CSCI 3110 Lecture Notes
Graph (1)

- **Definitions:**

1. **graph** – a set of vertices and edges that connect the vertices
   directed or undirected
2. **directed graph** – graph in which each edge is associated with an ordered pair of vertices (digraph)
3. **undirected graph** – graph in which each edge is associated with an unordered pair of vertices (undirected)
4. **adjacent** – a vertex v_1 in a graph is adjacent to v_2 if there is an edge from vertex v_2 to v_1
   directed
5. **path** – between two vertices is a sequence of edges that begins at one vertex and ends at another vertex, directed or undirected
6. **simple path** – a path that passes through each vertex only once directed or undirected
7. **cycle** – a path that begins and ends at the same vertex directed or undirected
8. **simple cycle** – a cycle that does not pass through other vertices more than once directed or undirected
9. **acyclic graph** – a graph without cycle directed
10. **connected graph** – there is a path between every two vertices (undirected)
11. **tree** – connected, acyclic graph with a specially designated node called the root undirected, typically, we do not consider the graph with two vertex a, and b, and an edge between them as a cyclic graph, ie., do not consider ab, ba to form a cycle
12. **complete graph** – A complete graph with *n* vertices (denoted K*n*) is a graph with *n* vertices in which each vertex is connected to each of the others (with one edge between each pair of vertices). Here are the first five complete graphs:

note difference between "binary tree is complete" vs. "complete graph"

13. **complete directed graph** – a directed graph of n vertices with exactly n*(n-1) edges.
14. **outdegree of a node** -- number of edges extending from the node (digraph)
15. **indegree of a node** – number of edges entering a node (digraph)
16. degree of a node – number of edges incident to a node (undirected)
17. **multigraph** – figure which has multiple occurrences of the same edge (2 or more edges between two vertices)
18. **network, or weighted graph** – graph in which each edge has an associated positive numerical weight
19. **strongly connected** – there is a path between every two vertices (digraph)
20. **weakly connected** – for every two vertices v_1 and v_2 in the graph, there is a path from vertex v_1 to v_2 or there is a path from v_2 to v_1 (digraph)

- Graph implementations and common graph operations

adjacency matrix → better for operation "Is there an edge from v_i to v_j"
adjacency list → better for operation "Find all vertices adjacent to v_i"

- **Graph traversal**

  a. A graph traversal visits all of the vertices that it can reach
     → a graph traversal visits all of the vertices in a graph if the graph is connected
     → If a graph is not connected, multiple traversals starting from unvisited node is capable of discovering the connected components of the graph

b. **depth first search (DFS)** – a graph traversal strategy in which a path from a vertex v proceeds as deeply into the graph as possible before backing up

```
DFS(in v:vertex)
{
      s.createStack()

      // push v onto the stack and mark it
      s.push(v)
      mark v as visited

      while (!s.isEmpty())
      {
          if (no unvisited vertices are adjacent to the vertex on the top of the stack)
                  s.pop()
          else
                  select an unvisited vertex u adjacent to the vertex on the top of the stack
                  s.push(u)
                  mark u as visited
      }
}
```

It is possible to get caught in a loop. To avoid loop, need to check the unvisited vertex u, visit it only if it has not been visited before.
DFS in a tree assumes no loop.

c. **breadth first search (BFS)** – a graph traversal strategy in which a path from a vertex v visits every vertex adjacent to v that it can before visiting any other vertex

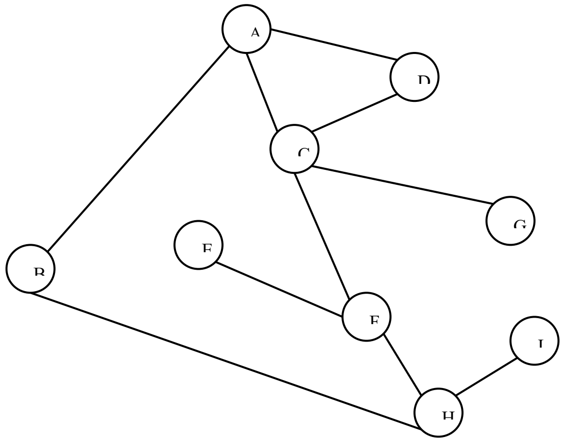
```
BFS(in v:vertex)
{
        q.CreateQueue()

        q.Enqueue(v)
        mark v as visited

        while (!q.IsEmpty())
        {
                q.Dequeue(w)

                for (each unvisited vertex u adjacent to w)
                {
                        mark u as visited
                        q.Enqueue(u)
                } // end for
        }
} // end while
```

Also susceptible to loop. (infinite)


A
D
C
G
E
B
F
I
H