ECE4050
Data Structures and Algorithms

Lecture 8: Graphs
A graph $G = (V, E)$ consists of a set of vertices $V$, and a set of edges $E$, such that each edge in $E$ is a connection between a pair of vertices in $V$.

The number of vertices is written $|V|$, and the number of edges is written $|E|$.
Paths and Cycles

Path: A sequence of vertices $v_1, v_2, \ldots, v_n$ of length $n-1$ with an edge from $v_i$ to $v_{i+1}$ for $1 \leq i < n$.

A path is simple if all vertices on the path are distinct.

A cycle is a path of length 3 or more that connects $v_i$ to itself.

A cycle is simple if the path is simple, except the first and last vertices are the same.
Figure 11.1 Examples of graphs and terminology. (a) A graph. (b) A directed graph (digraph). (c) A labeled (directed) graph with weights associated with the edges. In this example, there is a simple path from Vertex 0 to Vertex 3 containing Vertices 0, 1, and 3. Vertices 0, 1, 3, 2, 4, and 1 also form a path, but not a simple path because Vertex 1 appears twice. Vertices 1, 3, 2, 4, and 1 form a simple cycle.
Connected Components

An undirected graph is connected if there is at least one path from any vertex to any other.

The maximum connected subgraphs of an undirected graph are called connected components.

![Graph](image)

**Figure 11.2** An undirected graph with three connected components. Vertices 0, 1, 2, 3, and 4 form one connected component. Vertices 5 and 6 form a second connected component. Vertex 7 by itself forms a third connected component.
Directed Representation

Figure 11.3 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).
Undirected Representation

Which one is more efficient in terms of space or time?

Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).
Representation Costs

Adjacency Matrix: $\Theta(|V|^2)$ space.

Adjacency List: $\Theta(|V| + |E|)$ space.

$|E|$ could be as low as 0, or as high as $\Theta(|V|^2)$. Which representation actually requires less space depends on the number of edges.
Graph ADT

interface Graph {                 // Graph class ADT
    public void Init(int n);  // Initialize
    public int n();           // # of vertices
    public int e();           // # of edges
    public int first(int v);  // First neighbor
    public int next(int v, int w); // Neighbor
    public void setEdge(int i, int j, int wght);
    public void delEdge(int i, int j);
    public boolean isEdge(int i, int j);
    public int weight(int i, int j);
    public void setMark(int v, int val);
    public int getMark(int v);  // Get v's Mark
}

for (w = G->first(v); w < G->n(); w = G->next(v,w))
Graph Traversals

Some applications require visiting every vertex in the graph exactly once.

The application may require that vertices be visited in some special order based on graph topology.

Examples:
  • Artificial Intelligence Search
  • Shortest paths problems
Graph Traversals (2)

To insure visiting all vertices:

```c
void graphTraverse(Graph G) {
    int v;
    for (v=0; v<G.n(); v++)
        G.setMark(v, UNVISITED), // Initialize to make sure no second visit
    for (v=0; v<G.n(); v++)
        if (G.getMark(v) == UNVISITED)
            doTraverse(G, v);
}
```
Depth First Search (1)

// Depth first search
void DFS(Graph G, int v) {
    PreVisit(G, v);  // Take appropriate action
    G.setMark(v, VISITED);
    for (int w = G.first(v); w < G.n();
        w = G.next(v, w))
        if (G.getMark(w) == UNVISITED)
            DFS(G, w);
    PostVisit(G, v); // Take appropriate action
}
Depth First Search (2)

Cost: $\Theta(|V| + |E|)$. 
Breadth First Search (1)

Like DFS, but replace stack with a queue.
  • Visit vertex’s neighbors before continuing deeper in the tree.
Breadth First Search (2)

```java
void BFS(Graph G, int start) {
    Queue<Integer> Q = new AQueue<Integer>(G.n());
    Q.enqueue(start);
    G.setMark(start, VISITED);
    while (Q.length() > 0) { // For each vertex
        int v = Q.dequeue();
        PreVisit(G, v); // Take appropriate action
        for (int w = G.first(v); w < G.n();
            w = G.next(v, w))
            if (G.getMark(w) == UNVISITED) {
                // Put neighbors on Q
                G.setMark(w, VISITED);
                Q.enqueue(w);
            }
        PostVisit(G, v); // Take appropriate action
    }
}
```
Breadth First Search (3)
Problem: Given a set of jobs, courses, etc., with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.

This is modeled as a DAG (Directed Acyclic Graph)

The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a topological sort.

An acceptable topological sort for this example is J1, J2, J3, J4, J5, J6, J7.
Topological Sort based on DFS (2)

```java
void topsort(Graph G) {
    for (int i=0; i<G.n(); i++)
        G.setMark(i, UNVISITED);
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED)
            tophelp(G, i);
}

void tophelp(Graph G, int v) {
    G.setMark(v, VISITED);
    for (int w = G.first(v); w < G.n(); w = G.next(v, w))
        if (G.getMark(w) == UNVISITED)
            tophelp(G, w);
    printout(v);
}
```
void topsort(Graph G) {
    Queue<Integer> Q = new AQueue<Integer>(G.n());
    int[] Count = new int[G.n()];
    int v, w;
    for (v=0; v<G.n(); v++) Count[v] = 0;
    for (v=0; v<G.n(); v++)
        for (w=G.first(v); w<G.n(); w=G.next(v, w))
            Count[w]++;
    for (v=0; v<G.n(); v++)
        if (Count[v] == 0) Q.enqueue(v);
    while (Q.length() > 0) {
        v = Q.dequeue().intValue();
        printout(v);
        for (w=G.first(v); w<G.n(); w=G.next(v, w)) {
            Count[w]--;
            if (Count[w] == 0)
                Q.enqueue(w);
        }
    }
}
Shortest Paths Problems

Input: A graph with weights or costs associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:
- Find shortest path between two named vertices
- Find shortest path from S to all other vertices
- Find shortest path between all pairs of vertices
Shortest Paths Definitions

d(A, B) is the shortest distance from vertex A to B.

w(A, B) is the weight of the edge connecting A to B.

• If there is no such edge, then w(A, B) = ∞.
Single-Source Shortest Paths

Given start vertex \( s \), find the shortest path from \( s \) to all other vertices.

Visit vertices in some order, compute shortest paths for all vertices seen so far, then add shortest path to next vertex \((x)\).

Problem: Shortest path to a vertex already processed might go through \( x \).

Solution: Process vertices in order of distance from \( s \).
Example Graph
## Dijkstra’s Algorithm Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Process A</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>20</td>
<td>∞</td>
</tr>
<tr>
<td>Process C</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>Process B</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process D</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>Process E</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>18</td>
</tr>
</tbody>
</table>
Dijkstra’s Implementation

// Compute shortest path distances from s, store them in D
void Dijkstra(Graph G, int s, int[] D) {
    for (int i=0; i<G.n(); i++) // Initialize
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) {
        int v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (D[v] == Integer.MAX_VALUE) return;
        for (int w = G.first(v); w < G.n();
             w = G.next(v, w))
            if (D[w] > (D[v] + G.weight(v, w)))
                D[w] = D[v] + G.weight(v, w);
    }
}
Implementing minVertex

Issue: How to determine the next-closest vertex? (I.e., implement minVertex)

Scan through the table of current distances.

• Cost: $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$. 
int minVertex(Graph G, int[] D) {
    int v = 0;  // Initialize to unvisited vertex;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED)
            { v = i; break; }
    for (int i=0; i<G.n(); i++)
        // Now find smallest value
        if ((G.getMark(i) == UNVISITED) &&
            (D[i] < D[v]))
            v = i;
    return v;
}
Minimal Cost Spanning Trees

Minimal Cost Spanning Tree (MST) Problem:

Input: An undirected, connected graph G.
Output: The subgraph of G that
1) has minimum total cost as measured by summing the values of all the edges in the subset, and
2) keeps the vertices connected.

The number of edges of MST for G(V, E) is: $|V| - 1$
MST Example

(1) Start with any Vertex N in the graph, setting the MST to be N initially.
(2) At each step expanding the MST by selecting the least-cost edge from a vertex currently in the MST to a vertex not currently in the MST.
Prim’s MST Algorithm

// Compute a minimal-cost spanning tree
void Prim(Graph G, int s, int[] D, int[] V) {
    int v, w;
    for (int i=0; i<G.n(); i++)   // Initialize
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) {
        v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v);
        if (D[v] == Integer.MAX_VALUE) return;
        for (w=G.first(v); w<G.n(); w=G.next(v, w))
            if (D[w] > G.weight(v, w)) {
                D[w] = G.weight(v, w);
                V[w] = v;
            }
    }
}
Kruskal’s MST Algorithm (1)

Initially, each vertex is in its own MST.

Merge two MST’s that have the shortest edge between them.
  • Use a priority queue to order the unprocessed edges. Grab next one at each step.

How to tell if an edge connects two vertices already in the same MST?
  • Use the UNION/FIND algorithm with parent-pointer representation.
Kruskal’s MST Algorithm (2)

Initial: A | B | C | D | E | F

Step 1: A | B | C | 1 | E | F
Process edge (C, D)

Step 2: A | B | C | 1 | E | 1
Process edge (E, F)

Step 3: A | B | C | 1 | D | 2
Process edge (C, F)
Exercises

Prove the following implications regarding free trees.

(a) IF an undirected graph is connected and has no simple cycles, THEN the graph has $|V| - 1$ edges.

(b) IF an undirected graph has $|V| - 1$ edges and no cycles, THEN the graph is connected.
Exercises

(a) Draw the adjacency matrix representation for the graph of the figure.
(b) Draw the adjacency list representation for the same graph.
(c) If a pointer requires four bytes, a vertex label requires two bytes, and an edge weight requires two bytes, which representation requires more space for this graph?
(d) If a pointer requires four bytes, a vertex label requires one byte, and an edge weight requires two bytes, which representation requires more space for this graph?