Lecture 9
Graph Traversal

Euiseong Seo
(euiseong@skku.edu)
Need for Graphs

- One of unifying themes of computer science
- Closely related to many daily life problems
  - Navigation
  - Circuit generation
  - Social network services
  - Games
  - Computer networks
- How can a problem be represented as a graph?
- How to solve a graph problem?
Graph Notation

- A graph $G = (V, E)$
  - $V$ is a set of vertices (nodes)
  - $E$ is a set of edges
    - $E = (x, y)$ where $x, y \in V$
    - Ordered or unordered pairs of vertices from $V$

- Modeling of problems
  - What are vertices and edges in the followings?
    - Road networks
    - Human interactions
    - Program analysis
### Flavors of Graphs

- **Undirected or directed**
  - A graph is undirected if edge \((x, y) \in E\) implies that \((y,x) \in E\), too.
  - Otherwise, the graph is directed.
Flavors of Graphs

- Weighted or unweighted
  - If each edge of a graph is assigned a numerical value, or weight, the graph is a weighted graph
  - Otherwise, it is a unweighted graph
Cyclic or acyclic

- An acyclic graph does not contain any cycles
- Trees are connected acyclic undirected graphs
- Directed acyclic graphs are called DAGs
Data Structures for Graphs

- Assume that a graph $G = (V, E)$ contains $n$ vertices and $m$ edges

- Adjacency matrix
  
  - Use a $n \times n$ matrix $M$
  
  - $M[i,j] = 1$, if $(i,j) \in E$
  
  - $M[i,j] = 0$, if $(i,j) \notin E$

  - Pros
    - Easy to add or remove edges
    - Easy to find a specific edge, $(i,j)$, if exists

  - Cons
    - Waste of memory space for sparse graphs
Data Structures for Graphs

- Adjacency lists in lists

```
1 -> 2 -> 5 /  
2 -> 1 -> 5 -> 3 -> 4 /  
3 -> 2 -> 4 /  
4 -> 2 -> 5 -> 3 /  
5 -> 4 -> 1 -> 2 /  
```
Adjacency lists in matrices

- Use arrays instead of linked lists
- Looks like it combines the worst properties of both, but.

```
1  2
5  4

1  2  5
1  2  3  4
1  2
2  1  5  4  3
2  1  2
3  2  4
3  1  2
4  2  5  3
4  1  2  3
5  1  2  4
5

1  2
5  4

1  2  5
1  2
2  1  2
2  5
3  3  4
3  3
3  4
3  1
5  4
5
```
List in Array Representation

An undirected edge \((x,y)\) appears twice, once as \(y\) in \(x\)'s list and once as \(x\) in \(y\)'s list

```c
#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;
```
Reading a Graph

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:

```c
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 ++;
}
```
Tree Traversal

- BFS (Breadth First Search)
- DFS (Depth First Search)
BFS
Shortest path from $s$
Queue: s 2 3
Queue: s 2 3 5
Queue: 2 3 5 4
Queue: 2 3 5 4
Queue: 2 3 5 4

Undiscovered
Discovered
Top of queue
Finished
Queue: 3 5 4
Queue: 3 5 4 6
Queue: 4 6
Queue: 4 6 8
Queue: 6 8 7 9
Queue: 8 7 9
Queue: 7 9
Queue: 7 9
Queue: 7 9

1. Undiscovered
2. Discovered
3. Top of queue
4. Finished

Queue:

1. 7
2. 9

Graph:

- Node 0 (s) connected to 3, 1, and 5
- Node 1 connected to 2
- Node 2 connected to 4
- Node 3 connected to 6
- Node 4 connected to 5
- Node 5 connected to 1
- Node 6 connected to 3
- Node 7 connected to 3
- Node 8 connected to 3
Queue: 9
Queue: 9
Queue:

Since Queue is empty, STOP!
DFS

- Similar to Backtracking
  - Go as deep as you can
  - Next one is your siblings
- Stack is an ideal candidate

```
DFS(G, v)
for all edges e incident on v
    do if edge e is unexplored then
        w = opposite(v, e) // return the end point of e distant to v
        if vertex w is unexplored then
            mark e as a discovered edge
            recursively call DFS(G, w)
        else
            mark e as a back edge
```
Finding Paths

- BFS Tree from x is unique
- Parent[i] is the node that discovered node i during the BFS originated from x
- Finding the shortest path from x to y in a undirected graph
  - By following the chain of ancestors backward from y to the root

```c
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);
    }
}
```
Many seemingly complicated problems reduce to finding connected components

- 15-Puzzle

Connected components can be found by using repetitive application of DFS or BFS

```c
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:\n",c);
            dfs(g,i);
            printf("\n");
        }
}```
Topological Sorting

- One of the fundamental operations on DAGs
- Construct an ordering of the vertices such that all directed edges go from left to right
  - Cannot exist over cyclic graphs
- This gives us a way to process each vertex before any of its successors
  - Suppose we seek the shortest (or longest) path from $x$ to $y$
  - No vertex appearing after $y$ in the topological order can contribute to any such path
### Definition

- A topological sort of a DAG $G$ is a linear ordering of all its vertices such that if $G$ contains a link $(u,v)$, then node $u$ appears before node $v$ in the ordering.
- find source nodes (indegree = 0)
  - if there is no such node, the graph is NOT DAG

```
Sorted: -
```
Topological Sorting Algorithm

- span c; decrement in\_deg of a, b, e
  - store a in Queue since in\_deg becomes 0

Diagram:

- a (in\_deg=0)
- b (in\_deg=1)
- c (in\_deg=0)
- d (in\_deg=1)
- e (in\_deg=2)
- f (in\_deg=1)

Sorted: c
span $a$; decrement in\_deg of $b$, $f$
  
  - store $b$, $f$ in Queue since ...

Sorted: $c$ $a$
Topological Sorting Algorithm

- span b; store d in Queue

Sorted: c a b
- span f; decrement in\_deg of e
  - no node with in\_deg = 0 is found

Sorted: c a b f
span d; store e in Queue.

Sorted: c a b f d
Topological Sorting Algorithm

- span e; Queue is empty

Sorted: c a b f d e
Topological Sorting Algorithm

Topological sorting can be performed using DFS. However, a more straightforward algorithm does an analysis of the in-degrees of each vertex in a DAG. Any in-degree 0 vertex may safely be placed first in topological order. Deleting its outgoing edges may create new in-degree 0 vertices, continuing the process.

```c
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] ] ++;
}
```
Topological Sorting Algorithm

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);
}

SWE2004: Principles in Programming | Spring 2016 | Euiseong Seo (euiseong@skku.edu)
Playing with Wheels

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.
Playing with Wheels

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
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.
Mr. Ferdaus has created a special type of 4-digit lock named FeruLock shown in the picture on the left. It always shows a 4-digit value and has a specific unlock code (An integer value). The lock is unlocked only when the unlock code is displayed. This unlock code can be made to appear quickly with the help of some of the special buttons available with that lock. Each button has a number associated with it. When any of these buttons is pressed, the number associated with that button is added with the displayed value and so a new number is displayed. The lock always uses least significant 4 digits after addition. After creating such a lock, he has found that, it is also very difficult for him to unlock the Ferulock. As a very good friend of Ferdaus, your task is to create a program that will help him to unlock the Ferulock by pressing these buttons minimum number of times.

Input
There will be at most 100 test cases. For each test case, there will be 3 numbers: L, U and R where L (0000 ≤ L ≤ 9999) represents the current lock code, U (0000 ≤ U ≤ 9999) represents the unlock code and R (1 ≤ R ≤ 10) represents the number of available buttons. After that, there are R numbers (0 ≤ RVi ≤ 9999) in a line representing the value of buttons. The values of L, U, RVi will always be denoted by a four digit number (even if it is by padding with leading zeroes). Input will be terminated when L = U = R = 0.

Output
For each test case, there will be one line of output which represents the serial of output followed by the minimum number of button press required to unlock the lock. If it is not possible to unlock the lock, then print a line Permanently Locked instead (without quotes).

Sample Input

<table>
<thead>
<tr>
<th>Input</th>
<th>Output for Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 9999 1</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>0000 9999 1</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td></td>
</tr>
</tbody>
</table>

Permanently Locked

48
Unlock the Lock

Input
There will be at most 100 test cases. For each test case, there will be 3 numbers: L, U and R where L (0000 ≤ L ≤ 9999) represents the current lock code, U (0000 ≤ U ≤ 9999) represents the unlock code and R (1 ≤ R ≤ 10) represents the number of available buttons. After that, there are R numbers (0 ≤ RV_i ≤ 9999) in a line representing the value of buttons. The values of L, U, RV_i will always be denoted by a four digit number (even if it is by padding with leading zeroes). Input will be terminated when L = U = R = 0.

Output
For each test case, there will be one line of output which represents the serial of output followed by the minimum number of button press required to unlock the lock. If it is not possible to unlock the lock, then print a line “Permanently Locked” instead (without quotes).
Unlock the Lock

Sample
Input
0000 9999 1
1000
0000 9999 1
0001
5234 1212 3
1023 0101 0001
0 0 0

Output for Sample Input
Case 1: Permanently Locked
Case 2: 9999
Case 3: 48