

9

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.

9.1 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, discussed in Chapter 10.

- *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, and will be discussed in Section 9.5.

- *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.

9.2 Data Structures for Graphs

There are several possible ways 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 3,000 vertices and 6,000 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 stored, but the adjacency matrix will have $3,000 \times 3,000 = 9,000,000$ cells, almost all of them empty!

- *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.

- *Adjacency Lists in Matrices* — Adjacency lists can also be 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 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, including [Sed01].

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:

```
#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.

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;
}

```

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 ++;
}

```

}

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

```
print_graph(graph *g)
{
    int i,j;                               /* counters */

    for (i=1; i<=g->nvertices; i++) {
        printf("%d: ",i);
        for (j=0; j<g->degree[i]; j++)
            printf(" %d",g->edges[i][j]);
        printf("\n");
    }
}
```

9.3 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 difference 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.

9.3.1 Breadth-First Search

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. This information could be maintained using one enumerated type variable; we used two Boolean variables instead.

Once a vertex is discovered, it is placed on a queue, such as we implemented in Section 2.1.2. 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 */

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]);
            }
    }
}

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

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

```

9.3.2 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. By setting the functions to

```

process_vertex(int v)
{
}

process_edge(int x, int y)
{
    nedges = nedges + 1;
}

```

we get an accurate count of the number of edges. Many problems perform different actions on vertices or edges as they are encountered. These functions give us the freedom to easily customize our response.

One final degree of customization is provided by the Boolean predicate `valid_edge`, which allows us ignore the existence of certain edges in the graph during our traversal. Setting `valid_edge` to return true for all edges results in a full breadth-first search of the graph, and will be the case for our examples except `netflow` in Section 10.4.

9.3.3 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.

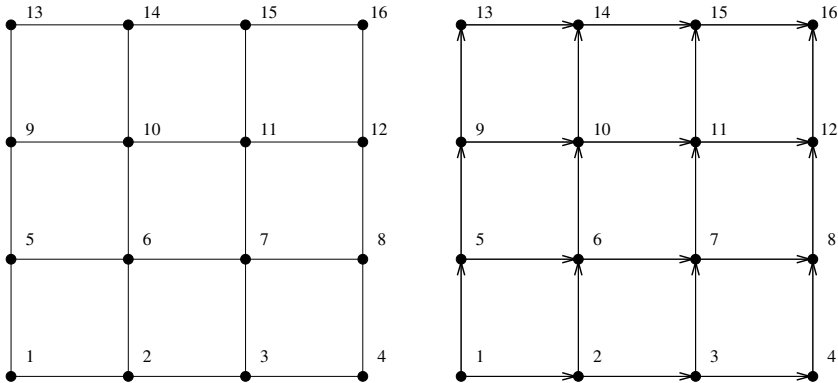


Figure 9.1. An undirected 4×4 grid-graph (l), with the DAG defined by edges going to higher-numbered vertices (r).

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);
    }
}
```

On our grid graph example (Figure 9.1) our algorithm generated the following parent relation:

vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
parent	-1	1	2	3	1	2	3	4	5	6	7	8	9	10	11	12

For the shortest path from the lower-left corner of the grid to the upper-right corner, this parent relation yields the path $\{1, 2, 3, 4, 8, 12, 16\}$. Of course, this shortest path is not unique; the number of such paths in this graph is counted in Section 6.3.

There are two points to remember about using breadth-first search to find a shortest path from x to y : First, the shortest path tree is only useful if BFS was performed with x as the root of the search. Second, BFS only gives the shortest path if the graph is unweighted. We will present algorithms for finding shortest paths in weighted graphs in Section 10.3.1.

9.4 Graph Traversal: Depth-First

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;
}
```

Rooted trees are a special type of graph (directed, acyclic, in-degrees of at most 1, with an order defined on the outgoing edges of each node). In-order, pre-order, and post-order traversals are all basically DFS, differing only in how they use the ordering of out-edges and when they process the vertex.

9.4.1 Finding Cycles

Depth-first search of an undirected graph partitions the edges into two classes, *tree edges* and *back edges*. The tree edges those encoded in the `parent` relation, the edges

which discover new vertices. Back edges are those whose other endpoint is an ancestor of the vertex being expanded, so they point back into the tree.

That all edges fall into these two classes is an amazing property of depth-first search. Why can't an edge go to a brother or cousin node instead of an ancestor? In DFS, all nodes reachable from a given vertex v are expanded before we finish with the traversal from v , so such topologies are impossible for undirected graphs. The case of DFS on directed graphs is somewhat more complicated but still highly structured.

Back edges are the key to finding a cycle in an undirected graph. If there is no back edge, all edges are tree edges, and no cycle exists in a tree. But any back edge going from x to an ancestor y creates a cycle with the path in the tree from y to x . Such a cycle is easy to find using `dfs`:

```
process_edge(int x, int y)
{
    if (parent[x] != y) { /* found back edge! */
        printf("Cycle from %d to %d:",y,x);
        find_path(y,x,parent);
        finished = TRUE;
    }
}

process_vertex(int v)
{
}
```

We use the `finished` flag to terminate after finding the first cycle in our 4×4 grid graph, which is 3 4 8 7 with (7,3) as the back edge.

9.4.2 Connected Components

A *connected component* of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices. These 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, since the vertex order does not matter. Basically, we search from the first vertex. 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, and so on 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");
        }
}

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

process_edge(int x, int y)
{
}

```

Variations on connected components are discussed in Section 10.1.2.

9.5 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 the edges represented precedence constraints, such that edge (x, y) means job x must be done before job y . Then any topological sort defines a legal schedule. Indeed, there can be many such orderings for a given DAG.

But the applications go deeper. 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.

Study the following implementation:

```

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] ] ++;
}

```

There are several things to observe. Our first step is computing the in-degrees of each vertex of the DAG, since the `degree` field of the graph data type records the out-degree of a vertex. These are the same for undirected graphs, but not directed ones.

Next, note that we use a queue here to maintain the in-degree 0 vertices, but only because we had one sitting around from Section 2.1.2. Any container will do, since the processing order does not matter for correctness. Different processing orders will yield different topological sorts.

The impact of processing orders is apparent in topologically sorting the directed grid in Figure 9.1, where all edges go from lower- to higher-numbered vertices. The sorted permutation $\{1, 2, \dots, 15, 16\}$ is a topological ordering, but our program repeatedly stripped off diagonals to find

1 2 5 3 6 9 4 7 10 13 8 11 14 12 15 16

Many other orderings are also possible.

Finally, note that this implementation does not actually delete the edges from the graph! It is sufficient to consider their impact on the in-degree and traverse them rather than delete them.

9.6 Problems

9.6.1 Bicoloring

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

```
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
```

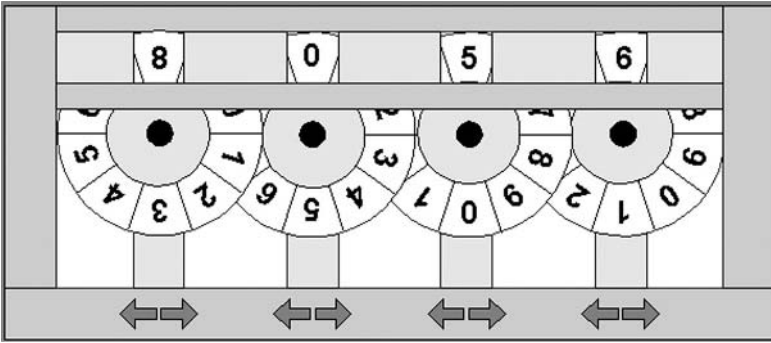
Sample Output

```
NOT BICOLORABLE.
BICOLORABLE.
```

9.6.2 *Playing With Wheels*

PC/UVa IDs: 110902/10067, **Popularity:** C, **Success rate:** average **Level:** 2

Consider the following mathematical machine. Digits ranging from 0 to 9 are printed consecutively (clockwise) on the periphery of each wheel. The topmost digits of the wheels form a four-digit integer. For example, in the following figure the wheels form the integer 8,056. Each wheel has two buttons associated with it. Pressing the button marked with a *left arrow* rotates the wheel one digit in the clockwise direction and pressing the one marked with the *right arrow* rotates it by one digit in the opposite direction.



We start with an initial configuration of the wheels, with the topmost digits forming the integer $S_1S_2S_3S_4$. You will be given a set of n forbidden configurations $F_{i_1}F_{i_2}F_{i_3}F_{i_4}$ ($1 \leq i \leq n$) and a target configuration $T_1T_2T_3T_4$. Your job is to write a program to calculate the minimum number of button presses required to transform the initial configuration to the target configuration without passing through a forbidden one.

Input

The first line of the input contains an integer N giving the number of test cases. A blank line then follows.

The first line of each test case contains the initial configuration of the wheels, specified by four digits. Two consecutive digits are separated by a space. The next line contains the target configuration. The third line contains an integer n giving the number of forbidden configurations. Each of the following n lines contains a forbidden configuration. There is a blank line between two consecutive input sets.

Output

For each test case in the input print a line containing the minimum number of button presses required. If the target configuration is not reachable print “-1”.

Sample Input

```
2
8 0 5 6
6 5 0 8
5
8 0 5 7
8 0 4 7
5 5 0 8
7 5 0 8
6 4 0 8

0 0 0 0
5 3 1 7
8
0 0 0 1
0 0 0 9
0 0 1 0
0 0 9 0
0 1 0 0
0 9 0 0
1 0 0 0
9 0 0 0
```

Sample Output

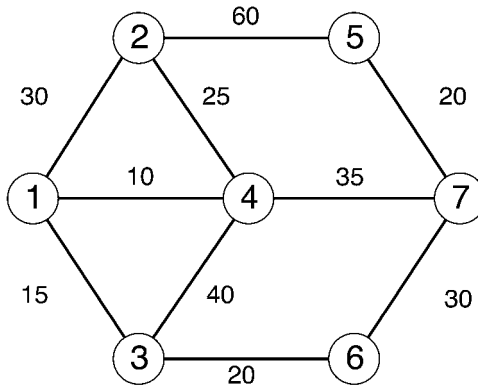
```
14
-1
```

9.6.3 The Tourist Guide

PC/UVa IDs: 110903/10099, **Popularity:** B, **Success rate:** average **Level:** 3

Mr. G. works as a tourist guide in Bangladesh. His current assignment is to show a group of tourists a distant city. As in all countries, certain pairs of cities are connected by two-way roads. Each pair of neighboring cities has a bus service that runs only between those two cities and uses the road that directly connects them. Each bus service has a particular limit on the maximum number of passengers it can carry. Mr. G. has a map showing the cities and the roads connecting them, as well as the service limit for each bus service.

It is not always possible for him to take all the tourists to the destination city in a single trip. For example, consider the following road map of seven cities, where the edges represent roads and the number written on each edge indicates the passenger limit of the associated bus service.



It will take at least five trips for Mr. G. to take 99 tourists from city 1 to city 7, since he has to ride the bus with each group. The best route to take is 1 - 2 - 4 - 7.

Help Mr. G. find the route to take all his tourists to the destination city in the minimum number of trips.

Input

The input will contain one or more test cases. The first line of each test case will contain two integers: N ($N \leq 100$) and R , representing the number of cities and the number of road segments, respectively. Each of the next R lines will contain three integers (C_1 , C_2 , and P) where C_1 and C_2 are the city numbers and P ($P > 1$) is the maximum number of passengers that can be carried by the bus service between the two cities. City numbers are positive integers ranging from 1 to N . The $(R + 1)$ th line will contain three integers (S , D , and T) representing, respectively, the starting city, the destination city, and the number of tourists to be guided.

The input will end with two zeros for N and R .

Output

For each test case in the input, first output the scenario number and then the minimum number of trips required for this case on a separate line. Print a blank line after the output for each test case.

Sample Input

```
7 10
1 2 30
1 3 15
1 4 10
2 4 25
2 5 60
3 4 40
3 6 20
4 7 35
5 7 20
6 7 30
1 7 99
0 0
```

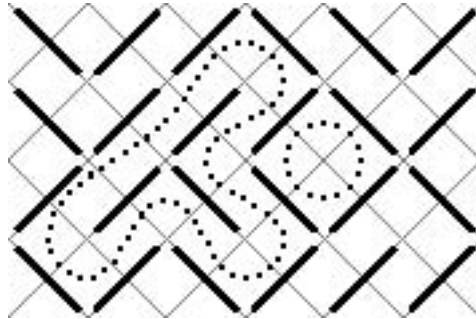
Sample Output

```
Scenario #1
Minimum Number of Trips = 5
```

9.6.4 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.

```

9.6.5 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, \dots, 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
```

9.6.6 Tower of Cubes

PC/UVa IDs: 110906/10051, **Popularity:** C, **Success rate:** high **Level:** 3

You are given N colorful cubes, each having a distinct weight. Cubes are not monochromatic – indeed, every face of a cube is colored with a different color. Your job is to build the tallest possible tower of cubes subject to the restrictions that (1) we never put a heavier cube on a lighter one, and (2) the bottom face of every cube (except the bottom one) must have the same color as the top face of the cube below it.

Input

The input may contain several test cases. The first line of each test case contains an integer N ($1 \leq N \leq 500$) indicating the number of cubes you are given. The i th of the next N lines contains the description of the i th cube. A cube is described by giving the colors of its faces in the following order: front, back, left, right, top, and bottom face. For your convenience colors are identified by integers in the range 1 to 100. You may assume that cubes are given in increasing order of their weights; that is, cube 1 is the lightest and cube N is the heaviest.

The input terminates with a value 0 for N .

Output

For each case, start by printing the test case number on its own line as shown in the sample output. On the next line, print the number of cubes in the tallest possible tower. The next line describes the cubes in your tower from top to bottom with one description per line. Each description gives the serial number of this cube in the input, followed by a single whitespace character and then the identification string (**front**, **back**, **left**, **right**, **top**, or **bottom** of the top face of the cube in the tower. There may be multiple solutions, but any one of them is acceptable.

Print a blank line between two successive test cases.

Sample Input

```
3
1 2 2 2 1 2
3 3 3 3 3 3
3 2 1 1 1 1
10
1 5 10 3 6 5
2 6 7 3 6 9
5 7 3 2 1 9
1 3 3 5 8 10
6 6 2 2 4 4
1 2 3 4 5 6
```

```
10 9 8 7 6 5
6 1 2 3 4 7
1 2 3 3 2 1
3 2 1 1 2 3
0
```

Sample Output

Case #1

```
2
2 front
3 front
```

Case #2

```
8
1 bottom
2 back
3 right
4 left
6 top
8 front
9 front
10 top
```

9.6.7 *From Dusk Till Dawn*

PC/UVa IDs: 110907/10187, **Popularity:** B, **Success rate:** average **Level:** 3

Vladimir has white skin, very long teeth and is 600 years old, but this is no problem because Vladimir is a vampire. Vladimir has never had any problems with being a vampire. In fact, he is a successful doctor who always takes the night shift and so has made many friends among his colleagues. He has an impressive trick which he loves to show at dinner parties: he can tell blood group by taste. Vladimir loves to travel, but being a vampire he must overcome three problems.

1. He can only travel by train, because he must take his coffin with him. Fortunately he can always travel first class because he has made a lot of money through long term investments.
2. He can only travel from dusk till dawn, namely, from 6 P.M. to 6 A.M. During the day he has must stay inside a train station.
3. He has to take something to eat with him. He needs one litre of blood per day, which he drinks at noon (12:00) inside his coffin.

Help Vladimir to find the shortest route between two given cities, so that he can travel with the minimum amount of blood. If he takes too much with him, people ask him funny questions like, “What are you doing with all that blood?”

Input

The first line of the input will contain a single number telling you the number of test cases.

Each test case specification begins with a single number telling you how many route specifications follow. Each route specification consists of the names of two cities, the departure time from city one, and the total traveling time, with all times in hours. Remember, Vladimir cannot use routes departing earlier than 18:00 or arriving later than 6:00.

There will be at most 100 cities and less than 1,000 connections. No route takes less than 1 hour or more than 24 hours, but Vladimir can use only routes within the 12 hours travel time from dusk till dawn.

All city names are at most 32 characters long. The last line contains two city names. The first is Vladimir’s start city; the second is Vladimir’s destination.

Output

For each test case you should output the number of the test case followed by “Vladimir needs # litre(s) of blood.” or “There is no route Vladimir can take.”

Sample Input

```
2
3
Ulm Muenchen 17 2
Ulm Muenchen 19 12
Ulm Muenchen 5 2
Ulm Muenchen
10
Lugoj Sibiu 12 6
Lugoj Sibiu 18 6
Lugoj Sibiu 24 5
Lugoj Medias 22 8
Lugoj Medias 18 8
Lugoj Reghin 17 4
Sibiu Reghin 19 9
Sibiu Medias 20 3
Reghin Medias 20 4
Reghin Bacau 24 6
Lugoj Bacau
```

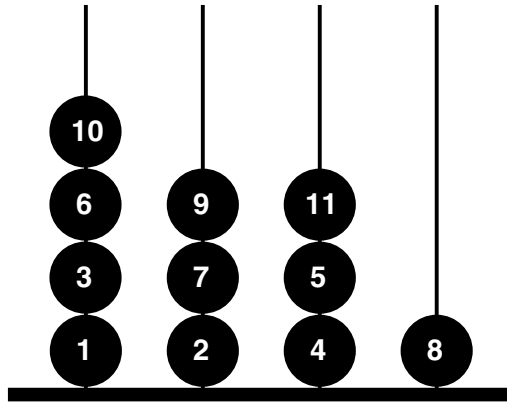
Sample Output

```
Test Case 1.
There is no route Vladimir can take.
Test Case 2.
Vladimir needs 2 litre(s) of blood.
```

9.6.8 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 $1, 2, 3, \dots, \infty$. 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

```
2
4
25
```

Sample Output

```
11
337
```

9.7 Hints

- 9.1 Can we color the graph during a single traversal?
- 9.2 What is the graph underlying this problem?
- 9.3 Can we reduce this problem to connectivity testing?
- 9.4 Does it pay to represent the graph explicitly, or just work on the matrix of slashes?
- 9.5 What is the graph underlying this problem?
- 9.6 Can we define a directed graph on the cubes such that the desired tower is a path in the graph?
- 9.7 Can this be represented as an *unweighted* graph problem for BFS?
- 9.8 Can the constraints be usefully modeled using a DAG?