CS 228: Introduction to Data Structures
Lecture 35
Monday, April 20, 2015

Representing Graphs

In what follows, V and E denote the number of nodes and edges in a graph, respectively. For simplicity, we begin by assuming that the nodes are numbered 0 through V–1. We will drop this assumption later.

An ***adjacency matrix*** is a V-by-V array of boolean values. Each row and column represents a vertex of the graph. Set the value at row i, column j to `true` if (i, j) is an edge of the graph.

If the graph is undirected, the adjacency matrix is *symmetric*: row i, column j has the same value as row j, column i (see Figure 5).

$$\begin{array}{r|cccccc|}
 & \text{Belgium} & \text{Denmark} & \text{France} & \text{Germany} & \text{Luxembourg} & \text{Netherlands} \\
\hline
\text{Belgium} & 0 & 0 & 1 & 1 & 1 & 1 \\
\text{Denmark} & 0 & 0 & 0 & 1 & 0 & 0 \\
\text{France} & 1 & 0 & 0 & 1 & 1 & 0 \\
\text{Germany} & 1 & 1 & 1 & 0 & 1 & 1 \\
\text{Luxembourg} & 1 & 0 & 1 & 1 & 0 & 0 \\
\text{Netherlands} & 1 & 0 & 0 & 1 & 0 & 0 \\
\hline
\end{array}$$

Figure 5. Adjacency matrix for the graph of Figure 2.

As the next figure shows, the adjacency matrix of a directed graph is not, in general, symmetric.

$$\begin{array}{c|cccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 1 \\
3 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

Figure 5. Adjacency matrix for the graph of Figure 1.

An ***adjacency list*** representation consists of an array with one entry per node. The entry for node v references the set of all neighbors of v (recall that u is a neighbor of v if (v,u) is an edge in the graph).

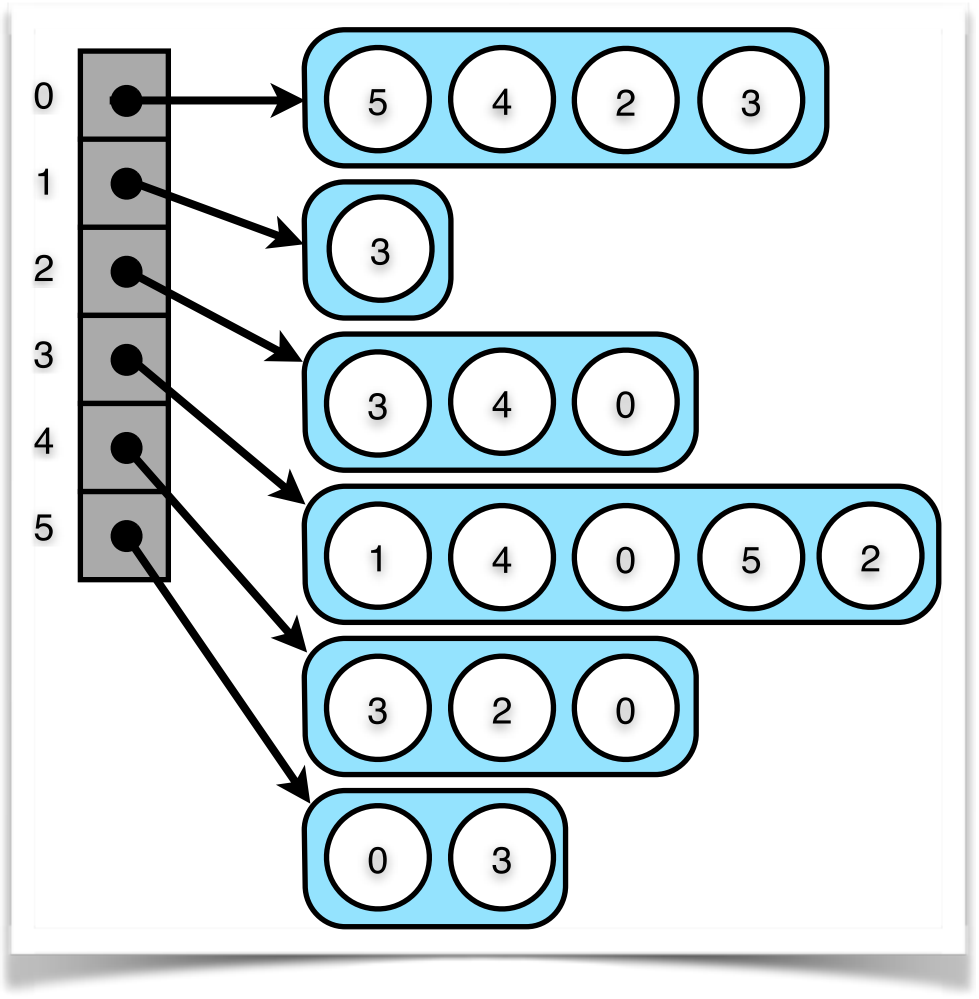

Figure 6. Adjacency list representation for the graph of Figure 2. Belgium = 0, Denmark = 1, France = 2, Germany = 3, Luxembourg = 4, Netherlands = 5.

For the street map example of Figure 1, the adjacency list representation would look like this.

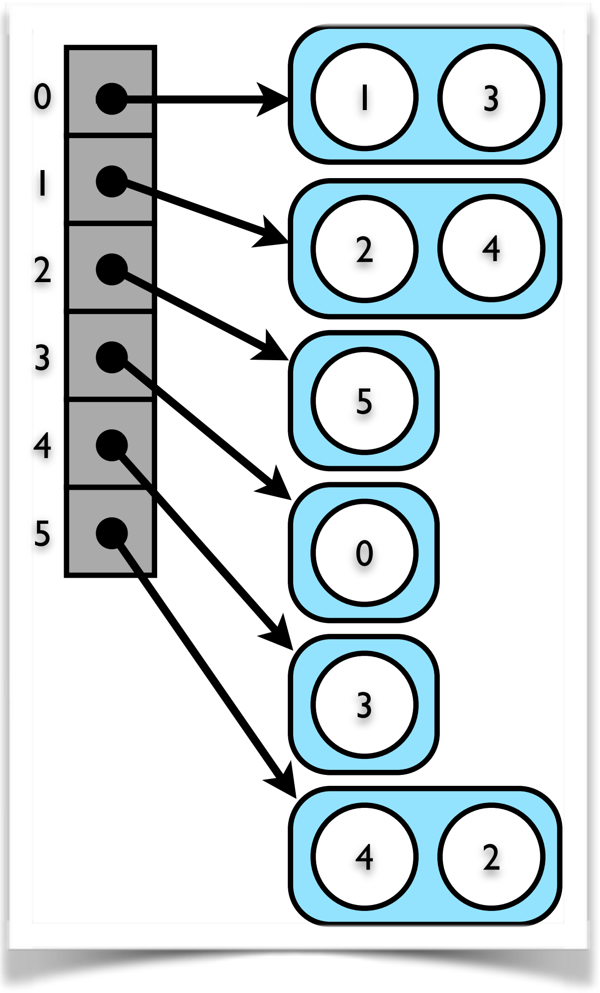

Figure 7. Adjacency list representation for the graph of Figure 1.

To represent the set of neighbors of a node, we can use any of the standard methods — arrays, linked lists, even binary search trees (because you can traverse a BST in linear time).

Adjacency Matrices versus Adjacency Lists

The next table compares adjacency matrices and adjacency lists for undirected graphs. V and E denote the number of nodes and edges, respectively.

	Adjacency Matrix	**Adjacency List**
Scanning the neighbors of v	O(V)	O(degree(v))
Test if u and v are neighbors	O(1)	O(min(degree(u), degree(v))
Space	O(V^2)	O(E + V)

Notes

- The time complexity to test whether two nodes are neighbors on an adjacency list assumes that we interleave scans of the lists of neighbors of u and v, stopping if u is found in v's list, v is found in u's list, or the end of either list is reached.
- The adjacency matrix bounds are identical for directed graphs.
- In a directed graph represented by an adjacency list, scanning the neighbors of u takes O(outdegree(u)) time.

- Scanning the neighbors of a node u in a directed graph represented by an adjacency list takes O(outdegree(u)) time; so does determining if v is a neighbor of u.

- The space requirement for an adjacency matrix is $O(V^2)$, because it is a $V \times V$ boolean matrix.

- The space requirement for an adjacency list is O(V+E):

  - There are V sets of neighbors (some might be empty).

  - For undirected graphs, each edge (u,v) contributes to *two* neighbor sets: v must be in the set of u's neighbors, and u must be in the set of v's neighbors.
    - Total contribution of all edges is 2E.

  - For directed graphs, each edge (u,v) contributes to only *one* neighbor set: v must be in the set of u's neighbors.
    - Total contribution of all edges is E.

The choice between adjacency matrices and adjacency lists depends on at least two factors: the number of nodes in the graph and the ***density*** of the graph. The density of a graph is the ratio of the number of edges to the maximum possible number of edges. For an undirected graph, the maximum number of edges is $V \cdot (V - 1)/2$, which is roughly $V^2/2$, and is achieved when there is an edge between every pair of nodes[1]. For a directed graph, the maximum number of edges is $V \cdot (V - 1)$, which is roughly V^2, because for every two nodes u and v, we can have two edges: (u,v) and (v,u).

- If a graph has few nodes, an adjacency matrix is a good choice, regardless of the density, because it is simple and lightweight.

- If the graph is large and *sparse* (i.e., not dense), an adjacency list is more space-efficient than an adjacency matrix. It is also more time-efficient for scanning the neighbors of a node.

- For large dense graphs, adjacency matrices are a good choice: they are simple, lightweight, and space- and time-efficient. Note, though, that large dense graphs are rare. For instance, as of early 2014, the Facebook graph had about 1.23 billion users (nodes), but the average user had 338 friends[2] — that's *sparse*.

Using `HashMap`s and `HashSet`s

The representations we have seen have some limitations:

[1] A graph like this is said to be ***complete***.

[2] http://www.theguardian.com/news/datablog/2014/feb/04/facebook-in-numbers-statistics

1. Nodes are often complex objects with names, like “Bob” or “Australia”; they are not just integers.

2. In an adjacency list, checking if a node u is adjacent to another node v is time-consuming: you have to scan the lists for u and v sequentially, which can take time linear in the number of nodes. Checking adjacency in an adjacency matrix takes O(1) time, but, as we saw, such matrices are not space-efficient for large sparse graphs.

3. Graphs are often dynamic: nodes and edges keep getting added and deleted (think, e.g., of all the friending and un-friending that goes on in Facebook).

We can address issue 1 by using a `Map` object to map names of nodes to their respective lists of neighbors. If we use a `HashMap` from vertex names to `List` objects, we can find the list for any node in O(1) (under the usual assumption about `HashMap`s).

We can address issue 2 — testing for adjacency quickly — by representing the neighbors as a `HashSet`, rather than a List. Thus, the underlying map is of the form

```
HashMap<V, HashSet<V>>
```

This means that each of the following operations takes O(1) time (under the usual assumptions about `HashMap`s and `HashSets`):

- accessing the set of neighbors of a node,
- testing if two nodes are adjacent,
- adding/deleting a node, and
- adding/deleting an edge.

The last two points address issue 3.

An implementation of the `HashMap`/`HashSet` representation of undirected graphs called `Graph<V>` is posted on Blackboard.

Graph Traversals

A ***graph traversal*** is a way to explore the nodes and edges of the graph. The idea is similar to traversing a tree. This is not surprising: a tree is just a special kind of graph. We will study two types of graph traversals: depth-first search (DFS) and breadth-first search (BFS). DFS and BFS can be used on both directed and undirected graphs. Java code and detailed examples for BFS and DFS are posted on Blackboard.

Note. Much of the material in the remaining lectures is based on the following book, which we shall refer to as "CLRS".

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms* (3rd ed.). MIT Press, 2009.

Breadth-First Search

Suppose we pick some node s as a starting node. The ***distance*** from s to a node v (possibly s itself) is the minimum number of edges in a (simple) path from s to v. The distance from s to itself is obviously 0.

BFS visits nodes by increasing distance from the starting node, s. BFS resembles level-order tree traversal. Like level-order tree traversal, BFS uses a queue Q. BFS also maintains a value dist(v) for each vertex v, which is an estimate of the distance from s to v. Initially,

- Q contains only s,
- dist(s) = 0 — the distance from s to itself is zero —, and
- dist(v) = ∞ for every node v other than s — since we haven't found any path from s to v yet[3] .

[3] In an actual program, instead of ∞, we'd use a very large number, such as `Integer.Max_Value`.

The algorithm then does the following while Q is not empty:

1. Let u be the node at the front of Q.

2. ***Process*** u: for each neighbor v of u, if we are discovering v for the first time, we make dist(v) = dist(u) + 1 and enqueue v.

3. Dequeue u.

Unlike level-order traversal of a tree, we may encounter the same node multiple times during a BFS (in a tree there is just one path from the root to any node, but this is not so for a general graph). To keep track of the status of the nodes, we need to maintain some additional information. We will explain this and other details next time.