CMSC 341
Lecture 20

Announcements

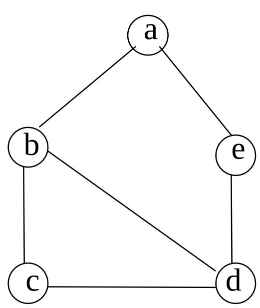
Basic Graph Definitions

A *graph* G = (V,E) consists of a finite set of *vertices*, V, and a set of *edges*, E. Each edge is a pair (v,w) where v, w ∈ V.

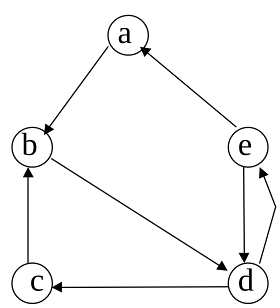
- V and E are sets, so each vertex v ∈ V is unique, and each edge e ∈ E is unique.
- Edges are sometimes called *arcs* or *lines*.
- Vertices are sometimes called *nodes* or *points*.

A *directed graph* is a graph in which the edges are ordered pairs. That is, (u,v) ≠ (v,u), u, v ∈ E. Directed graphs are sometimes called *digraphs*.

An *undirected graph* is a graph in which the edges are unordered pairs. That is, (u,v) = (v,u).



undirected graph



directed graph

Basic Graph Definitions (cont.)

Vertex v is *adjacent to* vertex w if and only if $(v,w) \in E$. (Book calls this *adjacent from*)

Vertex v is *adjacent from* vertex w if and only if $(w,v) \in E$.

An edge may also have:

- *weight* or *cost* -- an associated value
- *label* -- a unique name

The *degree* of a vertex u in an undirected graph is the number of vertices adjacent to u. Degree is also called *valence*.

The *indegree* (*outdegree*) of a vertex u in a directed graph is the number of vertices adjacent to (from) u.

Paths in Graphs

A *path* in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$ for $1 \le i < n$.

The *length* of a path in a graph is the number of edges on the path. The length of the path from a vertex to itself is 0.

A *simple path* is a path such that all vertices are distinct, except that the first and last may be the same.

A *cycle* in a graph is a path $w_1, w_2, w_3, \ldots, w_n$, $w \in V$ such that:

- there are at least two vertices on the path
- $w_1 = w_n$ (the path starts and ends on the same vertex)
- if any part of the path contains the subpath w_i, w_j, w_i, then each of the edges in the subpath is distinct.

A *simple cycle* is one in which the path is simple.

Connectedness in Graphs

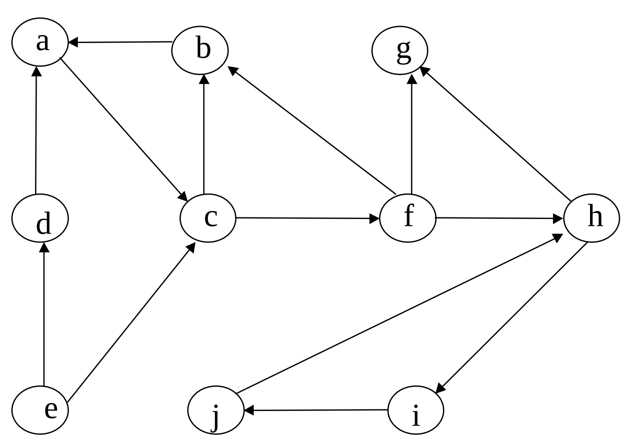
An undirected graph is *connected* if there is a path from every vertex to every other vertex.

A directed graph is *strongly connected* if there is a path from every vertex to every other vertex.

A directed graph is *weakly connected* if there would be a path from every vertex to every other vertex, disregarding the direction of the edges.

A *complete* graph is one in which there is an edge between every pair of vertices.

A *connected component* of a graph is any maximal connected subgraph. Connected components are sometimes simply called *components*.



A Graph ADT

Has some data elements

- vertices
- edges

Has some operations

- getDegree(u) -- returns the degree of vertex u (undirected graph)
- getInDegree(u) -- returns the indegree of vertex u (directed graph)
- getOutDegree(u) -- returns the outdegree of veretx u (directed graph)
- getAdjacent(u) -- returns a list of the vertices adjacent from a vertex u (directed and undirected graphs)
- isConnected(u,v) -- returns TRUE if vertices u and v are connected, FALSE otherwise (directed and undirected graphs)

Graph Traversals

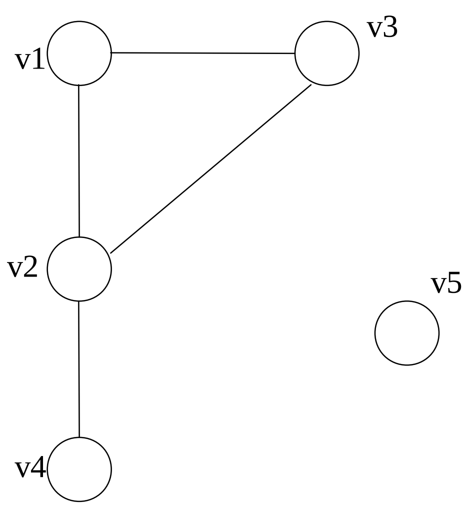
Like trees, can be traversed breadth-first or depth-first.

- Use stack for depth-first traversal.
- Use queue for breadth-first traversal.

Unlike trees, need to specifically guard against repeating a path from a cycle. Can mark each vertex as “visited” when we encounter it and not consider visited vertices more than once.

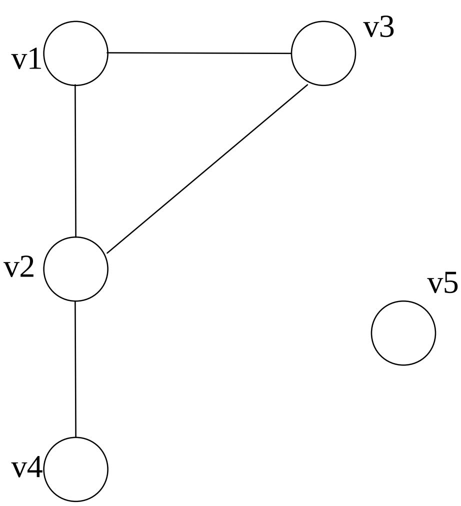
Breadth-First Traversal

```
Queue q = new Queue();
graphvertex u;

for all v, d[v] = ∞         // mark each vertex unvisited
q.enqueue(startvertex);     // start with any vertex
d[startvertex] = 0;         // mark visited
while (!q.isEmpty()) {
  u = q.dequeue();
  for (each vertex w adjacent from u)
      if (d[w] == ∞) {      // w not marked as visited
             d[w] = d[u]+1;      // mark visited
             q.enqueue(w);
             }
  }
```



Depth First Traversal

```
dfs(Graph G) {
   for (each v ∈ V)
        dfs(v)
   }

dfs(Vertex v) {
   markVisited(v);
   for(each vertex w adjacent from u)
        if ( w is not marked as visited)
                dfs(w)
   }
```



DFS (stack version)

```
Stack s = new Stack();
GraphVertex u;
GraphVertex startvertex = graph.getStartVertex();

s.push(startvertex);
markVisited(startvertex);
while (!s.isEmpty()) {
   u = s.Pop();
   for (each vertex w adjacent to u)
        if (w is not marked as visited) {
                markVisited(w);
                s.push(w);
                }
   }
```

Unweighted Shortest Path Problem

Unweighted shortest-path problem: Given as input an unweighted graph, G = (V,E), and a distinguished vertex, s, find the shortest unweighted path from s to every other vertex in G.

After running BFS algorithm with s as starting vertex, the shortest path length from s to i is given by d[i].

Weighted Shortest Path Problem

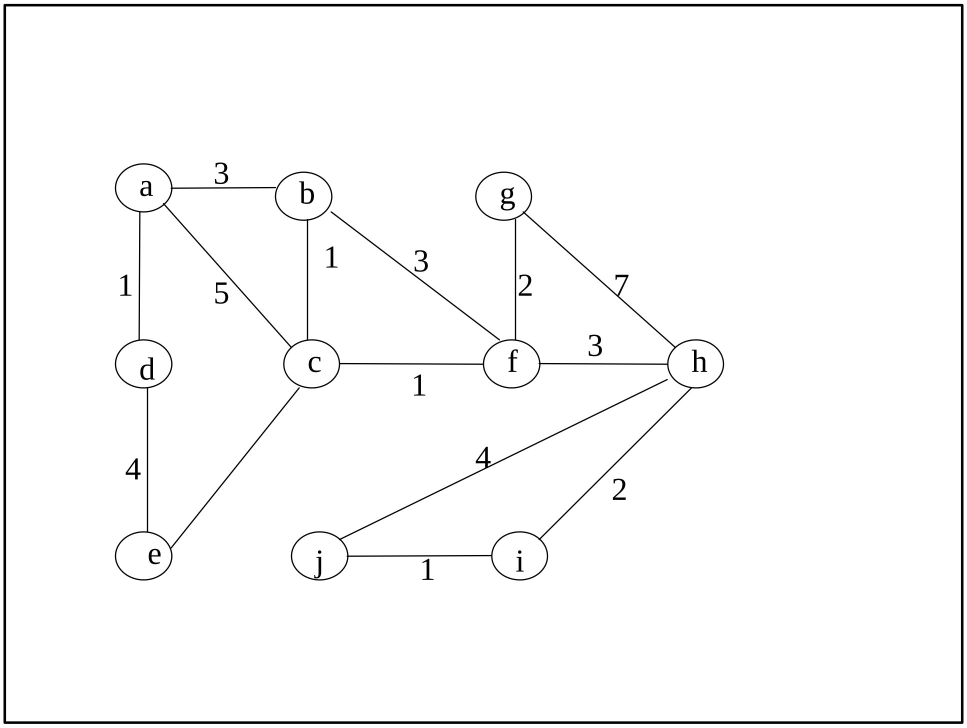
Single-source shortest-path problem: Given as input a weighted graph, G = (V,E), and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

Use Dijkstra's algorithm

- keep tentative distance for each vertex giving shortest path length using vertices visited so far
- keep vertex before this vertex (to allow printing of path)
- at each step choose the vertex with smallest distance among the unvisited vertices (greedy algorithm)

Dijkstra's Algorithm

```
Vertex v, w;
start.dist = 0;
for (;;) {
   v = smallest unknown distance vertex;
   if (v == NOT_A_VERTEX) break;
   v.known = TRUE;
   for each w adjacent to v
        if (!w.known)
                if (v.dist + cvw < w.dist) {
                        decrease (w.dist to v.dist + cvw);
                        w.path = v;
                        }
   }
```


a
b
g
c
d
f
h
e
j
i
3
1
5
1
3
2
7
3
1
4
4
2
1