Graphs

Definition (undirected, unweighted): A graph G, consists of a set of vertices, V, (or nodes) and a set of edges, E, such that each edge is associated with a pair of vertices. We write G = (V,E).

A directed graph is the same as above, but where each edge is associated with an *ordered* pair of vertices.

A weighted graph is the same as above, but where each edge *also* has an associated real number with it, known as the edge weight.

Data Structures to Store Graphs

Adjacency Matrix

The standard adjacency matrix stores a matrix as a 2-D array with each slot in A[i][j] being a 1 if there is an edge from vertex i to vertex j, or storing a 0 otherwise. Alternatively, each entry in the array is null if no edge is connecting those vertices, or an Edge object that stores all necessary information about the edge. If it's a weighted graph, A[i][j] stores the edge weight of the edge connecting i to j. With undirected graphs, A[i][j] = A[j][i]. For weighted graphs, if there is no edge from i to j, there are several options (store a large integer, store -1, store null).

Although these are very easy to work with mathematically, they are more inefficient than Edge lists for several tasks. For example, you must scan all vertices to find all the edges incident to a vertex. In a relatively sparse graph, using an adjacency matrix would be very inefficient for running some of the algorithms we will learn.

Edge List Structure

An edge list is an array of lists, where A[i] stores each edge that goes from vertex i. For an unweighted graph, the list could just be a list of integers (adjacent vertices) and for a weighted graph the list could be of edge objects.

Graph Definitions

A complete undirected unweighted graph is one where there is an edge connecting all possible pairs of vertices in a graph. The complete graph with n vertices is denoted as K_n.

A graph is bipartite if *there exists* a way to partition the set of vertices V, in the graph into two sets V_1 and V_2, where $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \varnothing$, such that each edge in E contains one vertex from V_1 and the other vertex from V_2.

A complete bipartite graph on m and n vertices is denoted by $K_{m,n}$ and consists of m+n vertices, with each of the first m vertices connected to all of the other n vertices, and no other vertices.

A path of length n from vertex v_0 to vertex v_n is an alternating sequence of n+1 vertices and n edges beginning with vertex v_0 and ending with vertex v_n in which edge e_i is incident upon vertices v_{i-1} and v_i. (The order in which these are connected matters for a path in a directed graph in the natural way.)

A connected graph is one where you any pair of vertices in the graph is connected by at least one path.

A graph G'= (V', E') is a subgraph of G = (V, E) if V' $\subseteq$ V, E' $\subseteq$ E, and for every edge e' $\in$ E', if e' is incident on v' and w', then both of these vertices is contained in V'.

The function dist(v, w), where v and w are two vertices in a graph is defined as the length of the shortest path from v to w.

The diameter of a graph is simply the maximum distance between any two vertices in the graph.

More Graph Definitions...

A simple path is one that contains no repeated vertices.

A cycle is a path of non-zero length from and to the same vertex with no repeated edges.

A simple cycle is a cycle with no repeated vertices except for the first and last one.

A Hamiltonian cycle is a simple cycle that contains all the vertices in the graph.

An Euler cycle is a cycle that contains every edge in the graph exactly once. Note that a vertex may be contained in an Euler cycle more than once. Typically, these are known as Euler circuits, because a circuit has no repeated edges.

Interestingly enough, there is a nice simple method for determining if a graph has an Euler circuit, but no such method exists to determine if a graph has a Hamiltonian cycle. The latter problem is an NP-Complete problem. It a nutshell, this means it is most-likely difficult to solve perfectly in polynomial time. We will cover this topic at the end of the course more thoroughly, hopefully.

The complement of a graph G is a graph G' which contains all the vertices of G, but for each edge that exists in G, it is NOT in G', and for each possible edge NOT in G, it IS in G'.

Two graphs G and G' are isomorphic if there is a one-to-one correspondence between the vertices of the two graphs such that the resulting adjacency matrices are identical.

Graph Coloring

For graph coloring, we will deal with unweighted undirected graphs. To color a graph, you must assign a color to each vertex in a graph such that no two vertices connected by an edge are the same color.

Thus, a graph where all vertices are connected (a complete graph) must have all of its vertices colored separate colors.

All bipartite graphs can be colored with only two colors, and all graphs that can be colored with two colors are bipartite. To see this, first simply note that we can two-color a bipartite graph by simply coloring all the vertices in V_1 one color and all the vertices in V_2 the other color. To see the latter result, given a two-coloring of a graph, simply separate the vertices by color, putting all blue vertices on one side and all the red ones on the other. These two groups specify the existence of sets V_1 and V_2, as designated by the definition of bipartite graphs.

The minimum number of colors that is necessary to color a graph is known as its chromatic number.

Interestingly enough, there is an efficient solution to determine whether or not a graph can be colored with two colors or not,

but no efficient solution currently exists to determine whether or not a graph can be colored using three colors.

Graph Traversals

Depth First Search

The general "rule" used in searching a graph using a depth first search is to search down a path from a particular source vertex as far as you can go. When you can go to farther, "backtrack" to the last vertex from which a different path could have been taken. Continue in this fashion, attempting to go as deep as possible down each path until each node has been visited.

The most difficult part of this algorithm is keeping track of what nodes have already been visited, so that the algorithm does not run ad infinitum. We can do this by labeling each visited node and labeling "discovery" and "back" edges.

The algorithm is as follows:

DFS(Graph G,vertex v):
For all edges e incident to the start vertex v do:
1) If e is unexplored
a) Let e connect v to w.
b) If w is unexplored, then
i) Label e as a discovery edge
ii) Recursively call DFS(G,w)
else
iii) Label e as a back edge

In pseudocode, for the simplest version we do the following, not worrying specifically about marking discovery or back edges:

```
DFS(Graph G, vertex v, boolean[] visited) {

    visited[v] = true;
    for (vertex u: neighbor of v)
        if (!visited[u])
            DFS(G, u, visited);
}
```

To prove that this algorithm visits all vertices in the connected component of the graph in which it starts, note the following:

Let the vertex u be the first vertex on any path from the source vertex that is not visited. That means that w, which is connected to u was visited, but by the algorithm given, it's clear that if this situation occurs, u must be visited, contradicting the assumption that u was unvisited.

Next, we must show that the algorithm terminates. If it does not, then there must exist a "search path" that never ends. But this is impossible. A search path ends when an already visited vertex is visited again. The longest path that exists without revisiting a vertex is of length V, the number of vertices in the graph.

The running time of DFS is O(V+E). To see this, note that each edge and vertex is visited at most twice. In order to get this efficiency, an adjacency list must be used. (An adjacency matrix can not be used to complete this algorithm that quickly.)

Breadth First Search

The idea in a breadth first search is opposite to a depth first search. Instead of searching down a single path until you can go no longer, you search all paths at an uniform depth from the source before moving onto deeper paths. Once again, we'll need to mark both edges and vertices based on what has been visited.

In essence, we only want to explore one "unit" away from a searched node before we move to a different node to search from. All in all, we will be adding nodes to the back of a queue to be ones to searched from in the future. In the implementation on the following page, a set of queues L_i are maintained, each storing a list of vertices a distance of i edges from the starting vertex. One can implement this algorithm with a single queue as well. Let L_i be the set of vertices visited that are a path length of i from the source vertex for the algorithm.

BFS(Graph G,vertex s):
1) Let L_0 be empty
2) Insert s into L_0.
3) Let i = 0
4) While L_i is not empty do the following:
A) Create an empty container L_{i+1}.
B) For each vertex v in L_i do
i) For all edges e incident to v
a) if e is unexplored, mark endpoint w.
b) if w is unexplored
Mark it.
Insert w into L_{i+1}.
Label e as a discovery edge.
else
Label e as a cross edge.
C) i = i+1

In code, we might do this more simply as follows:

```
BFS(ArrayList[] G, vertex v) {

    int[] dist = new int[G.length];
    Arrays.fill(dist, -1);
    dist[v] = 0;
    LinkedList q = new LinkedList<Integer>();
    q.offer(v);
    while (q.size() > 0) {
        int cur = q.poll();
        for (Integer next: (ArrayList<Integer>)G[cur])
        if (dist[next] == -1) {
            dist[next] = dist[cur] + 1;
            q.offer(next);
        }
    }
}

}
```

The basic idea here is that we have successive rounds and continue with our rounds until no new vertices are visited on a round. For each round, we look at each vertex connected to the vertex we came from. And from this vertex we look at all possible connected vertices.

This leaves no vertex unvisited because we continue to look for vertices until no new ones of a particular length are found. If there are no paths of length 10 to a new vertex, surely there can be no paths of length 11 to a new vertex. The algorithm also terminates since no path can be longer than the number of vertices in the graph.

Directed Graphs

Traversals

Both of the traversals are essentially the same on a directed graph. When you run the algorithms, you must simply pay attention to the direction of the edges. Furthermore, you must keep in mind that you will only visit edges that are reachable from the source vertex.

Mark and Sweep Algorithm for Garbage Collection

A mark bit is associated with each object created in a Java program. It indicates if the object is live or not. When the JVM notices that the memory heap is low, it suspends all threads, and clears all mark bits. To garbage collect, we go through each live stack of current threads and mark all these objects as live. Then we use a DFS to mark all objects reachable from these initial live objects. (In particular each object is viewed as a vertex and each reference as a directed edge.) This completes marking all live objects. Then we scan through the memory heap freeing all space that has NOT been marked.