Chapter 9

Graph Traversal

Summary of Chapter 9- from the book
Programming Challenges: The Programming Contest Training Manual
By: Steven S. Skiena , and Miguel A. Revilla
2003 Springer-Verlag New York, Inc.
ISBN: 0-387-00163-8

Graph Traversal:
Graphs are one of the unifying themes of computer science − an abstract representation which describes the organization of transportation systems, electrical circuits, human interactions, and telecommunication networks. That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.
In this chapter, we focus on problems which require only an elementary knowledge of graph algorithms, specifically the appropriate use of graph data structures and traversal algorithms. In Chapter 10, we will present problems relying on more advanced graph algorithms that find minimum spanning trees, shortest paths, and network flows.

Flavors of Graphs:
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 the source code of a computer program, the vertices may represent lines of code, with an edge connecting lines x and y if y can be the next statement executed after x. In analyzing human interactions, the vertices typically represent people, with edges connecting pairs of related souls.
There are several fundamental properties of graphs which impact the choice of data structures used to represent them and algorithms available to analyze them. 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. If not, we say that the graph is *directed*. Road networks *between* cities are typically undirected, since any large road has lanes going in both directions. Street networks *within* cities are almost always directed, because there are typically at least a few one-way streets lurking about. Program-flow graphs are typically directed, because the execution flows from one line into the next and changes direction only at branches. Most graphs of graph-theoretic interest are undirected.
- *Weighted vs. Unweighted* — In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight. Typical application-specific edge weights for road networks might be the distance, travel time, or maximum capacity between x and y. In *unweighted* graphs, there is no cost distinction between various edges and vertices.

  The difference between weighted and unweighted graphs becomes particularly apparent in finding the shortest path between two vertices. For unweighted graphs, the shortest path must have the fewest number of edges, and can be found using the breadth-first search algorithm discussed in this chapter. Shortest paths in weighted graphs requires more sophisticated algorithms.
- *Cyclic vs. Acyclic* — An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs. Trees are the simplest interesting graphs, and inherently recursive structures since cutting any edge leaves two smaller trees.
  Directed acyclic graphs are called *DAGs*. They arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y. An operation called *topological* sorting orders the vertices of a *DAG* so as to respect these precedence constraints. Topological sorting is typically the first step of any algorithm on a *DAG*.
- *Simple vs. Non-simple* — Certain types of edges complicate the task of working with graphs. A *self-loop* is an edge (x, x) involving only one vertex. An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.

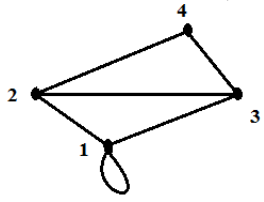
Both of these structures require special care in implementing graph algorithms. Hence any graph which avoids them is called *simple*.

- *Embedded vs. Topological* — A graph is *embedded* if the vertices and edges have been assigned geometric positions. Thus any drawing of a graph is an embedding, which may or may not have algorithmic significance.
  Occasionally, the structure of a graph is completely defined by the geometry of its embedding. For example, if we are given a collection of points in the plane, and seek the minimum cost tour visiting all of them (i.e., the traveling salesman problem), the underlying topology is the *complete graph* connecting each pair of vertices. The weights are typically defined by the Euclidean distance between each pair of points.
  Another example of topology from geometry arises in grids of points. Many problems on an $n \times m$ grid involve walking between neighboring points, so the edges are implicitly defined from the geometry.
- *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. The vertices of this implicit search graph are the states of the search vector, while edges link pairs of states which can be directly generated from each other. It is often easier to work with an implicit graph than explicitly constructing it before analysis.
- *Labeled vs. Unlabeled* — In *labeled* graphs, each vertex is assigned a unique name or identifier to distinguish it from all other vertices. In *unlabeled* graphs, no such distinctions have been made.
  Most graphs arising in applications are naturally and meaningfully labeled, such as city names in a transportation network. A common problem arising on graphs is that of *isomorphism testing*, determining whether the topological structure of two graphs are in fact identical if we ignore any labels. Such problems are typically solved using backtracking, by trying to assign each vertex in each graph a label such that the structures are identical.

Data Structures for Graphs:

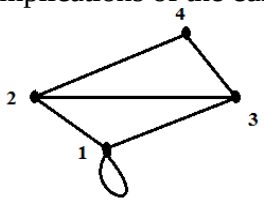
There are several possible w*ays to represent graphs. We discuss four useful representations below. We assume the graph G = (*V, *E*) contains *n* vertices and *m* edges.

- *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. This allows fast answers to the question "is (*i*, *j*) in *G*?", and rapid updates for edge insertion and deletion. It may use excessive space for graphs with many vertices and relatively few edges, however.
  Consider a graph which represents the street map of Manhattan in New York City. Every junction of two streets will be a vertex of the graph, with neighboring junctions connected by edges. How big is this graph? Manhattan is basically a grid of 15 avenues, each crossing roughly 200 streets. This gives us about 3000 vertices and 6000 edges, since each vertex neighbors four other vertices and each edge is shared between two vertices. Such a small amount of data should easily and efficiently be stored, but the adjacency matrix will have $3000 \times 3000 = 9000000$ cells, almost all of them empty!



$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

• *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 make it harder to ask whether a given edge (*i*, *j*) is in *G*, since we have to search through the appropriate list to find the edge. However, it is often surprisingly easy to design graph algorithms which avoid any need for such queries. Typically, we sweep through all the edges of the graph in one pass via a breadth-first or depths-first traversal, and update the implications of the current edge as we visit it.



For undirected graph the adjacency list is:
{{1, 1}, {1, 2}, {1, 3}, {2, 3}, {2, 4}, {4, 3}}

• *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 (or equivalently, a row of a matrix) by keeping a count *k* of how many elements there are, and packing them into the first *k* elements of the array. Now we can visit successive the elements from the first to last just like a list, but 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, there is a method to its madness. First, it is the simplest data structure to program, particularly for static graphs which do not change after they are built. Second, the space problem can in principle be eliminated by allocating the rows for each vertex dynamically, and making them exactly the right size.
To prove our point, we will use this representation in all our examples below.

• *Table of Edges* — An even simpler data structure is just to maintain an array or linked list of the edges. This is not as flexible as the other data structures at answering "who is adjacent to vertex x?" but it works just fine for certain simple procedures like Kruskal's minimum spanning tree algorithm.

As stated above, we will use adjacency lists in matrices as our basic data structure to represent graphs. It is not complicated to convert these routines to honest pointer-based adjacency lists. Sample code for adjacency lists and matrices can be found in many books.
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 × 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:

```
const int MAXV = 100;                       // maximum number of vertices
const int MAXDEGREE = 50;                   // maximum vertex outdegree

struct graph
{
        int edges[MAXV+1][MAXDEGREE];       // adjacency info
        int degree[MAXV+1];                 // out-degree of each vertex
        int nVertices;                      // number of vertices in graph
```

```
        int nEdges;                                     // number of edges in graph
};

void read_graph(graph *, bool);
void initialize_graph(graph *);
void insert_edge(graph *, int, int, bool);
void print_graph(graph *);

void main()
{
        graph *p, u;
        bool directed;

        p = &u;

        cout << "If the grpah is directed enter 1, otherwise enter 0: ";
        cin >> directed;
        read_graph(p, directed);
        print_graph(p);
}
```

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.
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.

```
void 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);

        cout << "Enter the number of vertices and edges: ";
        cin >> g -> nVertices >> m;
        for (i=1; i<=m; i++)
        {
                cout << "Enter the vertices of edge number " << i << ": ";
                cin >> x >> y;
                insert_edge(g,x,y,directed);
        }
}
```

```
void initialize_graph(graph *g)
{
        int i;                                          // counter
        g -> nVertices = 0;
        g -> nEdges = 0;
        for (i=1; i<=MAXV; i++)
                g->degree[i] = 0;
}
```

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:

```
void insert_edge(graph *g, int x, int y, bool directed)
{
        if (g->degree[x] > MAXDEGREE)
                cout << "Warning: insertion (" << x << ", " << y
                        << ") exceeds max degree" << endl;

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

        if (directed == false)
                insert_edge(g,y,x,true);
        else
                g->nEdges ++;
}
```

Printing the associated graph is now simply a matter of nested loops:

```
void print_graph(graph *g)
{
        for (int i=1; i<=g->nVertices; i++)
        {
                for (int j=0; j<g->degree[i]; j++)
                {
                        cout << "{" << i << ", " << g->edges[i][j] << "} ";
                }
                cout << endl;
        }
}
```

Sample input / output:

```
Mark "C:\Courses\0211490\Programming\Cpp Programs\adjacencyListsMatrices\[
If the grpah is directed enter 1, otherwise enter 0: 1
Enter the number of vertices and edges: 3 3
Enter the vertices of edge number 1: 1 2
Enter the vertices of edge number 2: 1 3
Enter the vertices of edge number 3: 2 3
{1, 2} {1, 3}
{2, 3}
```

Sample input / output:

```
Mark "C:\Courses\0211490\Programming\Cpp Programs\adjacencyListsMatrices
If the grpah is directed enter 1, otherwise enter 0: 0
Enter the number of vertices and edges: 3 3
Enter the vertices of edge number 1: 1 2
Enter the vertices of edge number 2: 1 3
Enter the vertices of edge number 3: 2 3
(1, 2) (1, 3)
(2, 1) (2, 3)
(3, 1) (3, 2)
```

Graph Traversal: Breadth-First:

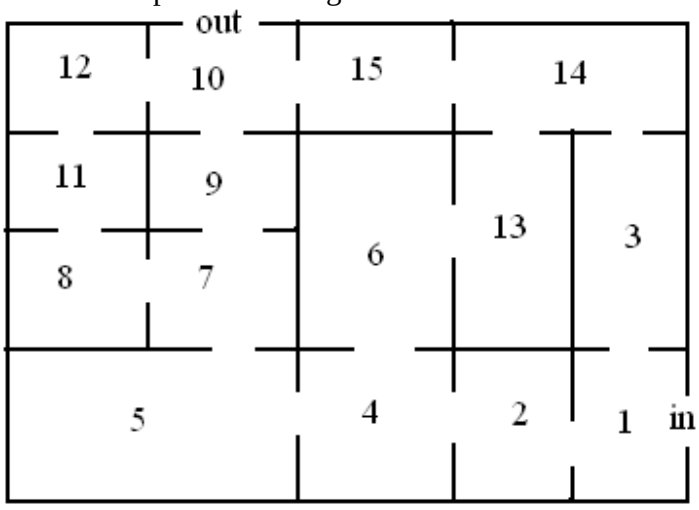
The basic operation in most graph algorithms is completely and systematically traversing the graph. We want to visit every vertex and every edge exactly once in some well-defined order. There are two primary traversal algorithms: *breadth-first search* (BFS) and *depth-first search* (DFS). For certain problems, it makes absolutely no different which one you use, but in other cases the distinction is crucial.

Both graph traversal procedures share one fundamental idea, namely, that it is necessary to mark the vertices we have seen before so we don't try to explore them again. Otherwise we get trapped in a maze and can't find our way out. BFS and DFS differ only in the order in which they explore vertices.

Breadth-first search is appropriate if (1) we don't care which order we visit the vertices and edges of the graph, so any order is appropriate or (2) we are interested in shortest paths on unweighted graphs.

Breadth-First Search:

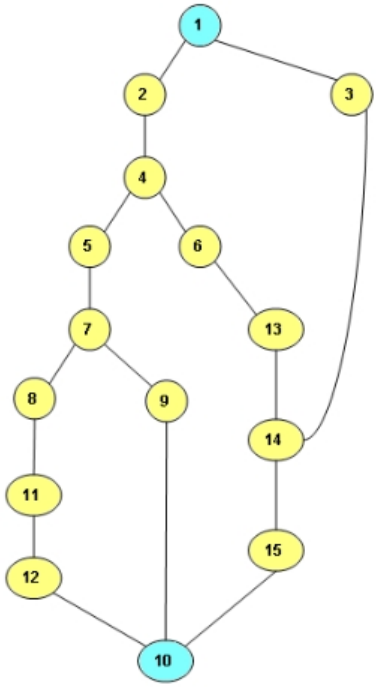
You are given the maze figure below and asked to find a way to the exit with a minimum number of decisions to make (a decision is required whenever there is at least one available direction to go). We have pointed out these critical positions and given them numbers.



On the basis of the above diagram we will draw a graph with the following rules :

- the vertices are the critical positions
- there is an edge from X to Y if we can go from X to Y by making exactly one decision

Also the circles colored in cyan are the start (1) and the finish (10).



It is easy to see now that the minimum length path is 1, 3, 14, 15, 10 with 4 decisions to make (the number of edges connecting the vertices). This is because we have an overview of the maze, we know every detail about it in advance. But what if we do not ? We would never be aware of the consequences of our current decision until we make it. What would be our strategy then ?

One possibility is to gather an infinite number of people (this is for instructional purposes only) and put them in the start position of the maze.

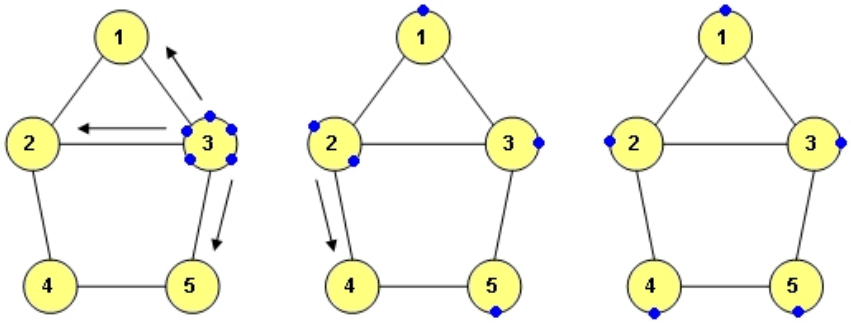
Then every time we are to make a decision, leave one single person in the current position and split the group into N smaller ones, where N is the number of the current possible decisions to make. These groups each go a different way and so on, until someone reaches the finish. It is clear that the path found by this group of people is of minimum length, since every group has to make only one decision at a time (the length of each step is the same for everyone) and this group made it first to the finish. Note that passing through a point that already has one person standing there is not permitted.

To reconstruct the minimum length path, let us assume that every person that is left in a certain point knows exactly the position where the group came from. For example the person in point 2 knows that the group came from the 1st position before it left him there. Now if we start with the last position of the winning group (the finish) we may go backwards as we know its previous position, and so on until we reach the start. We have now constructed the exact minimum path (in terms of decisions to make) to arrive at the finish of the maze, but in reversed order. The last operation is to reverse the path in order to find the correct one.

This is a version of Lee's algorithm for finding the shortest path between two particular vertices in an undirected graph. But let us now concentrate on the method of traversal, which in terms of graph theory is known as Breadth First Search (BFS in short). I assume you already know the algorithm, it is all in the way the groups of people move from one point to another - at each step the groups

separate into smaller ones, then go to the next positions and also leave a person behind. Remember it is not allowed to pass through a point with one person already standing there. This separation process continues until there are no more available positions to visit.

For a better understanding let us try and traverse the next graph starting from vertex 3 using this method. The steps are already described graphically below but let us make a few comments.



First there is a group of 5 people (which is enough for this situation) positioned in vertex 3. Then one person goes to vertex 1, one to vertex 5, two persons go to vertex 2 and one stays in 3. At the next phase of the process we clearly see that the person in vertex 1 has nowhere to go, since both its next possible positions (2 and 3) are occupied. Because 2 is smaller than 5 (considering the integer numbers order) we will search first for its available positions. The single one is vertex 4, so we leave one person in 2 and send one to 4. This seems like our last move - no person can move from its current position anymore since all are occupied.

The BFS method shows the vertices that are visited through each step of the traversal process.

In the example above we would get the following listing :

3, 1, 2, 5, 4

To implement this method in C++ we will use a queue to store the current position of each group of people and search for their available directions to go. Also we will use an additional Boolean array to store information about each vertex (whether it is occupied or not) :

visited [*k*] = *true if position k is occupied, otherwise visited* [*k*] = *false*

The Breadth First Search pseudocode looks like this (assuming x is the first node to start the traversal) :

```
for (all vertices 1<= i <= n) do
visited [ i ] = false
ADD x to Queue
visited [ x ] = true
while (Queue is not empty) do
extract information from the Queue to variable k
display k
for (all vertices 1<= i <= n) do
```

```
if (visited[ i ] == false AND there is an edge between k and i) then
ADD i to Queue
visited [ i ] = true
end if
end while
```

The minimum length path algorithm is very similar to this. The only difference is that we need to store the position from which every person came in the traversal process in order to reconstruct the path. We will achieve this using an array and displaying it in reverse order at the end of the traversal. Below you will find the listing that implements both BFS and the minimum path algorithm by encapsulating them inside the Graph class. We have also implemented a Queue class since both use this type of container.

```
#include <iostream>

using namespace std;

struct node
{
	int info;
	node *next;
};



class Queue
{
public:
	Queue();
	~Queue();
	bool isEmpty();
	void add(int);
	int get();
private:
	node *first, *last;
};

class Graph
{
public:
	Graph(int size = 2);
	~Graph();
	bool isConnected(int, int);
	// adds the (x, y) pair to the edge set
	void addEdge(int x, int y);
	// performs a Breadth First Search starting with node x
	void BFS(int x);
	// searches for the minimum length path
	// between the start and target vertices
	void minPath(int start, int target);
```

```
private :
        int n;
        int **A;
};

Queue::Queue()
{
        first = new node;
        first->next = NULL;
        last = first;
}

Queue::~Queue()
{
        delete first;
}

bool Queue::isEmpty()
{
        return (first->next == NULL);
}

void Queue::add(int x)
{
        node *aux = new node;
        aux->info = x;
        aux->next = NULL;
        last->next = aux;
        last = aux;
}

int Queue::get()
{
        node *aux = first->next;
        int value = aux->info;
        first->next = aux->next;
        if (last == aux) last = first;
                delete aux;
        return value;
}

Graph::Graph(int size)
{
        int i, j;
        if (size < 2)
                n = 2;
        else
                n = size;
```

```
        A = new int*[n];
        for (i = 0; i < n; ++i)

                A[i] = new int[n];
        for (i = 0; i < n; ++i)
                for (j = 0; j < n; ++j)
                        A[i][j] = 0;
}

Graph::~Graph()
{
        for (int i = 0; i < n; ++i)
                delete [] A[i];
        delete [] A;
}

bool Graph::isConnected(int x, int y)
{
        return (A[x-1][y-1] == 1);
}

void Graph::addEdge(int x, int y)
{
        A[x-1][y-1] = A[y-1][x-1] = 1;
}

void Graph::BFS(int x)
{
        Queue Q;
        bool *visited = new bool[n+1];
        int i;

        for (i = 1; i <= n; ++i)
                visited[i] = false;

        Q.add(x);
        visited[x] = true;
        cout << "Breadth First Search starting from vertex ";
        cout << x << " : " << endl;

        while (!Q.isEmpty())
        {
                int k = Q.get();
                cout << k << " ";
                for (i = 1; i <= n; ++i)
                if (isConnected(k, i) && !visited[i])
                {
                        Q.add(i);
```

```
                visited[i] = true;
            }
        }

        cout << endl;
        delete [] visited;
}

void Graph::minPath(int start, int target)
{
        Queue Q;
        int i, p, q;
        bool found;
        struct aux
        {
            int current, prev;
        };

        aux *X = new aux[n+1];
        int *Y = new int[n+1];
        bool *visited = new bool[n+1];
        for (i = 1; i <= n; ++i)
            visited[i] = false;

        Q.add(start);
        visited[start] = true;
        found = false;
        p = q = 0;
        X[0].current = start;
        X[0].prev = 0;

        while (!Q.isEmpty() && !found)
        {
            int k = Q.get();
            for (i = 1; i <= n && !found; ++i)
                if (isConnected(k, i) && !visited[i])
                {
                    Q.add(i);
                    ++q;
                    X[q].current = i;
                    X[q].prev = p;
                    visited[i] = true;

                    if (i == target)
                        found = true;
                }
            ++p;
        }
```

```
        cout << "The minimum length path from " << start;
        cout << " to " << target << " is : " << endl;
        p = 0;

        while (q)
        {
                Y[p] = X[q].current;
                q = X[q].prev;
                ++p;
        }

        Y[p] = X[0].current;
        for (q = 0; q <= p/2; ++q)
        {
                int temp = Y[q];
                Y[q] = Y[p-q];
                Y[p-q] = temp;
        }

        for (q = 0; q <= p; ++q)
                cout << Y[q] << " ";
        cout << endl;
        cout << "Length = " << q-1 << endl;

        delete [] visited;
        delete [] X;
        delete [] Y;
}

void Traversal()
{
        Graph g(5);
        g.addEdge(1, 2); g.addEdge(1, 3); g.addEdge(2, 4);
        g.addEdge(3, 5); g.addEdge(4, 5); g.addEdge(2, 3);
        g.BFS(3);
}

void Maze()
{
        Graph f(15);
        f.addEdge(1, 2); f.addEdge(1, 3); f.addEdge(2, 4);
        f.addEdge(3, 14); f.addEdge(4, 5); f.addEdge(4, 6);
        f.addEdge(5, 7); f.addEdge(6, 13); f.addEdge(7, 8);
        f.addEdge(7, 9); f.addEdge(8, 11); f.addEdge(9, 10);
        f.addEdge(10, 12); f.addEdge(10, 15); f.addEdge(11, 12);
        f.addEdge(13, 14); f.addEdge(14, 15);
        f.minPath(1, 10);
}
```

```
void main()
{
        Traversal();
        cout << endl;
        Maze();
}
```

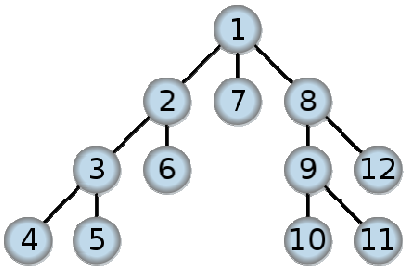
Sample input / output:

```
Mark "C:\Courses\0211490\Programming\Cpp Programs
Breadth First Search starting from vertex 3 :
3 1 2 5 4

The minimum length path from 1 to 10 is :
1 3 14 15 10
Length = 4
```

Graph Traversal: Depth-First:

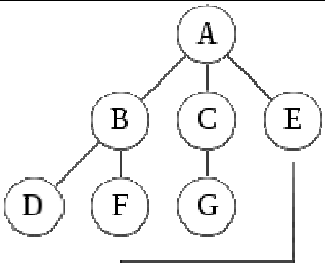
Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

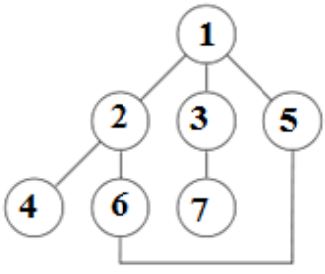
Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.



Order in which the nodes are expanded

For the following graph:





a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.

```
#include<iostream>

using namespace std;

class graph
{
private:
        int n;
        graph* next;
public:
        graph* read_graph(graph*);
        void dfs(int); //dfs for a single node
        void dfs(); //dfs of the entire graph
        void ftraverse(graph*);
};

graph *g[100];
int visit[100];
int dfs_span_tree[100][100];

graph* graph::read_graph(graph*head)
{
        int x;
        graph* last;
        head=last=NULL;

        cout<<"Enter adjecent node ,-1 to stop:\n";
        cin>>x;
        while(x!=-1)
        {
                graph*NEW;
                NEW=new graph;
                NEW->n=x;
                NEW->next=NULL;

                if(head==NULL)
                        head=NEW;
                else
                        last->next=NEW;

                last=NEW;

                cout<<"Enter adjecent node ,-1 to stop:\n";
                cin>>x;
```

```
        }

        return head;
}

void graph::ftraverse(graph*h)
{
        while(h!=NULL)
        {
                cout<<h->n<<"->";
                h=h->next;
        }

        cout<<"NULL"<<endl;
}

void graph::dfs(int x)
{
        cout<<"node "<<x<<" is visited\n";
        visit[x]=1;

        graph *p;
        p=g[x];
        while(p!=NULL)
        {
                int x1=p->n;
                if(visit[x1]==0)
                {
                        cout<<"from node "<<x<<' ';
                        //Add the edge to the dfs spanning tree
                        dfs_span_tree[x][x1]=1;
                        dfs(x1);
                }

                p=p->next;
        }
}

void graph::dfs()
{
        int i;
        cout<<"*********************************************************\n";
        cout<<"This program is to implement dfs for an unweighted graph \n";
        cout<<"*********************************************************\n";

        cout<<"Enter the no of nodes ::";
        cin>>n;
        for(i=1;i<=n;i++)
```

```
		g[i]=NULL;

	for(i=1;i<=n;i++)
	{
		cout<<"Enter the adjacent nodes to node no. "<<i<<endl;
		cout<<"*****************************************\n";
		g[i]=read_graph(g[i]);
	}

	//display the graph
	cout<<"\n\nThe entered graph is ::\n";
	for(i=1;i<=n;i++)
	{
		cout<<" < "<<i<<" > ::";
		ftraverse(g[i]);
	}

	for(i=1;i<=n;i++)
		visit[i]=0; //mark all nodes as unvisited

	cout<<"\nEnter the start vertex ::";
	int start;
	cin>>start;

	for(i=1;i<=n;i++)
		for(int j=1;j<=n;j++)
			dfs_span_tree[i][j]=0;

	cout<<"\nThe dfs for the above graph is ::\n";
	dfs(start);

	cout<<"\n\nThe required dfs spanning tree is ::\n";

	for(i=1;i<=n;i++)
	{
		for(int j=1;j<=n;j++)
			cout<<dfs_span_tree[i][j]<<' ';
		cout<<endl;
	}
}

int main()
{
	graph obj;
	obj.dfs();
	return 0;
}
```

Sample input / output:

```
************************************************************
This program is to implement dfs for an unweighted graph
************************************************************
Enter the no of nodes ::7
Enter the adjacent nodes to node no. 1
****************************************
Enter adjecent node ,-1 to stop:
2
Enter adjecent node ,-1 to stop:
3
Enter adjecent node ,-1 to stop:
5
Enter adjecent node ,-1 to stop:
-1
Enter the adjacent nodes to node no. 2
****************************************
Enter adjecent node ,-1 to stop:
1
Enter adjecent node ,-1 to stop:
4
Enter adjecent node ,-1 to stop:
-1
Enter the adjacent nodes to node no. 3
****************************************
Enter adjecent node ,-1 to stop:
1
Enter adjecent node ,-1 to stop:
7
Enter adjecent node ,-1 to stop:
-1
Enter the adjacent nodes to node no. 4
****************************************
Enter adjecent node ,-1 to stop:
2
Enter adjecent node ,-1 to stop:
-1
Enter the adjacent nodes to node no. 5
****************************************
Enter adjecent node ,-1 to stop:
1
Enter adjecent node ,-1 to stop:
6
Enter adjecent node ,-1 to stop:
-1
Enter the adjacent nodes to node no. 6
****************************************
Enter adjecent node ,-1 to stop:
2
Enter adjecent node ,-1 to stop:
5
Enter adjecent node ,-1 to stop:
-1
Enter the adjacent nodes to node no. 7
****************************************
Enter adjecent node ,-1 to stop:
3
Enter adjecent node ,-1 to stop:
-1


The entered graph is ::
 < 1 > ::2->3->5->NULL
 < 2 > ::1->4->NULL
 < 3 > ::1->7->NULL
 < 4 > ::2->NULL
 < 5 > ::1->6->NULL
 < 6 > ::2->5->NULL
 < 7 > ::3->NULL
```

```
Enter the start vertex ::1

The dfs for the above graph is ::
node 1 is visited
from node 1 node 2 is visited
from node 2 node 4 is visited
from node 1 node 3 is visited
from node 3 node 7 is visited
from node 1 node 5 is visited
from node 5 node 6 is visited


The required dfs spanning tree is ::
0 1 1 0 1 0 0
0 0 0 1 0 0 0
0 0 0 0 0 0 1
0 0 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
```

Topological Sorting:

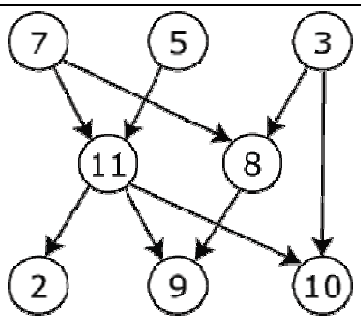
In graph theory, a topological sort or topological ordering of a *directed acyclic graph* (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.

More formally, define the reachability relation *R* over the nodes of the DAG such that *xRy* if and only if there is a directed path from *x* to *y*. Then, *R* is a partial order, and a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.

The canonical application of topological sorting (topological order) is in scheduling a sequence of jobs or tasks. The jobs are represented by vertices, and there is an edge from *x* to *y* if job *x* must be completed before job *y* can be started (for example, when washing clothes, the washing machine must finish before we put the clothes to dry). Then, a topological sort gives an order in which to perform the jobs.

In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, and resolving symbol dependencies in linkers.

Example:



The graph shown to the left has many valid topological sorts, including:

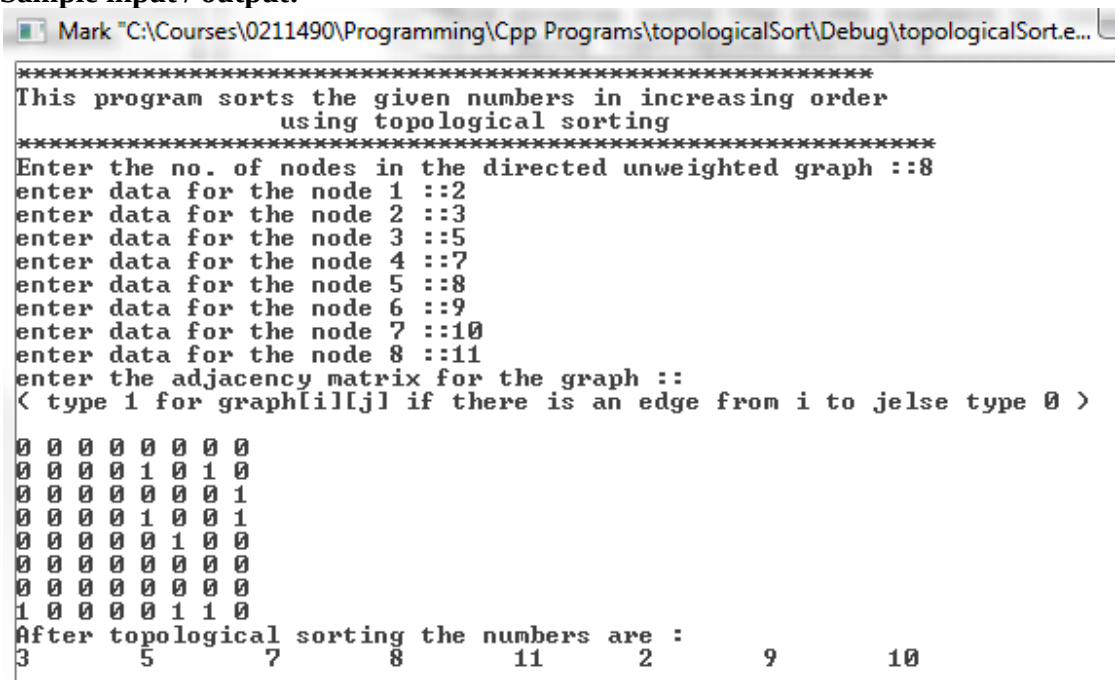
- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 9, 2 (least number of edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10

```
#include<iostream>

using namespace std;

class graph
{
private:
	int n;
	int data[20];
	int gptr[20][20];
public:
	void create();
	void topological();
};
void graph::create()
{
	int i, j;
	cout<<"************************************************************\n"
		<<"This program sorts the given numbers in increasing order\n"
		<<"\t\t using topological sorting\n"
		<<"************************************************************\n";
	cout<<"Enter the no. of nodes in the directed unweighted graph ::";
	cin>>n;
	for(i=1;i<=n;i++)
	{
		cout<<"enter data for the node "<<i<<" ::";
		cin>>data[i];
	}

	cout<<"enter the adjacency matrix for the graph ::\n";
	cout<<"( type 1 for graph[i][j] if there is an edge from i to j"
		<<"else type 0 )\n\n";

	for(i=1;i<=n;i++)
		for(j=1;j<=n;j++)
			cin>>gptr[i][j];
}

void graph::topological()
{
	int flag;
	int i,j;
	int poset[20],included[20];
	for(i=1;i<=n;i++)
	{
		poset[i]=0;
		included[i]=false;
	}
```

```
int k=1;
flag=true;
int zeroindegree;
int c=1;
while(flag==1)
{
	for(i=1;i<=n;i++)
	{
		if(!included[i])
		{
			zeroindegree=true;
			for(j=1;j<=n;j++)
				if(gptr[j][i]>0)
				{
					zeroindegree=false;
					break;
				}

			if(zeroindegree)
			{
				included[i]=true;
				poset[k]=data[i];
				k=k+1;
				for(j=1;j<=n;j++)
				{
					gptr[i][j]=-1;
					gptr[j][i]=-1;
				}
				break;
			}
		}
	}

	if(i==n+1)
	{
		if(zeroindegree==false)
		{
			cout<<"Graph is not acyclic\n";
			return;
		}
		else
		{
			poset[k]=data[i-1];
			k=k+1;
			flag=false;
		}
	}
}
```

```
        cout<<"After topological sorting the numbers are :\n";
        for(i=1;i<=n;i++)
                cout<<poset[i]<<"\t";

        cout<<endl<<endl;
}

void main()
{
        graph obj;
        obj.create();
        obj.topological();
}
```

Sample input / output:



Problems

Biocoloring

PC/UVa IDs:	110901/10004	**Popularity:**	A	**Success rate:**	high	**Level:**	1

The *four-color theorem* states that every planar map can be colored using only four colors in such a way that no region is colored using the same color as a neighbor. After being open for over 100 years, the theorem was proven in 1976 with the assistance of a computer.
Here you are asked to solve a simpler problem. Decide whether a given connected graph can be bicolored, i.e., can the vertices be painted red and black such that no two adjacent vertices have the same color.
To simplify the problem, you can assume the graph will be connected, undirected, and not contain self-loops (i.e., edges from a vertex to itself).

Input

The input consists of several test cases. Each test case starts with a line containing the number of vertices n, where $1 < n < 200$. Each vertex is labeled by a number from 0 to n−1. The second line contains the number of edges *l*. After this, *l* lines follow, each containing two vertex numbers specifying an edge.
An input with $n = 0$ marks the end of the input and is not to be processed.

Output

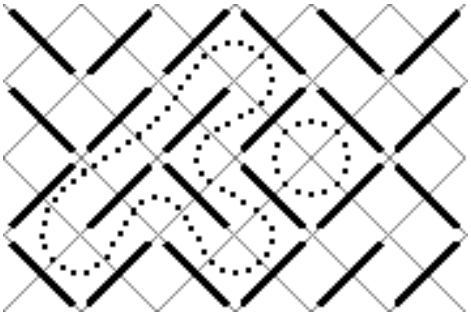
Decide whether the input graph can be bicolored, and print the result as shown below.

Sample Input	*Sample Output*
3 3 0 1 1 2 2 0 9 8 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8 0	NOT BICOLORABLE. BICOLORABLE.

Slash Maze

PC/UVa IDs:	110904/705	**Popularity:**	B	**Success rate:**	average	**Level:**	2

By filling a rectangle with slashes (/) and backslashes (\), you can generate nice little mazes. Here is an example:



As you can see, paths in the maze cannot branch, so the whole maze contains only (1) cyclic paths and (2) paths entering somewhere and leaving somewhere else. We are only interested in the cycles. There are exactly two of them in our example.
Your task is to write a program that counts the cycles and finds the length of the longest one. The length is defined as the number of small squares the cycle consists of (the ones bordered by gray lines in the picture). In this example, the long cycle has length 16 and the short one length 4.

Input

The input contains several maze descriptions. Each description begins with one line containing two integers w and h ($1 \leq w, h \leq 75$), representing the width and the height of the maze. The next *h* lines describe the maze itself and contain w characters each; all of these characters will be either "/" or "\".
The input is terminated by a test case beginning with $w = h = 0$. This case should not be processed.

Output

For each maze, first output the line "Maze #n:", where n is the number of the maze. Then, output the line "*k* Cycles; the longest has length *l*.", where *k* is the number of cycles in the maze and *l* the length of the longest of the cycles. If the maze is acyclic, output "There are no cycles."
Output a blank line after each test case.

Sample Input

```
6 4
\/\\/
\//\/
//\\\
\\//
3 3
///
\//
\\\
0 0
```

Sample Output

```
Maze #1:
2 Cycles; the longest has length 16.

Maze #2:
There are no cycles.
```

Edit Step Ladders

PC/UVa IDs:	110905/10029	**Popularity:**	B	**Success rate:**	low	**Level:**	3

An *edit step* is a transformation from one word *x* to another word *y* such that *x* and *y* are words in the dictionary, and *x* can be transformed to *y* by adding, deleting, or changing one letter. The transformations from *dig to dog* and from *dog to do* are both edit steps. An *edit step ladder* is a lexicographically ordered sequence of words $w_1, w_2, \ldots, w_n$ such that the transformation from w_i to w_i+1 is an edit step for all i from 1 to $n - 1$.
For a given dictionary, you are to compute the length of the longest edit step ladder.

Input

The input to your program consists of the dictionary: a set of lowercase words in lexicographic order at one word per line. No word exceeds 16 letters and there are at most 25,000 words in the dictionary.

Output

The output consists of a single integer, the number of words in the longest edit step ladder.

Sample Input

```
cat
dig
dog
fig
fin
fine
fog
log
wine
```

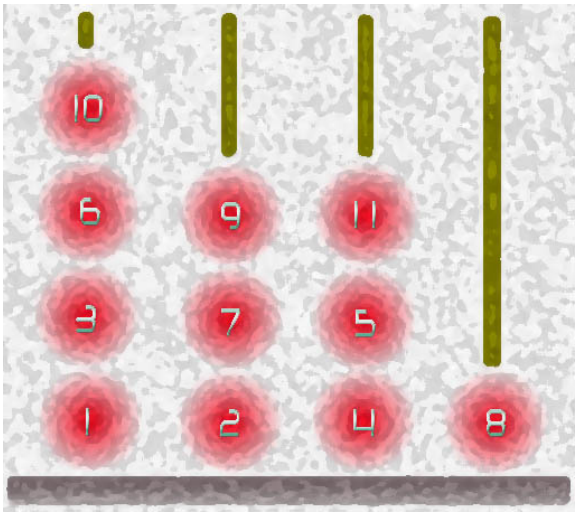
Sample Output

```
5
```

Hanoi Tower Troubles Again!

PC/UVa IDs:	110908/10276	**Popularity:**	B	**Success rate:**	high	**Level:**	3

There are many interesting variations on the Tower of Hanoi problem. This version consists of N pegs and one ball containing each number from 1, 2, 3, . . . , ∞. Whenever the sum of the numbers on two balls is *not* a perfect square (i.e., c^2 for some integer c), they will repel each other with such force that they can never touch each other.



The player must place balls on the pegs one by one, in order of increasing ball number (i.e., first ball 1, then ball 2, then ball 3. . .). The game ends where there is no non-repelling move.
The goal is to place as many balls on the pegs as possible. The figure above gives a best possible result for 4 pegs.

Input

The first line of the input contains a single integer T indicating the number of test cases ($1 \leq T \leq 50$). Each test case contains a single integer N ($1 \leq N \leq 50$) indicating the number of pegs available.

Output

For each test case, print a line containing an integer indicating the maximum number of balls that can be placed. Print "-1" if an infinite number of balls can be placed.

Sample Input	*Sample Output*
2 4 25	11 337