Course Notes for

CS 1501
Algorithm Implementation

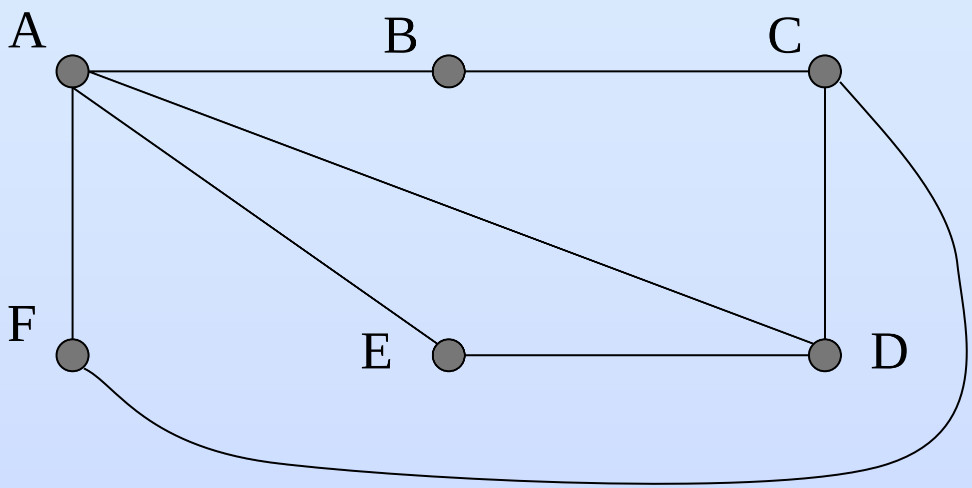
By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Graph: $G = (\mathbb{V}, \mathbb{E})$
  - Where $\mathbb{V}$ is a set of vertices and $\mathbb{E}$ is a set of edges connecting vertex pairs
  - Used to model many real life and computer-related situations
    - Ex: roads, airline routes, network connections, computer chips, state diagrams, dependencies, etc.
  - Ex:

    $\mathbb{V}$ = {A, B, C, D, E, F}

    $\mathbb{E}$ = {(A,B), (A,D), (A,E), (A,F), (B,C), (C,D), (C, F), (D, E)}
    - See next slide



AB
AD
AE
AF
BC
CD
CD
DE

- This is an undirected graph
  - Edges are bidirectional
  - Note that the picture / physical representation of this graph is not unique
  - We could draw several other "versions" of this graph

- Undirected graph
  - Edges are unordered pairs: (A,B) == (B,A)
- Directed graph
  - Edges are ordered pairs: (A, B) != (B,A)
- Adjacent vertices, or neighbors
  - Vertices connected by an edge

- Let v = $|\mathbb{V}|$ and e = $|\mathbb{E}|$
  - What is the relationship between v and e?
  - Let's look at two questions:
    1) Given v, what is the minimum value for e?
    2) Given v, what is the maximum value for e?

1) Given v, minimum e?
   - Graph definition does not state that any edges are required: 0

2) Given v, maximum e? (graphs with max edges are called complete graphs)
   - Directed graphs
     - Each vertex can be connected to each other vertex
     - "Self-edges" are typically allowed
       - Vertex connects to itself – used in situations such as transition diagrams
     - v vertices, each with v edges, for a total of **v^2 edges**

- Undirected graphs
  - "Self-edges" are typically not allowed
  - Each vertex can be connected to each other vertex, but (i,j) == (j,i) so the total edges is ½ of the total number of vertex pairs
  - Assuming v vertices, each can connect to v-1 others
    - This gives a total of (v)(v-1) vertex pairs
    - But since (i,j) == (j,i), the total number of edges is (v)(v-1)/2
- **If e <= vlgv, we say the graph is SPARSE**
- **If $e \approx v^2$, we say the graph is DENSE**

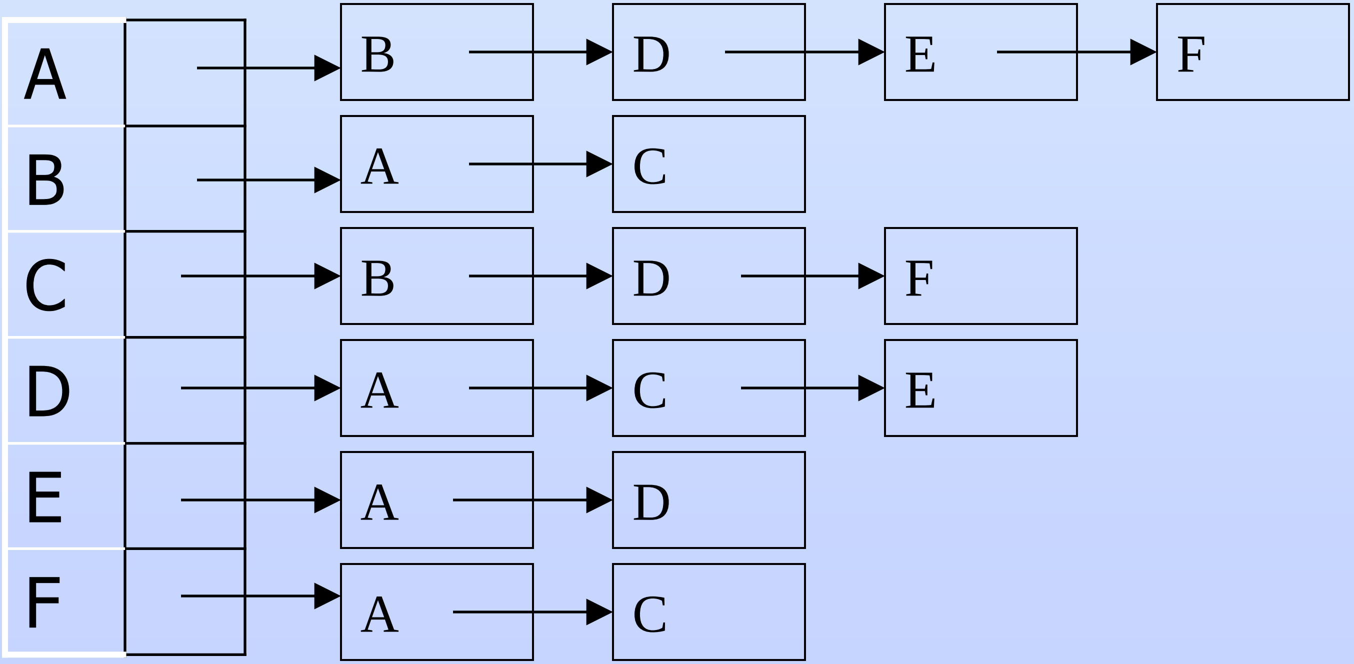
- Representing a graph on the computer
  - Most often we care only about the connectivity of the graph
    - Different representations in space of the same vertex pairs are considered to be the same graph
    - This is often but not always the case
  - Two primary representations of arbitrary graphs
    - Adjacency Matrix
      - Square matrix labeled on rows and columns with vertex ids
      - M[i][j] == 1 if edge (i,j) exists
      - M[i][j] == 0 otherwise

- Plusses:
  - Easy to use/understand
  - Can find edge (i,j) in Theta(1)
    - Ex: Is (B,C) in graph?
  - M^P gives number of paths of length P
- Minuses:
  - Theta(v^2) memory, regardless of e
  - Theta(v^2) time to initialize
  - Theta(v) to find neighbors of a vertex
    - Ex: Neighbors of D?

	A	B	C	D	E	F
A	0	1	0	1	1	1
B	1	0	1	0	0	0
C	0	1	0	1	0	1
D	1	0	1	0	1	0
E	1	0	0	1	0	0
F	1	0	1	0	0	0

Graph from slides 3,4

- Adjacency List
  - Array of linked lists
  - Each list [i] represents neighbors of vertex i



- Plusses:
  - Theta(e) memory
    - For sparse graphs this could be much less than v^2
  - Theta(d) to find neighbors of a vertex
    - d is the degree of a vertex (# of neighb)
    - For sparse graphs this could be much less than v
- Minuses
  - Theta(e) memory
    - For dense graphs, nodes will use more memory than simple location in adjacency matrix
    - Remember array multiway trie vs. dlb?
  - Theta(v) worst case to find one neighbor
    - Linked list gives sequential access to nodes – neighbor could be at end of the list

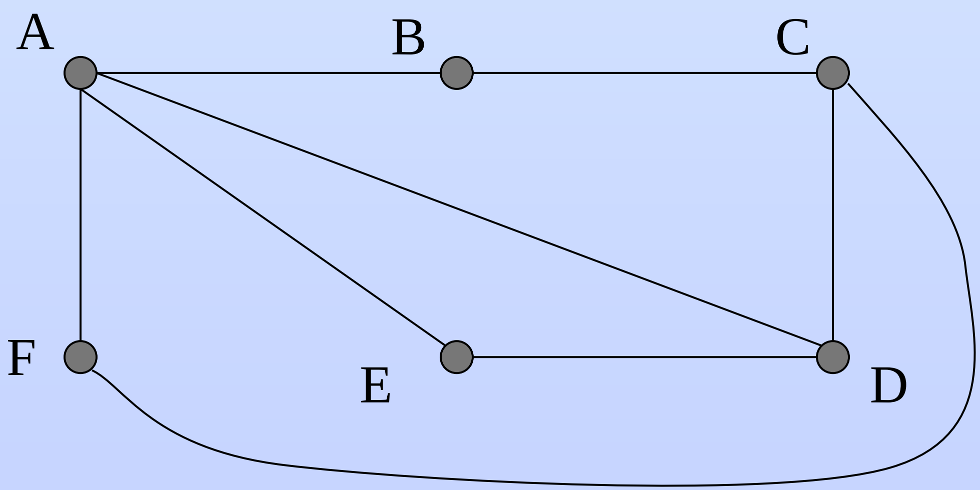
- Overall
  - Adjacency Matrix tends to be better for dense graphs
  - Adjacency List tends to be better for sparse graphs
    - We will compare these again when we look at some algorithms
  - Text uses adjacency list in examples
    - See Graph.java and next slide

```
public class Graph {
     private final int V;
     private int E;
     private Bag<Integer>[] adj;

     public void addEdge(int v, int w)
     {
          E++;
          adj[v].add(w);
          adj[w].add(v);
     }
...
}
public class Bag<Item> implements Iterable<Item> {
     private int N;
     private Node first;

     // helper linked list class
     private class Node {
          private Item item;
          private Node next;
     }
...
}
```

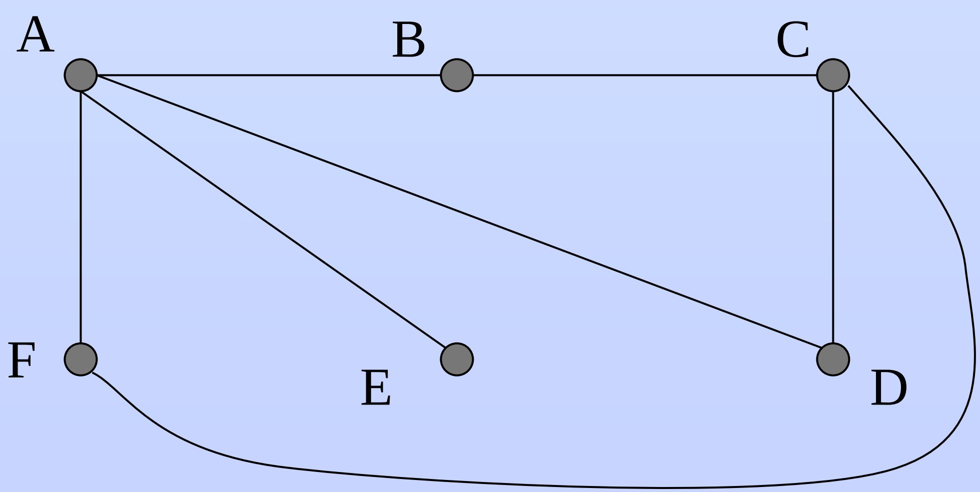
- A few more definitions…
  - Path: A sequence of adjacent vertices
  - Simple Path: A path in which no vertices are repeated
  - Simple Cycle: A simple path except that the last vertex is the same as the first



Path: ABC
Path: ADEAFC
Simple Path: ABCDE
Simple Cycle: ABCDEA

More Graph Definitions

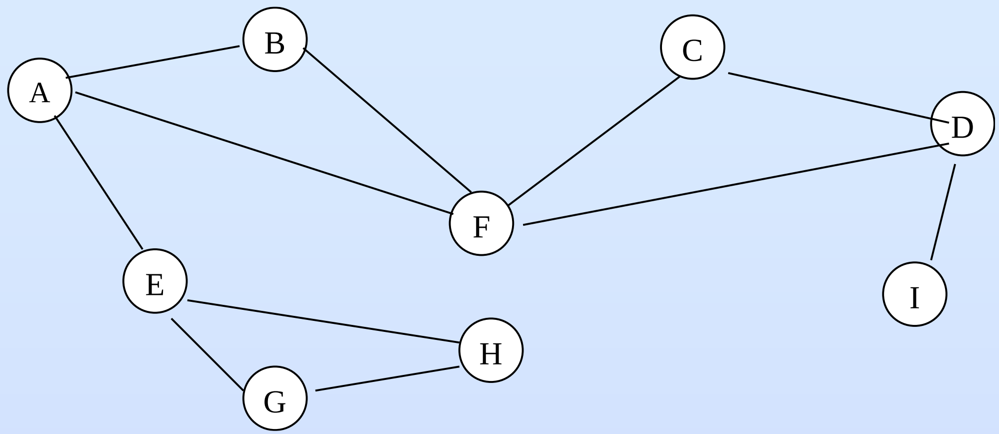
- Connected Graph: A graph in which a path exists between all vertex pairs
  - Connected Component: connected subgraph of a graph
- Acyclic Graph: A graph with no cycles
- Tree: A connected, acyclic graph
  - Has exactly v-1 edges



- This graph is connected
- If we remove (AB), (AD), (CF)
  - We now have two connected components:
    - {A,E,F} and {B,C,D}
- This graph is NOT a tree
  - It has cycles
- If we remove (AD), (CF)
  - It is a tree
  - It has (v-1) = 5 edges

- How to traverse a graph
  - Unlike linear data structures, it is not obvious how to systematically visit all vertices in a graph
  - Two famous, well-known traversals
    - Depth First Search (DFS)
      - Visit deep into the graph quickly, branching in other directions only when necessary
    - Breadth First Search (BFS)
      - Visit evenly in all directions
      - Visit all vertices distance i from starting point before visiting any vertices distance i+1 from starting point

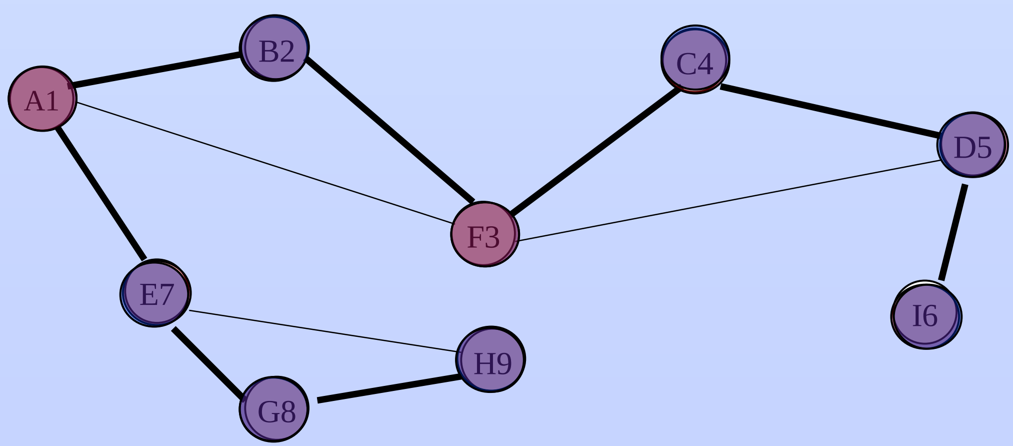
Original Graph



DFS

After DFS (bold edges traversed)

- Red circle: current vertex
- Blue circle: vertex has been visited



- Idea of DFS

```
Start at some vertex, V
Visit(V)

Visit(V)
   Mark V as visited
   foreach neighbor, W, of V
      if W is not visited
           Visit(W)
```

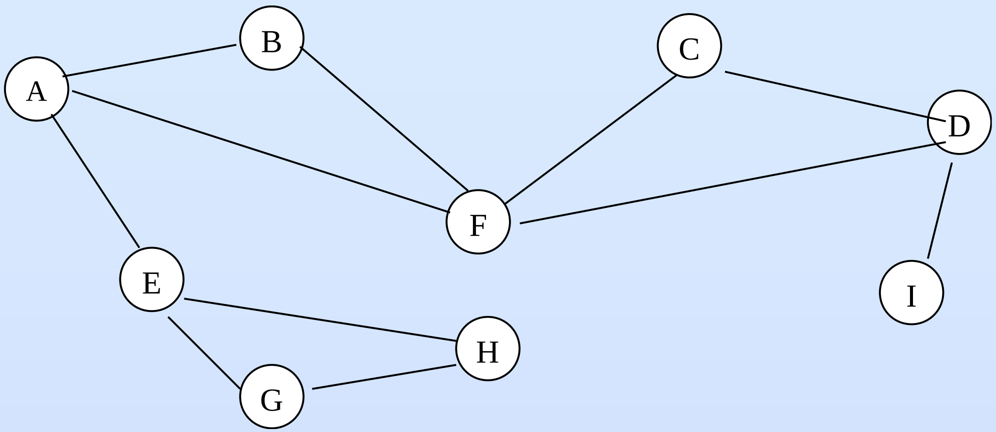
- Note that the order that neighbors of a vertex are stored is not specified
- In this trace we assume they are stored alphabetically by vertex
- Note that backtracking may be necessary to visit all vertices
  - Consider E, G, H
  - We backtrack to A before moving forward agai

- As you can see from the trace, DFS is usually done recursively
  - Current node recursively visits first unseen neighbor
  - What if we reach a "dead-end" (i.e. vertex with no unseen neighbors)?
    - Backtrack to previous call, and look for next unseen neighbor in that call
  - See also code in DepthFirstSearch.java
    - It is very similar to pseudocode in previous slide

- BFS is usually done using a queue
  - Current node puts all of its unseen neighbors into the queue
    - Vertices that have been seen but not yet visited are called the **FRINGE**
      - For BFS the fringe is the vertices in the queue
  - Front item in the queue is the next vertex to be visited
  - Algorithm continues until queue is empty
  - See trace in next slide
  - Also see code in BreadthFirstPaths.java

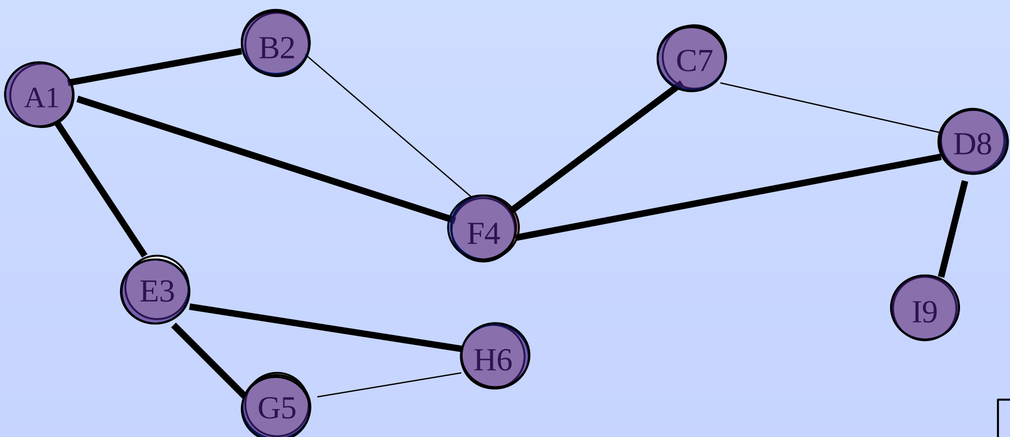
BFS

Original Graph



After BFS (bold edges traversed)

- Red circle: current vertex
- Blue circle: vertex has been visited



- Idea of BFS

```
Start at some vertex, V
Add V to the Q
while (!Q.isEmpty())
   Let V = Q.removeFront()
   Mark V as visited
   foreach neighbor, W, of V
      if W is unseen
         Mark W as seen
         Add W to the Q
```

- Assume same alphabetical ordering of neighbors as with DFS
- Vertices that have been seen but not visited are called the "fringe"
  - These are the vertices in the Q

Queue Front →

I	D	C	H	G	F	E	B	A

- Both DFS and BFS
  - Are initially called from "main" or other method
    - If the graph is connected, the main method will call DFS or BFS only one time, and (with a little extra code) a SPANNING TREE for the graph is built
    - If the graph is not connected, the main method would have to call DFS or BFS multiple times
      - First call of each will terminate with some vertices still unseen, and loop in "search" will iterate to the next unseen vertex, calling visit() [dfs() or bfs()] again
      - Each call (with a little extra code) will yield a spanning tree for a connected component of the graph
    - See DepthFirstPaths.java
    - See CC.java

Spanning Tree

```
public class DepthFirstPaths {
    private boolean[] marked;    // marked[v] = is there an s-v path?
    private int[] edgeTo;        // edgeTo[v] = last edge on s-v path
    private final int s;         // source vertex

    public DepthFirstPaths(Graph G, int s) {
        this.s = s;
        edgeTo = new int[G.V()];
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    // depth first search from v
    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }
```

- We are doing the DFS algorithm with an extra array – edgeTo[]
- For each vertex, when we visit it we mark it and we also note **where we are visiting from**
  - This is the "parent" vertex in the spanning tree
  - When the dfs method completes the edgeTo[] array will allow us to determine all edges in the spanning tree

```
if (edgeTo[i] == j)
then
    vertex j is the parent
    of vertex i in the
    spanning tree
```

Connected Components

```
public CC(Graph G) {
    marked = new boolean[G.V()];
    id = new int[G.V()];
    size = new int[G.V()];
    for (int v = 0; v < G.V(); v++) {
        if (!marked[v]) {
            dfs(G, v);
            count++;
        }
    }
}

private void dfs(Graph G, int v) {
    marked[v] = true;
    id[v] = count;
    size[v]++;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            dfs(G, w);
        }
    }
}
```

- id[v] indicates the connected component for vertex v, starting at 0
- There is one call to from CC to dfs() for each connected component
- If the graph is connected, all vertices will be in CC 0
  - In iteration 0 of loop, dfs() will be called
  - All nodes will be recursively visited and marked in dfs
  - When control returns to CC, marked[v] will be true for all vertices so no more calls to dfs() will be made
- Note that we could easily substitute "bfs" for "dfs"