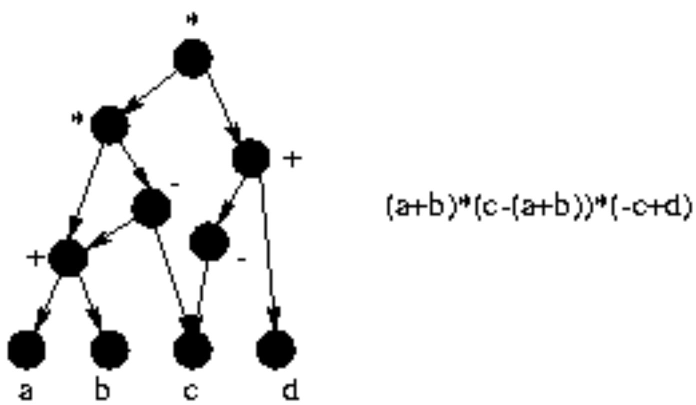
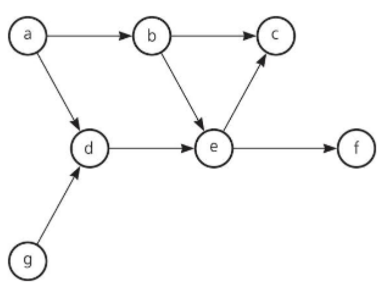
CSCI 3110 Graph (2)

A. Topological order – linear ordering of the vertices in a digraph(without cycles) in which vertex x precedes vertex y if there is a directed edge from x to y in the graph.

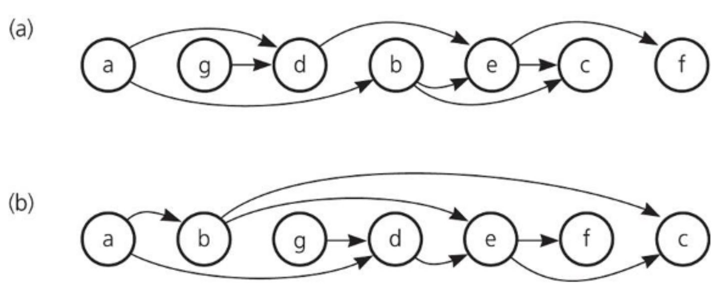
Application: compiler expression evaluation



B. Topological sort – a technique that arrange the vertices into topological order according to an acyclic digraph. A *topological sort* of a DAG *G* is a linear ordering of all its vertices such that if *G* contains an arc *(u, v),* then *u* appears before *v* in the order. <u>Many DAGs have more than one possible topological order.</u>



A directed graph without cycles



The topological orders (a) a, g, d, b, e, c, f and (b) a, b, g, d, e, f, c

Application of topological sort: Given Pre-requisites map of the department, what is a correct sequence of courses to take that satisfy the pre-requisite requirements

Solution 1: Arranges the vertices in theGraph into a topological order and places them in list aList.

1. Find a vertex that has no successor
2. Add the vertex to the beginning of a list
3. Remove that vertex from the graph, as well as all edges that lead from it
4. Repeat the previous steps until the graph is empty
   When the loop ends, the list of vertices will be in topological order

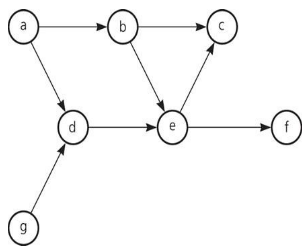
```
TopSort(in theGraph:Graph, out aList:List) {
        n=number of vertices in theGraph
        for (step = 1 through n)        {
                Select a vertex v that has no successors
                aList.Insert(1, v)
                Delete from theGraph vertex v and its edges
        }
}
```

Solution 2: DFS topological sorting algorithm

1. A modification of the iterative DFS algorithm
2. Push all vertices that have no predecessor onto a stack
3. Each time you pop a vertex from the stack, add it to the beginning of a list of vertices
4. When the traversal ends, the list of vertices will be in topological order

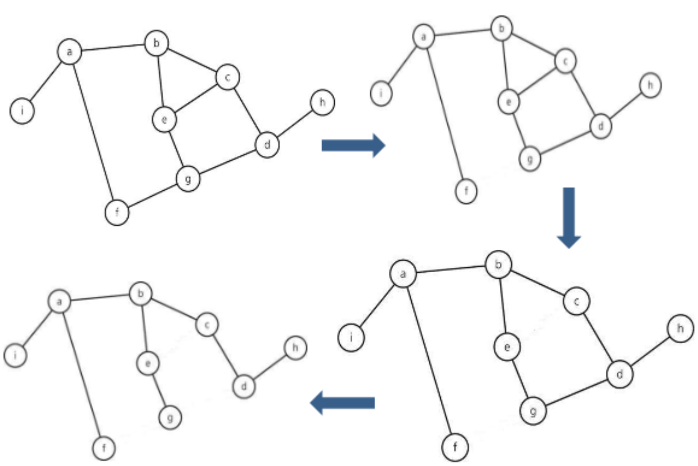
```
TopSort(in theGraph:Graph, out aList:list) {
     s.createStack()

     for (all vertices v in the graph)
        if (v has no predecessors)      {
                s.push(v)
                mark v as visited
        }

     while (!s.isEmpty()) {
        if (no unvisited vertices are adjacent to the vertex on the top of the stack){
                s.pop()
                aList.insert(1, v)
        }
        else    {
                select an unvisited vertex u adjacent to the vertex on top of the stack
                s.push(u)
                mark u as visited
        } // end if
    } // end while
}
```

A trace of topSort2



Action	Stack s (bottom to top)	List aList (beginning to end)
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	c
Push c	a g d e c	c
Pop c, add c to aList	a g d e	f c
Push f	a g d e f	e f c
Pop f, add f to aList	a g d e	d e f c
Pop e, add e to aList	a g d	g d e f c
Pop d, add d to aList	a g	g d e f c
Pop g, add g to aList	a	b g d e f c
Push b	a b	a b g d e f c
Pop b, add b to aList	a	
Pop a, add a to aList	(empty)	

C. Spanning Trees – a spanning tree of a connected undirected graph G is a subgraph of G that contains all of G's vertices and enough of its edges to form a **tree (connected, acyclic)**



- **Is spanning tree unique given an undirected graph?**
- **Would a spanning tree with N vertices have less than N-1 edges?**
- **Would a spanning tree with N vertices have more than N-1 edges?**

Detecting a cycle in an undirected connected graph

- A connected undirected graph that has n vertices must have at least $n - 1$ edges
- A connected undirected graph that has n vertices and exactly $n - 1$ edges cannot contain a cycle
- A connected undirected graph that has n vertices and more than $n - 1$ edges must contain at least one cycle

A graph may have many spanning trees; for instance the complete graph on four vertices

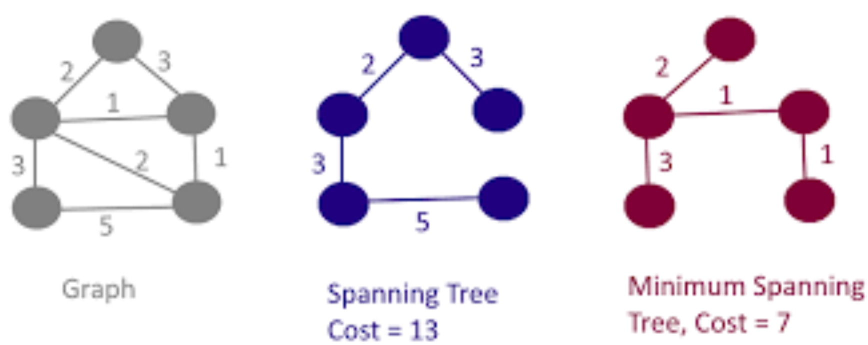
```
o---o
|\ /|
| x |
|/ \|
o---o
```

has sixteen spanning trees:

```
o---o    o---o    o   o    o---o
|   |    |        |   |        |
|   |    |        |   |        |
|   |    |        |   |        |
o   o    o---o    o---o    o---o

o---o    o   o    o   o    o   o
 \ /     |\ /      \ /      \ /|
  x      | x        x        x |
 / \     |/ \      / \      / \|
o   o    o   o    o---o    o   o

o   o    o---o    o   o    o---o
|\  |       /     |  /|     \
| \ |      /      | / |      \
|  \|     /       |/  |       \
o   o    o---o    o   o    o---o

o---o    o   o    o   o    o---o
|\       |  /      \  |       /|
| \      | /        \ |      / |
|  \     |/          \|     /  |
o   o    o---o    o---o    o   o
```

- **DFS spanning tree** – spanning tree created by executing a depth-first search of a graph's vertices.

- **BFS spanning tree** – spanning tree created by executing a breadth-first search of a graph's vertices.



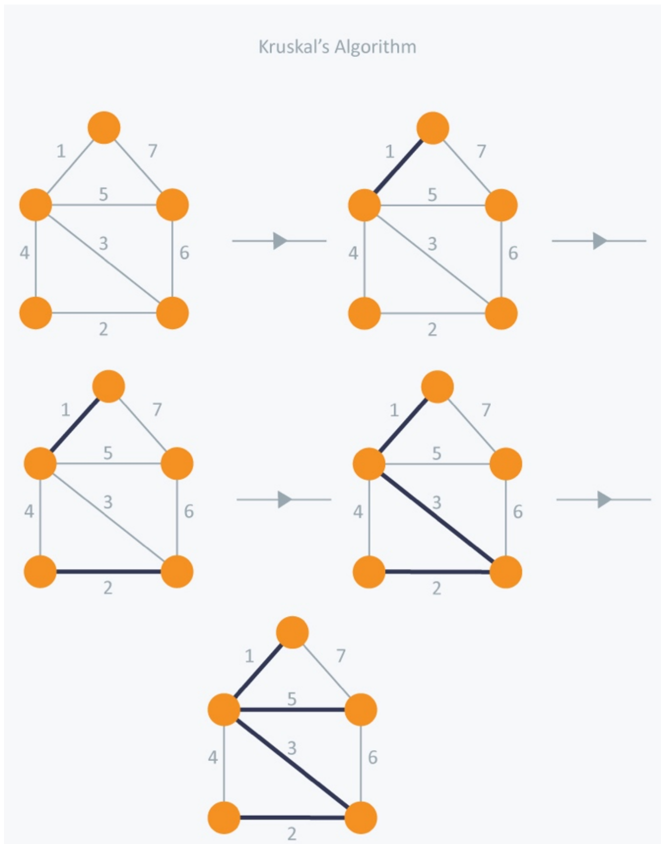
- **Minimum spanning tree(MSP)** – spanning tree of a weighted, connected, undirected graph which has minimal sum of edge-weights.

Application of MSP: When constructing a networks (of phone lines, or cable lines, …), how to design the network lines such that the smallest total amount of wire is used.

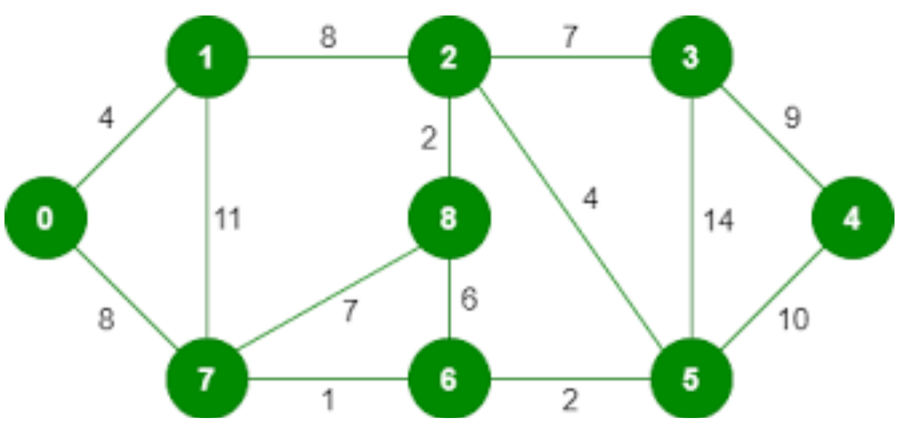
<u>Solution 1:</u> Kruskal's algorithm: O(ElogE), E is the number of edges

// main idea: at each step, add an edge to the intermediate spanning tree such that it does
// not form a cycle and it will result in a MST (with least total sum of edge weights)

    sort the edges of G in increasing order by weights
    keep a subgraph S of G, initially empty
    for each edge e in sorted list of edges
        if the endpoints of e are disconnected in S
            add e to S
    return S

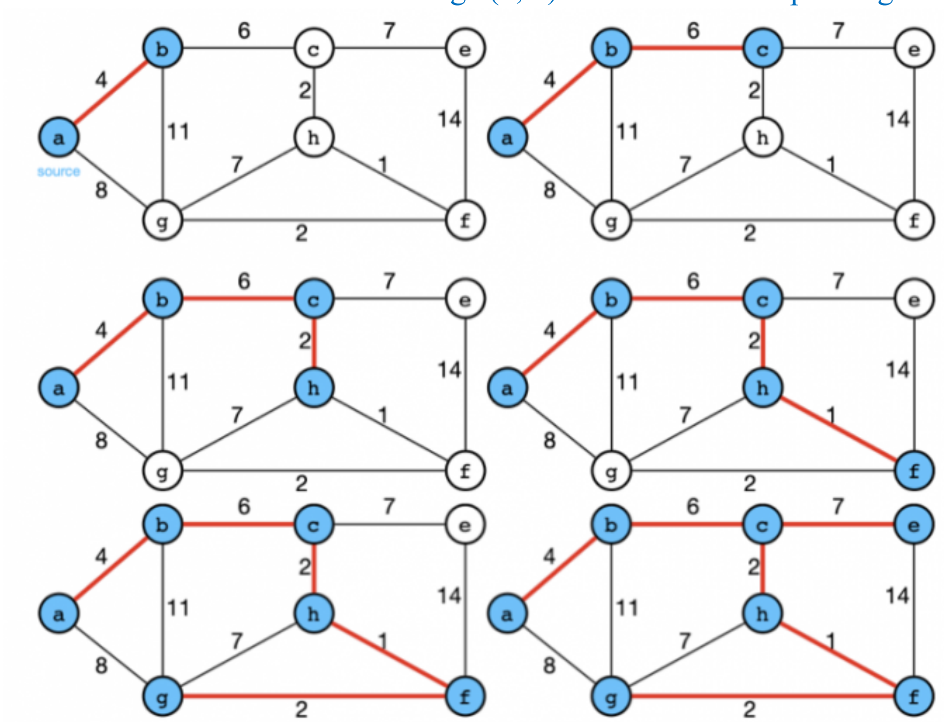


<u>Practice Problem:</u>

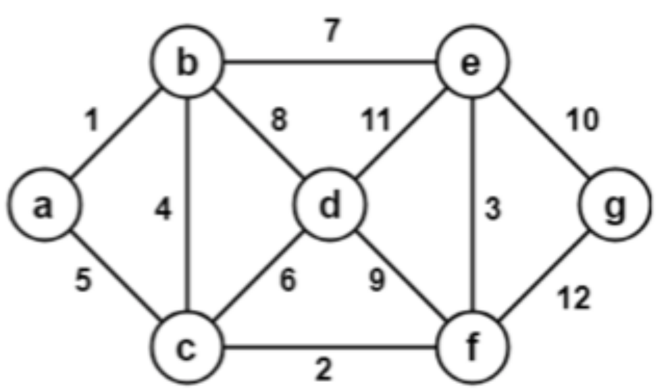


<u>Solution 2:</u> Prim's algorithm:
// the edges currently included always form a single tree

> **PrimsAlgorithm**(in v:Vertex)
> Mark vertex v as visited and include it in the minimum spanning tree
> while (there are unvisited vertices)
>> find the least-cost edge(v, u) from a visited vertex v to
>> some unvisited vertex u
>> Mark u as visited
>> Add the vertex u and the edge (v, u) to the minimum spanning tree



<u>Practice Problem:</u>



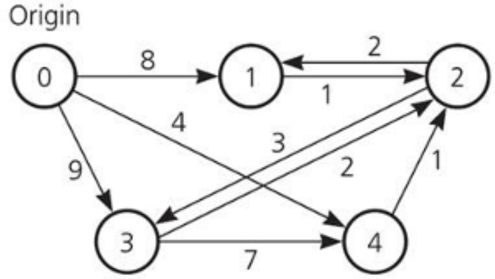
Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\|+\|E\|)\log\|V\|) = O(\|E\|\log\|V\|)$
Fibonacci heap and adjacency list	$O(\|E\|+\|V\|\log\|V\|)$

D. Shortest path – between two vertices in a weighted graph is the path that has the smallest sum of its edge weights.

Dijkstra's shortest path algorithm finds the shortest path from one vertex to all other vertices in a weighted, directed graph (all edge weights are nonnegative).

```
ShortestPath (in theGraph:Graph, in weight:WeightArray) {
    create a set vertexSet that contains only vertex 0 // initialization
    n = number of vertices in theGraph
    for (v=0 through n-1)
            weight[v] = matrix[0][v]
    for (step =2 to n)        {
            find the smallest weight[v] such that v is not in vertexSet
            add v to vertexSet

            // check weight[u] for all u not in vertexSet
            for (all vertices u not in vertexSet)
                    if (weight[u] > weight[v]+matrix[v][u])
                            weight[u] = weight[v]+matrix[v][u]
    }
}
```

(a)



Step	v	vertexSet	weight [0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

A trace of the shortest-path algorithm applied to the graph

Practice problems: Show the shortest path from the starting node s (or 0) to all other nodes in the graph.