Lecture 6 Note Wednesday, 02/23/2022

Announcements:
- The Lecture Recordings will be available on the following YouTube Playlists Link: https://youtube.com/playlist?list=PLZaTmV9UMKlgYpo2cAiMaEWxqyvbiXDFd

Graph
References:
Algorithm Design - Chapter 3

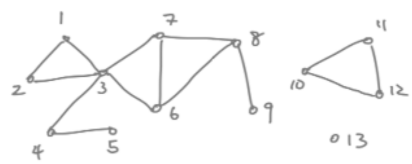
Basic Definitions
- A *graph* G is defined by a tuple (V,E).
  - V is the set of *vertices*, the nodes in a graph.
  - E is the set of *edges*, connections between two vertices. $E \subseteq V \times V$.
- In an *undirected graph* (which is the default), the edge (v_i, v_j) means: there is an edge from v_i to v_j and there is an edge from v_j to v_i.
- In a *directed graph* or *digraph*, (v_i, v_j) means there is an edge from v_i to v_j and (v_j, v_i) means there is an edge from v_j to v_i.
- Two vertices are *adjacent* if there is an edge between them.
- The neighbors of v is the set of vertices adjacent to v, denoted $Neighbor(v)$ or $N(v)$.
- The degree of v is the number of vertices connected to v, loop will count twice, denoted $degree(v)$ or $deg(v)$.
- A *simple graph* is a graph without loop, edge with (v,v), or multiple edges, edges connected same pair of vertices.
  - Let say there are n vertices and m edges, then number of edges in the simple graph will be $0 \leq m \leq \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$.
- Handshake Theorem:
  - $d_1 + d_2 + \cdots + d_n = 2 \cdot \#\, of\ edges$.
    $$\sum_{v \in V} \deg(v) = 2m$$
    - Give \$2 to every edge. Each edge then gives \$1 to its endpoints.

Examples of graphs in real life
- Transportation Networks
  - Vertices can represent airports or cities.
    Edges can represent the flights between airports or transportation between cities.
  - The edge can be directed for one-way flight, or it can be undirected for the two-way highway.
- Communication Networks
  - Vertices can represent computers. If two computers are connected in the network, then there is an edge between them.
- Information Networks – web graph
  - Vertices can represent websites. The edges can represent link from one webpage to another. The edge should be directed since a website can link to another, but the other website doesn’t need to link back.
- Social Networks
  - Would the Facebook graph be directed or undirected?
    - Undirected, two people are friend in Facebook if they are friend of each other.

      - The twitter graph will be directed, one person can follow another person, the other person doesn't need to follow back.
- Dependence Networks
  - For example, our major course list, the vertices can represent courses. The edges will represent the dependence, i.e., prerequisites.
  - Or in some large software system, you can have different functions represented by vertices, and the edges will be the dependency on one function will call on another function.

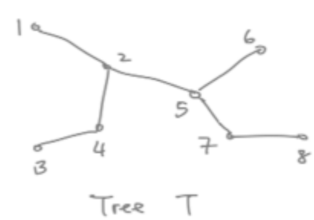
Graph Terminology

- A *path* P in an undirected graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_k$, such that (v_i, v_{i+1}) is an edge in G for $1 \leq i \leq k-1$.
  - Example:

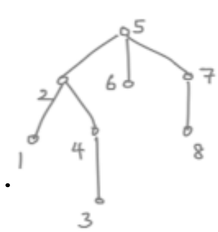


      - v_1, v_3, v_7, v_6, v_8 is a path (from v_1 to v_8)
      - v_2, v_3, v_5, v_4, v_3, v_6, v_9 is not a path, there is no edge between v_3 and v_5.
  - Edge vs path: an edge is a direct connection/link between two vertices. A path is a sequence of edges.
- A *simple path* does not revisit a vertex.
- A *circuit* is a path $v_1, v_2, \ldots, v_k$, where $v_1 = v_k$, which it ends at its starting vertex.
- A *cycle path* is a simple circuit, which it's a path $v_1, v_2, \ldots, v_k$ with $v_1 = v_k$ and $v_1, v_2, \ldots, v_{k-1}$ are distinct.
- Two vertices v_i and v_j are *connected* if there is a path from v_i to v_j in G.
- A graph G is *connected* if every pair of vertices in the graph G is connected. If a graph is not connected, *disconnected*, then you can break it up into components. A *component* is the maximumly connected set of vertices.
- Different types of graphs in simple undirected graph:
  - Path/Chain graph: n vertices and $n-1$ edges.
  - Cycle graph: n vertices and n edges.
  - Complete graph: n vertices and $m = \binom{n}{2} = \frac{n(n-1)}{2}$ edges.
  - Tree: a connected graph with no cycle. In other words, there is no way you can loop around.

Tree



  - There are many ways to draw a tree. The graph T is a tree, but it's not a typical tree you see.
  - You can choice any vertex in the graph and draw the graph with the root at this vertex.
  - T rooted at v_5



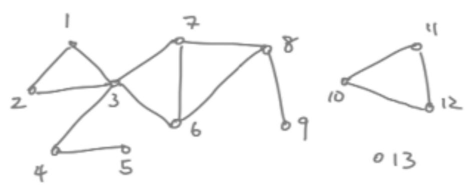
      - All the neighbors of v_5 will be *children* of v_5, you have v_2, v_6, and v_7. It doesn't matter which order you put down the children.
      - v_5 will be *parent* of v_2, v_6, and v_7.
      - The children of the same parent are called *siblings*.
      - The children of v and all the children of their children and so on are all *descendants* of v.
      - v_i is an *ancestor* of v_j if v_j is a descendant of v_i.
      - A leaf is the vertex with no children, i.e., v_1, v_3, v_6, v_8.

    - Tree can represent a lot of hierarchy relation.
- A tree with n vertices has exactly $n-1$ edges.
    - Proof: (parent-child relation)
        - Each vertex except the root has exactly one edge going up, connecting to its parent.
        - Each edge leads up from exactly one non-root vertex.

- Theorem: Let G be undirected graph on n vertices.
  Then any two of the following three statements imply the third statement.
    1) G is connected.
    2) G has no cycle.
    3) G has $n-1$ edged.
    - Combining any of the two statements, it will also give you the definition of tree.

Representing a graph

Let n be # of vertices and m be # of edges.



- Adjacency List

1	2,3
2	1,3
3	1,2,4,6,7
4	3,5
⋮	⋮
10	11,12
11	10,12
12	10,11
13	∅

    - You listed all the adjacent vertices (neighbors) of each vertex.
    - Number of neighbors of a vertex in a simple graph = its degree.
    - Size of Adjacency List: $n + 2m = O(n + m)$
    - m is at most $\binom{n}{2}$, which is n^2, but it can be much smaller. If G is a tree, then $O(n + m) = O(n)$.
    - We will prefer adjacency list when we try to save space.

- Adjacency Matrix

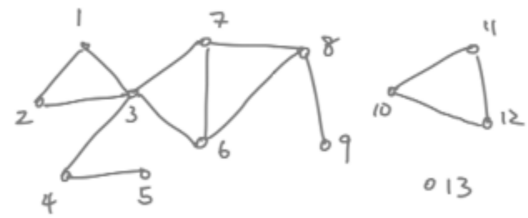
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

    - A n by n matrix where if there is an edge (v_i, v_j), then the entry v_i, v_j will be 1, otherwise 0.
    - Size of Adjacency Matrix: $n^2 = O(n^2)$
    - How long does it take you to check whether two vertices are adjacent in the adjacency list? the adjacency matrix?
        - List: At most $n-1$ (in simple graph).
        - Matrix: $O(1)$.

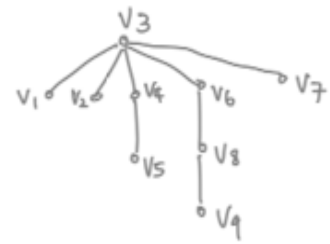
- Question: Do v_i and v_j have a path between them?
    - Connectivity Queues (are v_i and v_j connected?)
    - There are two algorithms can solve it.
        - BFS – Breadth First Search
        - DFS – Depth First Search

BFS (Breadth First Search) & DFS (Depth First Search)
- Input to both algorithms is the graph $G = (V, E)$ and a source/root node s.
- Output a tree rooted at s (containing all vertices connected to s).
- Both algorithms run in $O(n + m)$ time.

- However, they output different trees.
- BFS (Breadth First Search)
  - Starting at a node in the graph, the BFS algorithm explores all possible directions, computing one new layer in each step.
  - Layer L_1 consists of all nodes that are neighbors of s. (For notational reasons, we will sometimes use layer L_0 to denote the set consisting just of s.)
  - Assuming that we have defined layers $L_1, \ldots, L_j$, then layer L_{j+1} consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer L_j.



- Tree rooted at v_3 →



What to expect or prepare for the next class:
- BFS/DFS and properties of their result.
- Application of BFS
- Directed Graph/DAG

Reading Assignment
Algorithm Design: 3.1