Applications of Graph Traversal

Yonghui Wu
Stony Brook University
yhwu@fudan.edu.cn

Applications of Graph Traversal

- All vertices in a graph need to be visited exactly once. Such a process is called graph traversal.
- BFS and DFS are bases for many graph-related algorithms. Then based on BFS and DFS, topological sort and connectivity of undirected graphs are introduced.

Contents

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- topological sort
- connectivity of undirected graphs

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, $v \in V$:

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

- Initially $d[s]=0$; and for $v \in V-\{s\}$, $d[v]= -1$. The process for Breadth-First Search (BFS) is as follow.
- 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. Because u is the parent or the precursor for v, $d[v]=d[u]+1$.

- 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*.

```
void BFS(VLink G[ ], int v)  // BFS algorithm starting from source v in G
{ int w;
  visit v;
d[v]=0;              // distance d[v]
  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*. The algorithm of graph traversal based on BFS is as follow.

```
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);
}
```

Prime Path

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

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.

- Obviously, 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.

- Therefore, 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.

- Firstly 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. Therefore the directed graph needn’t to be stored, and we only need focus on calculating the shortest paths.

- The algorithm is as follow.
- 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:
- Step 3: Output the result: If the goal prime is gotten (*ans*≥0), then output the length of the shortest path *ans*; else output "Impossible".

DFS Algorithm

- DFS algorithm starts from a vertex *u*. Firstly vertex *u* is visited. Then unvisited vertices adjacent from *u* are selected one by one, and for each vertex DFS is initiated. The algorithm is as follow.
- void *DFS*(VLink *G*[ ], int *v*) // DFS starts from a vertex *v*
- { int *w*;
- *visited*[*v*] = 1; // Vertex *v* is visited.
- Get a vertex *w* adjacent from *v* (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 *v* (If there is no such a vertex *w*, *w*=-1.);
- }
- }

- *DFS*(*G*, *v*) visits the connected component containing vertex *v*. DFS for a graph is as follow.
- 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*);
- }

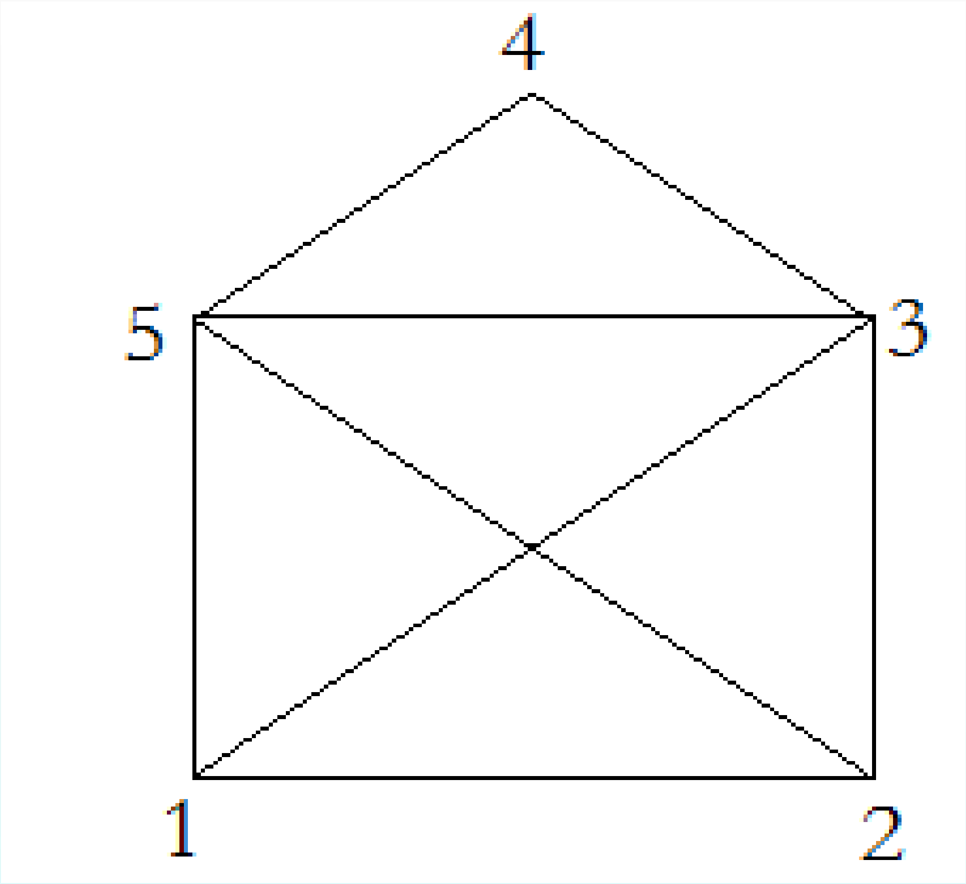
- For a graph with n vertices and e edges, the time complexity for DFS that initializes all vertices' marks is $O(n)$, and the time complexity for DFS is $O(e)$. Therefore, if $n \leq e$, the time complexity for DFS is $O(e)$.

The House Of Santa Claus

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

Analysis

- The House of Santa Claus is an undirected graph with 8 edges (Figure 11.2). 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. Because the graph is a connected graph, DFS for the graph starting from any vertex can visit all vertices and edges.


4
5
3
1
2

- 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 vertices exactly once. And the problem requires to list all possibilities by increasing order. Therefore DFS must visit all vertices starting from vertex 1.

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.
- There are two methods to implement Topological Sort: Deleting arcs, and Topological Sort implemented by DFS.

- 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 outputted, 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 $O(VE)$.

Following Orders

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

Topological Sort implemented by DFS

- Suppose x and y are vertices in a directed graph, and (x, y) is an arc. If x is in the set of vertices gotten by DFS(y), then arc (x, y) is a back edge. And its time complexity is $O(E)$.
- There is no cycle in a directed graph, if and only if there is no back edge in the graph.

- the algorithm of topological sort implemented by DFS is as follow.
- Suppose it takes one time unit to visit a vertex, the end time when vertex *u* and its descendants are all visited is *f*[*u*]. And *f*[*u*] can be calculated by DFS algorithm as follow. Obviously, if there exists a topological sort in the graph, there is no back edge in DFS traversal for the graph. That is, for any arc (*u*, *v*) in the graph, $f[v] < f[u]$.
- The topological sequence is stored in a stack *topo*. In *topo*, array *f*[ ] for vertices are in descending order from top to bottom.

- void *DFS-visit* (*u*); //DFS traversal for the subtree whose root is *u*
- { Set a visited mark for *u*;
- *time*=*time*+1;
- for each arc (*u*, *v*)
- if (*v* hasn't been visited)
- *DFS-visit* (*v*);
- *f*[*u*]=*time*;
- add *u* into stack *topo*;
- };

- Initially *time*=0, and set unvisited marks to all vertices. For every unvisited vertex *v*, *DFS-visit* (*v*) is called. Then stack *topo* and *f*[ ] can be gotten. If there exists an arc (*u*, *v*) in the graph such that *f*[*v*]>*f*[*u*], then (*u*, *v*) is a back edge, and topological sort fails; else all vertices from top to bottom in stack *topo* constitute a topological sequence.
- The time complexity for DFS is O(*E*), and the time complexity for adding all vertices into stack *topo* is O(1). Therefore, the time complexity for topological sort is O(*E*).

Sorting It All Out

- **Source: ACM East Central North America 2001**
- **IDs for Online Judge: POJ 1094, ZOJ 1060, UVA 2355**