505 22240 / ESOE 2012 Data Structures: Lecture 11
Graph Traversals and Weighted Graphs

◎Graph Traversals

・Visits each vertex once.

① Depth-first search (DFS): starts at an arbitrary vertex and searches a graph as "deeply" as possible as early as possible. ⇨ **preorder**

② Breadth-first search (BFS): starts by visiting an arbitrary vertex, then visits all vertices whose distance from the starting vertex is one, then all vertices with distance two, and so on. ⇨ **level-order**

① Depth-first search (DFS)

・Each vertex has boolean "visited" field that tells us if we've visited it before.

・Assume a strongly connected graph: there is a path from starting vertex to every other vertex.

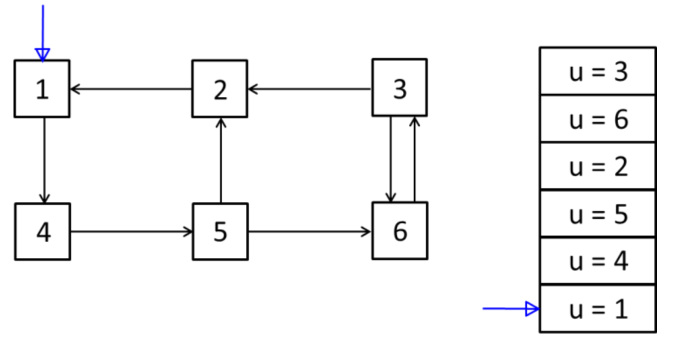
・Code:

```
class Graph {
    void dfs(Vertex* u) {
        u->visit( );
        u->visited = true;            // Mark the vertex u visited
        for (each vertex v such that (u, v) belongs to E) {
            if (!v->visited) {
                dfs(v);
            }
        }
    }
};
```

・A "visit( )" method is defined that performs some action on a specified vertex, e.g., count total population of the city graph above.

・e.g. street map: starting at vertex 1

⇨ Use stack to keep track of vertices.



・Runs in O(|V|+|E|) time with adjacency list.

・Runs in $O(|V|^2)$ time with adjacency matrix.

② Breadth-first search (BFS)

・We use a queue, so vertices are visited by distance from starting vertex.

・Code:

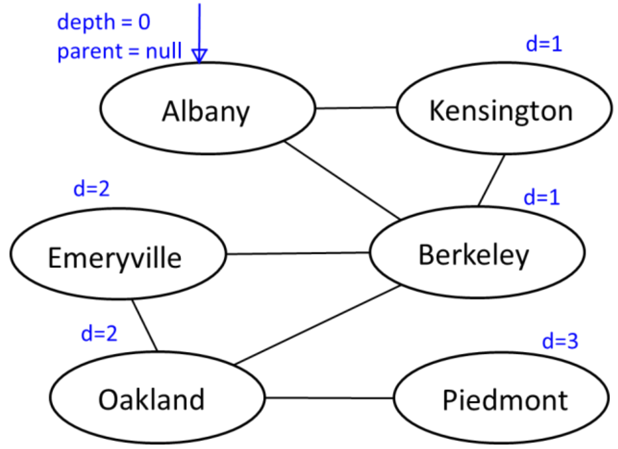
```
void bfs(Vertex* u) {
    u->visit(NULL);   // pass edge's origin vertex as a parameter
    u->visited = true;
    q = new Queue;
    q->enqueue(u);        // initially containing u
    while (q is not empty) {
        v = q->dequeue( );
        for (each vertex w such that (v, w) is an edge in E) {
```

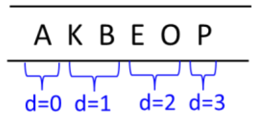
```
            if (!w->visited) {
                w->visit(v);
                w->visited = true;
                q->enqueue(w);
            }
        }
    }
}

class Vertex {
public:
    Vertex* parent;
    int depth;

    void visit(Vertex* origin) {
        this->parent = origin;
        if (origin == NULL) {
            this->depth = 0;
        } else {
            this->depth = origin->depth + 1;
        }
    }
};
```

· When edge (v, w) is traversed to visit w, depth of w = depth of v + 1, and v becomes the “parent” of w.

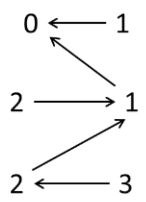
· e.g. city adjacency graph: starting from Albany.

⇨ Queue is shown in the right diagram.





· Find shortest path from any vertex to start vertex by following parent pointers.



· BFS runs in O(|V|+|E|) time with adjacency list.

· BFS runs in $O(|V|^2)$ time with adjacency matrix.

§ Weighted Graphs

· A graph with each edge labeled with a numerical weight.

→ express distance between two nodes, cost moving from one to other, resistance between two points,…etc.

① Adjacency matrix: array of ints / doubles / whatever.

② Adjacency list: each list node includes a weight.

★Two problems:

① Shortest path problem: (*Read It Yourself*)

② Minimum spanning tree:

Suppose you're wiring a house for electricity:

Each node is an outlet, or source of electricity.

Edges labeled with length of wire.

Connect all nodes with shortest length of wire?

◎Kruskal's Algorithm

・Let G = (V, E): undirected graph.

・"spanning tree" T = (V, F) of G is a graph with same vertices as G, and |V|-1 edges of G that form a tree.

・If G is not connected, T is a forest, a collection of trees.

・If G is weighted, a "minimum spanning tree" T of G is a spanning tree of G whose total weight (summed over all edges of T) is minimal.

① Create a new graph T with same vertices as G, but no edges (yet).

② Make list of all edges in G.

③ Sort edges by weight, lowest to highest.

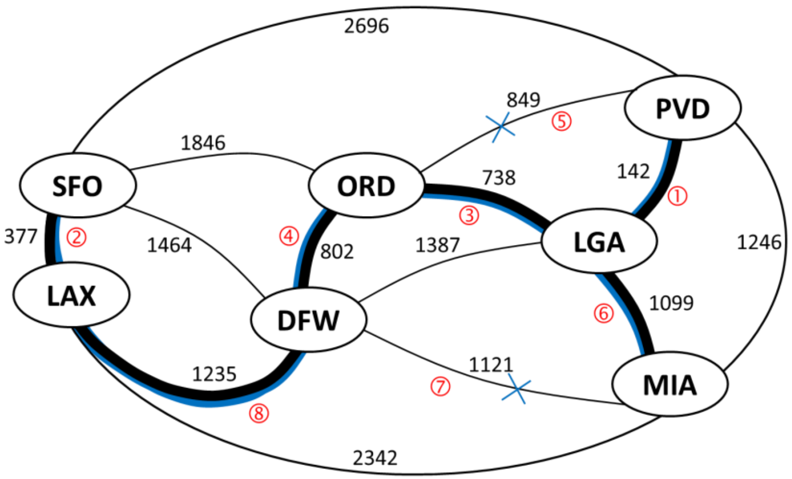
④ Iterate through edges in sorted order.

For each edge (u, w):

4a If u and w are not connected by a path in T, add (u, w) to T.

★Never adds (u, w) if some path connects u and w, T is guaranteed to be a tree (if G is connected) or forest (not).

・e.g.



・Running time:

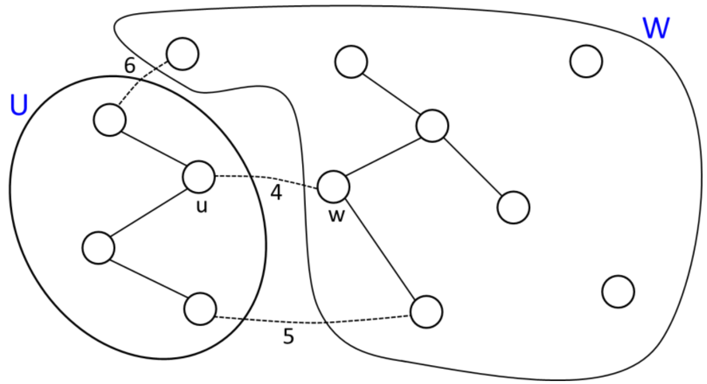
①~②: O(|V|) time

③: Sorting |E| edges takes O(|E| log|E|)

4a : < O(|E| log|E|)

・Kruskal's algorithm runs in O(|V|+|E| log |E|) time = O(|V|+|E| log |v|) time.

★Why does it work?



T in progress:

Considering adding an edge (u, w) to T.

Let U be set of nodes that have a path to u, W be all other nodes, including w.

⇨ (u, w) has the shortest length among all unconnected edges between U and W (due to sorted edges).