CSE 373

Graphs 3: Implementation
reading: Weiss Ch. 9

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Implementing a graph

- If we wanted to program an actual data structure to represent a graph, what information would we need to store?
  - for each vertex? for each edge?

- What kinds of questions would we want to be able to answer quickly:
  - about a vertex?
  - about edges / neighbors?
  - about paths?
  - about what edges exist in the graph?

- We'll explore three common graph implementation strategies:
  - edge list, adjacency list, adjacency matrix
Edge list

- **edge list**: An unordered list of all edges in the graph.
  - an array, array list, or linked list

- **advantages**:
  - easy to loop/iterate over all edges

- **disadvantages**:
  - hard to quickly tell if an edge exists from vertex A to B
  - hard to quickly find the degree of a vertex (how many edges touch it)
Graph operations

- Using an edge list, how would you find:
  - all neighbors of a given vertex?
  - the degree of a given vertex?
  - whether there is an edge from \( A \) to \( B \)?
  - whether there are any loops (self-edges)?

- What is the Big-Oh of each operation?
Adjacency matrix

- **adjacency matrix**: An $N \times N$ matrix where:
  - the non-diagonal entry $a[i,j]$ is the number of edges joining vertex $i$ and vertex $j$ (or the weight of the edge joining vertex $i$ and vertex $j$).
  - the diagonal entry $a[i,i]$ corresponds to the number of loops (self-connecting edges) at vertex $i$ (*often disallowed*).
  - in an undirected graph, $a[i,j] = a[j,i]$ for all $i, j$. (*diagonally symmetric*)

![Adjacency Matrix and Graph](image)
Graph operations

- Using an *adjacency matrix*, how would you find:
  - all neighbors of a given vertex?
  - the degree of a given vertex?
  - whether there is an edge from $A$ to $B$?
  - whether there are any loops (self-edges)?

- What is the Big-Oh of each operation?
Adj matrix pros / cons

- **advantages:**
  - fast to tell whether an edge exists between any two vertices $i$ and $j$ (and to get its weight)

- **disadvantage:**
  - consumes a lot of memory on sparse graphs (ones with few edges)
Adjacency list

• **adjacency list**: Stores edges as individual linked lists of references to each vertex's neighbors.
  - in unweighted graphs, the lists can simply be references to other vertices and thus use little memory
  - in undirected graphs, edge \((i, j)\) is stored in both \(i\)'s and \(j\)'s lists

![Diagram of adjacency list representation](image)
Graph operations

- Using an *adjacency list*, how would you find:
  - all neighbors of a given vertex?
  - the degree of a given vertex?
  - whether there is an edge from $A$ to $B$?
  - whether there are any loops (self-edges)?
- What is the Big-Oh of each operation?
Adj list pros / cons

• **advantages:**
  - new vertices can be added to the graph easily, and they can be connected with existing nodes simply by adding elements to the appropriate arrays;
  - easy to find all neighbors of a given vertex (and its degree)

• **disadvantages:**
  - determining whether an edge exists between two vertices requires $O(N)$ time, where $N$ is the average number of edges per node
Weighted/directed graphs

- **weighted:**
  - *adj. list*: store weight in each edge node
  - *adj. matrix*: store weight in each matrix box
- **directed:**
  - *adj. list*: edges appear only in start vertex's list
  - *adj. matrix*: no longer diagonally symmetric
## Runtime comparison

- \(|V|\) vertices, \(|E|\) edges
- no parallel edges
- no self-loops

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory usage</td>
<td>(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>Find all neighbors of (v)</td>
<td>(</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td>Is (v) a neighbor of (w)?</td>
<td>(</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td>add a vertex</td>
<td>(1)</td>
<td>(1)</td>
<td>(</td>
</tr>
<tr>
<td>add an edge</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>remove a vertex</td>
<td>(</td>
<td>E</td>
<td>)</td>
</tr>
<tr>
<td>remove an edge</td>
<td>(</td>
<td>E</td>
<td>)</td>
</tr>
</tbody>
</table>
Representing vertices

- Not all graphs have vertices/edges that are easily "numbered".
  - How do we represent lists or matrices of vertex/edge relationships?
  - How do we quickly look up edges or vertices near a given vertex?

- edge list:
  - `List<Edge>`

- adjacency list:
  - `Map<Vertex, List<Edge>>` or `Multimap<Vertex, Edge>`

- adjacency matrix:
  - `Map<Vertex, Map<Vertex, Edge>>` or `Table<Vertex, Vertex, Edge>`
A graph ADT

- As with other ADTs, we can create a Graph ADT interface:

```java
public interface Graph<V, E> {
    void addEdge(V v1, V v2, E e, int weight);
    void addVertex(V v);
    void clear();
    boolean containsEdge(E e);
    boolean containsEdge(V v1, V v2);
    boolean containsVertex(V v);
    int cost(List<V> path);
    int degree(V v);
    E edge(V v1, V v2);
    int edgeCount();
    Set<E> edges();
    int edgeWeight(V v1, V v2);
}
```
A graph ADT, cont'd.

// public interface Graph<V, E> {
  ...
  boolean isDirected();
  boolean isEmpty();
  boolean isReachable(V v1, V v2); // DFS
  boolean isWeighted();
  List<V> minimumWeightPath(V v); // Dijkstra's
  Set<V> neighbors(V v);
  int outDegree(V v);
  void removeEdge(V v1, V v2);
  void removeVertex(V v);
  List<V> shortestPath(V v1, V v2); // BFS
  String toString();
  int vertexCount();
  Set<V> vertices();
}
Info about vertices

• Information stored in each vertex (for internal use):
  ▪ can store various flags and fields for use by path search algorithms

```java
public class Vertex<V> {
    public int cost() {...}
    public int number() {...}
    public V previous() {...}
    public boolean visited() {...}
    public void setCost(int cost) {...}
    public void setNumber(int number) {...}
    public void setPrevious(V previous) {...}
    public void setVisited(boolean visited) {...}
    public void clear() {...} // reset dist, prev, visited
}
```
Info about edges

• Information stored in each edge (for internal use):

```java
public class Edge<V, E> {
    public boolean contains(V vertex) {...}
    public E edge() {...}
    public V end() {...}
    public V start() {...}
    public int weight() {...} // 1 if unweighted
}
```