CS490: Problem Solving in Computer Science

Lecture 6: Introductory Graph Theory

Dustin Tseng
Mike Li

Wednesday January 16, 2006

Outline

- Introduction
- Depth First Search
- Breadth First Search

- Introduction
- Depth First Search
- Breadth First Search

Definitions

- A graph, G, is a pair of sets (V, E)
- V is a finite set of vertices
- E is subset of $|V| \times |V|$ edges
- each edge is a pair of vertices
- undirected graphs: edges are unordered pairs
- directed graphs: edges are ordered pairs
- weighted graphs: each edge has an associated weight
- some graphs have self-edges, no graph can have duplicate edges
- the number of vertices is usually denoted by $n = |V|$
- the number of edges is usually denoted by $m = |E|$
- $0 \leq m \leq n^2$ if we allow self-edges and $0 \leq m \leq n(n-1)$ if we don't

Definitions

- a walk in a graph is a finite sequence of vertices $(v_1, v_2, ..., v_k)$, such that for all i between 1 and $k-1$, $(v_i, v_{i+1}) \in E$
- a path is a walk that never visits the same vertex twice.
- unweighted graph: the length of a path is the number of edges in it
- weighted graph: the length of a path is the sum of edge weights
- a cycle is a walk from a vertex back to itself
- an Euler cycle is a cycle that visits each edge exactly once
- a Hamiltonian cycle is a cycle that visits each vertex exactly once
- finding an Euler cycle in a graph can be done in O(n) time
- finding a Hamiltonian cycle is NP-hard

Definitions

- a graph is called connected if there is a path between any pair of vertices
- 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

Data-structures

There are several data-structures suited for representing graphis

- an adjacency matrix M is an n×n matrix of 0s and 1s
- M[i][j] is 1 iff the edge (i,j) ∈ E.
- an adjacenty list L is a set of lists, one for each vertex
- L[i] is a list of all vertices such that j is in the list iff (i,j) ∈ E
- a typical way of creating an adjacency list in C++ is to make a `vector< vector< int > > L`
- finally an edge list is a list (or a set) of edges
- the memory complexity for the three structures are $O(n^2)$, $O(m+n)$ and $O(m)$, respectively

- Introduction
- Depth First Search
- Breadth First Search

Introduction

- 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 this problem is DFS
- start by visiting v, mark it as "reachable" and then visit all of v's neighbours recursively.

Implementation

Using a adjacency matrix, we can implement the DFS very easily as follows:

```
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 a graph.
- then initialize all entreis of seen[] to false
- 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

More DFS

- DFS induces what is called a DFS tree to the graph - a rooted tree of recurive calls to dfs()
- one possible use of our simple implementation is flood fill - an algorithm that many paint programs use to implement the paint bucket tool
- for more complicated algorithms that rely on DFS, we often need what is called the White-Gray-Black DFS.
- start by considering each vertex as being white.
- color a vertex gray when we first reach it
- color a vertex black when dfs() returns

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;
}
```

White-Gray-Black DFS

There are several things going on here

- the seen[] array has been replaced by the color[]rray
- dfs() now has a extra parameter to indicate the parent of the vertex in the DFS tree
- white vertices are those not yet visited
- any vertex u is gray while dfs(u) is being executed
- black vertices are those already explored - all their children in the DFS tree as well as themselves are visited

White-Gray-Black DFS

The vertex coloring also gives us three types of edges:

- gray-to-white edges: edges of the DFS tree
- gray-to-gray edges: edges up the DFS tree
- gray-to-black edges: edge from one DFS tree branch to another, cross edge
- in undirected graphs, cross edges never happen (exercise)

Finally

- dfsNum[] assigns each vertex a DFS number
- the DFS number is from 0 to n-1, indecating the order in which the dfs() visited the vertices
- the variable num keeps track of the current DFS number and gets incremented each time a new vertex is visited

DFS for Bridge Detection

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.

DFS for Bridge Detection

How can we use DFS here?

- first since we are dealing with an undirected graph, there is no cross-edges
- then observe that, 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, no edge that is a member of a cycle can be a bridge.
- so conversely, if there is 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)

DFS for 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;
}
```

DFS for Bridge Detection

Once again there are several changes to note here

- dfs(u) 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 leastAncestor, which is updated in two places
- first when we find a back edge
- second when we call dfs() recursively on a subtree
- the new 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)

- Introduction
- Depth First Search
- Breadth First Search

Introduction

- in DFS, we explore a vertex's neighbours recursively, meaning that we reach as deep as possible first then go back and visit other neighbours.
- 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
- therefore BFS visits vertices in order of their breadth, or the distance from that vertex to the source
- BFS is an iterative algorithm, it also builds a tree.

BFS for Flood Fill

A simple problem that BFS is good at is the flood-fill problem mentioned previously.

```
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 floodfill
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 untouched 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 );
    }
}
```

BFS for Flood Fill

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
- the queue will keep the vertices in a breadth-first manner
- we start by marking the source vertex as seen, coloring it and pushing it onto the queue
- next we do the same for all its neighbours, mark, color and the push
- since BFS is iterative, we need not to write it as a function

BFS for Shortest Path

BFS has a special property that we can exploit. In BFS, the vertices closes (least number edges) to the source vertex is seen first. Hence, we can use BFS to compute the shortest distance between a source certex and any other vertex in an 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 shortestpath
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"
}
```

BFS for Shortest Path

```
// 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 untouched 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;
}
```

BFS for Shortest Path

- Note now BFS also has a White-Gray-Black colour[] array instead of seen[]
- The properties of these colours remain the same, but the BFS tree is different from the DFS tree on the same graph.
- the d[] array is added to keep track of distance from the source vertex, which is updated whenever we use an edge
- the pi[] array tells us what the parent of a vertex is in the shortest path that we have found using BFS
- after this BFS implementation, a value of `d[v] = inf` indicates that v was never visited. Hence BFS also solves the reachability problem.

BFS vs DFS

So, when is DFS better and when is BFS better?

- the answer depends on the type of the problem we want to solve
- BFS visits each layer one at a time, good when
  - if we know the solution is at a low depth
  - if we need to look at every vertex anyway (e.g. flood fill)
- DFS visits neighbours recursively
- we can actually choose which neighour to visit first
- there are heuristics we can follow to make our decision and improve the average performance

What about space and time complexity?

What Else?

- References:
  - Frank and Igor's CS490 notes.
  - Cormen, Thomas H., et al. Introduction to Algorithms.
- server problems
- midterm 1 on Monday, Feb. 6
- seminar schedule is updated
- Good Luck!