# Chapter 11 Application of Graph Traversal

Yonghui Wu
School of Computer Science, Fudan University
yhwu@fudan.edu.cn
WeChat : 13817360465

# Chapter 11 Application of Graph Traversal

- 11.1 BFS Algorithm
- 11.2 DFS Algorithm
- 11.3 Topological Sort
- 11.4 Connectivity of Undirected Graphs

# 11.1 BFS Algorithm

Given a graph $G(V, E)$ and a source vertex $s$ in $G$, Breadth-First Search (BFS) visits all vertices that can be reached from $s$ layer by layer, and calculate distances from $s$ to all vertices (that is, numbers of edges from $s$ to these vertices). The distance from $s$ to vertex $v$ $d[v]$ is as follow, $v \in V$:

$$d[v] = \begin{cases} -1 & \text{if s and v are not connected} \\ \text{the length of the shortest path from s to v} & \text{otherwise} \end{cases}$$

# The process for Breadth-First Search (BFS)

- Initially $d[s]=0$; and for $v \in V-\{s\}$, $d[v]= -1$.
- Every visited vertex $u$ is processed in order:
  - for every vertex $v$ that is adjacent to $u$ and is not visited, that is $(u, v) \in E$, and $d[v]=-1$, $v$ will be visited.
- $u$ is the parent or the precursor for $v$, $d[v]=d[u]+1$.

- Because the traversal order is based on hierarchy, and the traversal is implemented through the "First In First Out (FIFO)" access rule, a queue $Q$ is used to store visited vertices:
  - Initially  source vertex $s$ is added into queue $Q$, and $d[s]=0$.
  - Then, vertex $u$ which is the front is deleted from queue $Q$; vertices which aren't visited and are adjacent to $u$, that is, for such a vertex $v$, $(u, v)\in E$, and $d[v]=-1$, are visited in order: $d[v]=d[u]+1$; and vertex $v$ is added into queue $Q$.
  - The process repeats until queue $Q$ is empty.
- BFS traversal starts from source $s$, visits all connected vertices, and forms a BFS traversal tree whose root is $s$.

- BFS algorithm starting from source $u$, visits all vertices that can be reached from $u$ top-down and layer by layer.

```
• void BFS(VLink G[ ], int v)  // BFS algorithm starting from source v in G
• { int w;
•    visit v;  d[v]=0;   ADDQ(Q, v);            // v is added into queue Q
•    while (!EMPTYQ(Q))      // while queue Q is not empty, visit other vertices
•   {  v=DELQ(Q);           // the front is deleted from queue Q
•       Get the first adjacent vertex w for vertex v ( if there is no adjacent vertex for v, w=-1);
•       while (w != -1)
•         {    if (d[w] == -1)        // if vertex w hasn't been visited
•                {    visit w;    ADDQ(Q,w);     // adjacent vertex w is added into queue Q
•                     d[w] =d[v]+1;    // distance d[w]            }
•                Get the next adjacent vertex w for vertex v;
•         }
•   }
• }
```

- $BFS(G, v)$ can visit all vertices that can be reached from $v$ in $G$, that is, vertices in the connected component containing $v$.

```
• void TRAVEL_BFS (VLink G[ ], int d[ ], int n)
• { int i;
•   for (i = 0; i < n; i ++)        // Initialization
•      d[i] =-1;
•   for (i = 0; i < n; i ++)        // BFS for all unvisited vertices
•     if (d[i] == -1)
•       BFS(G, i);
• }
```

# 11.1.1 Prime Path

- **Source: ACM Northwestern Europe 2006**
- **IDs for Online Judge: POJ 3126**

- The ministers of the cabinet were quite upset by the message from the Chief of Security stating that they would all have to change the four-digit room numbers on their offices.
- — It is a matter of security to change such things every now and then, to keep the enemy in the dark.
- — But look, I have chosen my number 1033 for good reasons. I am the Prime minister, you know!
- — I know, so therefore your new number 8179 is also a prime. You will just have to paste four new digits over the four old ones on your office door.
- — No, it’s not that simple. Suppose that I change the first digit to an 8, then the number will read 8033 which is not a prime!
- — I see, being the prime minister you cannot stand having a non-prime number on your door even for a few seconds.
- — Correct! So I must invent a scheme for going from 1033 to 8179 by a path of prime numbers where only one digit is changed from one prime to the next prime.

- Now, the minister of finance, who had been eavesdropping, intervened.
- — No unnecessary expenditure, please! I happen to know that the price of a digit is one pound.
- — Hmm, in that case I need a computer program to minimize the cost. You don't know some very cheap software gurus, do you?
- — In fact, I do. You see, there is this programming contest going on... Help the prime minister to find the cheapest prime path between any two given four-digit primes! The first digit must be nonzero, of course. Here is a solution in the case above.
- 1033
- 1733
- 3733
- 3739
- 3779
- 8779
- 8179
- The cost of this solution is 6 pounds. Note that the digit 1 which got pasted over in step 2 can not be reused in the last step – a new 1 must be purchased.

- **Input**
- One line with a positive number: the number of test cases (at most 100). Then for each test case, one line with two numbers separated by a blank. Both numbers are four-digit primes (without leading zeros).
- **Output**
- One line for each case, either with a number stating the minimal cost or containing the word Impossible.

# Analysis

- Every number is a four-digit number. There are 10 possible values for each digit ([0..9]), and the first digit must be nonzero.
- The problem is represented by a graph:
    - the initial prime and all primes gotten by changing a digit are vertices.
    - If prime $a$ can be changed into prime $b$ by changing a digit, there is an arc $(a, b)$ whose length is 1 connecting two vertices corresponding to $a$ and $b$ respectively.

- If there is a path from initial prime $x$ to goal prime $y$, then the number of arcs in the path is the cost; else there is no solution.
- Solving the problem is to calculate the shortest path from initial prime $x$ to goal prime $y$, and BFS is used to find the shortest path.

- Suppose
  - array *s*[ ] is used to store lengthes of the shortest pathes for all gotten primes;
  - the type for elements in queue *h*[ ] is *struct*,
    - where *h*[ ].*k* and *h*[ ].*step* are used to store primes and lengthes of pathes respectively,
  - pointers for the front and the rear of *h* are *l* and *r* respectively.

- First, sieve method is used to calculate all primes between 2 and 9999, and all primes are put into array $p$.
- Only the minimal cost is required to calculate for the problem.
- The directed graph needn’t to be stored, and we only need focus on calculating the shortest paths.

- Step 1: Initialization.
- The initial prime $x$ is added into queue $h$.
  - Its path length is 0 ($h[1].k=x$; $h[1].step=0$; ). The minimal cost $ans$ is initialized -1.

- Step 2: Front *h*[l] is operated as follow:
- If the front is the goal prime ( *h*[l].*k*==*y* ), then note down the length of the path (*ans*=*h*[l].*step*) and exceed the loop;
- Enumerate all possibilities for the front: enumerate the number of digit *i* from 1 to 4, enumerate value *j* for digit *i* from 0 to 9, and the first digit must be nonzero (!((*j*==0)&&(*i*==4))):
  - Get the number *tk* by changing the front *h*[l].*k*'s digit *i* into *j*;
  - If *tk* is a composite number (*p*[*tk*]==true), then continue to enumerate;
  - Get the length of the path *ts* for prime number *tk* (*ts*=*h*[l].*step*+1);
  - If *ts* is not the shortest (*ts*≥*s*[*tk*]), then continue to enumerate;
  - If *tk* is the goal prime (*tk*==*y*), then note down the length of the path (*ans*=*ts*) and exceed the loop;
  - Note down the length of the path for prime *tk* (*s*[*tk*]=*ts*);
  - Add prime *tk* and its length of the path (*r*++; *h*[*r*].*k*=*tk*; *h*[*r*].*step*=*ts*; ) into the queue;
- If the queue is empty (*l*==*r*) or the goal prime has been gotten (*ans*≥0), then exceed the loop;
- The front is deleted from queue (*l*++);

- Step 3: Output the result:
  - If the goal prime is gotten ($ans \geq 0$), then output the length of the shortest path $ans$; else output “Impossible”.

# 11.2 DFS Algorithm

- DFS (Depth-First Search) algorithm starts from a vertex $u$.
  - First vertex $u$ is visited.
  - Then unvisited vertices adjacent from $u$ are selected one by one, and for each vertex DFS is initiated.

# *DFS*(*G*, *u*) visits the connected component containing vertex *u*.

- void *DFS*(VLink *G*[ ], int *u*) // DFS starts from a vertex *u*
- { int *w*;
- *visited*[*u*] = 1; // Vertex *u* is visited.
- Get a vertex *w* adjacent from *u* (If there is no such a vertex *w*, *w*=-1.);
- while (*w* != -1) // adjacent vertices are selected one by one
- { if (*visited*[*w*] == 0) //If vertex *w* hasn't been visited
- { *visited*[*w*]=1;
- *DFS*(*G*, *w*) ; //Recursion
- }
- Get the next vertex *w* adjacent from *u* (If there is no such a vertex *w*, *w*=-1);
- }
- }

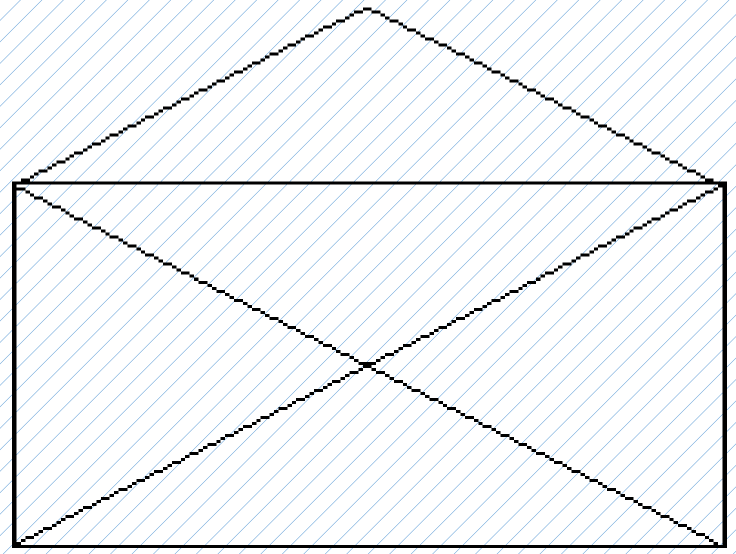
# DFS for a graph

- void *TRAVEL_DFS*(VLink *G*[ ], int *visited*[ ], int *n*)
- { int *i*;
- for (*i* = 0; *i* < *n*; *i* ++) //Initialization
- *visited*[*i*] = 0;
- for (*i* = 0; *i* < *n*; *i* ++) // DFS for every unvisited vertex
- if (*visited*[*i*] == 0)
- *DFS*(*G*, *i*);
- }

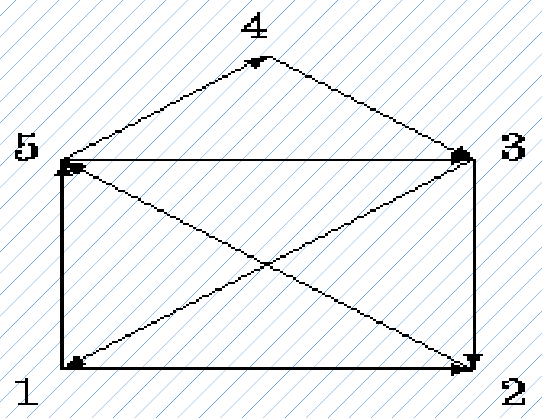
# 11.2.1 The House Of Santa Claus

- **Source: ACM Scholastic Programming Contest ETH Regional Contest 1994**
- **IDs for Online Judge: UVA 291**

- In your childhood you most likely had to solve the riddle of the house of Santa Claus. Do you remember that the importance was on drawing the house in a stretch without lifting the pencil and not drawing a line twice? As a reminder it has to look like shown in Figure.



- Well, a couple of years later, like now, you have to ``draw" the house again but on the computer. As one possibility is not enough, we require *all* the possibilities when starting in the lower left corner. Follow the example in Figure while defining your scetch.
- This Sequence would give the Outputline 153125432

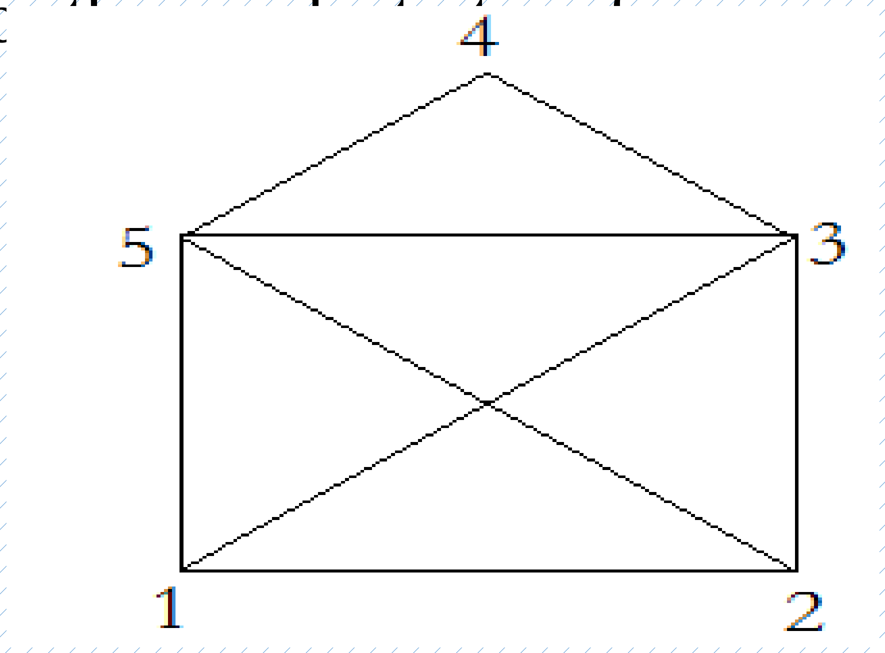


- All the possibilities have to be listed in the outputfile by increasing order, meaning that 1234... is listed before 1235... .

- **Output**
- 12435123
- 13245123
- ...
- 15123421

# Analysis

- The House of Santa Claus is an undirected graph with 8 edges (Figure).
- A symmetrical adjacency matrix *map*[ ][ ] is used to represent the graph. In the diagonal of the matrix, *map*[1][4], *map*[4][1], *map*[2][4], and *map*[4][2] are 0, and other elements are 1.
- The graph is a connected graph, DFS fo
vertex can visit all vertices and edges.



- The problem requires you to implement “drawing the house in a stretch without lifting the pencil and not drawing a line twice”. That is, the drawing must cover all 8 edges exactly once.
- The problem requires to list all possibilities by increasing order. Therefore DFS must visit all vertices starting from vertex 1.

# 11.3 Topological Sort

- Sort for a linear list is to sort elements based on keys' ascending or descending order.
- Topological Sort is different with sort for a linear list.
- Topological Sort is to sort all vertices in a Directed Acyclic Graph (DAG) into a linear sequence. If there is an arc $(u, v)$ in DAG, $u$ appears before $v$ in the sequence.

# Deleting arcs

- Step 1: Select a vertex whose in-degree is 0, and output the vertex;
- Step 2: Delete the vertex and arcs which start at the vertex, that is, in-degrees for vertices at which arcs end decrease 1;
- Repeat above steps.
  - If all vertices are output, the process of topological sort ends;
  - else there exists cycles in the graph, and there is no topological sort in the graph.
- The time complexity for the algorithm is O($E$).

- Using the algorithm for deleting arcs once, we can get one topological sort.
- Using recursive method, this algorithm is applied for all vertices whose in-degree is 0 successively, all topological sorts can be gotten.

# 11.3.1 Following Orders

- **Source: Duke Internet Programming Contest 1993**
- **IDs for Online Judge: POJ 1270, UVA 124**

- Order is an important concept in mathematics and in computer science. For example, Zorn's Lemma states: ``a partially ordered set in which every chain has an upper bound contains a maximal element.'' Order is also important in reasoning about the fix-point semantics of programs.
- This problem involves neither Zorn's Lemma nor fix-point semantics, but does involve order.
- Given a list of variable constraints of the form $x < y$, you are to write a program that prints all orderings of the variables that are consistent with the constraints.
- For example, given the constraints $x < y$ and $x < z$ there are two orderings of the variables $x$, $y$, and $z$ that are consistent with these constraints: $x\ y\ z$ and $x\ z\ y$.

- **Input**
- The input consists of a sequence of constraint specifications. A specification consists of two lines: a list of variables on one line followed by a list of contraints on the next line. A constraint is given by a pair of variables, where x y indicates that $x < y$.
- All variables are single character, lower-case letters. There will be at least two variables, and no more than 20 variables in a specification. There will be at least one constraint, and no more than 50 constraints in a specification. There will be at least one, and no more than 300 orderings consistent with the contraints in a specification.
- Input is terminated by end-of-file.
- **Output**
- For each constraint specification, all orderings consistent with the constraints should be printed. Orderings are printed in lexicographical (alphabetical) order, one per line.
- Output for different constraint specifications is separated by a blank line.

# Analysis

- Every variable (letter) is represented as a vertex, and a constraint $x<y$ is represented as an arc $(x, y)$.
- Therefore a list of contraints is represented as a directed graph.

① A directed graph is constructed based on the input.

Suppose *var* is the string for a list of variables. Because there are spaces in the string, *var*[0], *var*[2], *var*[4], ...... , are vertices, and the number of vertices is $\left\lfloor \frac{length(\text{var})}{2} \right\rfloor + 1$.

Suppose *v* is the string for a list of contraints. Array *pre* is used to store the sequence for vertices' in-degrees , where *pre*[*ch*] is the in-degree for vertex *ch*. Array *pre* is calculated as follow.

for (int *i*=0; *i*<the length of *v*; *i*+= 4) ++*pre*[the *i*+2-th letter in *v*];

② Get all Topological Sorts through DFS

All Topological Sorts in a directed graph can be gotten through DFS. Initial state is a subsequence *res* whose length is *dep*-1:

*dfs*(*dep*, *res*) {

If a Topological Sort is gotten (*dep*==*N*+1), then output *res* and backtrack (return);

Search vertex *i* whose in-degree is 0 (*has*[i]&& *pre*[i]==0, 'a'≤*i*≤'z'):

{ Delete vertex *i* (*has*[*i*]=false);

Delete all arcs which start form vertex *i* ( for(int *k*=0;*k*< the length of *v*; *k*+=4) if (the *k*th character in *v* ==*i*)--*pre*[the *k*+2-th character in *v*] );

*dfs*(*dep*+1, *res*+*i*);

return to the state before the recursion ( for(int *k*=0; *k*< the length of *v*; *k*+=4) if ( the *k*-th character of *v*==*i*) ++*pre*[the *k*+2-th character in *v*]; *has*[*i*]=true );

}

}

Obviously *dfs*(1, "") is called recursively and all Topological Sorts can be gotten.