Lecture 9: Analysis of Algorithms (CS483 - 001)

Amarda Shehu

Spring 2017
1. Graphs
   - Definition of a Graph
   - Omnipresence of Graphs

2. Graph Representations
   - Adjacency List Representation
   - Adjacency Matrix Representation
   - Alternative Graph Representations

3. Solving Problems with Graph Algorithms
What is a Graph?

Graph \( G = (V, E) \)
- \( V \): set of vertices
- \( E \): set of edges consisting of pairs of vertices from \( V \)

\[
V = \{ v_0, v_1, v_2, v_3, v_4 \}
\]
\[
E = \{ (v_0, v_1), (v_0, v_3), (v_1, v_2), (v_1, v_4) \}
\]
First Graph Problem

Seven Bridges of Koenigsberg [1736]:
Find a route that crosses each bridge exactly once. Posed by Leonard Euler [1707 - 1783].

modified from wikipedia

What is the minimum number of bridges that need to be added so that there exists a route that crosses each bridge exactly once?
Road Networks as Graphs
Outline of Today’s Class

Graphs

Graph Representations

Solving Problems with Graph Algorithms

Definition of a Graph

Omnipresence of Graphs

Airline Routes as Graphs

Figure: http://www.airlineroutemaps.com/
Social Networks as Graphs

Figure: http://hbr.idnet.net/images/
The Internet as a Graph

Visualization of the various routes through a portion of the Internet.

**Figure:** Credit: Matt Britt

**Figure:** Credit: Young Hyun, CAIDA
Websites as Graphs

**Figure:** http://www.google.com

**Figure:** Credit: Marcel Salathe
http://www.aharef.info

**Figure:** http://www.cs.gmu.edu

**Figure:** http://www.apple.com
Biological Networks as Graphs

Figure: Adapted from A. Barabasi, University of Notre Dame
Applications of Graphs

- Compilers
- Databases
- Neural Networks
- Machine Learning
- Artificial Intelligence
- Robotics
- Computational Biology
- ...

Amarda Shehu
Lecture 9: Analysis of Algorithms (CS483 - 001)
A graph $G = (V, E)$ is a pair consisting of:
- a set $V$ of vertices (or nodes)
- a set $E \subseteq V \times V$ of edges (or arcs)
  - edge $e_i \in E$ is a pair $(u, v)$ connecting vertices $u$ and $v$

A graph $G = (V, E)$ is:
- directed (referred to as a digraph) if $E$ is a set of ordered pairs of vertices. The edges here are often referred to as directed edges or arrows.
- undirected if $E$ is a set of unordered pairs of vertices.
- weighted if there are weights associated with the edges.
Illustrations of Types of Graphs

- **Figure:** undirected graph
- **Figure:** multigraph
- **Figure:** directed graph
- **Figure:** weighted graph
General Definition of a Graph

In a graph $G = (V, E)$:

- $E$ may be a set of unordered pairs of vertices not necessarily distinct. More than one edge can connect two vertices.
- An edge in $E$ may connect more than two vertices.
- These graphs are referred to as multigraphs or pseudo-graphs.
Focusing on Simple Graphs

Simple Graphs

- A simple graph, or a strict graph, is an unweighted, undirected graph containing no loops or multiple edges.
- Given that \( E \subseteq V \times V \), \( |E| \in O(|V|^2) \).
- If a graph is connected, \( |E| \geq |V| - 1 \).
- Combining the two, show that \( \lg(|E|) \in \theta(\lg(|V|)) \).
A **subgraph** $H$ of $G = (V, E)$ is $H = (V_1, E_1)$ where $V_1 \subseteq V$ and $E_1 \subseteq E$, where $\forall e = (k, j) \in E_1$, $k, j \in V_1$.

A **path** is a sequence of vertices, where each pair of successive vertices is connected by an edge.

The **length of the path** is the number of edges in the path.

A **simple path** contains unique vertices.

A **cycle** is a simple path with the same first and last vertex.

Two vertices are **adjacent** if they are connected by an edge.

The **neighbors** of a vertex are all the vertices adjacent to it.

The **degree** of a vertex is the number of its neighbors.

A graph is **connected** if $\exists$ a path between every pair of vertices.

A **tree** is a connected graph with no cycles.
Graph Representations

- A graph can be represented as an **adjacency list**.
- A graph can be represented as an **adjacency matrix**.
Adjacency List Representation

struct elist
{
    int vto;
    struct elist *next;
};

struct vlist
{
    int v;
    elist *edges;
    struct vlist *next;
};
# Basic Graph Functionality

<table>
<thead>
<tr>
<th>Function</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(v)</td>
<td>(O(</td>
</tr>
<tr>
<td>hasVertex(v)</td>
<td>(O(\text{find}(v)))</td>
</tr>
<tr>
<td>hasEdge(v_i, v_j)</td>
<td>(O(\text{find}(v_i) + \text{deg}(v_i)))</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>(O(1))</td>
</tr>
<tr>
<td>insertEdge(v_i, v_j)</td>
<td>(O(\text{find}(v_i)))</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>(O(</td>
</tr>
<tr>
<td>removeEdge(v_i, v_j)</td>
<td>(O(\text{find}(v_i) + \text{deg}(v_i)))</td>
</tr>
<tr>
<td>outEdges(v)</td>
<td>(O(\text{find}(v) + \text{deg}(v)))</td>
</tr>
<tr>
<td>inEdges(v)</td>
<td>(O(</td>
</tr>
<tr>
<td>overall memory</td>
<td>(O(</td>
</tr>
</tbody>
</table>

**Handshaking Lemma:** \(\sum_{v \in V} |\text{elist}(v)| = 2|E|\) for undirected graphs. \(O(|V| + |E|)\) storage \(\Rightarrow\) **sparse** representation.
The adjacency list of a vertex can be implemented as a linked list.
The list of vertices themselves can be implemented using:
- A linked list
- A binary search tree
- A hash table

In a standard implementation, each edge list has two fields, a data field and a pointer:
- The data field contains adjacent vertex name and edge information
- The pointer points to next adjacent vertex
Adjacency Matrix Representation

\[ M[i][j] = 1 \text{ iff } (v_i, v_j) \in E \]

<table>
<thead>
<tr>
<th></th>
<th>( V_0 )</th>
<th>( V_1 )</th>
<th>( V_2 )</th>
<th>( V_3 )</th>
<th>( V_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_0 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( V_1 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( V_2 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( V_3 )</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( V_4 )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```cpp
bool M[n][n];

bool **M;

using namespace std;

vector<vector<bool>> M;
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>find($v$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>hasVertex($v$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>hasEdge($v_i$, $v_j$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertVertex($v$)</td>
<td>$O(</td>
</tr>
<tr>
<td>insertEdge($v_i$, $v_j$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$O(</td>
</tr>
<tr>
<td>removeEdge($v_i$, $v_j$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>outEdges($v$)</td>
<td>$O(</td>
</tr>
<tr>
<td>inEdges($v$)</td>
<td>$O(</td>
</tr>
<tr>
<td>overall memory</td>
<td>$O(</td>
</tr>
</tbody>
</table>

$O(|V|^2)$ storage $\Rightarrow$ **dense** representation.
### Comparing The Two Representations

<table>
<thead>
<tr>
<th>Function</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>find($v$)</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>hasVertex($v$)</td>
<td>$O(\text{find}(v))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>hasEdge($v_i$, $v_j$)</td>
<td>$O(\text{find}(v_i) + \text{deg}(v_i))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertVertex($v$)</td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
<tr>
<td>insertEdge($v_i$, $v_j$)</td>
<td>$O(\text{find}(v_i))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>removeEdge($v_i$, $v_j$)</td>
<td>$O(\text{find}(v_i) + \text{deg}(v_i))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>outEdges($v$)</td>
<td>$O(\text{find}(v) + \text{deg}(v))$</td>
<td>$O(</td>
</tr>
<tr>
<td>inEdges($v$)</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>overall memory</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Operation</td>
<td>Method</td>
<td>Time Complexity</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Fast to query</td>
<td>[hasVertex, hasEdge]</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Fast to scan</td>
<td>[outEdges]</td>
<td>$O(</td>
</tr>
<tr>
<td>Fast to insert</td>
<td>[insertVertex, insertEdge]</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Fast to remove</td>
<td>[removeEdge]</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Graph Representation: Hash Map

- Vertex set as a hash map
  - key: vertex
  - data: outgoing edges
- Outgoing edges of each vertex as a hash set

using namespace std::ext;
hash_map<key, hash_set<key> >

vertex    outgoing edges
## Comparing The Three Representations

<table>
<thead>
<tr>
<th>Function</th>
<th>Adj. List</th>
<th>Adj. Matrix</th>
<th>Hash Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(v)</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>hasVertex(v)</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>hasEdge(v_i, v_j)</td>
<td>$O(</td>
<td>V</td>
<td>+ \text{deg}(v_i))$</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>$O(1)$</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>insertEdge(v_i, v_j)</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>removeEdge(v_i, v_j)</td>
<td>$O(</td>
<td>V</td>
<td>+ \text{deg}(v_i))$</td>
</tr>
<tr>
<td>outEdges(v)</td>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>inEdges(v)</td>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>overall memory</td>
<td>$O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>
Graph modeling: Problem Solving with Graph Algorithms

- Identify the vertices and the edges in your problem formulation
- Identify the objective of the problem
- State this objective in graph terms
- Implementation:
  - Construct the graph from the input instance
  - Run the suitable graph algorithm on the graph
  - Convert the output into a suitable/required format