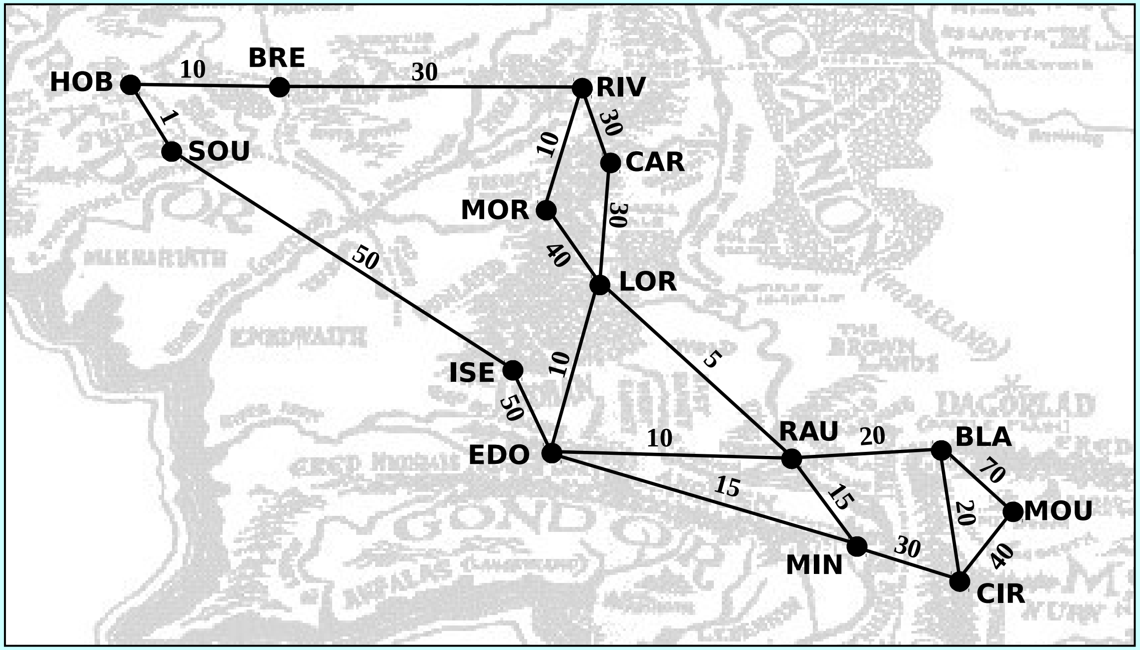
- One of the most useful algorithms for computing the shortest paths in a graph was developed by Edsger W. Dijkstra in 1959.
- The strategy is similar to the breadth-first search algorithm you used to implement the word-ladder program in Assignment #2. The major difference are:
  - The queue used to hold the paths delivers items in increasing order of total cost rather than in the traditional first-in/first-out order. Such queues are called ***priority queues***.
  - The algorithm keeps track of all nodes to which the total distance has already been fixed. Distances are fixed whenever you dequeue a path from the priority queue.

Shortest Path



Exercise: Dijkstra's Algorithm

Find the shortest path from Hobbiton (**HOB**) to Lorien (**LOR**):



HOB (0)

HOB→SOU (1)

HOB→BRE (10)

HOB→BRE→RIV (40)

HOB→BRE→RIV→MOR (50)

HOB→SOU→ISE (51)

HOB→BRE→RIV→CAR (70)

HOB→BRE→RIV→MOR→LOR (90)

HOB→BRE→RIV→CAR→LOR (100)

HOB→SOU→ISE→EDO (101)

Kruskal’s Algorithm

- In many cases, finding the shortest path is not as important as as minimizing the cost of a network as a whole. A set of arcs that connects every node in a graph at the smallest possible cost is called a ***minimum spanning tree***.
- The following algorithm for finding a minimum spanning tree was developed by Joseph Kruskal in 1956:
  - Start with a new empty graph with the same nodes as the original one but an empty set of arcs.
  - Sort all the arcs in the graph in order of increasing cost.
  - Go through the arcs in order and add each one to the new graph if the endpoints of that arc are not already connected by a path.
- This process can be made more efficient by maintaining sets of nodes in the new graph, as described on the next slide.

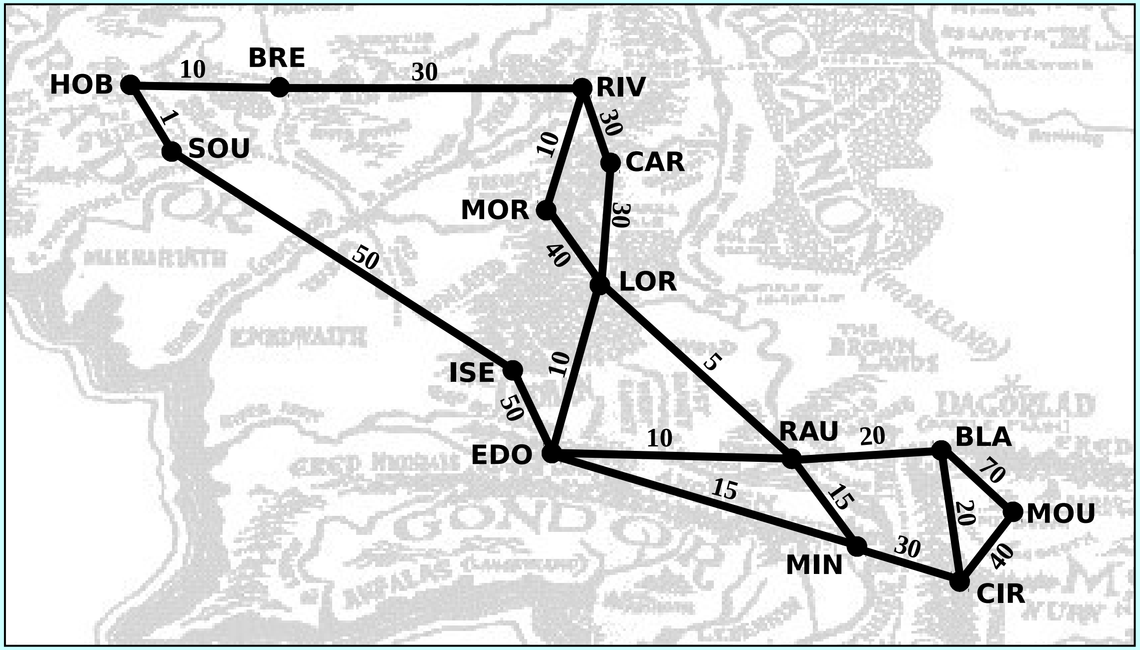
Combining Sets in Kruskal's Algorithm

- Implementing Kruskal's algorithm requires you need to build a new graph containing the spanning tree. As you do, you will generate sets of disconnected trees, which are called ***forests***.
- At the beginning of the process, every node is the graph is in a set all by itself. After that, you combine nodes together by choosing an arc and then taking one of the following actions:

1. *The nodes at the endpoints of the arc are in different sets.* In this case, you include the edge in the spanning tree and combine the sets together.
2. *The endpoints are in the same set.* In this case, there is already a path between these two nodes, which means that you don't need this arc.

Exercise: Minimum Spanning Tree

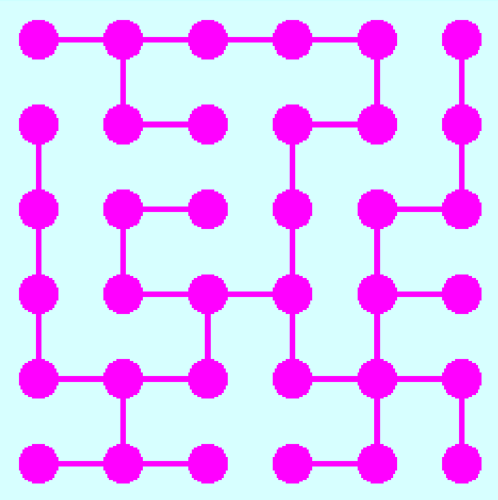
Apply Kruskal’s algorithm to find a minimum spanning tree:



1: **HOB → SOU**
5: **LOR → RAU**
10: **BRE → HOB**
10: **EDO → LOR**
10: **EDO → RAU**
10: **MOR → RIV**
15: **EDO → MIN**
15: **MIN → RAU**
20: **BLA → CIR**
20: **BLA → RAU**
30: **BRE → RIV**
30: **CAR → LOR**
30: **CAR → RIV**
30: **CIR → MIN**
40: **CIR → MOU**
40: **LOR → MOR**
50: **EDO → ISE**
50: **ISE → SOU**
70: **BLA → MOU**

An Application of Kruskal’s Algorithm

- Suppose that you have a graph that looks like this:



- What would happen if you applied Kruskal’s algorithm for finding a minimum spanning tree, assuming that you choose the arcs in a random order?

The End