Part 10. Graphs

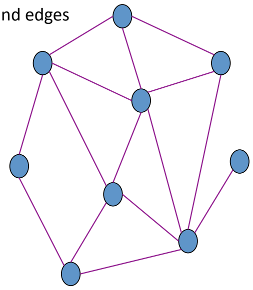
CS 200 Algorithms and Data Structures

Outline

- **Introduction**
- Terminology
- Implementing Graphs
- Graph Traversals
- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits

Graphs

A collection of nodes and edges

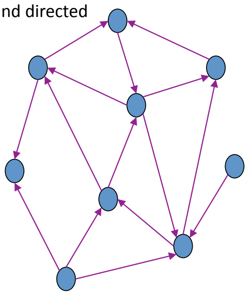


What can this represent?

- A computer network
- Abstraction of a map
- Social network

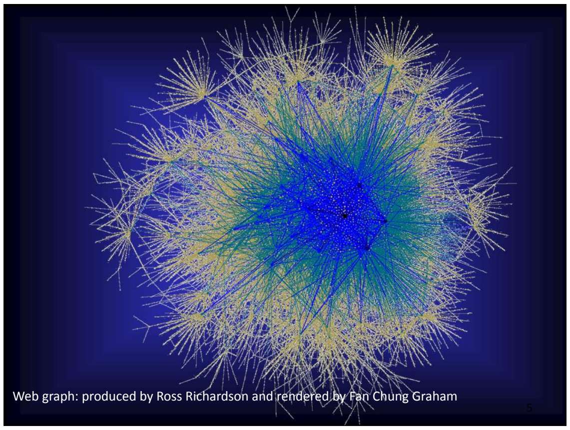
Directed Graphs

A collection of nodes and directed edges



Sometimes we want to represent directionality:

- Unidirectional network connections
- One way streets
- The web



Web graph: produced by Ross Richardson and rendered by Fan Chung Graham

Example

- "Follow the money" example (the international science and engineering visualization challenge, 2009, Science magazine and National Science Foundation)
  - http://www.sciencemag.org/site/special/vis2009/show/
- Reference

The Scaling Laws of Human Travel, D. Brockmann, L. Hufnagel and T. Geisel, *Nature* ***439, 462 (2006)***
  - http://rocs.northwestern.edu/index_assets/brockmann2006nature.pdf

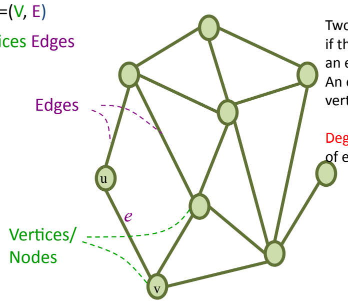
Outline

- Introduction
- **Terminology**
- Implementing Graphs
- Graph Traversals
- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits

Graph Terminology

G=(V, E)

Vertices Edges



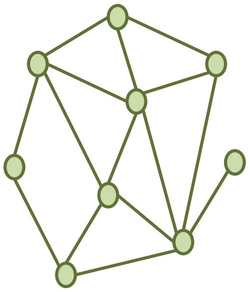
Two vertices are adjacent if they are connected by an edge.

An edge is incident on two vertices

Degree of a vertex: number of edges incident on it

Graph terminology: 14.1 in W&M, 9.1 in Rosen

Graph Terminology



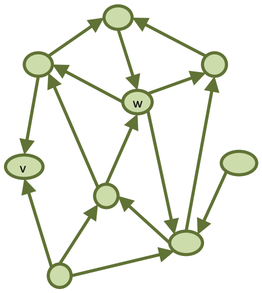
Self loop (loop): an edge that connects a vertex to itself

Simple graph: no self loops and no two edges connect the same vertices

Multigraph: may have multiple edges connecting the same vertices

Pseudograph: multigraph with self-loops

Directed Graphs



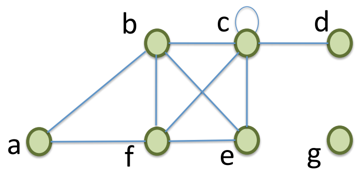
Indegree: number of incoming edges

Outdegree: number of outgoing edges

The degree of a vertex

- The degree of a vertex in an undirected graph
  - the number of edges incident with it
  - except that a loop at a vertex contributes twice to the degree of that vertex.

Example



deg(a) = 2
deg(b) = deg(f) = 4
deg(c)=6

deg(d) = 1
deg(e) = 3
deg(g) = 0

Theorem 10-1: The Handshaking Theorem

- Let G=(V,E) be an undirected graph. Then

$$\sum_{v\in V}\deg(v) = 2|E|$$

- How many edges are there in a graph with 10 vertices each of degree six?
  - 10 * 6 /2= 30

Theorem 10-2

- An undirected graph has an even number of vertices of odd degree.
- Proof:

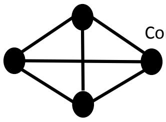
Theorem 10-3

- Let G=(V,E) be a graph with directed edges. Then

$$\underset{\text{indegree}}{\sum_{v\in V}\deg^{-}(v)} = \underset{\text{outdegree}}{\sum_{v\in V}\deg^{+}(v)} = |E|$$

Complete Graphs

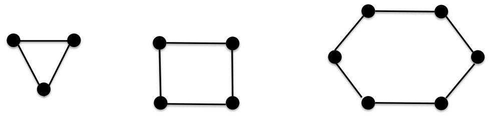
- Simple graph that contains exactly one edge between each pair of distinct vertices.



Complete Graph

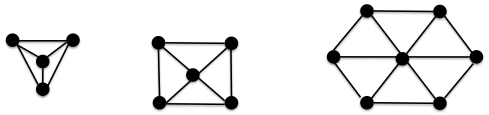
Cycles

The cycle C_n, $n \geq 3$, consists of n vertices $v_1, v_2, \ldots, v_n$ and edges $\{v_1, v_2\}$, $\{v_2, v_3\}$,…., and $\{v_{n-1}, v_n\}$.

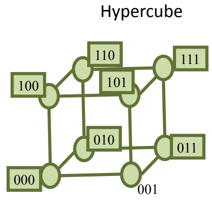


Wheels

- We obtain the wheel W_n when we add an additional vertex to the cycle C_n, for $n \leq 3$, and connect this new vertex to each of the n vertices in C_n, by new edges

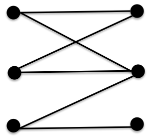


n-Cubes (n-dimensional hypercube)

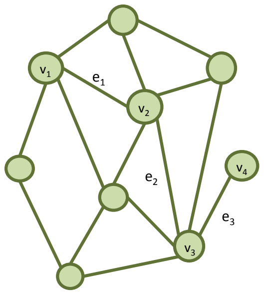


Bipartite Grphs

- A simple graph G is called ***bipartite***, if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 and a vertex in V_2.



Paths



- **Path:** a sequence of edges
- (e_1, e_2, e_3) is a path of length 3 from v_1 to v_4
- In a simple graph a path can be represented as a sequence of vertices

Outline

- Introduction
- Terminology
- **Implementing Graphs**
- Graph Traversals
- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits

Graph ADT

- Create
- Empty?
- Number of vertices?
- Number of edges?
- Edge exists between two vertices?
- Add a vertex
- Add an edge
- Delete a vertex (and any edges adjacent to it)
- Delete an edge
- Retrieve a vertex

Classes for a Weighted Undirected Graph

- Vertex:???
- Edge: ???
- Graph:
  - organized collection of vertices and edges

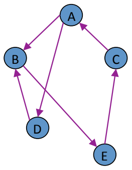
1. Adjacency Matrix Implementation
2. Adjacency List Implementation

Adjacency Matrix Implementation

Vertices

Label	Index
A	0
B	1
C	2
D	3
E	4

mapping of vertex labels to array indices



Edges

square matrix of edges

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

Adjacency Matrix: indicates whether an edge exists between two vertices

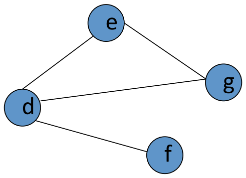
Possible Java implementation: int [][] A;

Adjacency Matrix Implementation

- Directed graph: A[i][j] is 1 if there is an edge from vertex i to vertex j
- Undirected graph: A[i][j] is 1 if there is an edge between vertex i and vertex j.
  What's the relationship between A[i][j] and A[j][i]?
- Weighted graph: A[i][j] gives the weight of the edge

Example

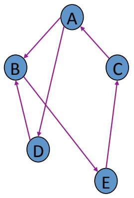
- Adjacency matrix?

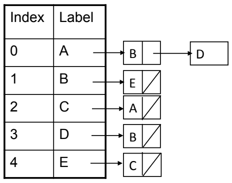


	d	e	f	g
d	0	1	1	1
e	1	0	0	1
f	1	0	0	0
g	1	1	0	0

Adjacency List Implementation

Each vertex has a list of outgoing edges





Index	Label
0	A
1	B
2	C
3	D
4	E

mapping of vertex labels to list of edges

Adjacency List Implementation

- For undirected graphs each edge appears twice. Why?

Which Implementation?

- Which implementation best supports common Graph Operations:
  - Is there an edge between vertex i and vertex j?
  - Find all vertices adjacent to vertex j
- Which best uses space?

Implementation: Edge Class

```
Class Edge {
   private Integer v,w;   // vertices
   private int weight;
   public Edge(Integer first, Integer second,
     int edgeWeight){
     v = first; w = second; weight =
     edgeWeight; }
   public int getWeight() {
     return weight; }
   public Integer getV() {
     return v; }
   public Integer getW() {
     return w; }
```

Implementation: Graph Class

```
class Graph {
   private int numVertices;
   private int numEdges;
   private Vector<TreeMap<Integer, Integer>>
     adjList;

   public Graph(int n) {
      numVertices = n; numEdges = 0;
      adjList = new Vector<TreeMap<Integer, Integer>>
        ();
      for (int i=0; i<numVertices; i++) {
         adjList.add(new TreeMap<Integer, Integer>());
      }
   }
```

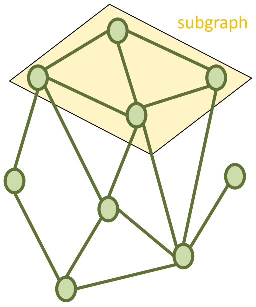
Implementation: Graph Class

```
public void addEdge(Integer v, Integer, w,
  int weight){
  // precondition:  the vertices v and w
  must exist
   //postcondition:  the edge (v,w) is part
  of the graph

  adjList.get(v).put(w, weight);
  adjList.get(w).put(v, weight);
  numEdges++;
}
```

Graph Terminology

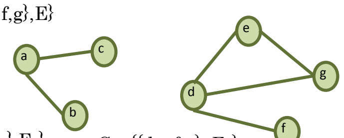
A subgraph of a graph G = (V,E) is a graph (V',E') such that V' is a subset of V and, E' is a subset of E



Connected Components

- An undirected graph is called **connected** if there is a path between every pair of vertices of the graph.
- A **connected component** of a graph G is a connected subgraph of G that is not a proper subgraph of another connected subgraph of G.

G={{a,b,c,d,e,f,g},E}



G_1={{a,b,c},E_1} G_2={{d,e,f,g}, E_2}

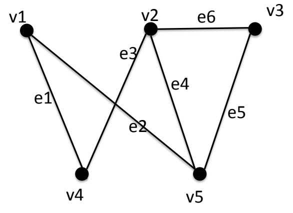
Other Representations

- Incidence Matrices

Let G = (V,E) be an undirected graph. Suppose that $v_1, v_2, \ldots v_n$ are the vertices and $e_1, e_2, \ldots, e_m$ are the edges of G. Then the incidence matrix with respect to this ordering of V and E is the n x m matrix M = $[m_{ij}]$, where

$$M_{ij} = \begin{cases} 1 & \text{when edge ej is incident with vi,} \\ 0 & \text{otherwise} \end{cases}$$

Example



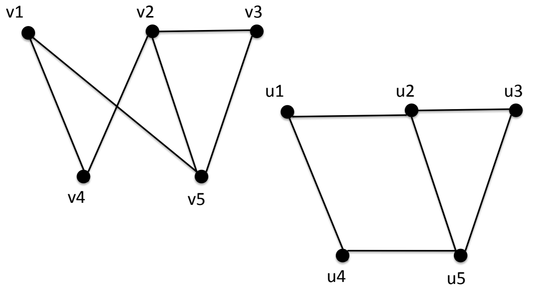
$$\begin{array}{c|cccccc} & e1 & e2 & e3 & e4 & e5 & e6 \\ \hline v1 & 1 & 1 & 0 & 0 & 0 & 0 \\ v2 & 0 & 0 & 1 & 1 & 0 & 1 \\ v3 & 0 & 0 & 0 & 0 & 1 & 1 \\ v4 & 1 & 0 & 1 & 0 & 0 & 0 \\ v5 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Useful to represent multiple edges and loops

Isomorphism of Graphs

- Definition
- The simple graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ are ***isomorphic*** if there is a one-to-one and onto function f from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2, for all a and b in V_1. Such a function f is called an ***isomorphism***.

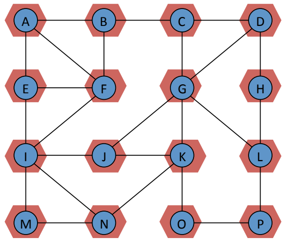
Example



Outline

- Introduction
- Terminology
- Implementing Graphs
- **Graph Traversals**
- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits

Graph Traversal



Graph Traversal

Depth First Search (DFS)

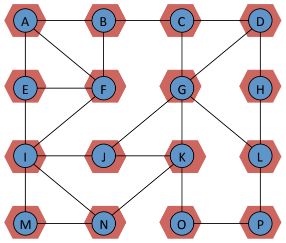
```
dfs(in v:Vertex)
  mark v as visited
  for (each unvisited vertex u adjacent to v)
   dfs(u)
```

- Need to track visited nodes
- Order of visiting nodes is not completely specified
- Is there a difference between directed undirected graphs?
- Which graph implementation?

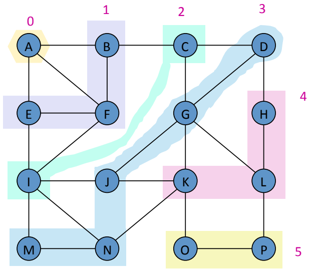
Iterative DFS

```
dfs(in v:Vertex)
    s – stack for keeping track of active vertices
    s.push(v)
    mark v as visited
    while(!s.isEmpty()) {
      if (no unvisited vertices adjacent to the vertex
      on top of the stack) {
        s.pop()  \\backtrack
      else {
        select unvisited vertex u adjacent to vertex on
      top of the stack
        s.push(u)
        mark u as visited
      }
    }
```

DFS Example



Graph Traversal – Breadth First Search (BFS)



BFS

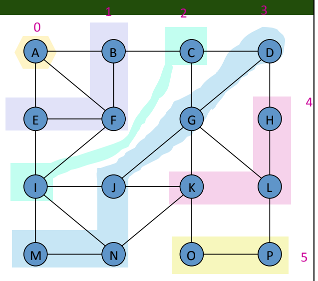
- Similar to level order tree traversal
- DFS is a last visited first explored strategy
- BFS is a first visited first explored strategy

BFS

```
bfs(in v:Vertex)
  q – queue of nodes to be
  processed
  q.enque(v)
  mark v as visited
  while(!q.isEmpty()) {
   w = q.dequeue()
   for (each unvisited vertex u
  adjacent to w) {
      mark u as visited
      q.enqueue(u)
   }
  }
```

Trace this example

```
bfs(in v:Vertex)
  q – queue of nodes to
  be processed
  q.enque(v)
  mark v as visited
  while(!q.isEmpty()) {
   w = q.dequeue()
   for (each unvisited
        vertex u adjacen
        t to w) {
       mark u as visited
       q.enqueue(u)
   }
  }
```



Graph Traversal

- Properties of BFS and DFS:
  - Visit all vertices that are reachable from a given vertex
  - Therefore DFS(v) and BFS(v) visit a connected component
- Computation time for DFS, BFS for a connected graph: O(|V| + |E|)

Reachability

- Reachability
  - *v* is reachable from *u*
    - if there is a (directed) path from *u* to *v*
  - solve using BFS or DFS
- Transitive Closure (G*)
  - G* has edge from *u* to *v* if v is reachable from u.