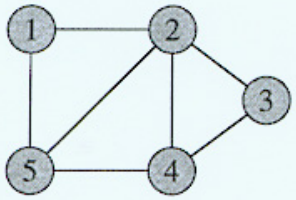
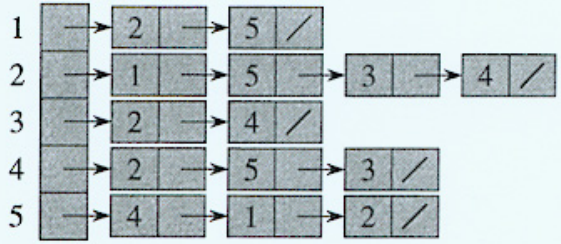
Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu



(a)

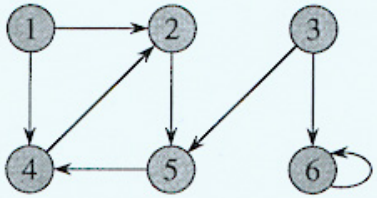


(b)

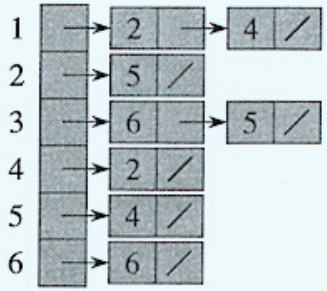
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Vertex and Edge classes

```
Class Edge {
    public Vertex dest;
    public double weight;

    public Edge (Vertex d,
                 double w) {
        dest = d;
        weight = w;
    }
}
```

```
Class Vertex {
    public String Name;
    public AnyType extraInfo;
    public List adj;
    public int dist; // double?
    public Vertex prev;
    public Vertex (String s) {
        Name = s;
        adj = new LinkedList();
        reset();
    }
    public reset () {
        dist=INFNT; path=null;
    }
}
```

Topological Sort

```
void topSort () {
        for( int j = 0; j < N; j++) {
                Vertex v = findVertexOfIndegZero();
                if (v == null)
                        return; //  Cycle found
                v.topologicalNum = j;
                for each vertex w adjacent to v
                        w.inDegree--; // use extraInfo field
        }
}
```

Time Complexity = O(n + m)

Unweighted SP algorithm

```
Void BFS (Vertex s) { // same as unweighted SP
   Queue <Vertex> Q = new Queue <>;
   for each Vertex v except s { v.dist = INFNT;}
   s.dist = 0; s.prev = null;
   Q.enqueue(s);
   while ( !Q.isEmpty() ) {
         v = Q.dequeue();
         for each vertex w adjacent to v
            if (w.dist == INFNT) {
                  w.dist = v.dist + 1;
                  w.prev = v;
                  Q.enqueue(w);
            }
   }
}
```

Time Complexity = O(n + m)

Dijkstra's SP algorithm

```
void Dijkstra (Vertex s) { // same as weighted SP
    PriorityQueue <Vertex> Q = new PriorityQueue <>;
    for each Vertex v except s { v.dist = INFNT; Q.insert(v); }
    s.dist = 0; s.prev= null;
    Q.insert(s);
    while ( !Q.isEmpty() ) {
          v = Q.deleteMin();
          for each vertex w adjacent to v
              if (w.dist > v.dist + weight of edge (v,w)) {
                    w.dist = v.dist + weight of edge (v,w);
                    w.prev= v;
                    Q.updatePriority(w, v.dist + weight of edge (v,w));
              }
    }
}
```

Time Complexity = O(n log n + m + m log n) = O(m log n)

Spanning Trees and MST

- **Graph** G(V,E)
- **Path**: sequence of edges
- **Cycle**: closed path
- A **subgraph** G' of G is given by G'(V', E')
- A **tree** is a graph with no cycles
- A connected graph has a path between all vertices
- A spanning tree is a tree that connects all vertices

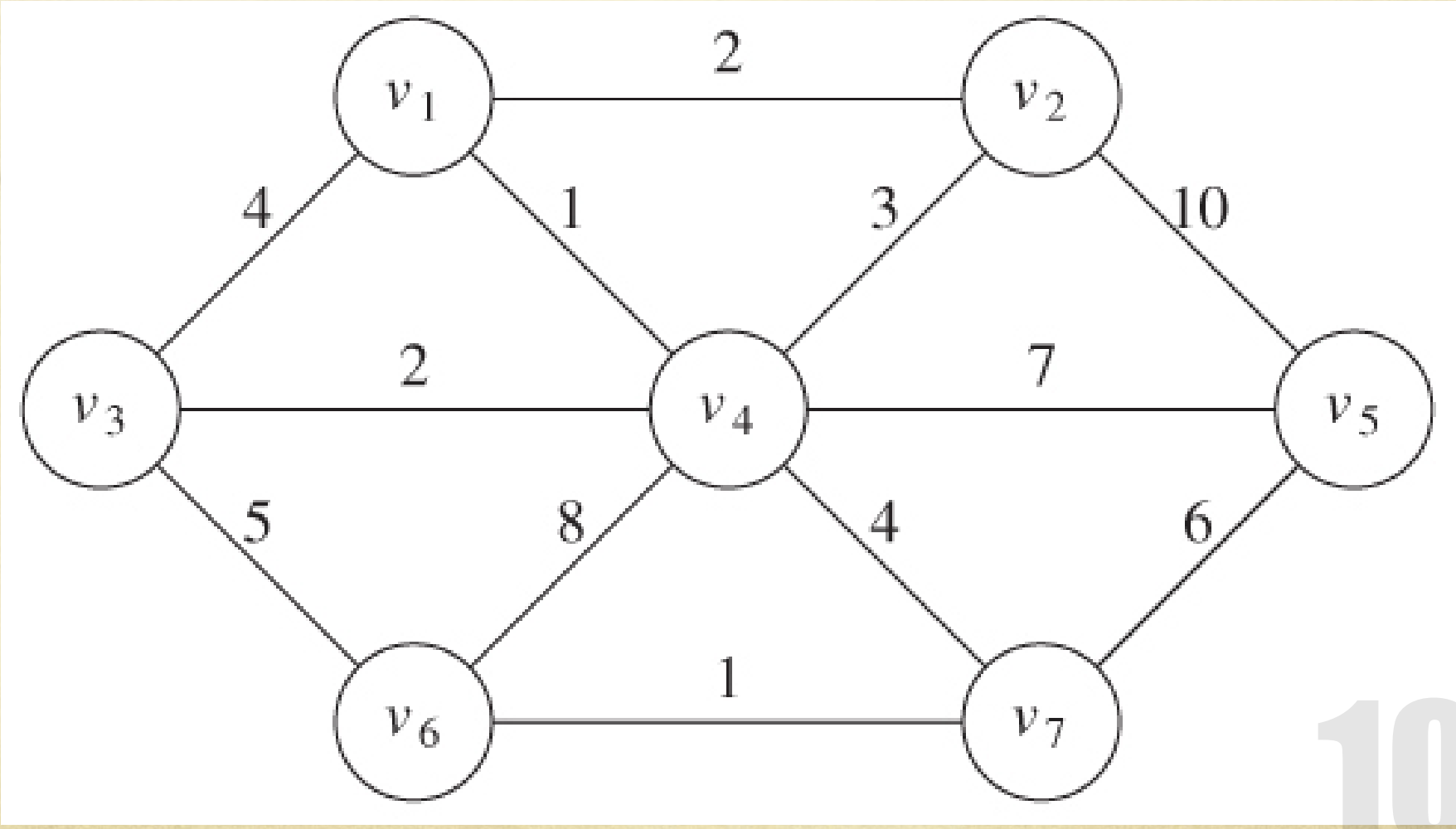
Minimum Spanning Trees

- **Input**: Weighted Graph, G(V,E) with n vertices & m edges, and edge weights w(e) for edge e
- **Output**: Find a spanning tree of minimum total weight

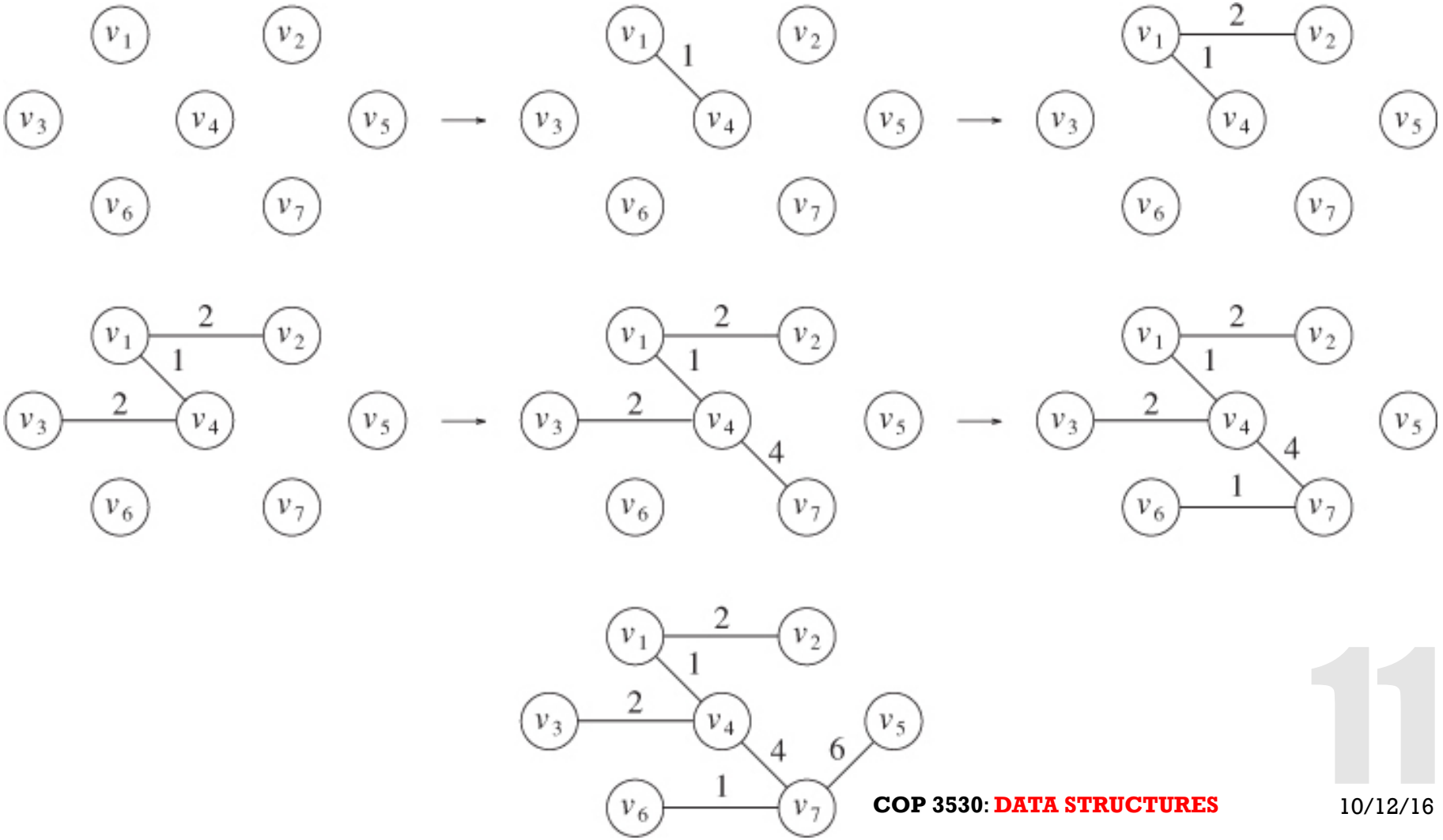
Observations and Facts

- Every spanning tree has n-1 edges.
- A spanning tree connects all vertices of a graph.
- A spanning tree has no cycles.
- If there is a cycle in the original graph, any spanning tree must miss at least one edge.
  - MST misses one heaviest edge on that cycle.
- Given any partition of vertices into (S, V-S), every spanning tree must include at least one edge from S to V-S.
  - MST includes one lightest edge from E(S,V-S).

MST – Prim's Algorithm



Prim's Algorithm



Reminder of Dijkstra's Alg

```
void Dijkstra (Vertex s) { // same as weighted SP
    PriorityQueue <Vertex> Q = new PriorityQueue <>;
    for each Vertex v except s { v.dist = INFNT; Q.insert(v); }
    s.dist = 0; s.prev= null;
    Q.insert(s);
    while ( !Q.isEmpty() ) {
          v = Q.deleteMin();
          for each vertex w adjacent to v
              if (w.dist > v.dist + weight of edge (v,w)) {
                    w.dist = v.dist + weight of edge (v,w);
                    w.prev= v;
                    Q.updatePriority(w, v.dist + weight of edge (v,w));
              }
    }
}
```

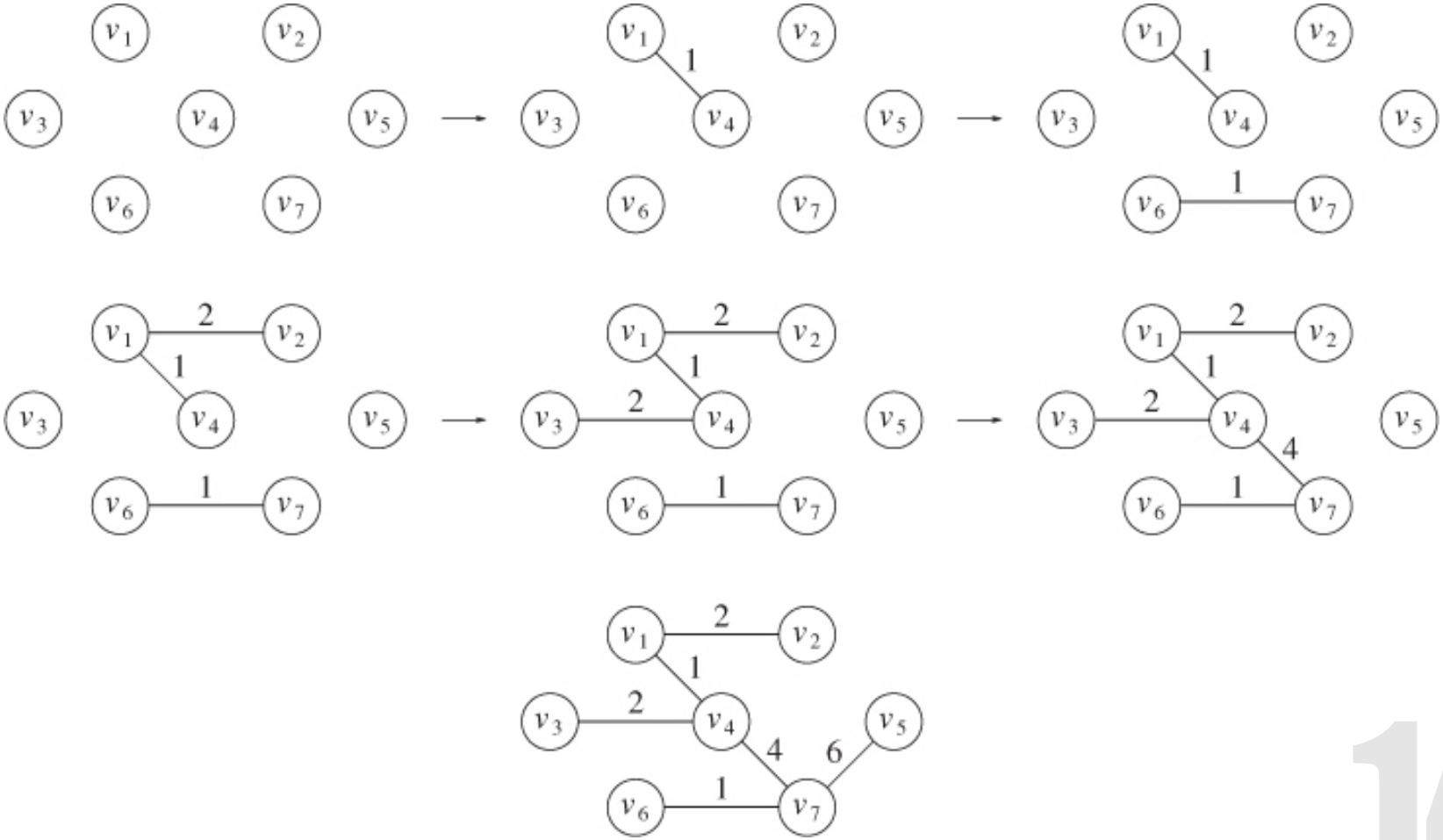

Time Complexity = O(n log n + m + m log n) = O(m log n)

Prim's Algorithm

```
ArrayList <Edges> Prim (Vertex s) { // computes MST
   PriorityQueue <Vertex> Q = new PriorityQueue <>;
   for each Vertex v except s { v.dist = INFNT; Q.insert(v); }
   s.dist = 0; s.prev= null;
   Q.insert(s);
   while ( !Q.isEmpty() ) {
         v = Q.deleteMin();
         for each vertex w adjacent to v
            if (w.dist > weight of edge (v,w)) {
                  w.dist = weight of edge (v,w);
                  w.prev= v;
                  Q.updatePriority(w, weight of edge (v,w));
            }
   }
  for each Vertex v except s { compile List mst with edges (v.prev, v); }
  return mst;
}
```

Time Complexity = O(n log n + m + m log n) = O(m log n)

Kruskal's Algorithm



Pseudocode: Kruskal's

```
ArrayList <Edges> Kruskal( List <Edge> edges, int N) {
        PriorityQueue<Edge> Q = new PriorityQueue<>(edges);
        List<Edge> mst = new ArrayList<>();
        while mst.size() != N-1) {
                Edge e = Q.deleteMin();
                if (EdgeConnectsDifferentComponents(e)) {
                        mst.add(e);
                        UpdateComponents(e);
                }
        }
        return mst;
}
```

Time Complexity = O(m log m) = O(m log n)