CS61B, Spring 2003 **Discussion #13** **Amir Kamil**
UC Berkeley **4/14/03**

Topics: Graph Algorithms

1 Graph Algorithms

There are many algorithms that can be applied to graphs. Many of these are actually used in the real world, such as Dijkstra's algorithm to find shortest paths. We will discuss a few here.

1.1 Topological Sort

Graphs are sometimes used to represent "before and after" relationships. For example, you need to think through a design for a program before you start coding. These two steps can be represented as vertices, and the relationship between them as a directed edge from the first to the second.

On such graphs, it is useful to determine which steps must come before others. The *topological sort* algorithm computes an ordering on a graph such that if vertex α is earlier than vertex β in the ordering, there is no path from β to α. In other words, you cannot get from a vertex later in the ordering to a vertex earlier in the ordering. Of course, topological sort works only on directed acyclic graphs.

The simplest topological sort algorithm is to repeatedly remove vertices with in-degree of 0 from the graph. The edges belonging to the vertex are also removed, reducing the in-degree of adjacent vertices. This is done until the graph is empty, or until no vertex without incoming edges exists, in which case the sort fails.

1.2 Dijkstra's Algorithm

Graphs are very often used to represent distances between locations, and an obvious necessity is to find shortest paths between these locations. *Dijkstra's algorithm* can be used to compute shortest paths from a starting vertex to each other vertex. We will discuss first how to compute shortest distances.

Dijkstra's algorithm is in a class called *greedy algorithms*. Greedy algorithms work by choosing a local optimum value at each step. In the case of shortest paths, always choosing the local optimum results in the global optimum as well.

A starting vertex is chosen, from which all distances will be computed. Each vertex in the graph is assigned a distance value, initially ∞ for all but the starting vertex, which is given a distance of 0. Then the vertex with the minimum distance is examined, and any vertices adjacent to the current vertex have their distances updated to the current vertex's distance plus the edge between the current and the adjacent vertex, if this update will reduce that distance value. Then the vertex with the next minimum distance is examined, and so on.

Computation of the actual path is not much harder. Each vertex keeps track of a *back edge*. When a vertex's distance is updated, the back edge is set to be an edge between that vertex and the vertex that caused the update. Then to get the path between a vertex α and the starting vertex, follow the back edges from α back to the start. The path from the start to α is the reverse of this.

1.3 Minimum Spanning Trees

Another useful question about weighted graphs is to find which edges in the graph must remain such that the graph is connected, but the total amount of weight in the remaining edges is minimized. Such a result is called a *minimum spanning tree* (MST). The two algorithms to compute MSTs are *Kruskal's algorithm* and *Prim's algorithm*.

1.3.1 Kruskal's Algorithm

Kruskal's algorithm is straightforward. The vertices are separated into individual sets, and the edges ordered by weight. Then each edge is examined in order, and if the two corresponding vertices are in different sets,

```
Vector topologicalSort(Graph g) {
  Vector res = new Vector();
  while (!g.isEmpty()) {
    Vertex v = findNextVertex(g);
    if (v == null) {
      return null; // sort failed
    }
    g.remove(v);
    res.add(v);
  }
  return res;
}
```

Figure 1: Java algorithm for topological sort. The method to find a vertex with no incoming edges is left as an exercise.

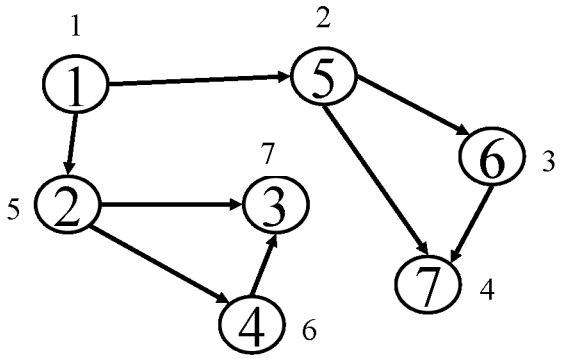
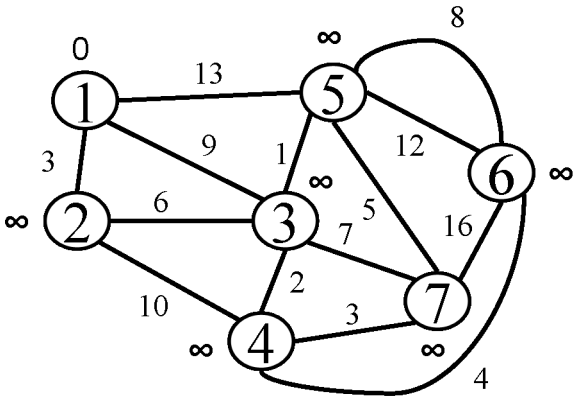


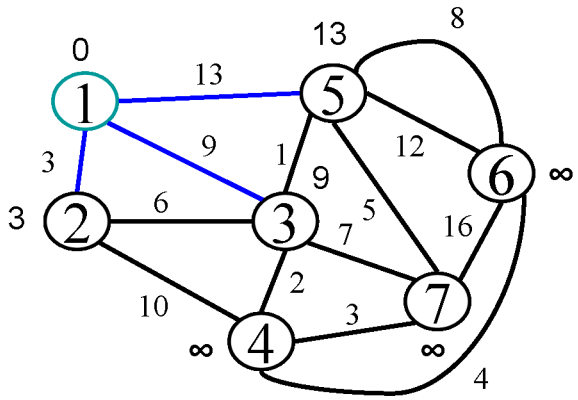
Figure 2: A topological sorting of a graph.

```
Edge[] Dijkstra(Graph g, Vertex s):
  Construct an array D[] with one cell for each vertex in g;
  Construct an array of edges E[] with one cell for each vertex in g;
  Set D[s] = 0 where s is the starting vertex;
  Set D[u] = (really big) for all vertices u except the starting  vertex;
  Construct a min priority queue P containing each node n, with D[n] as  its priority;
  while P is not empty do:
    v = P.removeMinElement();
    for each vertex z still in P adjacent to v do:
      if D[v] + w(v, z) < D[z] then:
        D[z] = D[v] + w(v, z);
        E[z] = edge(v, z);
        Reorder P according to the new priorities D[];
      fi;
    od;
  od
  return E;
```

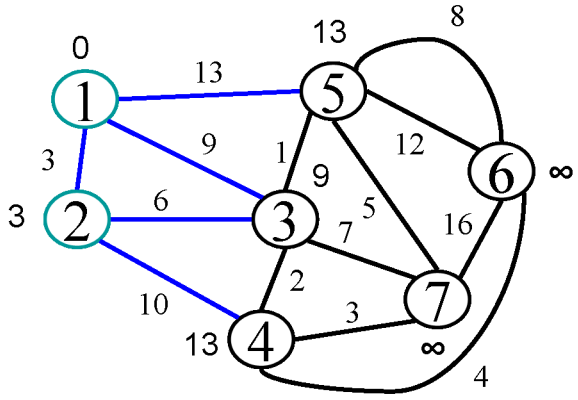
Figure 3: Dijkstra's algorithm. w(v, z) above refers to the weight of the edge connecting v and z.



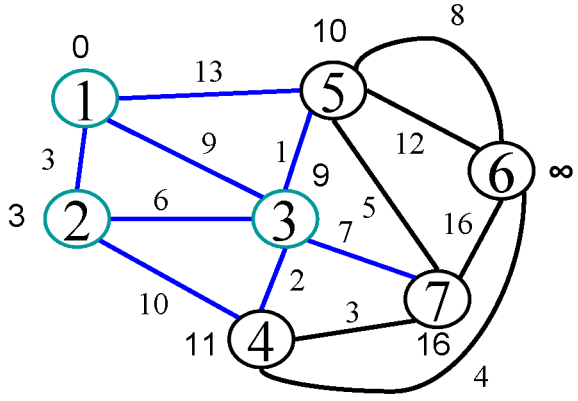
P.Q.
1 (0)
2 (∞)
3 (∞)
4 (∞)
5 (∞)
6 (∞)
7 (∞)



P.Q.
x1 (0)
2 (3)
3 (9)
5 (13)
4 (∞)
6 (∞)
7 (∞)



P.Q.
x1 (0)
x2 (3)
3 (9)
5 (13)
4 (13)
6 (∞)
7 (∞)



P.Q.
x1 (0)
x2 (3)
x3 (9)
5 (10)
4 (11)
7 (16)
6 (∞)

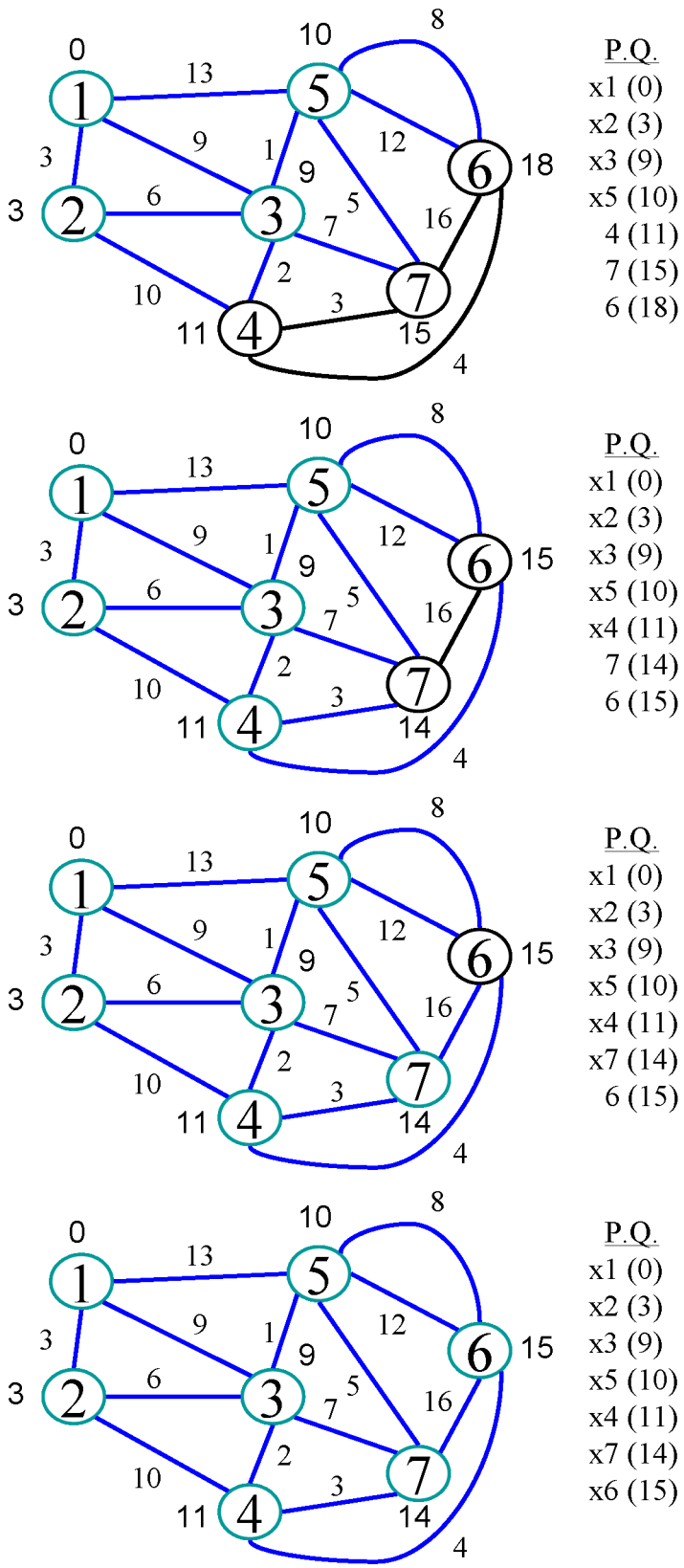


Figure 4: Dijkstra's algorithm applied to a graph, starting at node 1.

```
Tree Kruskal(Graph g):
  For each vertex in g, create a set with that vertex in it;
  Construct a min priority queue P containing all edges (u, v), with their weights as priorities;
  Construct an empty tree T;
  while T does not contain all the vertices in g do:
    E = P.removeMinElement().
    if E.u and E.v are not in the same set then:
      Add E to T;
      Merge the sets containing E.u and E.v;
    fi;
  od;
  return T;
```

Figure 5: Kruskal's algorithm for computing MSTs.

```
Tree Prim(Graph g, Vertex s):
  Construct an empty tree T;
  Construct an array D[] with one cell for each vertex in g;
  Construct an array of edges E[] with one cell for each vertex in g;
  Set D[s] = 0 where s is the starting vertex;
  Set D[u] = (really big) for all vertices u except the starting  vertex;
  Construct a min priority queue P containing each node n, with D[n] as  its priority;
  while P is not empty do:
    v = P.removeMinElement();
    add v and E[v] to T;
    for each vertex z still in P adjacent to v do:
      if w(v, z) < D[z] then:
        D[z] = w(v, z);
        E[z] = edge(v, z);
        Reorder P according to the new priorities D[];
      fi;
    od;
  od
  return T;
```

Figure 6: Prim's algorithm. w(v, z) above refers to the weight of the edge connecting v and z.

the two sets are combined and the edge added to the result tree. Since the algorithm always chooses the minimum-weight edge between two portions of the graph, the result is a minimum spanning tree.

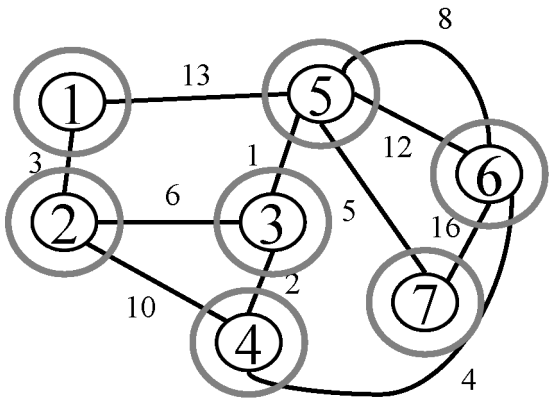
1.3.2 Prim's Algorithm

Prim's algorithm is another greedy algorithm, this time to find minimum spanning trees. Instead of always choosing the global optimal edge like Kruskal's algorithm, it always chooses the local optimal edge. It turns out that this will also result in a minimum spanning tree, though not necessarily the same one.

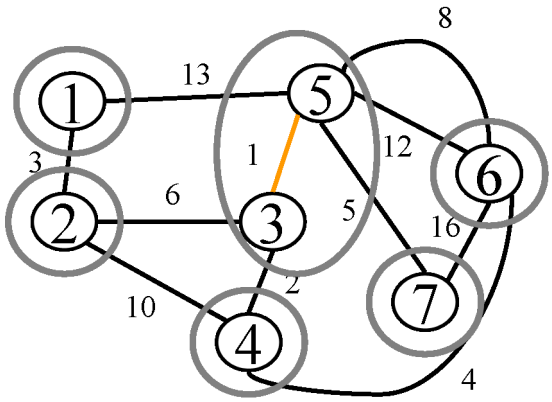
The actual algorithm is almost identical to Dijkstra's algorithm. The only difference is that distances are updated to only the weight of an edge between vertices, not the sum of the weight and the previous vertex's distance. The back edges compose the minimum spanning tree.

1.4 Cycle Detection

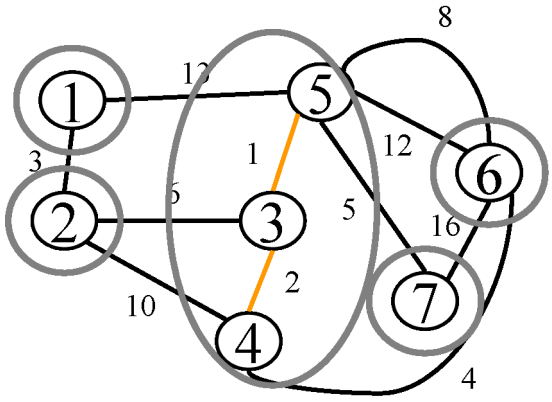
Cycle detection on a graph is a bit different than on a tree due to the fact that a graph node can have multiple parents. On a tree, the algorithm for detecting a cycle is to do a depth first search, marking nodes



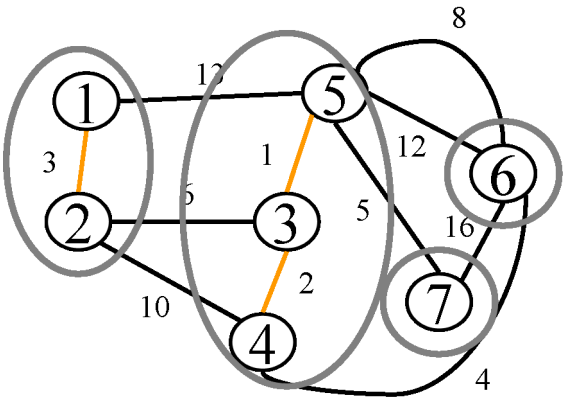
Edges
3-5 (1)
3-4 (2)
1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)



Edges
x3-5 (1)
3-4 (2)
1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)



Edges
x3-5 (1)
x3-4 (2)
1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)



Edges
x3-5 (1)
x3-4 (2)
x1-2 (3)
4-6 (4)
5-7 (5)
2-3 (6)
5-6 (8)
2-4 (10)
5-6 (12)
1-5 (13)
6-7 (16)

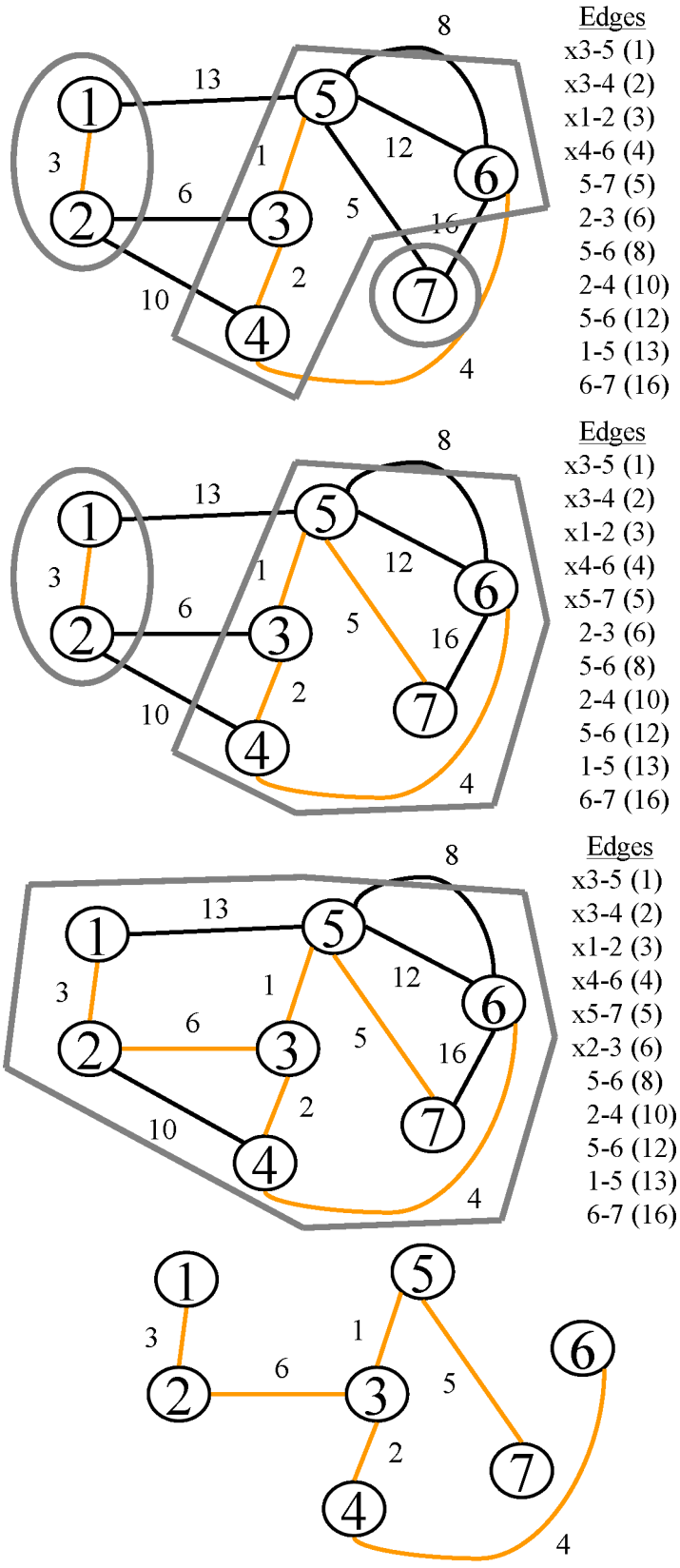
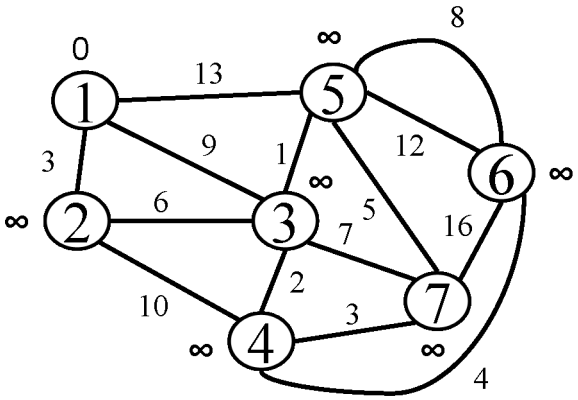
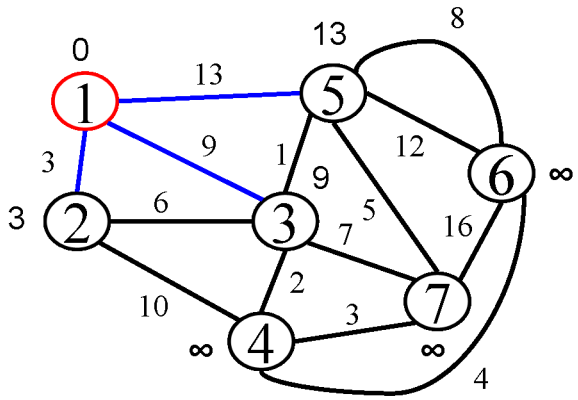


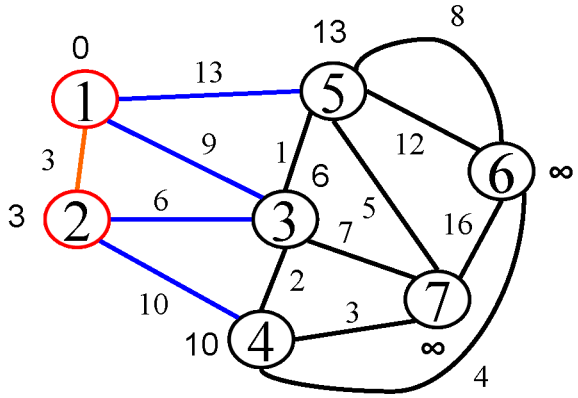
Figure 7: Kruskal's algorithm applied to a graph.



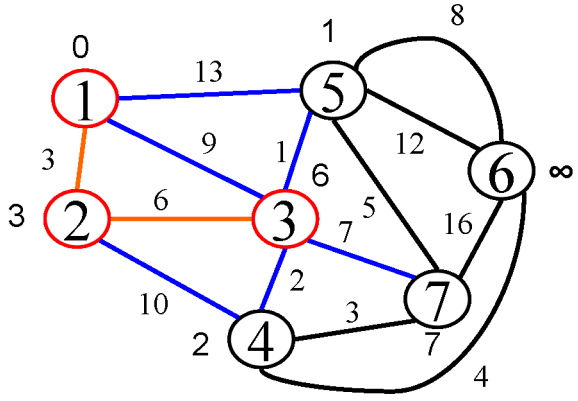
P.Q.
1 (0)
2 (∞)
3 (∞)
4 (∞)
5 (∞)
6 (∞)
7 (∞)



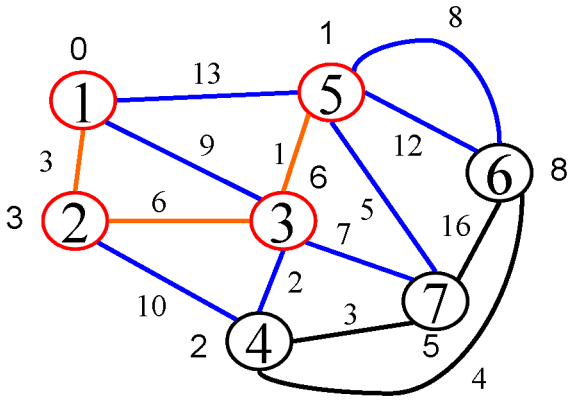
P.Q.
x1 (0)
2 (3)
3 (9)
5 (13)
4 (∞)
6 (∞)
7 (∞)



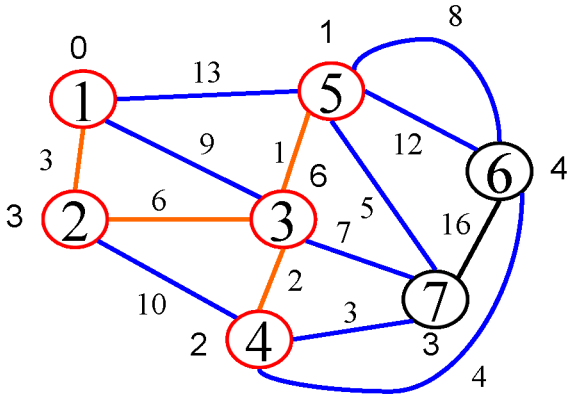
P.Q.
x1 (0)
x2 (3)
3 (6)
4 (10)
5 (13)
6 (∞)
7 (∞)



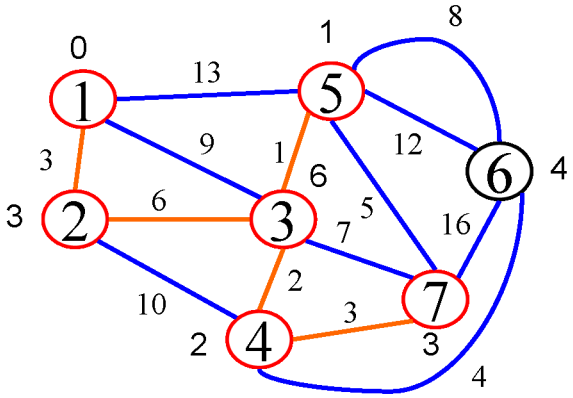
P.Q.
x1 (0)
x2 (3)
x3 (6)
5 (1)
4 (2)
7 (7)
6 (∞)



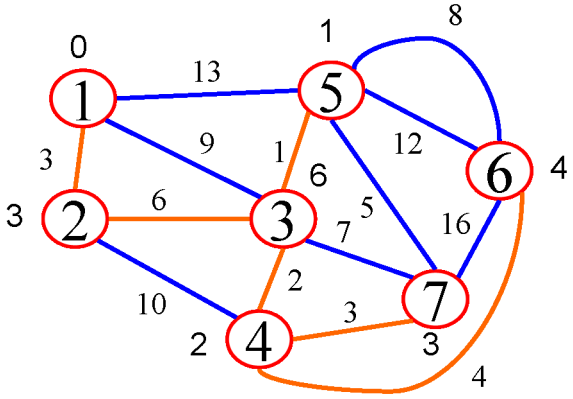
P.Q.
x1 (0)
x2 (3)
x3 (6)
x5 (1)
4 (2)
7 (5)
6 (8)



P.Q.
x1 (0)
x2 (3)
x3 (6)
x5 (1)
x4 (2)
7 (3)
6 (4)



P.Q.
x1 (0)
x2 (3)
x3 (6)
x5 (1)
x4 (2)
x7 (3)
6 (4)



P.Q.
x1 (0)
x2 (3)
x3 (6)
x5 (1)
x4 (2)
x7 (3)
x6 (4)

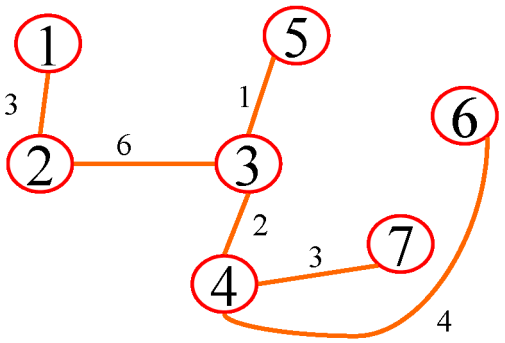


Figure 8: Prim's algorithm applied to a graph.

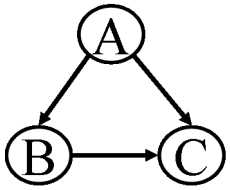


Figure 9: An acyclic graph on which the tree cycle detection algorithm would fail.

as they are encountered. If a previously marked node is seen again, then a cycle exists. This won't work on a graph. The graph in figure 9 will be falsely reported to have a cycle, since node C will be seen twice in a DFS starting at node A.

The cycle detection algorithm for trees can easily be modified to work for graphs. The key is that in a DFS of an acyclic graph, a node whose descendants have all been visited can be seen again without implying a cycle. However, if a node is seen a second time before all of its descendants have been visited, then there must be a cycle. Can you see why this is? Suppose there is a cycle containing node A. Then this means that A must be reachable from one of its descendants. So when the DFS is visiting that descendant, it will see A again, before it has finished visiting all of A's descendants. So there is a cycle.

In order to detect cycles, we use a modified depth first search called a *colored DFS*. All nodes are initially marked *white*. When a node is encountered, it is marked *grey*, and when its descendants are completely visited, it is marked *black*. If a grey node is ever encountered, then there is a cycle.

```
boolean containsCycle(Graph g):
  for each vertex v in g do:
    v.mark = WHITE;
  od;
  for each vertex v in g do:
    if v.mark == WHITE then:
      if visit(g, v) then:
        return TRUE;
      fi;
    fi;
  od;
  return FALSE;

boolean visit(Graph g, Vertex v):
  v.mark = GREY;
  for each edge (v, u) in g do:
    if u.mark == GREY then:
      return TRUE;
    else if u.mark == WHITE then:
      if visit(g, u) then:
        return TRUE;
      fi;
    fi;
  od;
  v.mark = BLACK;
  return FALSE;
```

Figure 10: Cycle detection algorithm.