

CS231 Fall 2013 Lecture 32

Stephanie R Taylor

November 18, 2013

1 Administrative Topics

- Come to celebration of Colby in C.S. tomorrow!
- Kyle is away this week, so I am subbing for him in lab.

2 Graphs

2.1 Introduction

There are two defining properties of trees: a tree is connected and there are no cycles. That is, there is exactly one path from each node to each other node. What if we relax those two restrictions? Then any node can be connected to any other nodes it wishes.

A graph consists of a finite set of nodes, each of which is connected to some (0 or more) of the nodes. Some graphs allow nodes to be connected to themselves (loops) and allow multiple edges between two nodes, but usually a graph has either 0 or 1 edges between each pair of distinct nodes. [draw examples including complete graph and totally disconnected graphs]

Who cares?

- roads between cities

- communication networks
- oil pipelines
- personal relationships between people
- networks of cells in the circadian clock
- rooms in caves with passages between them

Digraphs

Sometimes its good to have one-way edges. In that case, we call the graph a digraph (short for directed graph). Well mostly deal with undirected graphs.

Other variations

Sometimes a weight is added to each edge, for example, indicating the capacity of the pipeline or channel or the length of the road between cities. Or other information, for example, if the graph has a variety of types of edges, might be included. Also, Nodes may hold other information, e.g., info about the cities or SimObjects.

2.2 Implementation 1: Adjacency list

How do we implement a graph? [let them suggest things] One way is to have a Graph object storing a list of nodes and, for each node, you keep a list of nodes, the neighbors of this node. [Draw a picture.] Heres what the code might look like:

```
public class Graph {
    List<GraphNode> nodes;

    ...
}

public class GraphNode {
    List<GraphNode> neighbors;

    ...
}
```

What is its disadvantage? [Very slow to lookup a neighbor if the graph is nearly complete. Time is $O(n)$.] This implementation is good if, instead of searching whether two nodes are connected by an edge, you are interested in traversing the graph. This is a good OO implementation since each GraphNode object is responsible for telling you its neighbors instead of the graph itself storing all that information.

This implementation is also much more like the implementation of binary trees in that each node keeps track of the neighbor or children pointers. However, in the case of a graph, you need an list (or set) instead of just two pointers, since a node can have arbitrarily many children or neighbors. A Graph corresponds in this way to a BinaryTree object and the nodes correspond to nodes. But there is one more difference: Why does a Graph maintain a list of nodes, instead of just one starting "root" node, like a BinaryTree does? [A graph might be disconnected]

2.3 Implementation 2: Hash table

Nodes have a map of key=direction, value=neighbor pairs

Suppose each node has at most 4 neighbors: one on the north, east, west, and south sides? Can we simplify things in that case? Do we really need to store the neighbors of a node in a list? [use 4 instance variables] This isn't so bad, but is there a way to still put all the neighbors in a data structure? [use an array where 0 means north, 1 means east, etc.] This works too, but is kind of ugly having to remember what direction 0,1,2,3 stand for. Better? [use constants for the 4 integers EAST, WEST, etc.] Good. An alternative is to store the data in a table. Remember tables or dictionaries in Python? `sides = dict()` `sides['triangle'] = 3` `sides['rectangle'] = 4` or `sides = 'triangle':3, 'rectangle':4`

Java has a similar thing called "HashMap":

```
HashMap sides = new HashMap();
sides.put("triangle",3); sides.put("rectangle",4);
System.out.println(sides.get("triangle"));
```

You can use a HashMap in each node:

```
HashMap neighbors = new HashMap();
neighbors.put("East", otherNode1);
```

```
neighbors.put("West", otherNode2);
```

2.4 Implementation 3: Graphs and Nodes containing Sets of nodes

One minor problem with this implementation is that the nodes in a graph or the neighbors of a node are sets, not ordered lists.

```
public class Graph {
    Set<GraphNode> nodes;

    ...
}

public class GraphNode {
    Set<GraphNode> neighbors;

    ...
}
```

2.5 Implementation 4: Adjacency matrix

This is the most straightforward implementation of a simple graph with n nodes since it is based on the realization that the most basic thing about a graph is which nodes are connected to which. This information can be stored in a 2-D array of 0s and 1s, 0s meaning no edge and 1s meaning an edge. This matrix is called the adjacency matrix. It results in the following implementation:

```
public class Graph {
    int [][] edges;

    public Graph(int numNodes)
    {
        edges = new int[numNodes][numNodes];
        //... fill in the edges array with 0 s and 1 s somehow...
    }
}

//in main...
```

```

Graph g = new Graph(5);
if ( g.edges[4][2] == 1 )
    //node with index 4 is connected by an edge to the node
    //with index 2.

```

[Give an example of undirected and directed graphs.]

There are lots of nice features about this scheme, the main one being that in time $O(1)$ you can determine if two given nodes are connected by an edge.

Enhancements

- If you want your graph to allow loops and multiple edges, what can you do? [allow diagonal and all entries to be values greater than 1.]
- Instead, if you want each edge to be weighted by a given amount, you can allow entries to be arbitrary values.
- What if you want both and maybe some other information about each edge such as its color? [have array `Edge[][]` where each `Edge` objects stores all the info regarding the connection between the two nodes]
- What if you want to store information about each node? [Can have a separate array of `Node` objects, indexed in the same order.]

Disadvantages of the adjacency matrix

This implementation is actually used a lot. What is the biggest disadvantage of this scheme, even with the enhancements? [What if there are a huge number of nodes, say 10,000, and only 100,000 edges. The array must have size 100,000,000 even though 0.1% of the array has non-zero entries.] For that reason, the adjacency matrix implementation should only be used if you know in advance that your graph is going to have either a small number of vertices or a lot of edges.

2.6 Drawbacks of these implementations

All of these implementations assume the necessary information can be stored in nodes. But what if edges have properties? For example, if they have colors

or weights? (Note: For the adjacency matrix, the weights are easily stored, but colors would require another data structure).

Also, for the first three implementations, If each node has lots of neighbors, it takes time to find out if there is an edge between two given nodes (have to search the array or set, which can have size n).

Alternatives to speed it up and store more information:

- Could have each node storing a Set of Edge objects, each of which stores its two endpoints and other information.

2.7 Graph problems

Now, what do people do with graphs? [GIVE THEM HANDOUT]

1. Is there a path that uses each edge exactly once? [Eulerian Path Problem]
2. Is there a path that goes through each node exactly once? [Hamilton Path Problem]
3. Is there a circuit that uses each edge exactly once? [Eulerian circuit Problem]
4. Is there a circuit that goes through each node exactly once? [Hamilton circuit Problem]
5. Does it have particular data in it? [a search problem]
6. Is it connected?
7. Are there any cycles?
8. Is there a path between two given nodes? If so, what is it? [Project 10]
9. What is the shortest path (least number of edges), if any, between two given nodes? [Shortest Path Problem]
10. What is the shortest path (sum of weights of edges) between two given nodes?

11. What are the lengths of the shortest paths between all pairs of nodes? [All-pairs Shortest Path Problem]
12. What is the shortest tour of all nodes in a complete graph (where shortest means the least sum of the weights of edges)? [Travelling Salesperson Problem]
13. Can the nodes be colored red and blue so that red nodes have edges only to blue nodes and vice versa? [Bipartite Graph Problem]
14. What is the fewest number of colors needed to color the nodes so that adjacent nodes have different colors? [Chromatic Number of a graph]
15. Can the graph be drawn in the plane so that no edges cross? [Planarity Problem]
16. How many paths of a given length are there between two given nodes?
17. Can each node be represented by an interval on the real line so that adjacent nodes correspond to overlapping intervals? [Interval Graph Problem]

2.8 Traversing a graph

In many of the problems, we need to use some graph traversal algorithm where we are visiting all nodes. For example, all three problems for your project involve traversals. All these traversals can be done recursively or with stacks or queues. During the traversal, there are three categories of nodes: Nodes already visited, nodes that are unvisited neighbors of visited nodes, and the remaining nodes. We will mark the nodes when visited and we will keep a collection *C* of unvisited neighbors of visited nodes.

General graph traversal algorithm:

```

unmark all nodes
create a new empty collection C of nodes
choose a starting node and add it to C
while C is not empty do:
    remove a node n from C
    mark n as visited
    for all neighbors p of n:
```

```
if p is unmarked and not already in C:  
    add p to C
```