CSE 326: Data Structures

Graphs

James Fogarty
Spring 2009
Graphs

Formalism representing relationships among objects

Graph \( G = (V, E) \)

- **Set of vertices**
  (aka nodes):
  \( V = \{v_1, v_2, \ldots, v_n\} \)

- **Set of edges**:
  \( E = \{e_1, e_2, \ldots, e_m\} \)
  where each \( e_i \) connects one vertex to another \( (v_j, v_k) \)

Graphs can be *directed* or *undirected*
Graphs

• Graphs are not quite an ADT
  – Operations are unclear

• Many algorithms developed for graphs

• Many important problems can be solved by formulating them as graphs, then applying a standard graph algorithm
Examples of Graphs

• The web
  – Vertices are webpages
  – Each edge is a link from one page to another

• Social networks
  – Vertices are people
  – Edges connect friends

• Call graph of a program
  – Vertices are subroutines
  – Edges are calls and returns
**Undirected Graphs**

In *undirected* graphs, edges have no specific direction (edges are always two-way):

Thus, \((u, v) \in E\) implies \((v, u) \in E\). Only one of these edges needs to be in the set; the other is implicit.

*Degree* of a vertex: number of edges containing that vertex. (Same as number of adjacent vertices.)
Directed Graphs

In *directed* graphs (aka *digraphs*), edges have a specific direction:

Thus, \((u,v) \in E\) does not imply \((v,u) \in E\).

*In-degree* of a vertex: number of inbound edges.  
*Out-degree* of a vertex: number of outbound edges.
Graphs

Notation

\[ |\mathbf{V}| = \text{number of vertices} \]
\[ |\mathbf{E}| = \text{number of edges} \]

- \( \mathbf{v} \) is adjacent to \( \mathbf{u} \) if \( (\mathbf{u}, \mathbf{v}) \in \mathbf{E} \)
  - neighbor of = adjacent to
  - Order matters for directed edges

- It is possible to have an edge \( (\mathbf{v}, \mathbf{v}) \)
  - This edge type called a loop.
Weighted Graphs

Each edge has an associated weight or cost.

![Diagram of weighted graph with cities and weights]

- Clinton to Mukilteo: 20
- Kingston to Edmonds: 30
- Bainbridge to Seattle: 35
- Bremerton to Seattle: 60
Paths and Cycles

- A *path* is a list of vertices \( \{v_1, v_2, ..., v_n\} \) such that \((v_i, v_{i+1}) \in E\) for all \(0 \leq i < n\).
- A *cycle* is a path that begins and ends at the same node.
- A graph is *acyclic* if it contains no cycles.

\[ p = \{Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle\} \]
Path Length and Cost

- **Path length**: the number of edges in the path
- **Path cost**: the sum of the costs of each edge

For path $P$:
- $\text{length}(P) = 5$
- $\text{cost}(P) = 11.5$

How would you ensure $\text{length}(p) = \text{cost}(p)$ for all $p$?
Simple Paths and Cycles

A *simple path* repeats no vertices (except that the first can also 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 is also a simple path
Paths/Cycles in Directed Graphs

Consider this directed graph:

```
A ----> B ----> C ----> D
     |          |
     |          |
     V          V
     B ----> C
```

Is there a path from A to D? No

Does the graph contain any cycles? No
Undirected Graph Connectivity

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

**Connected graph**

**Disconnected graph**

A *complete undirected* graph has an edge between every pair of vertices:

(Complete = *fully connected.*)
Directed Graph Connectivity

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 directed* graph has a directed edge between every pair of vertices. (Again, complete = *fully connected*.)
A tree is a graph that is:
- *undirected*
- *acyclic*
- *connected*

That don’t look like any tree I ever seen
Rooted Trees

We are more accustomed to:

- Rooted trees (a tree node that is “special”)
- Directed edges from parents to children (parent closer to root).

A rooted tree (root indicated in red) drawn two ways

Rooted tree with directed edges from parents to children.

Characteristics of this graph?

- Directed, acyclic, weakly connected, path from root to every other node
Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

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

\[{ \text{Rooted, directed tree} } \subseteq \{ \text{DAG} \} \subseteq \{ \text{Graph} \} \]
How many edges $|E|$ in a graph with $|V|$ vertices?

$0 \leq |E| \leq |V|^2$

What if the graph is directed?

$0 \leq |E| \leq 2|V|^2$

What if it is undirected and connected?

$|V|-1 \leq |E| \leq |V|^2$

Can the following bounds be simplified?

- Arbitrary graph: $O(|E| + |V|^2)$
- Undirected, connected: $O(|E| \log|V| + |V| \log|V|)$

Some (semi-standard) terminology:

- A graph is *sparse* if it has $O(|V|)$ edges (upper bound).
- A graph is *dense* if it has $\Theta(|V|^2)$ edges.
What’s the data structure?

- Think about what we want to support
- What is the common query?
- Which edges are adjacent to a vertex?
Representation 1: Adjacency Matrix

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

Runtimes:
- Iterate over vertices? \(O(|V|)\)
- Iterate over edges? \(O(|V|^2)\)
- Iterate edges adj. to vertex? \(O(|V|)\)
- Existence of edge? \(O(1)\)

Space requirements? \(O(|V|^2)\)
Best for what kinds of graphs? dense
Representation 2: Adjacency List

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

**Runtimes:**
- Iterate over vertices? $O(|V|)$
- Iterate over edges? $O(|E|)$
- Iterate edges adj. to vertex? $O(d)$
- Existence of edge? $O(d)$
- Space requirements? $O(|V|+|E|)$
- Best for what kinds of graphs? sparse
Representing Undirected Graphs

What do these representations look like for an undirected graph?

Adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency list:

A
B
A
C
B
C  D
D  C
Some Applications: Moving Around Washington

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

Edge labels:
What’s the *fastest way* to get from Seattle to Pullman?

Edge labels:

Distance, speed limit
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? How about 4th and Seneca?
Application: Topological Sort

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

What kind of input graph is allowed? \( \text{DAG} \)

Is the output unique? No, often called a partial ordering
Topological Sort: Take One

1. Label each vertex with its *in-degree* (# of inbound edges)

2. **While** there are vertices remaining:
   a. Choose a vertex \( v \) of *in-degree zero*; output \( v \)
   b. Reduce the in-degree of all vertices adjacent to \( v \)
   c. Remove \( v \) from the list of vertices

*Runtime:* \( O(|V|^2 + |E|) \) \( O(|V|^2) \)
void Graph::topsort()
{
    Vertex v, w;

    labelEachVertexWithItsInDegree();

    for (int counter=0; counter < NUM_VERTICES;
        counter++){
        v = findNewVertexOfDegreeZero();
        v.topologicalNum = counter;
        for each w adjacent to v
            w.indegree--;
    }
}

What's the bottleneck?
Finding a new vertex
Topological Sort: Take Two

1. Label each vertex with its in-degree
2. Initialize a queue $Q$ to contain all in-degree zero vertices
3. While $Q$ not empty
   a. $v = Q$.dequeue; output $v$
   b. Reduce the in-degree of all vertices adjacent to $v$
   c. If new in-degree of any such vertex $u$ is zero $Q$.enqueue($u$)

Is the use of a queue here important?

Runtime: $O(|V| + |E|)$

No, can use a stack, list, set, box, etc. Changes behavior, but not the fact the result is a topological sort
void Graph::topsort(){
    Queue q(NUM_VERTICES);
    int counter = 0;
    Vertex v, w;
    labelEachVertexWithItsIn-degree();

    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);
    }
}
Graph Traversals

• Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  – Mark visited vertices so do not enter an infinite loop

• Either can be used to determine connectivity:
  – Is there a path between two given vertices?
  – Is the graph (weakly) connected?

• Which one:
  – Uses a queue?
  – Uses a stack?
  – Always finds the shortest path (for unweighted graphs)?