Graph Theory Definitions

First, a few definitions. A graph, G, is a pair of sets (V, E), where V is a finite set of vertices and E is a subset of VxV – a set of edges. That is, each edge is a pair of vertices. In directed graphs, each edge is an ordered pair – it has a tail vertex and a head vertex. In undirected graphs, each edge is an unordered pair of vertices. Some graphs can have self-edges – edges that connect a vertex to itself. Because E is a set, a graph cannot have duplicate edges. In weighted graphs, each edge has an associated weight – simply a number assigned to the edge.

Usually, the number of vertices in a graph is denoted by n = |V|, and the number of edges is denoted by m = |E|. Note that $0 \le m \le n^2$ if we allow self-edges and $0 \le m \le n(n-1)$ if we don't.

A walk in a graph is a finite sequence of vertices (v1, v2, ..., vk) such that for all i between 1 and k-1, $(v_i, v_{i+1}) \in E$. That is, each pair of neighbouring vertices in a walk must have an edge between them. This is called a walk from v_1 to v_k. A path is a walk that never visits the same vertex twice. The length of a path is the number of edges in it (for unweighted graphs) or the sum of edge weights (for weighted graphs). A cycle is a walk from some vertex u to u. An Euler cycle is a cycle that visits each edge exactly once. A Hamiltonian cycle is a cycle that visits each vertex exactly once. It's interesting that finding an Euler cycle in a graph can be done in O(n) time, but finding a Hamiltonian cycle is NP-hard – no one knows if it can be done in polynomial time.

There are several data-structures suited for representing graphs. An adjacency matrix, M, is an n-by-n matrix of zeroes and ones, where M[i][j] is 1 if and only if the edge (i, j) is in E. It requires $O(n^2)$ memory and can answer in constant time the question, "Does G have an edge (i,j)?" An adjacency list, L, is a set of lists, one for each vertex, where L[i] is a list of all vertices j, such that we have an edge (i,j). Since there are n such lists (one per vertex) and for each edge, there is one entry in the corresponding list, the structure requires $O(n+m)$ memory. A typical way of creating an adjacency list in C++ is to make a "vector< vector< int > > L;", where each vertex in an integer, and L[u] is a vector of all vertices connected to u by an edge. Finally, another common graph data-structure is simply an edge list – a list (or a set) of edges. It requires $O(m)$ space and can be represented by a set of pairs or a vector of pairs in C++.

An undirected graph is called connected if for every pair of vertices, u and v, there is a path from u to v. A subgraph of G=(V,E) is a graph G'=(V', E') where $V' \subset V$ and $E' \subset E$. A connected component of G is a maximal connected subgraph of G.

Depth First Search (DFS)

One of the most basic problems on graphs is the Graph Reachability problem: given a graph G and a vertex v in G, which other vertices can be reached by a path starting from v? One of the simplest algorithms for it is DFS – start by visiting v, mark it as "reachable" and then visit all of v's neighbours recursively. Suppose that we have an adjacency matrix representation of a graph. Then the simplest possible DFS implementation would looks like this.

Example 1:

```
 bool M[128][128];  // adjacency matrix (can have at most 128 vertices)
bool seen[128];    // which vertices have been visited by dfs()
int n;             // number of vertices

void dfs( int u ) {
    seen[u] = true;
    for( int v = 0; v < n; v++ ) if( !seen[v] && M[u][v] ) dfs( v );
}
```

To use dfs(), first initialize M to contain the adjacency matrix of some graph. Then, initialize all entries of seen[] to false (we have not seen any vertices yet). Then finally, call dfs(u) on some vertex u. After dfs(u) returns, seen[v] will be true for exactly those vertices v that can be reached by a path from u. Note that DFS induces what is called a DFS tree of the graph – a rooted tree of recursive calls to dfs(). One possible use of this implementation is flood fill – an algorithm that many paint programs use to implement the paint bucket tool.

However, for more complicated algorithms that rely on DFS, we often need what is called the White-Gray-Black DFS. First, consider each vertex as being white. Now we need to add several lines to the dfs() function – first, colour the vertex u gray when we first reach it; and second, color it black when dfs() returns. The changes are in bold.

Example 2: White-Gray-Black DFS

```
 bool M[128][128]; // adjacency matrix
int colour[128];  // 0 is white, 1 is gray and 2 is black
int dfsNum[128], num;       // array of DFS numbers, and the current DFS number
int n;                      // the number of vertices

// p is u's parent in the DFS tree
void dfs( int u, int p ) {
    colour[u] = 1;
    dfsNum[u] = num++;
    for( int v = 0; v < n; v++ ) if( M[u][v] && v != p ) {
        if( colour[v] == 0 ) {
            // (u,v) is a forward edge
            dfs( v, u );
        }
        else if( colour[v] == 1 ) {
            // (u,v) is a back edge
        }
        else {
            // (u,v) is a cross edge
        }
    }
    colour[u] = 2;
}

int main() {
    // ... fill up M[][] with an adjacency matrix
    // ... set n to be the number of vertices

    for( int i = 0; i < n; i++ ) colour[i] = 0;
    num = 0;
    dfs( 0, -1 );

    return 0;
}
```

There are several things going on here. First of all, note that the seen[] array has been replaced by the colour[] array. We also have an extra parameter to dfs() - the parent of u in the DFS tree. Now look at the first and last lines of dfs(). White vertices are those we haven't visited yet. Any vertex u is gray while the call dfs(u) is being executed (there can be several such calls on the stack at any moment). Black vertices are those we have already explored – that means visited them and all of their children in the DFS tree.

The vertex colouring also gives us 3 types of edges: gray-to-white edges are forward edges (edges of the DFS tree), gray-to-gray edges are back edges (edges up the DFS tree) and gray-to-black edges are cross edges (edges from one DFS tree branch to another). In undirected graphs, cross edges never appear. (Prove this to yourself.) All of these labellings depend on the root vertex that we call dfs() on. In the above example, the root is the vertex 0. What we do with the different types of edges depends on the particular algorithm that we are implementing.

Finally, the dfsNum[] array simply assigns to each vertex a DFS number – a number from 0 to n-1 that indicates the order in which the dfs() function visited the vertices. The variable num keeps track of the current DFS number and is incremented each time a new vertex is visited.

One example where White-Gray-Black DFS does a great job is the Bridge Detection problem. In a connected graph, G, a bridge is any edge that, if removed, would make the graph disconnected. For example, if G represents a network of telephone cables between cities, a bridge would be any cable between a pair of cities that is "important" in the sense that a malfunction in that cable would cause some pair of cities to be disconnected.

We are dealing with an undirected graph in the Bridge Detection problem, so we don't have cross-edges. The main observation is this: if some vertex u has a back edge pointing to it, then nothing below u in the DFS tree can be a bridge. The reason is that each back edge gives us a cycle, and no edge that is a member of a cycle can be a bridge. By the converse argument, if we have a vertex v whose parent in the DFS tree is u, and no ancestor of v has a back edge pointing to it, then (u, v) is a bridge.

Example 3: Bridge Detection

```
bool M[128][128];          // adjacency matrix
int colour[128];           // 0 is white, 1 is gray and 2 is black
int dfsNum[128], num;      // DFS numbers
int n;                     // the number of vertices

// returns the smallest DFS number that has a back edge pointing to it
// in the DFS subtree rooted at u
int dfs( int u, int p ) {
    colour[u] = 1;
    dfsNum[u] = num++;
    int leastAncestor = num;
    for( int v = 0; v < n; v++ ) if( M[u][v] && v != p ) {
        if( colour[v] == 0 ) {
            // (u,v) is a forward edge
            int rec = dfs( v, u );
            if( rec > dfsNum[u] )
                cout << "Bridge: " << u << " " << v << endl;
            leastAncestor = min( leastAncestor, rec );
        }
```

```
            else {
                // (u,v) is a back edge
                leastAncestor = min( leastAncestor, dfsNum[v] );
            }
        }
        colour[u] = 2;
        return leastAncestor;
    }

    int main() {
        // ... fill up M[][] with an adjacency matrix
        // ... set n to be the number of vertices

        for( int i = 0; i < n; i++ ) colour[i] = 0;
        num = 0;

        dfs( 0, -1 );

        return 0;
    }
```

There are several changes to note here. First of all, dfs() now returns an int – the smallest DFS number reachable via a back edge from some vertex in the DFS subtree of u. This DFS number is stored in the variable called leastAncestor, which is updated in two places – when we find a back edge and when we call dfs() recursively on a subtree. The rest of the new code (the if statement) says, "If a recursive call to dfs(v,u) returns a larger number than the DFS number of u, then we have found the bridge (u,v)."

Breadth First Search (BFS)

In Depth-First Search (DFS), we explore a vertex's neighbours recursively, meaning that we reach as much depth as possible first, then go back and visit othe rneighbours (and hence the name Depth-First). Another useful search algorithm is the Breadth-First Search (BFS). In BFS, we start with one vertex in a **visited set**, the source vertex. Then, at each step, we visit the entire layer of unvisited vertices reachable by some vertex in the visited set, and add them to the visited set. Doing so, BFS visits vertices in order of their **breadth**, or simply the distance from that vertex to the source. BFS builds its own Breadth-First tree, and is an iterative algorithm.

A simple problem that BFS is good at is the flood-fill problem, mentioned in the DFS section. A flood-fill simply fills all vertices reachable by some source vertex with the same colour, much like the paint bucket tool in image-editing programs. The idea is to visit the vertices in a breadth-first manner using BFS, and colour each vertex as we reach it.

Example 4: Flood-fill

```
     bool M[128][128];  // adjacency matrix (can have at most 128 vertices)
    bool seen[128];    // which vertices have been visited
    int n;             // number of vertices

    // ... Initialize M to be the adjacency matrix
    queue<int> q;  // The BFS queue to represent the visited set
    int s = 0;     // the source vertex
```

```
// BFS flood-fill
for( int v = 0; v < n; v++ ) seen[v] = false;   // set all vertices to be "unvisited"
seen[s] = true;
DoColouring( s, some_color );
q.push( s );

while ( !q.empty() ) {
    int u = q.front();  // get first un-touched vertex
    q.pop();
    for( int v = 0; v < n; v++ ) if( !seen[v] && M[u][v] ) {
        seen[v] = true;
        DoColouring( v, some_color );
        q.push( v );
    }
}
```

Now, several things are going on in this example. We use a queue to represent the visited set because a queue will keep the vertices in order of when they were first visited. This means the queue will keep the vertices in a **breadth-first** manner. We first start off colouring the source vertex, marking it as **seen**, and pushing it onto the queue. Now, for each vertex in the queue, we simply do the same for all of its neighbours – colour them, mark them as seen, and push them onto the queue. Note that BFS is an iterative algorithm, so we did not write it as a function.

BFS also has a White-Gray-Black version, although slightly less common. The properties of the coluring of vertices remain the same – white-gray are forward edges, gray-gray are upward edges, and gray-black are cross edges. On the other hand, BFS has a special property that we can exploit: in BFS, the vertices closest to the source vertex is seen first (by closest we man the fewest amount of edges used). Hence, we can use BFS to compute the shortest distance between a source vertex and any other vertex in an unweighted graph. Here is an example:

Example 5: Shortest path in unweighted graph

```
bool M[128][128];  // adjacency matrix (can have at most 128 vertices)
int colour[128];   // 0 is white, 1 is gray and 2 is black
int d[128];        // d[v] is the distance from source to v
int pi[128];       // pi[v] is the parent of v in the shortest path from source to v
int n;             // number of vertices

// ... Initialize M to be the adjacency matrix
queue<int> q;  // The BFS queue to represent the visited set
int s = 0;     // the source vertex

// BFS shortest-path
const int Inf = INT_MAX;  // Infinity!!
for( int v = 0; v < n; v++ ) {
    colour[v] = 0;    // set all vertices to be "unvisited"
    d[v]      = Inf;  // distance is Infinity initially, meaning "cannot be reached"
    pi[v]     = -1;   // -1 is not a vertex, meaning "no parent so far"
}

// Initializing properties of the source vertex
colour[s] = 1;
d[s]      = 0;  // distance from s to itself is 0
pi[s]     = -1; // no parent for source vertex
q.push( s );

while ( !q.empty() ) {
    int u = q.front();  // get first un-touched vertex
    q.pop();
```

```
        for( int v = 0; v < n; v++ ) {
            if( colour[v] == 0 && M[u][v] ) {  // (u,v) is edge, v is white
                colour[v] = 1;
                d[v]  = d[u] + 1;  // one more edge used, increment distance by 1
                pi[v] = u;         // using edge (u,v), so parent of v is u
                q.push( v );
            }
        }
        colour[u] = 2;
    }
```

This version of BFS has several differences. First, it replaces the **seen** boolean array with a White-Gray-Black **colour** array. Note that, although the properties of these colours remain the same, the BFS tree is **different** from the DFS tree on the same graph. A back-edge in the DFS tree may not be a back-edge in the BFS tree.

The d[] array is added to keep track of the distance from the source vertex. This is updated whenever we use an edge. At the same time, we also keep a pi[] array to tell us what the parent of a vertex is in the shortest path that we have found using BFS. This allows us to compute the actual shortest path that we have found. Note that, after this BFS algorithm, a value of `d[v] = Inf` indicates that v was never visited in the BFS (ie. v is not reachable by s). Hence, BFS solves the Reachability problem too.

So, when is DFS better, and when is BFS better? This question depends on the type of the problem that we are trying to solve. BFS visits each layer one at a time, and so if you know the solution you are searching for is at a low depth, or if your problem requires looking at every vertex anyway (like flood-fill), then BFS is good at this. BFS is also good with undirected graphs because we can compute shortest-distances, useful for many other purposes.

On the other hand, since DFS visits neighbours recursively, we have a choice on which neighbour to visit first. When describing DFS, we arbitrarily pick a neighbour of a vertex to recurse on. However, we can actually use heuristics to decide which path to search first, in hopes of improving the average run-time of the algorithm. An example of this is the A*-search in artificial intelligence. The DFS-tree is also more useful than the BFS-tree. Later you will see how we find strongly-connected components of a graph easily using the DFS-tree.

Another key difference between BFS and DFS is their memory usage. BFS keeps the entire frontier in memory, and so its memory size is dependent on how **dense** the search space is. On the other hand, DFS's memory usage depends solely on the **depth** of the search space, since it only recurse on one neighbour at a time. The memory complexity is often a key consideration in determining whether to use DFS or BFS.

References

- Cormen, Thomas H., et al. Introduction to Algorithms. , 2nd ed. Cambridge: The MIT press, 2002.