LEC11: BREADTH-FIRST SEARCH

Lecture slide courtesy of Prof. Steven Skiena
TRAVERING A GRAPH

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.
  + E.g. printing or copying graphs, and converting between alternate representations
- For **efficiency**, we must make sure we don’t visit each edge repeatedly.
- 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.
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 -> discovered -> processed.
We must also maintain a structure containing all the vertices we have discovered but not yet completely processed.

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

To completely process 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. (do nothing to vertices already processed & vertices *discovered* but not *processed*.)
CORRECTNESS OF GRAPH TRAVERSAL

- Each undirected edge will be considered exactly twice,
  - once when each of its endpoints is explored.
- Directed edges will be considered only once,
  - when exploring the source vertex.
- Every edge and vertex in the connected component must eventually be visited.
  - Suppose that there exists a vertex \( u \) that remains unvisited, whose neighbor \( v \) was visited.
  - This neighbor \( v \) will eventually be explored, after which we will certainly visit \( u \).
  - Thus, we must find everything that is there to be found.
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.
  + In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discoverer $u$ to the discovered $v$. We thus denote $u$ to be the parent of $v$. 

BFS(G, s)

for each vertex \( u \in V[G] - \{s\} \) do

\[ state[u] = \text{“undiscovered”} \]

\[ p[u] = \text{nil}, \text{i.e. no parent is in the BFS tree} \]

\[ state[s] = \text{“discovered”} \]

\[ p[s] = \text{NULL} \]

\[ Q = \{s\} \]

while \( Q = \emptyset \) do

\[ u = \text{dequeue}[Q] \]

process vertex \( u \) as desired

for each \( v \in Adj[u] \) do

process edge \((u, v)\) as desired

if \( state[v] = \text{“undiscovered”} \) then

\[ state[v] = \text{“discovered”} \]

\[ p[v] = u \]

enqueue\([Q, v]\)

\[ state[u] = \text{“processed”} \]
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.
BFS IMPLEMENTATION - INITIALIZING BFS

Global Variables:
- bool processed[MAXV+1];
- bool discovered[MAXV+1];
- int parent[MAXV+1];

Each vertex is initialized as undiscovered:

initialize search(graph *g)
{
    int i;
    for (i=1; i<=g->nvertices; i++) {
        processed[i] = discovered[i] = FALSE;
        parent[i] = -1;
    }
}

BFS IMPLEMENTATION

Once a vertex is discovered, it is placed on a queue

```
bfs(graph *g, int start)
{
    queue q; /* queue of nodes to visit */
    int v;    /* current vertex */
    int y;       /* successor vertex */
    edgenode *p;     /* temporary pointer */

    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);
    }
}
```
EXPLOITING TRAVERSAL

- The exact behavior of bfs depends upon the functions process vertex early(), process vertex late(), and process edge(). By setting the functions to

- By setting the active functions to

  ```c
  process_vertex(int v) {
    printf("processed vertex %d\n",v);
  }
  process_edge(int x, int y) {
    printf("processed edge (%d,%d) ",x,y);
  }
  ```

- we print each vertex and edge exactly once.
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.
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.
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.
FINDING PATH EXAMPLE

```c
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);
    }
}
```

For the shortest path from 1 to 4, upper-right corner, this parent relation yields the path \{1, 5, 4\}. 
A graph is *connected* if there is a path between any two vertices.

The *connected components* of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices.

The *components* are 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.

- EX> 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.

Connected components can be found using BFS

- Anything we discover during a BFS must be part of the same connected component.
- 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 = 0;  /* component number */
    int i;     /* counter */

    initialize_search(g);
    for (i=1; i<=g->nvertices; i++){
        if (discovered[i] == FALSE) {
            c = c + 1;
            printf("Component %d:",c);
            bfs(g,i);
            printf("\n");
        }
    }
}

process_vertex_early(int v) { printf(" %d",v); }

process_edge(int x, int y) { }

\textbf{O}(n + m)
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.

The goal is to **use as few colors** as possible.

Vertex coloring problems often arise in **scheduling applications**

- Ex> register allocation in compilers
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 “married-to” graph in a heterosexual world. Men have marry only with women, and vice versa.
- Thus gender defines a legal two-coloring.
Solution Scratch (Augmented BFS):

+ Whenever we discover a new vertex, color it the opposite of its parent.

+ Check for conflict:
  - We check whether any nondiscovery edge links two vertices of the same color.

+ We will have constructed a proper two-coloring whenever we terminate without conflict.
FINDING A TWO-COLORING

twocolor(graph *g)
{
    int i; /* counter */
    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) f
            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 we wish.