Lecture 11:
Breadth-First Search

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

Problem of the Day

Present correct and efficient algorithms to convert between the following graph data structures, for an undirected graph G with n vertices and m edges. You must give the time complexity of each algorithm.

1. Convert from an adjacency matrix to adjacency lists.

2. Convert from an adjacency list to an incidence matrix. An incidence matrix M has a row for each vertex and a column for each edge, such that $M[i, j] = 1$ if vertex i is part of edge j, otherwise $M[i, j] = 0$.
3. Convert from an incidence matrix to adjacency lists.

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph.

For *efficiency*, we must make sure we visit each edge at most twice.

For *correctness*, we must do the traversal in a systematic way so that we don't miss anything.

Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.

Marking Vertices

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Each vertex will always be in one of the following three states:

- *undiscovered* – the vertex in its initial, virgin state.
- *discovered* – the vertex after we have encountered it, but before we have checked out all its incident edges.
- *processed* – the vertex after we have visited all its incident edges.

Obviously, a vertex cannot be *processed* before we discover it, so over the course of the traversal the state of each vertex progresses from *undiscovered* to *discovered* to *processed*.

To Do List

We must also maintain a structure containing all the vertices we have discovered but not yet completely explored.

Initially, only a single start vertex is considered to be discovered.

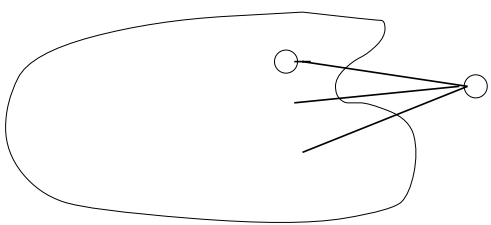
To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it *discovered* and add it to the list of work to do.

Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Suppose not, ie. there exists a vertex which was unvisited whose neighbor *was* visited. This neighbor will eventually be explored so we *would* visit it:



Breadth-First Traversal

The basic operation in most graph algorithms is completely and systematically traversing the graph. We want to visit every vertex and every edge exactly once in some well-defined order.

Breadth-first search is appropriate if we are interested in shortest paths on unweighted graphs.

Data Structures for BFS

We use two Boolean arrays to maintain our knowledge about each vertex in the graph.

A vertex is `discovered` the first time we visit it.

A vertex is considered `processed` after we have traversed all outgoing edges from it.

Once a vertex is discovered, it is placed on a FIFO queue. Thus the oldest vertices / closest to the root are expanded first.

```
bool processed[MAXV];
bool discovered[MAXV];
int parent[MAXV];
```

Initializing BFS

```
initialize_search(graph *g)
{
        int i;

        for (i=1; i<=g->nvertices; i++) {
                processed[i] = discovered[i] = FALSE;
                parent[i] = -1;
        }
}
```

BFS Implementation

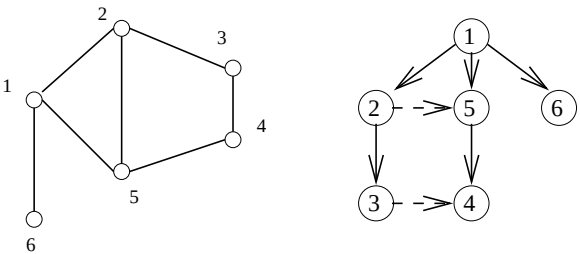
```
bfs(graph *g, int start)
{
        queue q;
        int v;
        int y;
        edgenode *p;

        init_queue(&q);
        enqueue(&q,start);
        discovered[start] = TRUE;

        while (empty_queue(&q) == FALSE) {
                v = dequeue(&q);
                process_vertex_early(v);
                processed[v] = TRUE;
                p = g->edges[v];
                while (p != NULL) {
                        y = p->y;
                        if ((processed[y] == FALSE) || g->directed)
                                process_edge(v,y);
                        if (discovered[y] == FALSE) {
```

```
                enqueue(&q,y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p− >next;
        }
        process_vertex_late(v);
    }
}
```

BFS Example



Exploiting Traversal

We can easily customize what the traversal does as it makes one official visit to each edge and each vertex. By setting the functions to

```
process_vertex(int v)
{
        printf("processed vertex %d ",v);
}

process_edge(int x, int y)
{
        printf("processed edge (%d,%d) ",x,y);
}
```

we print each vertex and edge exactly once.

Finding Paths

The `parent` array set within `bfs()` is very useful for finding interesting paths through a graph.

The vertex which discovered vertex i is defined as `parent[i]`.

The parent relation defines a tree of discovery with the initial search node as the root of the tree.

Shortest Paths and BFS

In BFS vertices are discovered in order of increasing distance from the root, so this tree has a very important property.
The unique tree path from the root to any node $x \in V$ uses the smallest number of edges (or equivalently, intermediate nodes) possible on any root-to-x path in the graph.

Recursion and Path Finding

We can reconstruct this path by following the chain of ancestors from x to the root. Note that we have to work backward. We cannot find the path from the root to x, since that does not follow the direction of the parent pointers. Instead, we must find the path from x to the root.

```
find_path(int start, int end, int parents[])
{
        if ((start == end) || (end == -1))
                printf("%d",start);
        else {
                find_path(start,parents[end],parents);
                printf(" %d",end);
        }
}
```

Connected Components

The *connected components* of an undirected graph are the separate "pieces" of the graph such that there is no connection between the pieces.

Many seemingly complicated problems reduce to finding or counting connected components. For example, testing whether a puzzle such as Rubik's cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected.

Anything we discover during a BFS must be part of the same connected component. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found:

Implementation

```
connected_components(graph *g)
{
        int c;
        int i;

        initialize_search(g);

        c = 0;
        for (i=1; i<=g– >nvertices; i++)
                if (discovered[i] == FALSE) {
                        c = c+1;
                        printf(”Component %d:”,c);
                        bfs(g,i);
                }
}
```

Two-Coloring Graphs

The *vertex coloring* problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.

A graph is *bipartite* if it can be colored without conflicts while using only two colors. Bipartite graphs are important because they arise naturally in many applications.

For example, consider the "had-sex-with" graph in a heterosexual world. Men have sex only with women, and vice versa. Thus gender defines a legal two-coloring.

Finding a Two-Coloring

We can augment breadth-first search so that whenever we discover a new vertex, we color it the opposite of its parent.

```
twocolor(graph *g)
{
        int i;

        for (i=1; i<=(g– >nvertices); i++)
                color[i] = UNCOLORED;

        bipartite = TRUE;

        initialize_search(&g);

        for (i=1; i<=(g– >nvertices); i++)
                if (discovered[i] == FALSE) {
                        color[i] = WHITE;
                        bfs(g,i);
                }
}
```

```
process_edge(int x, int y)
{
        if (color[x] == color[y]) {
                bipartite = FALSE;
                printf(”Warning: graph not bipartite, due to (%d,%d)”,x,y);
        }

        color[y] = complement(color[x]);
}


complement(int color)
{
if (color == WHITE) return(BLACK);
if (color == BLACK) return(WHITE);

return(UNCOLORED);
}
```

We can assign the first vertex in any connected component to be whatever color/sex we wish.