CLASS NOTES, CS 3137

1 Why Graphs?

Graphs are mathematical objects that have been studied for hundreds of years. They basically deal with connections between entities. With the onset of computation, graphs have served as very useful abstractions in solving a number of problems. The development of fast and efficient algorithms to compute on graphs continues to be an ongoing and vibrant research area in Computer Science. Some graph applications in computation include:

- Maps: Maps naturally express connections and paths between locations. Finding the shortest path or other constrained path finding is a hallmark of graph computation.
- Web Browsing: in actuality, the web is just one big giant graph, with websites (url's) linked to other web sites. Graph algorithms are central to locating information on the web.
- Circuits: We can model electrical circuits as graphs, with wires connecting different components (transistors, resistors, capacitors etc.). Using graph alogorithms we can design chips and ask questions about electrical flows, open circuits, timings etc.
- Scheduling: Large jobs with many components and pieces can be modeled as a graph, with timing constraints on jobs and pre-requisite jobs that need to be performed in a certain order.
- Matching Algorithms: Finding links (connections) between large groups of people and institutions can be modeled as a graph search.
- Computer networks: think about a local area network (LAN) with lots of computers connected together - this can be modeled as a graph. We can use this model to predict delays, find speedups in communication etc.

There are many more applications as well. It is important to note that some of these graphs (e.g. the web, 10 million transistor chip) are extremely large, and this puts a premium on finding efficient graph algorithms.

2 Graph Terminology

1. A **graph G** is comprised of two sets **V** and **E**, V=**Vertices**, E = **Edges** (edges are pairs of vertices). The graph is denoted G=(V,E), and edge set is E(G), vertex set V(G). Vertices are also called **nodes**, and edges are also called **arcs**.
2. Edges are unordered in an **undirected graph**: $(V_1, V_2) \equiv (V_2, V_1)$ Edges are ordered in a **directed graph**: $(V_1, V_2) \neq (V_2, V_1)$. A directed graph is also called a **digraph**
3. We restrict for now to **simple** graphs (i.e., no **self loops**)$V_1 \neq V_2$ for all edges. Also since E and V are sets we do not allow **multi-edges** i.e. (V_1, V_2) can only appear once. A graph with multi-edges is called a **multi-graph**.
4. Two vertices (V_1, V_2) in an undirected graph are said to be **adjacent** if edge (V_1, V_2) is **incident** to both V_1 and V_2. In directed graph, given edge (V_1, V_2) V_1 is adjacent to V_2, and V_2 is adjacent from V_1. We can also say that V_1 is a **predecessor** of V_2 and V_2 is a **successor** of V_1.

5. A **Subgraph** G' of G is such that G'=(V',E') where E'(G') is a subset of E(G) and V'(G') is a subset of V(G).

6. A **Path** from V_p to V_q in G is a sequence of vertices $V_p, V_{i1}, V_{i2}, \dots, V_{iN}, V_q$, such that $(V_p, V_{i1}), (V_{i1}, V_{i2}), \dots, (V_{iN}, V_q)$ are edges in E(G). A **simple path** is one which all vertices (except first and last) are distinct.

7. **Length** of the path is the number of edges in it, which is 1 less than the number of nodes on the path. A trivial path of length 0 (with no edges) exists between a node and itself.

8. A **Cycle** is a simple path in which the first and last vertices are the same. For an undirected graph, a simple cycle must have at path of 3 or more that starts and ends at the same node and doesn't visit any node more thatn once. An **acyclic** graph is a graph that contains no cycles.

9. Two vertices, V_1 and V_2 are **connected** if a path exists from V_1 to V_2. In an undirected graph if there is a path from V_2 to V_1 then there must also be a path from V_1 to V_2.

10. An undirected graph is **connected** if a path exists between all vertex pairs.

11. A **directed graph** is **strongly connected** if for every pair of vertices, V_1 and V_2, there is a path from V_1 to V_2 and a path from V_2 to V_1.

12. If a directed graph is not strongly connected, but the underlying graph without directions on the arcs is connected, then we call this directed graph **weakly connected**.

13. A **Connected Component** is a subgraph that is a connected subgraph of G.

14. The **degree** of a vertex is the number of edges that are incident to the vertex. In a directed graph, we use the terms **in-degree** and **out-degree**.

    The number of edges in an undirected graph is $E = \frac{1}{2}\sum_{i=1}^{N} d_i$, d_i = degree of vertex i.

15. Given an undirected graph with N vertices, there is a maximum of $\frac{N(N-1)}{2}$ edges that can exist.

    - Informal Proof. If N vertices, then there are $N * (N-1)$ edges from each of the N vertices to each of the $N-1$ other vertices. But since exactly half of these are duplicate edges, then there are a total of $\frac{N*(N-1)}{2}$
    - Proof By Induction. Basis: N=1. No edges and the formula holds.
    - Induction Step: Assume true for N vertices, show true for N+1 vertices.
    - By induction: if a graph with N vertices has $\frac{N(N-1)}{2}$ edges show graph with N+1 vertices has $\frac{(N+1)N}{2}$ edges. If we add a vertex to a graph with N vertices, we have to add a total of N new edges. So this graph will have the amount of edges in a graph with N vertices (which we know by the induction step) plus N more edges:

$$\frac{N\cdot(N-1)}{2} + N = \frac{N\cdot(N-1)+2N}{2} = \frac{N^2-N+2N}{2} = \frac{N^2+N}{2} = \frac{(N+1)\cdot N}{2} \tag{1}$$

3 Representing Graphs: Edge Lists

A simple but poor way to represent a graph is by an edge list. Assuming the graph has no nodes that are disconnected (i.e no edges incident to the node), we can simply list all edges as pairs of vertices. If the graph is directed, we can assume an ordering on the vertices for the directed edges: (a,b) is an edge *from* a *to* b in a directed graph.

Edge lists do not allow us to easily find connectivity information in the graph. A search through the entire set of edges is needed to find the connectivity information.

Typically, edge lists are ways to store graphs off-line and either an adjacency list or adjcanency matrix can be created from the edge lists.

4 Representing Graphs: Adjacency Matrix

- Matrix A such that $a_{ij} = 1$ if edge exists between V_i and V_j and 0 if no edge. If we assume no self loops (edges from a vertex to itself) then all diagonal entries will be 0: $a_{ii} = 0$
- Adjacency Matrix for a directed Graph:

  G=(V,E)= ($[V_1, V_2, V_3, V_4 V_5, V_6]$, $[(V_1, V_2), (V_1, V_3), (V_2, V_3), (V_3, V_4), (V_4, V_1), (V_4, V_5), (V_4, V_6), (V_5, V_4)]$)

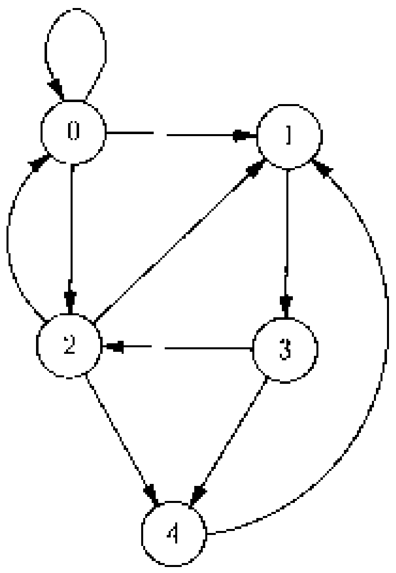
	$V1$	$V2$	$V3$	$V4$	$V5$	$V6$
$V1$	0	1	1	0	0	0
$V2$	0	0	1	0	0	0
$V3$	0	0	0	1	0	0
$V4$	1	0	0	0	1	1
$V5$	0	0	0	1	0	0
$V6$	0	0	0	0	0	0

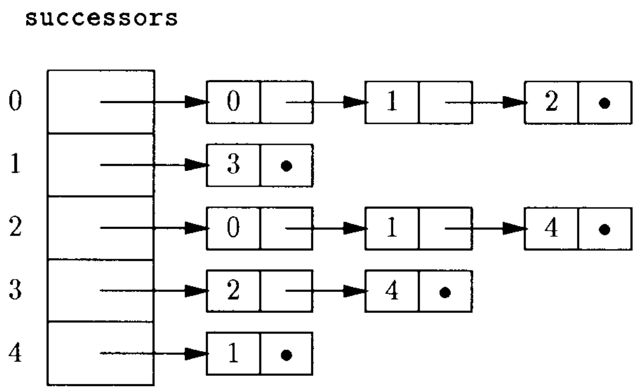
- In an undirected graph, the matrix is **symmetric** $(a_{ij} = a_{ji})$ so we need only store the upper or lower half of the matrix.
- With an adjacency matrix, most algorithms are $O(N^2)$ since we need to process the whole NxN matrix.
- In a directed graph, Column sum = in-degree of vertex and row sum = out degree of vertex

5 Representing Graphs: Adjacency Lists

We store for each vertex a linked list of its successors. The array Successors below is an array of pointers to a linked list of vertex nodes that contain a vertex which is adjacent to the node. The index of the array Successors is used to designate which vertex's successor list we are pointing to.

In an undirected graph, since each edge (V1,V2) is bidirectional, we will have an entry in the adjacency list twice for each node: once on a linkled list pointing from V1's entry in the successors list, and once from V2's entry in the successors list.





6 Depth First Search

Depth First Search (DFS) is a generalization of preorder traversal. If we do preorder traversal on a tree, we process a tree node v, and then recursively process each of its children (you can think of children as adjacent vertices of a graph). Because of a tree's structure, we do not have to worry about processing a node twice. However, with graphs, this can occur since a single node can have more than 1 "connection" (edge) to other nodes. So to prevent this we simply mark each vertex as visited, and then we will never process it twice.

DFS Search continually searches deeper into the graph until it comes upon nodes it has already visited. It then backtracks to continue searching for nodes it hasn't yet visited.

```
/* pseudo code for Depth First Search of Graph using an Adjacency List */
/* v is vertex number and index to adjacency list successor array */

void dfs( Vertex v)
{
  v.visited= true;
  for each w adjacent to v
    if( !w.visited)
        dfs(w);
}
```

Depth First search can be used to find all of the following:

- Testing whether a Graph G is connected. An undiercted graph is connected **if and only if** a DFS starting from ANY node does visits all the nodes
- Computing the Connected Components of G. We simply call DFS from any node, and list the nodes the DFS reaches. These are connected components. By repeating the DFS call to any unmarked nodes, we then find other connected components. We can create a depth first search forest of trees, each tree contianing connected components of the graph.

- Computing a spanning tree of G, if G is connected. A **Spanning Tree** is an acyclic graph that contains all the vertices of a graph G (this only makes sense if the graph is connected). We simply do a DFS on the graph, and mark all the edges we use to visit an umarked vertex. The result is a spanning tree.
- Computing a cycle in G or reporting that G has no cycles. This is done by checking to see if the DFS algorithm ever tries to visit a node that is already marked as visited. If it does, then the graph has a cycle.
- Computing a Path between two given vertices of G (if one exists). Simply start DFS at one of the vertices, see if it visits the other vertex.

7 Breadth First Search

Breadth First Search is similar to the level-order search we performed on trees. You visit the initial node, mark it as visited, and then place any successor nodes on a queue, marking them as visited. Then you begin dequeueing nodes. Each time a node is dequeued, you place any of its unvisited successor nodes on the queue and mark them as VISITED. Eventually the queue empties and you are done.

```
void bfs(Vertex v)
{
   v.visited=true;
   EnQueue on Q each unvisited vertex w adjacent to v and mark as visited
   while (Q not empty) {
      node= DeQueue(Q);
      EnQueue on Q each unvisited vertex w adjacent to node and mark as visited
}
```

8 Topological Sorting

1. The nodes of a directed graph without any cycles can be placed into a special ordering called a *topological sort*. This is a partial ordering of the nodes such that if there is a path from vertex v_i to v_j, then v_i appears before v_j in the ordering.

2. A common use for topological sorting is in determining prerequisites for courses. It can also be used to order the way parts of a job are done to insure that things that need to be done first are.

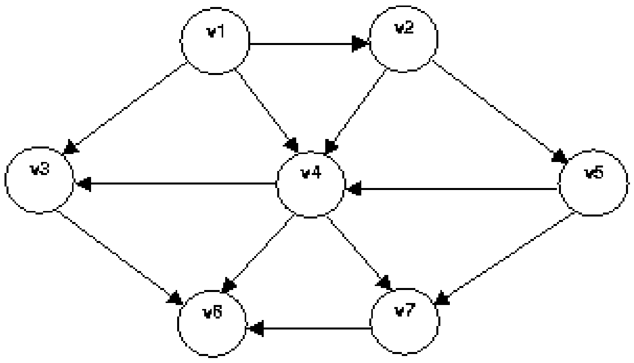
```
void topsort() throws CycleFound
{
  Queue q;
  int counter=0;
  Vertex v,w;

  q= new Queue();
  for each vertex v
    if (v.indegree == 0)
     q.enqueue(v);

  while (!q.isEmpty())
  {
    v=q.dequeue();
    v.topNum = ++counter;

    for each w adjacent to v
      if(--w.indegree == 0)
         q.enqueue(w);
  }

  if(counter != NUM_VERTICES)
     throw new CycleFound();
}
```

You can also perform topological sorting by using Depth First Search. Create a Depth First Search Tree, and then do a postorder traversal numbering the nodes of the tree. The reverse of the postorder numbering will give you the topological listing of the nodes.

9 Minimum Spanning Trees

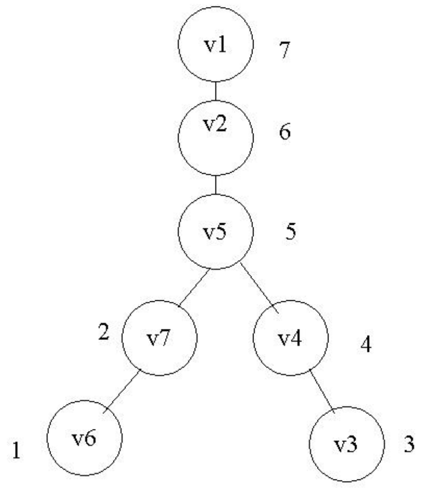
1. A **Spanning Tree** is an acyclic graph that contains all the vertices of a graph G. This only makes sense if the graph is connected!

2. A Spanning Tree on a connected graph of N nodes has exactly $N - 1$ edges (can be shown with a simple proof by induction)



...............Indegree Before Dequeue Number

Vertex	1	2	3	4	5	6	7
v1	0	0	0	0	0	0	0
v2	1	0	0	0	0	0	0
v3	2	1	1	1	0	0	0
v4	3	2	1	0	0	0	0
v5	1	1	0	0	0	0	0
v6	3	3	3	3	2	1	0
v7	2	2	2	1	0	0	0
Enqueue	v1	v2	v5	v4	v3,v7		v6
Dequeue	v1	v2	v5	v4	v3	v7	v6



Topological order: v1, v2, v5, v4, v3, v7, v6

Figure 1: Top: Acyclic Graph. Middle: Toplogical Sort of Graph showing order nodes are processed by dequeing nodes with zero in-degree. Bottom: Computing Toplogical sort using Depth First Search Tree of the graph, and its postorder numbering. By listing the nodes in the reverse of the postorder numbering, we generate a topological sort of the nodes. Note: graph must be acyclic.

3. One way to form a spanning tree is take a cycle that includes every node in the graph, and remove any edge. We can also see that removing the edge of greatest cost will reduce the cost of the spanning tree created over removing a lower cost edge. So there must be a way to find the spanning tree of minimal overall cost or **Minimum Spanning Tree (MST)**.

4. A MST can be used to reduce the cost of connecting nodes. For example, the nodes might represent cities that need to be connected by electrical or gas lines, and reducing the cost of the lines is equivalent to finding the MST.

5. Given graph G, Can we find the MST?

```
 1 -------(16)-------2
 | \               / | \
 |  \             /  |  \
 |   \           /   |   \
 |  (21)      (11)   |   (5)
 |     \       /     |     \
 |      \     /      |      \
 |       \   /       |       \
(19)       6        (6)       3
 |        / \        |       /
 |       /   \       |      /
 |      /     \      |     /
 |   (33)    (14)    |  (10)
 |    /         \    |   /
 |   /           \   |  /
 |  /             \  | /
 | /               \ |/
 5-------(18)--------4
```

6. Algorithm **- Kruskal's Method**. Select an edge of minimum length only if it doesn't make a cycle. Keep doing this until we have selected $N - 1$ edges.

   - Initially, we maintain a forest of N single node trees representing each vertex in the graph. We then link these vertices as we add edges of minimum cost to build up a spanning tree. We note that each of the nodes in a linked tree is part of a set. If at any time an edge is added that causes a cycle, we reject that edge.
   - We can determine if the proposed edge causes a cycle by maintaining a Union-Find data structure (see below) that tells us if two nodes in an edge are already connected in the tree. Adding this edge will then cause a cycle, so we reject this edge as part of the MST. Eventually, we build up a tree that includes all the nodes in the graph, and we stop.
   - We use a priority queue prioritized by increasing edge weight to always get the next lowest cost edge.

7. Pseudo Code for Kruskal:

```
Place all edges in a Priority Queue prioritized by minimum edge cost;

Make each vertex a single element set - a forest of single node trees;

While there is more than one tree in the forest do:
   Delete edge E of minimum length from the Priority Queue (E=(v,w));
   If vertices v and w are not already members of the same set then
      Add edge E=(v,w) as edge of the MST
      Perform a Set Union on the sets containing v and w;
```

The single node sets become multiple node sets through the Set Union operation. At any time if we want to add an edge, we make sure that the 2 vertices that define the edge are not in the same set - adding this edge will create a a cycle.

8. Algorithm - **Prim's Method** - Choose any node, and select the minimum cost edge containing that node. Continue to select an adjacent edge of least cost to the tree built so far, and add it if it doesn't make a cycle: Continually Select edge (u,v) where vertex u is in the tree so far and vertex v is not.

```
  PRIM (Start node 1)    Kruskal
edge   cost             edge   cost
(1-2)  16               (2-3)    5
(2-3)   5               (2-4)    6
(2-4)   6               (2-6)   11
(2-6)  11               (2-1)   16
(4-5)  18               (4-5)   18

Total  56                       56
```

9. Algorithm - **RemoveMax Edges**. Start with connected Graph G, and continually remoce the most expensive edges while maintaining the graph's connectivity.

```
RemoveMax Edges removes these edges to form MST:
(5-6)
(1-6)
(1-5)
(6-4)
(3-4)
```

10. How to implement **Union-Find Tree**. Define an array parent[] of integers defined by the following rule: if *i* lies at the root of the rooted tree for its set, then parent[*i*]=*i*. Otherwise parent[*i*] = value of its parent.

11. **Simple Find Algorithm**

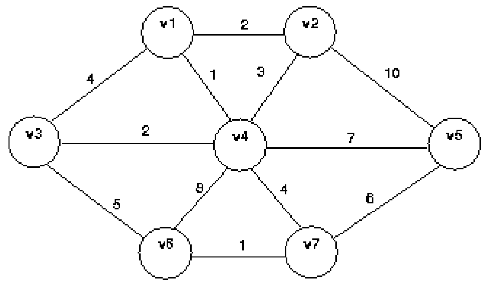
```
find (x)
  while(parent[x]!=x)
     x=parent[x]
  return(x)
```

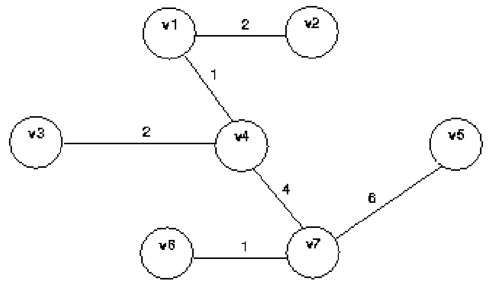
12. To perform Union, we find roots of the trees that are to be unioned, and make one root then parent of the other (graft one tree onto another).

```
union(x,y)
      px=find(x)
      py=find(y)
      parent[px]=py
```

13. Proof of Kruskal's MST algorithm. The idea is to show that the Spanning Tree we create by Kruskal's method can not be of higher total edge cost than the minimal Spanning Tree, so Kruskal's tree is the MST.

    - Let $e_1, e_2, \ldots, e_m$ be the dges of Graph G in order of their cost, smallest first. This is the order that the edges will be considered by Kruskal. Let K by the spanning tree created by Kruskal's algorithm, and let T be the MST of Graph G.
    - We now prove K and T are the same. Assume they are not. Then there must be some edge that is in one but not the other. Let e_i be the first edge in the ordering of edges that appears in either K or T but not both. We examine both cases, and show that this cannot be - that the edges in both K and T are the same.
    - *Case I:* Edge e_i is in T but not in K. If e_i is not in K, then it was rejected since it caused a cycle with the vertices already in K. But these same vertices must also be part of T, and if T has added edge e_i, then T must also have a cycle. But since T is by definition a MST, T cannot have a cycle. So edge e_i cannot be in T but not K.
    - *Case II:* Edge e_i is in K but not in T. Let e_i connect nodes u and v. Since T is connected, there must be some acyclic path in T between u and v; call it path Q. Since Q does not contain the edge e_i that also connects u and v, adding e_i to Q forms a simple cycle in the graph. We now analyze the two cases¡ where e_i is the largest cost edge or not.
        - If edge e_i has the highest label (most costly edge), then the edges of path Q contain edges from $[e_1, e_2, \ldots, e_{i-1}]$. Since T and K agree on all edges up until e_i, then all the edges of path Q are in K. But since e_i is also in K, then K has a cycle, and Kruskal's method will not allow a cycle. We thus rule out edge e_i having the highest edge label.
        - Assume there is some edge f on path Q whose edge label is higher than e_i. Suppose f connects nodes w and x. If we remove edge f from T, we can replace it with edge e_i, and reduce the cost of the tree since we have substituted a lower cost edge for f. We claim the tree is stil connected, since f connected w and x, and they still must be connected by the removal of f and insertion of e_i. However, this means that T was not minimal before the insertion of edge e_i, and therefore e_i must have been in T, contradicitng our assumption that e_i was in K but not T.
    - We have now shown that K and T must have the same edge set, and therefore K is minimal.





Edge	Weight	Action
(v1-v4)	1	Accepted
(v6-v7)	1	Accepted
(v1-v2)	2	Accepted
(v3-v4)	2	Accepted
(v2-v4)	3	Rejected
(v1-v3)	4	Rejected
(v4-v7)	4	Accepted
(v3-v6)	5	Rejected
(v5-v7)	6	Accepted

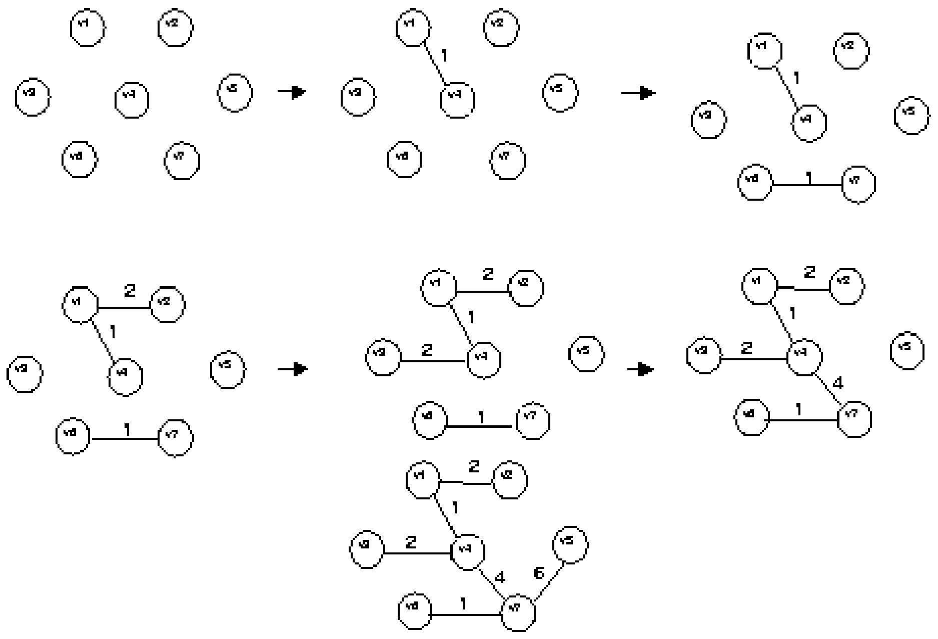


Figure 2: Kruskal’s Method of fidning MST. Top: Original Undirected Graph. Middle: MST. Bottom: Stages of accepting edges in Kruskal’s Method