Graphs

Graphs are one of the unifying themes of computer science.

A graph $G = (V, E)$ is defined by a set of *vertices* V, and a set of *edges* E consisting of ordered or unordered pairs of vertices from V.

In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges. In analyzing human interactions, the vertices typically represent people, with edges connecting pairs of related souls.

The first step in any graph problem is determining which flavor of graph you are dealing with:

- *Undirected vs. Directed* – A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

- *Weighted vs. Unweighted* – In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.

- *Cyclic vs. Acyclic* – An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs.

  Directed acyclic graphs are called *DAGs*.

- *Implicit vs. Explicit* – Many graphs are not explicitly constructed and then traversed, but built as we use them. A good example is in backtrack search.

Data Structures for Graphs

There are several possible ways to represent graphs. We assume the graph $G = (V, E)$ contains n vertices and m edges.

- *Adjacency Matrix* – We can represent G using an $n \times n$ matrix M, where element $M[i, j]$ is, say, 1, if (i, j) is an edge of G, and 0 if it isn't. It may use excessive space for graphs with many vertices and relatively few edges, however.

- *Adjacency Lists in Lists* – We can more efficiently represent sparse graphs by using linked lists to store the neighbors adjacent to each vertex. Adjacency lists require pointers but are not frightening once you have experience with linked structures.

- *Adjacency Lists in Matrices* – Adjacency lists can also embedded in matrices, thus eliminating the need for pointers. We can represent a list in an array counting how many elements there are, and packing them into the first elements of the array. Now we can visit successive the elements from the first to last by incrementing an index in a loop instead of cruising through pointers.

  This data structure looks like it combines the worst properties of adjacency matrices (large space) with the worst properties of adjacency lists (the need to search for edges). However, it is simple to program and understand – and what I will use in my examples.

List in Array Representation

We represent a graph using the following data type. For each graph, we keep count of the number of vertices, and assign each vertex a unique number from 1 to **nvertices**. We represent the edges in an MAXV $\times$ MAXDEGREE array, so each vertex can be adjacent to MAXDEGREE others. By defining MAXDEGREE to be MAXV, we can represent any simple graph, but this is wasteful of space for low-degree graphs:

```
#define MAXV            100       /* maximum number of vertices */
#define MAXDEGREE       50        /* maximum vertex outdegree */

typedef struct {
   int edges[MAXV+1][MAXDEGREE];  /* adjacency info */
   int degree[MAXV+1];            /* outdegree of each vertex */
   int nvertices;                 /* number of vertices in graph */
   int nedges;                    /* number of edges in graph */
} graph;
```

We represent a directed edge (x, y) by the integer y in x's adjacency list, which is located in the subarray **graph->edges[x]**. The degree field counts the number of meaningful entries for the given vertex. An undirected edge (x, y) appears twice in any adjacency-based graph structure, once as y in x's list, and once as x in y's list.

Reading a Graph

To demonstrate the use of this data structure, we show how to read in a graph from a file. A typical graph format consists of an initial line featuring the number of vertices and edges in the graph, followed by a listing of the edges at one vertex pair per line.

```
read_graph(graph *g, bool directed)
{
        int i;                                  /* counter */
        int m;                                  /* number of edges */
        int x, y;                               /* vertices in edge (x,y) */

        initialize_graph(g);

        scanf("%d %d",&(g->nvertices),&m);

        for (i=1; i<=m; i++) {
                scanf("%d %d",&x,&y);
                insert_edge(g,x,y,directed);
        }
}

initialize_graph(graph *g)
{
        int i;                                  /* counter */

        g -> nvertices = 0;
        g -> nedges = 0;

        for (i=1; i<=MAXV; i++) g->degree[i] = 0;
}
```

Inserting an Edge

The critical routine is **insert_edge**. We parameterize it with a Boolean flag **directed** to identify whether we need to insert two copies of each edge or only one. Note the use of recursion to solve the problem:

```
insert_edge(graph *g, int x, int y, bool directed)
{
    if (g->degree[x] > MAXDEGREE)
        printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);

    g->edges[x][g->degree[x]] = y;
    g->degree[x] ++;

    if (directed == FALSE)
            insert_edge(g,y,x,TRUE);
    else
            g->nedges ++;
}
```

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.

Our breadth-first search implementation **bfs** uses 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. Thus each vertex passes from undiscovered to discovered to processed over the course of the search.

Once a vertex is discovered, it is placed on a queue. Since we process these vertices in first-in, first-out order, the oldest vertices are expanded first, which are exactly those closest to the root:

```
bool processed[MAXV];   /* which vertices have been processed */
bool discovered[MAXV];  /* which vertices have been found */
int parent[MAXV];       /* discovery relation */

initialize_search(graph *g)
{
        int i;                          /* counter */

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

BFS Implementation

```
bfs(graph *g, int start)
{
        queue q;                /* queue of vertices to visit */
        int v;                  /* current vertex */
        int i;                  /* counter */

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

        while (empty(&q) == FALSE) {
                v = dequeue(&q);
                process_vertex(v);
                processed[v] = TRUE;
                for (i=0; i<g->degree[v]; i++)
                    if (valid_edge(g->edges[v][i]) == TRUE) {
                        if (discovered[g->edges[v][i]] == FALSE) {
                                enqueue(&q,g->edges[v][i]);
                                discovered[g->edges[v][i]] = TRUE;
                                parent[g->edges[v][i]] = v;
                        }
                        if (processed[g->edges[v][i]] == FALSE)
                                process_edge(v,g->edges[v][i]);
                    }
        }
}
```

Exploiting Traversal

The exact behavior of **bfs** depends upon the functions **process_vertex()** and **process_edge()**. Through these functions, 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\n",v);
}

process_edge(int x, int y)
{
        printf("processed edge (%d,%d)\n",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]`. Every vertex is discovered during the course of traversal, so except for the root every node has a parent. The parent relation defines a tree of discovery with the initial search node as the root of the tree.

Because vertices are discovered in order of increasing distance from the root, 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.

Since this is the reverse of how we normally want the path, we can either (1) store it and then explicitly reverse it using a stack, or (2) let recursion reverse it for us, as in the following slick routine:

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

Depth-First Search

Depth-first search uses essentially the same idea as backtracking. Both involve exhaustively searching all possibilities by advancing if it is possible, and backing up as soon as there is no unexplored possibility for further advancement. Both are most easily understood as recursive algorithms.

Depth-first search can be thought of as breadth-first search with a stack instead of a queue. The beauty of implementing **dfs** recursively is that recursion eliminates the need to keep an explicit stack:

```
dfs(graph *g, int v)
{
        int i;                          /* counter */
        int y;                          /* successor vertex */

        if (finished) return;           /* allow for search termination */

        discovered[v] = TRUE;
        process_vertex(v);

        for (i=0; i<g->degree[v]; i++) {
                y = g->edges[v][i];
                if (valid_edge(g->edges[v][i]) == TRUE) {
                        if (discovered[y] == FALSE) {
                                parent[y] = v;
                                dfs(g,y);
                        } else
                                if (processed[y] == FALSE)
                                        process_edge(v,y);
                }
                if (finished) return;
        }

        processed[v] = TRUE;
}
```

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.

An amazing number of 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.

Connected components can easily be found using depth-first search or breadth-first search. Anything we discover during this search 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:

```
connected_components(graph *g)
{
        int c;                          /* component number */
        int i;                          /* counter */

        initialize_search(g);

        c = 0;
        for (i=1; i<=g->nvertices; i++)
                if (discovered[i] == FALSE) {
                        c = c+1;
                        printf("Component %d:",c);
                        dfs(g,i);
                        printf("\n");
                }
}
```

Topological Sorting

Topological sorting is the fundamental operation on directed acyclic graphs (DAGs). It constructs an ordering of the vertices such that all directed edges go from left to right. Such an ordering clearly cannot exist if the graph contains any directed cycles, because there is no way you can keep going right on a line and still return back to where you started from!

The importance of topological sorting is that it gives us a way to process each vertex before any of its successors.

Suppose we seek the shortest (or longest) path from x to y in a DAG. Certainly no vertex appearing after y in the topological order can contribute to any such path, because there will be no way to get back to y. We can appropriately process all the vertices from left to right in topological order, considering the impact of their outgoing edges, and know that we will look at everything we need before we need it.

Topological sorting can be performed efficiently by using a version of depth-first search. However, a more straightforward algorithm is based on an analysis of the in-degrees of each vertex in a DAG. If a vertex has no incoming edges, i.e., has in-degree 0, we may safely place it first in topological order. Deleting its outgoing edges may create new in-degree 0 vertices. This process will continue until all vertices have been placed in the ordering; if not, the graph contained a cycle and was not a DAG in the first place.

```
topsort(graph *g, int sorted[])
{
        int indegree[MAXV];             /* indegree of each vertex */
        queue zeroin;                   /* vertices of indegree 0 */
        int x, y;                       /* current and next vertex */
        int i, j;                       /* counters */

        compute_indegrees(g,indegree);
        init_queue(&zeroin);
        for (i=1; i<=g->nvertices; i++)
                if (indegree[i] == 0) enqueue(&zeroin,i);

        j=0;
        while (empty(&zeroin) == FALSE) {
                j = j+1;
                x = dequeue(&zeroin);
                sorted[j] = x;
                for (i=0; i<g->degree[x]; i++) {
                        y = g->edges[x][i];
                        indegree[y] --;
                        if (indegree[y] == 0) enqueue(&zeroin,y);
                }
        }

        if (j != g->nvertices)
                printf("Not a DAG -- only %d vertices found\n",j);
}

compute_indegrees(graph *g, int in[])
{
        int i,j;                        /* counters */

        for (i=1; i<=g->nvertices; i++) in[i] = 0;

        for (i=1; i<=g->nvertices; i++)
                for (j=0; j<g->degree[i]; j++) in[ g->edges[i][j] ] ++;
}
```

Assigned Problems

110902 (Playing with Wheels) – Move from one number to another number using a minimum number of steps, while avoiding forbidden states. What is the graph, and is it weighted or unweighted?

110903 (The Tourist Guide) – Minimize the number of trips a guide needs to make in order to ferry a group of tourists from one place to another. What is the graph theoretic problem being solved here?

110906 (Tower of Cubes) – Build the tallest possible of cubes such that touching face colors match and lighter cubes are always on top of heavier cubes. What is the graph, and is it directed or undirected?

110907 (From Dusk Till Dawn) – Find the shortest train trip between two different cities subject to a vampire's particular constraints. What is the graph, and is it weighted or unweighted?