Lecture 16: Graphs, Adjacency Matrices and Lists, DFS

Review Exercises

1. Suppose you are merging the two sorted lists 1,3,5 and 2,3,4. After 3 elements have been added to the merged list, which elements remain in each list?
2. Partition the list 9,4,1,7,2,3,5 around the median.
3. ($\star$) Explain how to give an iterative (non-recursive) implementation of quick sort.
4. ($\star$) Explain how to give an iterative (non-recursive) implementation of merge sort.

Review Solutions

1. The first list will have 5 and the second will have 3,4.
2. 2,3,1,4,7,9,5 is a valid partition about the median.
3. You can use a stack to simulate the implicit tree traversal that quick sort performs. Instead of making a recursive call we simply push a pair of indices onto the stack (the left and right indices of the subarray to quick sort). While the stack is non-empty, the algorithm pops the next pair of indices from the stack, partitions the corresponding subarray, and pushes two new pairs determining the left and right halves. The stack begins with the pair $(0, n-1)$.

   Alternatively, we can use a queue and perform a level-by-level traversal of the implicit tree. This is the same algorithm as above but we replace the stack with a queue. The stack method should be more memory efficient when the implicit tree is fairly well balanced.
4. We can also perform a level-by-level traversal here, but we start with the leaves and move upwards. Initially we push all of the index pairs corresponding to the leaves on the queue. That is, we push $(0, 0)$ then $(1, 1)$, etc. The algorithm dequeues 2 pairs off the queue at a time, merges the corresponding lists, and then enqueues the combined pair. We stop when only 1 pair remains on the queue. The one technical detail is that we can only merge adjacent subarrays, so if the two pairs we get aren't adjacent, we simply enqueue the first, and dequeue a new pair (this occurs when we have an odd number of subarrays at a given level of the implicit tree).

   Below we give iterative implementations of both merge and quick sort using int arrays to store the pairs of indices.

IterativeSorts.java

```
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Random;

public class IterativeSorts
{
        public static void mergeSort(int[] arr)
        {
                ArrayDeque<int[]> queue = new ArrayDeque<>();
                int[] tmp = new int[arr.length];
                for (int i = 0; i < arr.length; ++i)
                        queue.add(new int[]{i,i});
                while (queue.size() > 1)
                {
                        int[] f = queue.poll();
                        int[] s = queue.poll();
                        if (f[1]+1 != s[0]) {
                                queue.add(f);
                                f = s;
                                s = queue.poll();
                        }
                        SortUtils.merge(arr, f[0], f[1], s[1], tmp);
                        f[1] = s[1];
                        queue.add(f);
                }
        }
        public static void quickSort(int[] arr, Random ran)
        {
                ArrayDeque<int[]> stack = new ArrayDeque<>();
                stack.add(new int[]{0,arr.length-1});
                while (!stack.isEmpty())
                {
                        int[] inds = stack.pollLast();
                        if (inds[0] >= inds[1]) continue;
                        int p = ran.nextInt(inds[1]-inds[0]+1)+inds[0];
                        p = SortUtils.partition(arr,inds[0],inds[1],p);
                        stack.add(new int[]{inds[0],p-1});
                        stack.add(new int[]{p+1,inds[1]});
                }
        }
        public static void main(String[] args)
        {
                Random ran = new Random();
                int[] arr = {1,2,3,4,5,6,7,8,9,10,11};
                SortUtils.shuffle(arr, ran);
                System.out.println(Arrays.toString(arr));
                mergeSort(arr);
                System.out.println(Arrays.toString(arr));
                SortUtils.shuffle(arr, ran);
                System.out.println(Arrays.toString(arr));
                quickSort(arr,ran);
                System.out.println(Arrays.toString(arr));
        }
}
```

Graphs: Definitions and Terminology

A graph is a set of vertices V (sometimes called nodes) and a set of edges E, which are links between the nodes. We often use the numbers $0, 1, \ldots, n-1$ to represent the vertices. Graphs are either directed (edges have a direction) or undirected (edges don't have a direction). Pictorially, vertices are typically drawn as circles. Edges in a directed graph are drawn as lines with arrows connecting the corresponding vertices, and edges in an undirected graph are just lines between vertices. A *loop* is a directed edge from a vertex to itself. Undirected graphs do not have loops. Sometimes we willl want to assign weights or costs to each edge. Such a graph is called a *weighted graph*. Sometimes graphs have *multiple edges* (in the same direction) between two vertices. By default, we will assume our graphs do not have multiple edges.

The *degree* of a vertex is the number of edges it touches (formally we say the number of edges incident on it). For a directed graph, a vertex v has separate indegree and outdegree for the number of edges coming in to v, and the number of edges leaving v, respectively.

A *path* from v to w is a sequence of distinct vertices starting with v and ending with w such that each subsequent vertex is connected by an edge (the trivial path has 1 vertex). The length of a path is the number of edges on it. If the graph is directed, the edges must go in the correct direction. More precisely, if the vertices of the path are $x_0, x_1, \ldots, x_n$ then the directed edges on the path must go from x_i to x_{i+1}. A *walk* is the same as a path, except the vertices need not be distinct. A *cycle* is the same as a path, but the first and last vertices are the same (all others are distinct). We require a cycle to have at least 1 edge (the smallest possible cycle is a single loop), and for all the edges to be distinct. The length of a cycle or walk is again the number of edges in it.

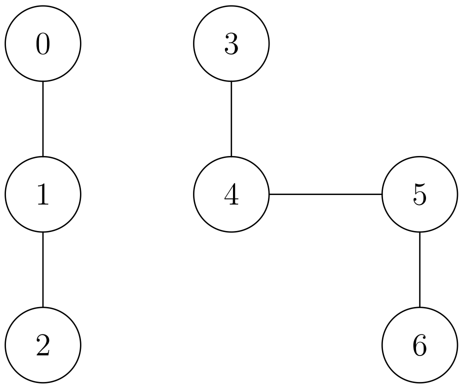
An undirected graph is called *connected* if every pair of vertices is connected by a path from one vertex to the other. A directed graph is called *strongly connected* if there is a (directed) path between every pair of vertices v, w. That is, if you fix vertices x, y there will be a path from x to y and a path from y to x. In an undirected graph, if you fix a vertex v and consider the set C of all vertices w reachable from v (i.e., all vertices w such that there is a path from v to w) then C is called a *connected component*. This means that an undirected graph is connected iff it has 1 connected component. The analogous concept for a directed graph is called a *strongly connected component*.

A non-empty undirected graph is called a *tree* if it is connected and has no cycles (*acyclic*). A tree is called *rooted* if a node is distinguished as the root. An important property of a tree (which could also be used as a definition) is that for every pair of vertices there is exactly 1 path connecting them. To see why, note there must be at least one path by connectedness. If there were 2 or more paths, we could use parts of those paths to form a cycle. A *directed tree* is a directed graph such that each pair of vertices is connected by exactly 1 path in one direction. That is, for any fixed vertices v, w either there is a path from v to w or a path from w to v but not both. When we use the term tree, we mean undirected tree.

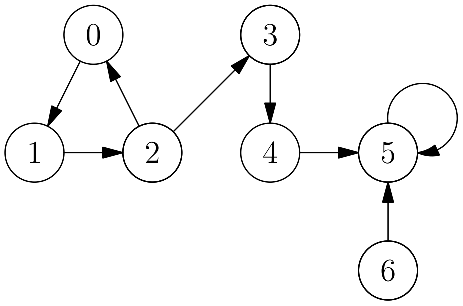
Graph Terminology Exercises

1. For each of the following graphs, determine which of the following properties holds: undirected, directed, connected, acyclic, strongly connected. Also give the number of connected (or strongly connected) components, the length of the longest path in the graph, and, if it has a cycle, the length of the shortest cycle (called the *girth*).

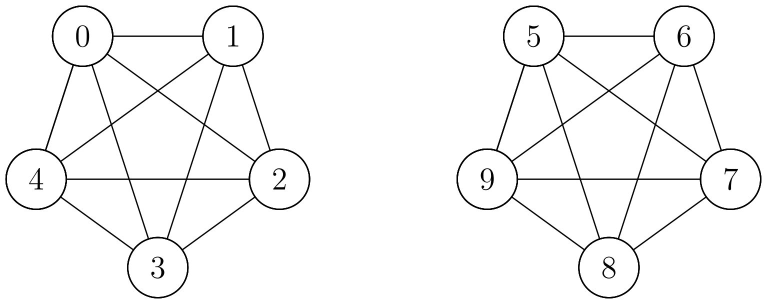
   (a)



   (b)



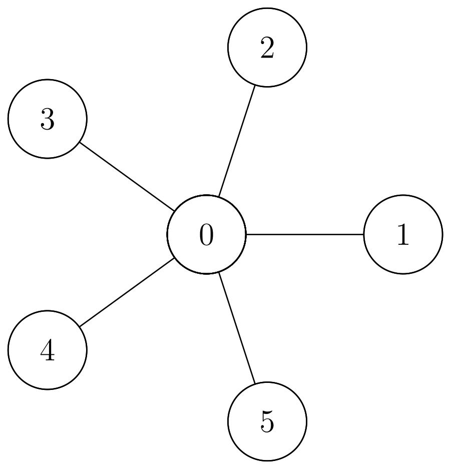
   (c)



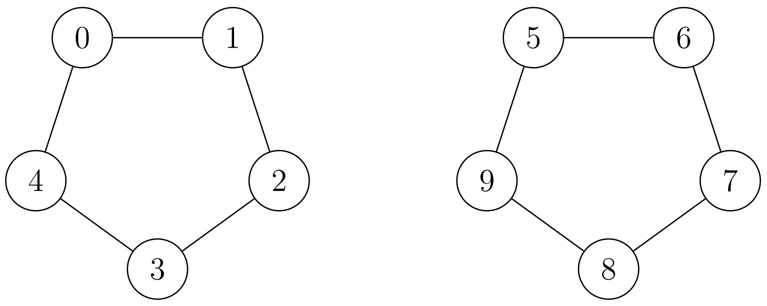
2. Draw a tree where 1 vertex has degree 5 and the rest have degree 1.
3. Draw an undirected graph with 2 connected components such that every vertex has degree 2.
4. Suppose a graph with n vertices has the maximum possible edges (as usual we assume no multiple edges). Give that number of edges if the graph is:
   (a) Undirected
   (b) Directed
   (c) Undirected, but has 2 components.
   (d) Undirected with 2 components of equal size.
5. As usual, here the term tree means undirected tree.
   (a) Show that removing any edge from a tree will leave a disconnected graph. Thus a tree is a minimal connected graph. [Hint: Suppose not.]
   (b) Show that a tree with at least 2 vertices must have a vertex of degree 1. [Hint: Suppose not.]
   (c) ($\star$) Show that a tree with n vertices must have $n-1$ edges.

Graph Terminology Solutions

1. (a) Undirected, acyclic, 2 connected components. The longest path has length 3.
   Note that each connected component forms a tree. This is why an undirected acyclic graph is called a *forest.*
   (b) Directed, 5 strongly connected components. The longest path has length 5, and the shortest cycle has length 1.
   (c) Undirected, 2 connected components. The longest path has length 4 and the shortest cycle has length 3.
2. This is called a star graph with 5 points.



3.



4. (a) $\binom{n}{2}$
   (b) n^2
   (c) $\binom{n-1}{2}$
   (d) $2\binom{n/2}{2}$

   When a graph is close to the maximum number of edges we call it *dense*. If a graph has a low number of edges we call it *sparse*.

5. (a) Suppose removing the edge between v and w does not disconnect the graph. Then there must be a path p in the graph that doesn't include the edge between v and w. But the path p along with edge vw forms a cycle. This contradicts the assumption that our graph was acyclic.
   (b) Suppose every vertex has degree at least 2. Start at a vertex v_0 and construct a path by repeatedly going to a new vertex you haven't visited before. At some point this stops at the vertex v_k since all of its neighbors have been visited already. But there is an unused edge coming out of v_k to one of the v_i for $i < k$ since the degree of v_k is at least 2. This shows the original graph had the cycle

$$v_i, v_{i+1}, \ldots, v_k, v_i.$$

(c) Proof by induction. For the base case, note that a tree of size 1 has 0 edges. For the inductive case, assume all trees of size n have $n - 1$ edges. Let T be some tree with $n + 1$ vertices. By the previous part it must have a vertex of degree 0. Removing this vertex (and the associated edge) leaves a smaller tree with $n - 1$ edges (by induction). Thus T has n edges.

An alternative idea goes as follows. Suppose tree T has n vertices. We will grow the tree T by adding one edge at a time. Each edge we add must be between 2 different connected components since we cannot form a cycle. This must occur exactly $n - 1$ times to leave us with 1 connected component, since each edge will decrease the number of connected components by 1.

Data Structures: Adjacency Matrices and Lists

In this section we discuss how to store a graph. In both of the following data structures we will store directed graphs. If we want to store undirected graphs, we will represent each undirected edge as 2 directed edges (in both directions).

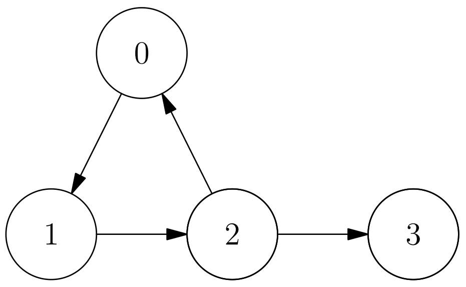
An *adjacency matrix* is an $n \times n$ boolean matrix. Let adj denote this matrix. If adj[i][j] is true, then there is a directed edge from i to j. To model a weighted graph, we can store an auxiliary matrix of costs (i.e., costs[i][j] stores the cost of the edge from i to j). We cannot model multiple edges with an adjacency matrix.

An *adjacency list* as an array of lists of vertices. In Java, we can represent this as an ArrayList<ArrayList<Integer>> adj. The list adj.get(v) stores all vertices w such that there is an edge from v to w. To store weights we can use an auxiliary data structure: an adjacency list of costs, or an adjacency matrix of costs. As an alternative, we can store Edge objects in our adjacency list which contain the terminating vertex and the weight. To model multiple edges we simply allow repeats in our lists.

Adjacency lists are more popular than adjacency matrices but they have their tradeoffs, as we will see in a moment.

Graph Data Structures Exercises

1. For the graph below, show how it would be stored in an adjacency matrix (for both edges and weights) and in an adjacency list (for both edges and weights).



2. What are the worst-case space requirements for an adjacency matrix and an adjacency list.

3. Give the worst-case runtimes of the following operations for an adjacency matrix and an adjacency list.

   (a) Determine if there is an edge from i to j.

   (b) Output all of the edges of the graph.

   (c) List the neighbors of a vertex v (i.e., the vertices w such that there is an edge from v to w).

   (d) List all vertices that have v as a neighbor (i.e., the vertices w such that there is an edge from w to v).

   (e) Output the degrees of every vertex.

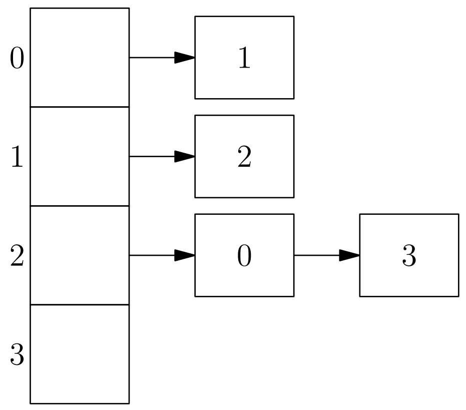
4. ($\star$) Think about how to compute if an undirected graph is connected. You can use either an adjacency matrix of an adjacency list.

Graph Data Structures Solutions

1. The adjacency matrix is given below.

$$\begin{array}{c|c|c|c|c|}
 & 0 & 1 & 2 & 3 \\ \hline
0 & \text{F} & \text{T} & \text{F} & \text{F} \\ \hline
1 & \text{F} & \text{F} & \text{T} & \text{F} \\ \hline
2 & \text{T} & \text{F} & \text{F} & \text{T} \\ \hline
3 & \text{F} & \text{F} & \text{F} & \text{F} \\ \hline
\end{array}$$

   The adjacency list is depicted below.



2. $\Theta(|V|^2)$ for an adjacency matrix and $\Theta(|V| + |E|)$ for an adjacency list.

3. (a) $\Theta(1)$ for an adjacency matrix. For an adjacency list it is $\Theta(\deg(i))$ where $\deg(i)$ is the degree of i. This can be $\Theta(|V|)$ in the worst-case. The worst case occurs when $\deg(i) = |V| - 1$.

   (b) $\Theta(|V|^2)$ for an adjacency matrix and $\Theta(|V| + |E|)$ for an adjacency list. Thus the list is far superior when the graph is sparse.

   (c) $\Theta(|V|)$ for an adjacency matrix and $\Theta(\deg(v))$ for an adjacency list.

   (d) $\Theta(|V|)$ for an adjacency matrix (loop over the column), and $\Theta(|V| + |E|)$ for an adjacency list (may need to look at every edge).

   (e) $\Theta(|V|^2)$ for an adjacency matrix and $\Theta(|V|)$ for an adjacency list.

4. One method is to perform a DFS (see next section) of the graph and see if you can reach every node from the first node you started at.

Depth First Search (DFS)

As with trees, we now discuss how to traverse the vertices of a graph. Unlike our tree traversals from earlier in the class, we have to be careful not to repeatedly traverse the same nodes since we can have multiple ways to arrive at the same vertex. To do this we will maintain an array that stores which vertices have been visited by our traversal. Thus, once we mark a vertex as visited, we will never allow it to be visited again preventing the issue mentioned above. Below we give an implementation of DFS for an adjacency matrix.

DFS.java

```
public class DFS
{
    public static final int UNVISITED = 0;
    public static final int VISITING = 1;
    public static final int FINISHED = 2;
    public static void dfs(boolean[][] adj)
    {
        int[] state = new int[adj.length];
        //Loops over each node in the graph
        //in case we haven't visited it yet
        for (int v = 0; v < adj.length; ++v)
            dfs(adj,v,state);
    }

    public static void dfs(boolean[][] adj,
            int v, int[] state)
    {
        if (state[v] != UNVISITED) return;
        System.out.printf("Visiting Vertex %d\n",v);
        state[v] = VISITING;
        for (int w = 0; w < adj.length; ++w)
            if (adj[v][w])
                dfs(adj,w,state);
        System.out.printf("Finishing Vertex %d\n",v);
        state[v] = FINISHED;
    }
    public static void main(String[] args)
    {
        boolean[][] adj = {
                {false,true,true},
                {true,false,true},
                {true,true,false}
        };
        dfs(adj);
    }
}
```

Using DFS we can answer many interesting questions about a graph.