Lecture 12

Graph Review

In this class we will look at undirected and directed graphs. In either case, we will think of E as our set of edges (ordered pairs of vertices) and V as our set of vertices. The main data structure we will use is the adjacency list: if $V = \{0, \ldots, n-1\}$ then

```
ArrayList<Integer> adj[] = new ArrayList<Integer>[n];
for (int i = 0; i < adj.length; ++i)
    adj[i] = new ArrayList<Integer>();
```

The list adj[i] is the collection of neighbors of i. An alternate data structure that is sometimes useful is the adjacency matrix:

```
boolean[][] adj = new boolean[n][n];
```

The value adj[i][j] is true if there is an edge from i to j. In an undirected graph we have adj[i][j]==adj[j][i].

If the edges have weights, you could store a parallel adjacency list or matrix of the weights, or you could make an adjacency list of objects (say pairs) that store the endpoint and the weight.

In undirected graphs, we say that two vertices p, q are connected if there is a path from p to q. A path is a sequence of distinct vertices $v_1, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$, for $1 \leq i \leq n-1$. The relationship "is connected" is an equivalence relation on the vertices of a graph that partitions V into equivalence classes called connected components. A graph with a single connected component is called connected. For directed graphs, the corresponding concept is called strongly connected. We say p, q are strongly connected if there is a path from p to q and a path from q to p. Analogously, we can partition V into strongly connected components. If we contract each strongly connected component to a single vertex, the resulting graph is a dag (which we define next).

A cycle in a graph is a sequence of vertices $v_1, \ldots, v_n$ with $v_1 = v_n$ and $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$. A directed acyclic graph is called a dag.

Exercises

1. A forest is an undirected acyclic graph. If a forest with n vertices has C components, how many edges will it have?

2. What is the maximum number of edges in a directed graph with n vertices that has no loops (edges of the form (v, v))?

3. A walk is like a path, except the vertices need not be distinct. How would you compute, for each pair of vertices (i, j), whether there is a walk of length k from i to j?

Solutions

1. $n - C$ edges.

2. $n^2 - n$.

3. Compute A^k where A is the adjacency matrix, multiplication is "and" and addition is "or". If instead we use an adjacency matrix where 1 is used for true, and 0 is used for false, then A^k counts the number of walks of length k.

Depth First and Breadth First Search

On either directed or undirected graphs we can use depth first and breadth first search to answer a large variety of questions. The code for depth first search follows:

```
static int INIT = 0, PROCESSING = 1, FINISHED = 2;
//DFS starting a given vertex vert
void dfs(ArrayList<Integer>[] adj, int vert, int[] state)
{
  if (state[vert] != INIT) return;
  state[vert] = PROCESSING;
  for (int i = 0; i < adj[vert].size(); ++i)
     dfs(adj,adj[vert].get(i),state);
  state[vert] = FINISHED;
}
void dfs(ArrayList<Integer>[] adj)
{
  int[] state = new int[adj.length];
  for (int i = 0; i < state.length; ++i) if (state[i] == INIT)
     dfs(adj,i,state);
}
```

The code for breadth first search follows:

```
//BFS starting at a given vertex vert
void bfs(ArrayList<Integer>[] adj, int vert, int[] state)
{
  Queue<Integer> queue = new ArrayDeque<Integer>();
  queue.add(vert);
  while (!queue.isEmpty())
  {
    int v = queue.poll();
    for (int i = 0; i < adj[v].size(); ++i)
    {
      int w = adj[v].get(i);
      if (state[w] == INIT)
      {
        queue.add(w);
```

```
          state[w] = PROCESSING;
        }
      }
      state[v] = FINISHED;
    }
  }
  void bfs(ArrayList<Integer>[] adj)
  {
    int[] state = new int[adj.length];
    for (int i = 0; i < state.length; ++i) if (state[i] == INIT)
       bfs(adj,i,state);
  }
```

The two algorithms will only process each vertex once, so they each determine a forest in a graph. It is sometimes useful to classify edges in the graph by their relationship to the constructed forest. An edge participating in the forest is called a tree edge. An edge from a node to one of its ancestors in the forest is called a back edge. An edge from a vertex to one of its non-immediate children is a forward edge. The remaining edges are called cross edges (only possible in directed graphs). The non-tree edges are found by DFS when it is iterating over the neighbors of a node that are not in the INIT state.

Search Application Exercises

1. Show how to compute the number components in a graph. Give the runtime of your algorithm for an adjacency list and an adjacency matrix.
2. In an undirected graph, a bridge is an edge whose removal increases the number of connected components. Give a simple algorithm for finding all bridges of a graph.
3. You are on an $n \times n$ grid located at the lower left corner $(0,0)$. Each turn you can take one step horizontally or vertically. You are also given a list of obstacles L which you must avoid. Determine the length of the shortest path from $(0,0)$ to $(n-1,n-1)$.
4. A graph is called 2-colorable (or bipartite) if each vertex can be assigned the color red or blue such that adjacent vertices have different colors. Give an algorithm to determine if a graph is bipartite.

Solutions

1. Run DFS. In the outer loop, count the number of times we must call DFS on a vertex in the INIT state. The runtime is $O(E+V)$ for an adjacency list and $O(V^2)$ for an adjacency matrix.
2. Count the number of components. Then, one at a time, remove each edge and see if it changes the number of components. The runtime is $O(VE)$.

3. Run BFS starting from $(0, 0)$ avoiding obstacles and staying within the confines of the grid. Can be helpful to use an array

   ```
   int[][] dirs = { {0,1},{0,-1},{1,0},{-1,0} };
   ```

   to ease iterating over neighbors.

4. Run DFS or BFS. On each connected component the search will alternate coloring neighbors. If it ever finds an already colored neighbor of the wrong color, the graph is not bipartite.

DFS Tree Exercises

1. When performing a DFS (on undirected or directed), when a node v has just begun processing, what can be said of the other nodes that are processing?

2. Give an algorithm to determine if a directed graph is acyclic.

Solutions

1. They must be ancestors in the DFS tree.

2. It must have a cycle.

3. Run DFS and test if any of the neighbors of a node being processed are in state PROCESSING. If so, there must be a cycle as we have an edge from a node to an ancestor in the DFS tree. If this never occurs, the graph is acyclic. To see this, suppose u is in a cycle, and suppose u is the first element of the cycle reached by DFS. Then every element of the cycle will be a descendent of u, so at least one of them will have u as a neighbor.

More Advanced BFS and DFS I

As we have seen, breadth first search processes vertices in order of their distance to the root vertex, where each edge has length 1. If each edge has a positive weight or length associated with it, we can still process vertices in distance-to-root order by replacing the queue with a priority queue. This is called Dijkstra's algorithm and runs in $O(E \lg V)$ (the $O(V \lg V)$ Fibonacci Heap variant is too slow due to its large constant). If at each step the priority queue stores the distance to the finished vertices instead of the distance-to-root you get Prim's algorithm for computing an MST.

A particularly useful usage of DFS is to topologically sort a dag. The output of a topological sort is a list of vertices such that if w follows v then there can be no path from w to v. The idea is that we will generate the reverse of a topological sort using DFS. Remember that once DFS finishes a vertex, it has processed or finished all vertices reachable from that vertex.

```
void dfs(ArrayList<Integer>[] adj, int vert, int[] state,
   ArrayList<Integer> list)
{
  if (state[vert] != INIT) return;
  state[vert] = PROCESSING;
  for (int i = 0; i < adj[vert].size(); ++i)
     dfs(adj,adj[vert].get(i),state);
  state[vert] = FINISHED;
  list.add(vert);
}
ArrayList<Integer> topsort(ArrayList<Integer>[] adj)
{
  ArrayList<Integer> list = new ArrayList<Integer>();
  int[] state = new int[adj.length];
  for (int i = 0; i < state.length; ++i) if (state[i] == INIT)
     dfs(adj,i,state,list);
  Collections.reverse(list);
  return list;
}
```

Exercises

1. If you reverse all of the edges of a dag, is the resulting graph still a dag? If yes, what can be said about the topological sort of the reversed dag?

2. Give code to list every divisor of n such that if $a \mid b$, then a appears before b in your list.

3. In an undirected graph, suppose you are doing a DFS and have processed one child c_1 of the root. After the child has finished, there is another child c_2 still in the INIT state that must be processed. What happens to the graph if we remove the root?

4. Suppose we are running DFS and have begun processing v. Also assume each vertex is assigned a number by DFS based on the processing order. Consider T_v, the v-rooted subtree of the DFS tree that has been fully built when v is finished. Each edge emanating from a node u of T_v either goes to a vertex with higher, equal, or lower number than v. For each case, describe the edges and the nodes they terminate for undirected graphs.

Solutions

1. Reversing edges turns cycles into cycles, and reversing twice returns the original graph, so the reverse of a dag is a dag. The topological sort of the reversed dag is the reverse of a topological sort of the original dag (there is a path from $u \to v$ in G iff there is a path from $v \to u$ in the reverse of G).

2. Model the divisors of n as a graph where $a \to b$ if $a \mid b$. Then we run topological sort. This can be implemented simply (but slowly) with the following code:

```
HashSet<Integer> visited = new HashSet<Integer>();
void dfs(int n)
{
  if (visited.contains(n)) return;
  visited.add(n);
  for (int i = 2; i*i <= n; ++i)
  {
    if (n % i != 0) continue;
    dfs(i);
    dfs(n/i);
  }
  dfs(1);
  System.out.println(n);
}
```

3. It must disconnect c_1 from c_2 as the vertex c_2 cannot be reachable from c_1 except through the root.

4. If the terminating node of the edge has

   (a) Lower number than v: It must be an ancestor of v. If not, it would be finished, and thus could not be reached by this non-tree edge.

   (b) Same number as v: Then it must be a tree edge (i.e., v is the parent of u in T_v), or it is a "back edge", and thus there is a cycle containing u and v.

   (c) Higher number than v: to another element of T_v.

More Advanced BFS and DFS II

Another application of DFS is to find bridges and articulation points in an undirected graph. A bridge is an edge whose removal increases the number of connected components. An articulation point is a vertex whose removal increases the number of components. To find these we need to study back edges. Let's focus our study on one connected component of our graph. First, consider the root of our DFS tree. DFS will first visit one of the root's children. Afterward, if any of its remaining children are unprocessed, it will visit them as well. If there are any unvisited children after the first, then the root must be an articulation point (the unvisited child is not reachable from the previous children, except via the root).

For non-root nodes, we use the following idea. The DFS algorithm will process each node in some order, so each node is assigned a DFS number. Suppose DFS returns the lowest number of its descendents, and their immediate neighbors, but ignores neighbors that are immediate parents in the DFS tree (i.e., it ignores tree edges). Then a non-root node v is an articulation point if any of the DFS calls on its children return values less than or equal to that of v (i.e., the child's only path to lower numbered nodes goes through v). If the returned value from DFS on a child c is strictly less than v, then the edge (v, c) is a bridge. To see why this works, suppose we are processing v's child c in the DFS tree. DFS will then build a tree rooted at c consisting of reachable yet unvisited nodes. Every edge emanating from this c-rooted tree is either

1. An edge to a vertex with a higher number than v: an edge from the c-rooted tree to itself.
2. An edge to v (either the tree edge, or another edge).
3. An edge to a vertex with a lower number than v: must be an edge to one of v's ancestors (if it wasn't an ancestor, it would be finished, and hence the edge couldn't exist).

```
static int dfsnum = 0, rootChildren = 0;
//Parent is -1 on root
int dfs(ArrayList<Integer>[] adj, int vert, int parent, int[] state,
   int[] num)
{
  if (state[vert] != INIT) return num[vert];
  state[vert] = PROCESSING;
  int minNum = num[vert] = dfsnum++;
  for (int i = 0; i < adj[vert].size(); ++i)
  {
    int c = adj[vert].get(i);
    if (parent == -1 && state[c] == INIT) rootChildren++;
    if (c != parent)
    {
      int cVal = dfs(adj,c,vert,state,num);
      if (cVal > num[vert]) System.out.printf("(%d,%d) is a 
          bridge\n",vert,c);
      minNum = Math.min(cVal, minNum);
    }
  }
  if (cVal >= num[vert]) System.out.printf("%d is an articulation 
      point\n",vert);
  state[vert] = FINISHED;
  return minNum;
}
void print(ArrayList<Integer>[] adj)
{
  int[] state = new int[adj.length], num = new int[adj.length];
  for (int i = 0; i < state.length; ++i)
      if (state[i] == INIT)
      {
        rootChildren = 0;
```

```
            dfs(adj,i,-1,state,num);
            if (rootChildren > 1) System.out.printf("%d is an 
                articulation point\n",i);
        }
}
```