Graphs
Chapter 9 in Weiss

Today's Outline
• Announcements
  – HW#5 due NOW
  – Midterm #2 – this Friday
    • Topics, sample exam posted

• Graphs

Graph... ADT?
• Not quite an ADT…
  operations not clear

• A formalism for representing
  relationships between objects

Graph \(G = (V, E)\)
  – Set of vertices:
    \(V = \{v_1, v_2, \ldots, v_n\}\)
  – Set of edges:
    \(E = \{e_1, e_2, \ldots, e_m\}\)
  where each \(e_i\) connects two
  vertices \((v_{i1}, v_{i2})\)

Graph Definitions
In directed graphs, edges have a specific direction:

Han \(\rightarrow\) Leia

In undirected graphs, they don’t (edges are two-way):

v is adjacent to u if \((u, v) \in E\)

More Definitions:
Simple Paths and Cycles
A simple path repeats no vertices (except that the first can be the last):

\(p = \{\text{Seattle, Salt Lake City, San Francisco, Dallas}\}\)
\(p = \{\text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle}\}\)

A cycle is a path that starts and ends at the same node:

\(p = \{\text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle}\}\)
\(p = \{\text{Seattle, Salt Lake City, Seattle, San Francisco, Seattle}\}\)

A simple cycle is a cycle that repeats no vertices except
that the first vertex is also the last (in undirected
graphs, no edge can be repeated)

Trees as Graphs
• Every tree is a graph!
• Not all graphs are trees!

A graph is a tree if
  – There are no cycles
    (directed or undirected)
  – There is a path from the
    root to every node
Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

Aside: If program call-graph is a DAG, then all procedure calls can be in-lined.

Graph Connectivity

Undirected graphs are connected if there is a path between any two vertices.

Directed graphs are strongly connected if there is a path from any one vertex to any other.

Directed graphs are weakly connected if there is a path between any two vertices, ignoring direction.

A complete graph has an edge between every pair of vertices.

Graph Representations

0. List of vertices + list of edges
1. 2-D matrix of vertices (marking edges in the cells) "adjacency matrix"
2. List of vertices each with a list of adjacent vertices "adjacency list"

Things we might want to do:
• iterate over vertices
• iterate over edges
• iterate over vertices adj. to a vertex
• check whether an edge exists

Some Applications: Moving Around Washington

What’s the shortest way to get from Seattle to Pullman?

Some Applications: Reliability of Communication

If Wenatchee’s phone exchange goes down, can Seattle still talk to Pullman?
Some Applications: Bus Routes in Downtown Seattle

If we're at 3rd and Pine, how can we get to 1st and University using Metro?

Representation 1: Adjacency Matrix

A $|V| \times |V|$ array in which an element $(u, v)$ is true if and only if there is an edge from $u$ to $v$.

Representation 2: Adjacency List

A $|V|$-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices.

Representation

- adjacency matrix:
  $$A[u][v] = \begin{cases} \text{weight} & \text{if } (u, v) \in E \\ 0 & \text{if } (u, v) \notin E \end{cases}$$

- adjacency list:
Application: Topological Sort

Given a directed graph, $G = (V,E)$, output all the vertices in $V$ such that no vertex is output before any other vertex with an edge to it.

Is the output unique?

Valid Topological Sorts:

```cpp
void Graph::topsort(){
    Vertex v, w;
    labelEachVertexWithItsInDegree();
    for(int count=0; count<NUM_VERTICES; count++) {
        v = findNewVertexOfDegreeZero();
        v.topoNum = count;
        for each w adjacent to v
            w.indegree--;
    }
}
```

```cpp
void Graph::topsort(){
    Vertex v, w;
    labelEachVertexWithItsInDegree();
    for(int count=0; count<NUM_VERTICES; count++) {
        v = findNewVertexOfDegreeZero();
        v.topoNum = count;
        for each w adjacent to v
            w.indegree--;
    }
}
```

```cpp
void Graph::topsort(){
    Queue q(NUM_VERTICES); int counter = 0; Vertex v, w;
    labelEachVertexWithItsInDegree();
    q.makeEmpty();
    for each vertex v
        if (v.indegree == 0)
            q.enqueue(v);
    while (!q.isEmpty()) {
        v = q.dequeue();
        v.topologicalNum = ++counter;
        for each w adjacent to v
            if (--w.indegree == 0)
                q.enqueue(w);
    }
}
```

Runtime: