CS310 - Advanced Data Structures and Algorithms

Spring 2014

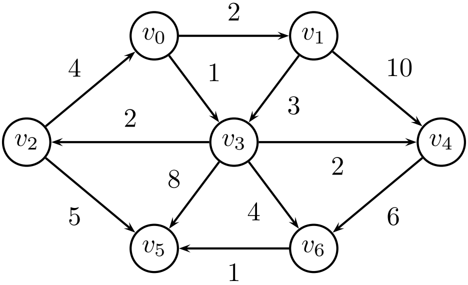
April 17, 2014

Graph – Definitions

- Graph – a mathematical construction that describes objects and relations between them.
- A graph consists of a set of vertices and a set of edges that connect the vertices:
- G = (V, E) where V is the set of vertices (nodes) and E is the set of edges (arcs)
- Each edge is an ordered pair (v,w) or an ordered triplet (v, w,W) where $v, w \in V$ and W is the weight.
- A directed graph (digraph) is one whose edge pair is ordered.
- Vertex w is adjacent to vertex v if and only if $(v, w) \in E$

A Directed Graph Example

$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$
$E = \{(V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10),$
$(V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2),$
$(V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1)\}$



Definitions

Definition (Path)

A sequence of vertices $w_1 \ldots w_n$ connected by edges s.t. $\{w_i, w_{i+1}\} \in E$ for each i=1..n.

Definition (Path Length)

Number of edges on the path.

Definition (Weighted Path Length)

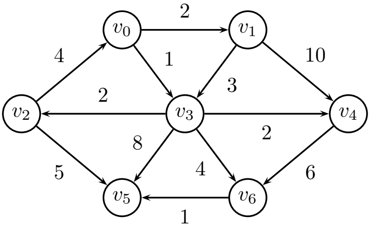
In a weighted graph, the sum of the costs of the edges on the path

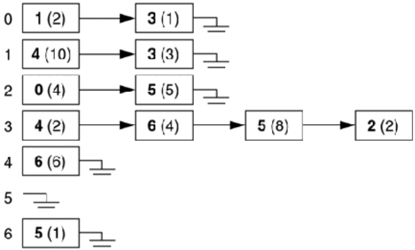
Definition (Cycle)

A path that begins and ends at the same vertex and contains at least one edge

Graph Representation

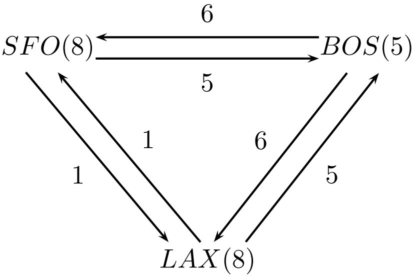
- Use a 2-dimensional array called adjacency matrix, a[v][w] = edge cost
- Nonexistent edges initialized to INFINITY
- For sparse graphs, use an adjacency list which contains a list of adjacent indices and weights.





Example – Traveling Between Cities

- For example, we have AirportNodes (LAX, 8) and (SFO, 8) and (BOS, 5) with edges with integer weight hours of flight time: LAX-SFO: 1 hr, SFO-LAX, 1hr., LAX-BOS: 5hr, BOS-LAX: 6hr. SFO-BOS: 5hr. BOS-SFO: 6hr.
- This is a directed graph.



Implementation

For a graph of Airports:

```
public class Airport {
    private String name;
    // ''BOS'', ''LAX'', etc.
    private int timezone;
    ...
    public boolean equals(Object x) {...
}
```

Implementation

- The simplest approach is to make a graph of String vertices, and use the airport names. We map from the names to the other info.
- Alternatively, we could use vertices of class Airport, with equals based on name.
- For a flight from "BOS" to "LAX", we want an edge. That edge may not need an application-specific type. But sometimes it is useful to have an app-defined edge type.
- JGraphT solution: Graph<V, E>, where E can be "DefaultEdge" if you don't want to bother with your own edge type.
- You do need to come up with type V, often String or Integer.

JGraphT – Open Source Graph Package

- If an application chooses Graph<String, String>, it is promising to come up with unique Strings for vertices, and also unique Strings for edges.
- The Graph object will remember all these ids and support access by them.
- If an application chooses Graph<String, DefaultEdge>, it is only promising to come up with unique Strings for vertices.
- The application can find edges of interest by how they are connected to its vertices.
- The Graph itself will come up with unique DefaultEdge objects for edges, which the application can use or ignore.

JGraphT: Open-Source Package

- Graph is an interface, like Set, etc.
- There are concrete classes and SimpleGraph and SimpleDirctedGraph that implement Graph, so we can put:
  ```
  Graph<String,DefaultEdge> g = new
  SimpleGraph<String,DefaultEdge>();
  ```
- Vertex operations on a Graph object like g: pretty straightforward:
  ```
  boolean addVertex(V v);
  boolean removeVertex(V v);
  boolean containsVertex(V v);
  Set<V> vertexSet();
  ```

Edge Operations

```
 // let Graph gen.  edge
E addEdge(V sourceVertex, V targetVertex);
// app edge
boolean addEdge(V sourceVertex, V targetVertex, E e);
// app edge
boolean containsEdge(V sourceVertex, V targetVertex);
boolean containsEdge(E e);
V getEdgeSource(E e);
V getEdgeTarget(E e);
double getEdgeWeight(E e); // For weighted graphs,
ignore now
Set<E> edgeSet();
```

Edge Operations

```
 // edges for vertices:
Set<E> getAllEdges(V sourceVertex, V targetVertex);
Set<E> edgesOf(V vertex);
// all edges touching vertex
E removeEdge(V sourceVertex, V targetVertex);
boolean removeEdge(E e);
Set<E> removeAllEdges(V sourceVertex, V
targetVertex);
removeAllEdges(Collection<?  extends E> edges);
```

Graph Operations – Undirected Graphs

- These are all valid for directed and undirected graphs.
- Only a few operations need special interpretation for undirected graphs: In an undirected graph, there is (conceptually) an edge from one vertex to the other, but it stands for a symmetric connection.
- containsEdge(V, V): the qualifying edge can go from target to source

Graph Operations – Undirected Graphs

- getEdgeSource, Target: the actual source and target (may be backwards to what you are expecting)
- getAllEdges(V,V): some returned edges may go from target to source
- removeAllEdges(V,V): removes ones connected the other way too
- removeEdge(V,V): removes one which may be connected the other way

Edge Operations for Directed Graphs

- For directed graphs, edgesOf will return inbound edges as well as outbound, unlike what is described by an adjacency list (which only lists outbound edges).
- In addition to Graph, there are subinterfaces DirectedGraph and UndirectedGraph:

```
 DirectedGraph:
int inDegreeOf(V vertex);
Set<E> incomingEdgesOf(V vertex);
int outDegreeOf(V vertex);
Set<E> outgoingEdgesOf(V vertex);
UndirectedGraph:
int degreeOf(V vertex);
```

Implementing a Graph

- The concrete class SimpleGraph implements UndirectedGraph, while SimpleDirectedGraph implements DirectedGraph.
- Above we put:
- `Graph<String,DefaultEdge> g = new SimpleGraph<String,DefaultEdge>();`
- Alternatively, we could put:
- `UndirectedGraph<String,DefaultEdge> g = new SimpleGraph<String,DefaultEdge>();`
- if we wanted to use degreeOf as well as all the Graph methods.

Getting Started with JGraphT

- Unzip graph package.
- Set up a Java project in eclipse with output directory “build”, replacing the default “bin”.
- See handout for loading and displaying a graph.
- See Demo.

Graph Demo

- Command Line Execution:
- LoadGraph: Its input file, test.dat, is in the top-level directory, so the command line needs "../test.dat" when running from build, the top of the classes tree in this project, so that the classpath is set properly by default.

```
> cd build
> java cs310.LoadGraph ../test.dat
```

DFS Demo

- DFSDemo: because it uses the JGraph library for Swing support, and that library is in lib/jgraph.jar, you need a more complicated command line with the .jar added to the classpath, as follows:

  ```
  cd build
  java -cp ../lib/jgraph.jar;.  cs310.DFSDemo
  ```

  (Note semicolon, then dot: this puts both the jar and . on the classpath).
- In Linux/UNIX, replace the semicolon with a colon:

  ```
  cd build
  java -cp ../lib/jgraph.jar:.  cs310.DFSDemo
  ```

Airport Example

```
Graph<String,DefaultEdge> g = new
SimpleDirectedGraph<String,DefaultEdge>();
// provided concrete class
g.addVertex(‘‘BOS’’);
g.addVertex(‘‘LAX’’);
DefaultEdge e1 = g.addEdge(‘‘BOS’’,‘‘LAX’’);
DefaultEdge e2 = g.addEdge(‘‘LAX’’, ‘‘BOS’’);
// now e1 and e2 uniquely id these two edges
Map<DefaultEdge,Double> hours = new
HashMap<DefaultEdge,Double>();
hours.put(e1, 5.1);
hours.put(e2, 6.0);
// slower going west against winds
...
```

Airport Example

- Alternatively, create an Edge type with .equals and .hashCode based on some unique id for each edge. Then use g.addEdge("BOS","LAX", e) to put it into the graph. It can have hours as a field, so don't need hours HashMap. Looping through vertices in the graph:

```
 for (String airport:  g.vertexSet()) {
// do something with vertex named airport
}
```

- Looping through adjacent vertices to a vertex v: find hops out of LAX

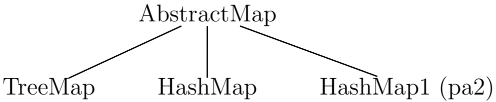
```
 for (DefaultEdge e:g.edgesOf(''LAX'')) {
    if (g.getEdgeSource(e).equals(''LAX'')) {
        // filter edges
        double hrs = hours.get(e);
        // get hours for edge
        // do something with this hop
        }
}
```

Airport Example

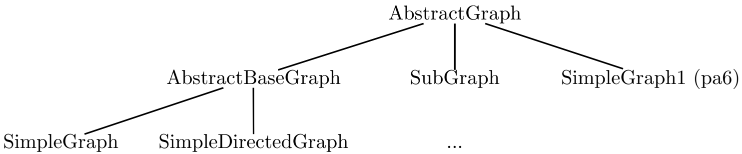
- Alternatively, use DirectedGraph for the type of g, and that lets you use DirectedGraph's outgoingEdgesOf(V), which gives you the right set of edges.
- Hopefully you can see how to write a "hello, world" program using a Graph with this API: Set up a simple Vertex type, and a main with a new SimpleDirectedGraph, then a few addVertexs and addEdges, then a vertexSet() iteration to see that the nodes are really in the graph, and an edgeSet iteration to check the edges.
- Or look at LoadGraph.java.

JgraphT Hierarchy

- Recall from pa2 how the AbstractMap provided useful base code for HashMap, JDK or our own. It holds the common code for TreeMap vs. HashMap:



- Similarly, JGraphT has a AbstractGraph class at the top of the implementation inheritance hierarchy:



JGraphT Implementation

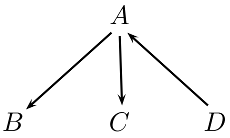
- The AbstractGraph implements, for example, containsEdge using getEdge, and removeAllEdges(V,V) using getAllEdges and removeEdge(E).
- It has no data structures, i.e., fields to collections of hold graph vertices or edges.
- AbstractBaseGraph has the master data structures, basically the collection of adjacency lists, for the JGraphT implementation classes.

JGraphT Implementation

- Since we are simplifying that setup, we need to have our SimpleGraph1 class instead of AbstractBaseGraph, extending from AbstractGraph.
- SimpleGraph1 is mostly implemented – you just finish it up.
- InternalEdge (corresponding to JGraphT IntrusiveEdge): holds the actual from, to vertices for a particular edge. Never returned to clients.

SimpleGraph1 Data Structures

- `Map<V, Set<E>> vertexList; // for each vertex, the Set of edges touching it`
- `Map<E, InternalEdge>> edgeList; // for each client edge, the InternalEdge that says how its connected`
- Small Example: Vertices: A, B, C, D – vertices, unique. e1, e2, e3 – client-known edges, unique.



Graph Example

- 4 V's 3 E's, 3 InternalEdges ie1, ie2, ie3
  - ie1 has object refs to A, B
  - ie2 has object refs to A,C
  - ie3 has object refs to D,A
- vertexList:
  - A: {e1, e2, e3} (e1, e2 outbound, e3 inbound edges)
  - B: {e1} (inbound edge)
  - C: {e2} (inbound edge)
  - D: {e3} (outbound edge)
- edgeList:
  - e1: ie1
  - e2: ie2
  - e3: ie3
- getEdge(A, D): look up A in vertexList, find e1, e2, e3. Iterate through them, looking up ies in edgeList, ie1, ie2, ie3, checking for connection from A to D or D to A.

Notes on Weiss Implementation (Optional)

- Weiss, pg. 536: vertexMap is Map from String to Vertex, pg. 535, which has List<Edge> for outgoing edges.
- Edge only knows dest Vertex. So different data structure but also using general idea of adjacency lists, and no generics of its own.
- Also, Vertex has "extra" fields dist, prev, and scratch for algorithm use.
- Weiss, pg. 532: We can use the same kind of input file. See test.dat.
- The Graph table has ids and adjacency lists. In addition, in the dark gray area, it holds intermediate results for a graph algorithm.
- JGraphT graphs don't hold "extra" info for algorithms.

Notes on Weiss Implementation (Optional)

- It is easy to have the same effect by setting up another Map from vertex/vertex name to (dist, prev) objects.
- JGraphT graphs are more general-purpose than Weiss's.
- What this is saying is that many shortest-path calculations are going to need "dist" and "prev" for each Node, and these will be changed as the algorithm calculates.