Lecture Notes 8

Graphs

- Graph – Pairwise connections of vertices via edges
- Examples:
  - Map – Intersection and road
  - Web Content – Page and link
  - Circuit – Device and wire
  - Schedule – Job and constraint
  - Commerce – Customer and transaction
  - Network – Site and connection
  - Software – Method and call (or invocation)
  - Social Network – Person and friendship
- Undirected Graphs
  - Multigraphs – Parallel edges are allowed
  - Simple Graphs – No self loops or parallel edges
  - Adjacent Vertices – Two vertices directly connected via an edge
  - Degree Vertex – Number of edges incident to the vertex
  - Subgraph – Subset of the graphs edges and associated vertices
  - Path – Sequence of vertices connected by edges
  - Simple Path – Path with no repeated vertices
  - Cycle – Path with at least one edge whose first and last vertices are the same
  - Simple Cycle – Cycle with only beginning and end vertices repeated
  - Path Length – Number of edges in the path
  - Connected Vertices – A path exists between the vertices
  - Connected Graph – All vertices are connected to every other vertex
  - Acyclic Graph – Graph with no cycles
  - Tree – An acyclic graph
  - Forest – Disjoint set of trees
  - Spanning Tree – A subgraph of the connected graph containing all vertices
  - Spanning Forest – The union of spanning trees
  - Graph G with V vertices is a trees if any is true:
    - G has V – 1 edges and no cycles
    - G has V – 1 edges and is connected
    - G is connected but removing any edge disconnects it
    - G is acyclic, but adding any edge creates a cycle
    - Every pair of vertices in G has exactly one simple path connecting them
  - Bipartide Graph – Vertices can divide into two sets, and edges only connect between sets
  - Implementations
    - Adjacency Matrix – Boolean for every pair of edges
    - Array of Edges – Array containing the pairs of vertices

        - Array of Adjacency Lists – Array of vertex with list of adjacent nodes
    - Depth First Search (DFS) – Search that traverses deeper (or further away) from start first
        - At each vertex mark visited
        - Traverse edges to next unvisited node
        - Once all edges are exhausted return to previous node
    - Path Finding from Source
        - Use DFS from source node
        - Add previous edge to search when visiting node
        - Reconstruct path in reverse from destination to source
    - Breadth First Search (BFS) – Visit vertices in order of further distance away
        - Edge count BFS can be implemented with FIFO
    - Connected Components
        - Use DFS to find all reached from first vertex
        - Repeat for each unmarked vertex
    - Symbol Graph
        - Vertices are named
        - Use symbol table to translate name to vertex index
        - Have vector of names to translate back
    - Degrees of Separation – Use BFS to find the distance from source to all others
- Directed Graphs – Edges are directed
    - Indegree – Number of edges coming in to vertex
    - Outdegree – Number of edges leaving the vertex
    - Reachability – Use DFS from single source to find all reachable vertices
    - Directed Acyclic Graphs (DAGs) – A directed graph with no cycles
    - Scheduling Problems – Directed edges are placed between a task and a dependency
    - Precedence Constrained Scheduling – Scheduling tasks by their precedence
    - Topological Sort – Order vertices in order such that directed edges point from earlier vertex
        - Post order of DFS Order
    - Strong Connectivity – All vertices are strongly connected
        - Strongly Connected Vertices – For v and w, both can reach each other (only exists if a directed cycle contains both of them)
        - Kosaraju-Sharir Algorith
            - Calculate Post order of DFS Order from G_R
            - Do DFS Order of G but with the previously calculated Post order
- Minimum Spanning Trees
    - Edge Weighted Graph – Graph where edges have weights
    - Minimum Spanning Tree – Spanning tree of a graph with minimum weight sum
    - Cut – Partitioning of graph vertices into two nonempty disjoint sets
    - Crossing Edge – Connects vertex in one partition to another of a cut graph
    - Edge Weighted Graph Implementation – Store weight instead of bool for edges
    - Prim's Algorithm – Grow the tree by adding the minimum cut edge until tree is built
        - Use vector to store added vertices

        - Use vector to store the edges of the MST
        - Use a min heap to store the candidate edges by weight
        - Lazy Prim's O(E lg E)
            - For every new vertex added put all edges to unattached vertices in min heap
            - While edges in min heap consider adding new vertex if hasn't already been added
        - Eager Prim's O(E lg V) time, O(V) extra space compared to Lazy
            - Improvement can avoid adding to min heap if saw a better candidate to add in future
    - Kruskal's Algorithm – Add edges that do not form a cycle, consider all edges in sorted order
        - O(E lg E) time, O(E) space
        - Use priority queue for edges
        - Use find-union structure for detecting cycles
- Shortest Paths
    - Edge-Weighted Digraphs – Directed graph with weights attached to each edge
    - Shortest Path – Directed path from vertex *s* to *t* where no other path has lower weight
    - Shortest Path Assumed Properties
        - Paths are directed
        - Weights are not necessarily distance
        - Not all vertices may be reachable
        - Paths may repeat vertices and edges
        - Shortest paths normally are simple
        - There isn't necessarily a single shortest path
        - Parallel edges and self-loops may be present
        - Negative weights introduce complications
    - Single Source Shortest Path
        - Edge relaxation – When a better path is found from *s* to *w* via *v*, the path weight is updated to go through *v*
        - Dijkstra's Algorithm – Solves SSSP with no negative weights
            - Requires O(V) space and O(E lg V) time
            - Initialize all distances as infinite except source which is zero
            - Initialize all previous nodes to invalid
            - Insert the source vertex in priority queue (sorted by distance
            - While have vertices in priority queue
            - Get nearest vertex and check if any edges can be relaxed, add unvisited nodes to priority queue
        - Edge Weighted DAG
            - O(E + V) time
            - Sort vertices in topological order
            - Consider each edge and relax the edges as necessary
            - Longest path possible by negating path weight and finding shortest path (useful for finding critical path in scheduling)
        - Bellman Ford – Solves SSSP negative weights allowed (no negative cycles)

- Requires O(V) space and O(E·V) time
- Initialize all distances as infinite except source which is zero
- Initialize all previous nodes to invalid
- Consider each edge in any order, and do V passes relaxing where necessary