CS201_Project_Report

SID:11912725
Name: 周民涛

The Topic I Chosen:

Demo of certain interesting algorithms. This may include exact steps of how the algorithm runs with given parameters. For example, (extended) Euclidean algorithm, RSA algorithm, Chinese Remainder Theorem, Roy-Warshall algorithm, topological sorting, Dijkstra algorithm, DFS/BFS algorithm, Tree traversal, Minimum spanning tree, etc.
In my report, I will show these demo:

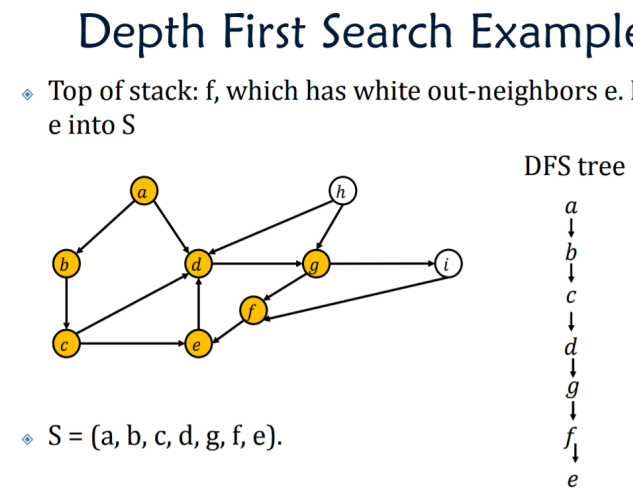
1. *DFS algorithm*
2. *BFS algorithm*
3. *Dijkstra algorithm*
4. *Topological sorting*
5. *Tree traversa algorithm*

1.DFS algorithm

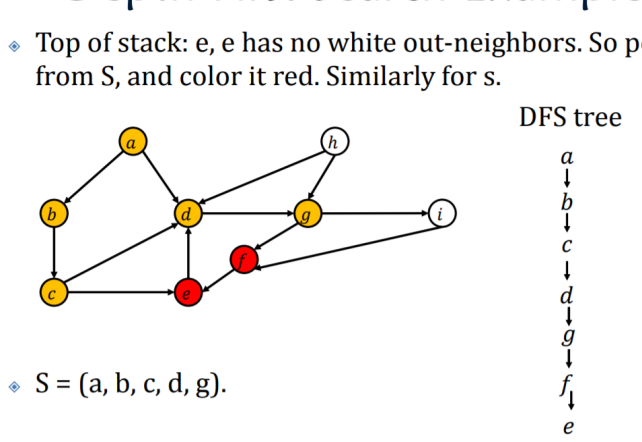
Introduction

Breadth-first search(BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Step



--->



Time complexity

$$O(|V| + |E|)$$

Application

Finding connected components.
Topological sorting.
Finding the bridges of a graph.

key code

```
ArrayList<Node> DFSsearch(){
    paintWhite();
    ArrayList<Node> roots=new ArrayList<>();
    for(Node a:allNode) {
        if(a.color==0) {
            roots.add(a);
            toplogicalorder.addAll(DFSsearch(a));
        }
    }
    return roots;
}
ArrayList<Node> DFSsearch(Node tar){
    Stack<Node> stack=new Stack<>();
    ArrayList<Node> out=new ArrayList<>();
    if(tar.color==0) {
        stack.push(tar);
        tar.color = 1;
        tar.child=new ArrayList<>();
        while (stack.getTop() != null) {
            Node now = stack.getTop();
            boolean have=false;
            for (Node i : now.connect) {
                if (i.color == 0) {
                    have=true;
                    stack.push(i);
                    now.child.add(i);
                    i.color = 1;
                    break;
                }
            }
            if(!have){
                stack.pop();
                now.color=2;
                out.add(now);
            }
        }
    }
    return out;
}
```

2.BFS algorithm

Introduction

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

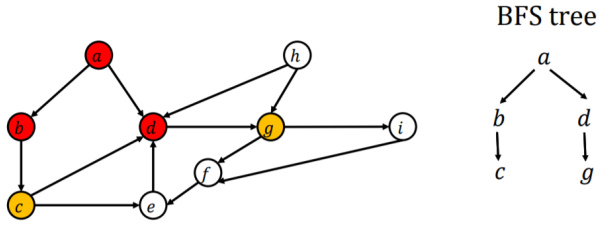
Step

Repeat the following until S is empty
Let v be the vertex that currently tops the stack S (do notremove v from S)
 Does v still have a white out-neighbor

- If yes: let it be u.
  - Push u into S, and color u yellow
  - Make u a child of v in the DFS-tree T
- If no, pop v from S, and color v red (meaning v is visited)
  - If there are still white vertices, repeat the above by restarting from an arbitrary white vertex v', creating a new DFS tree rooted at v'.

Breadth First Search Example

- After dequeuing d:

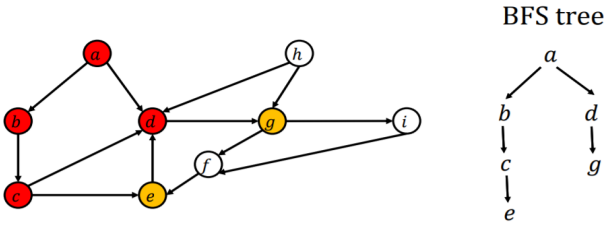


- Q = (c, g)

---->

Breadth First Search Example

- After dequeuing c:



- Q = (g, e)
- d is not enqueue again as it is red now

Time complexity

$$O(|V| + |E|)$$

Application

to find the topological order
To find the shortest path

key code

```
void BFSsearch(int nodeIndex){
    paintWhite();
    BFSsearch(allNode.get(nodeIndex));
}
void BFSsearch(Node a){
    Stack<Node> queue=new Stack<>();
    if(a.color==0) {
        queue.push(a);
        a.color = 1;
        a.child = new ArrayList<>();
        while (queue.getTop() != null) {
            Node now = queue.getFirst();
            now.child = new ArrayList<>();
            for (Node i : now.connect) {
                if (i.color == 0) {
                    i.depth=now.depth+1;
                    queue.push(i);
                    now.child.add(i);
                    i.color = 1;
                }
            }
            now.color = 2;
            queue.removeFirst();
        }
    }
}
```

3.Dijkstra algorithm

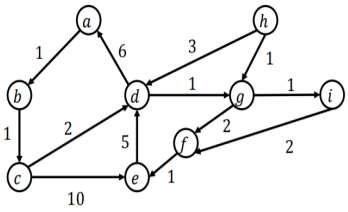
Introduction

Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road

networks.

Step

- Relax the out-going edge of g

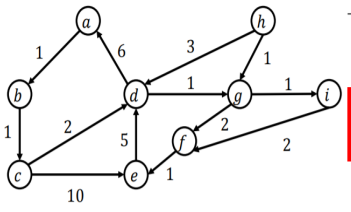


Vertex v	dist(v)	parent(v
a	8	d
b	∞	nil
c	0	nil
d	2	c
e	10	c
f	5	g
g	3	d
h	∞	nil
i	4	g

- S={a,b,e,f,h,i}
- Note that g has been removed!

---------->

- Relax the out-going edge of i



Vertex v	dist(v)	parent(v
a	8	d
b	∞	nil
c	0	nil
d	2	c
e	10	c
f	5	g
g	3	d
h	∞	nil
i	4	g

- S={a,b,e,f,h}
- Note that i has been removed!

Time complexity

$$O((|V| + |E|) * log|V|)$$

Application

To find the shortest path form source to anyplace

Example:

input

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

the first line is the sourse and the number of the place

given a adjacency matrix, the number is distance

sample Input:

```
0 9
0 4  0 0  0  0  0 8  0
4 0  8 0  0  0  0 11 0
0 8  0 7  0  4  0 0  2
0 0  7 0  9  14 0 0  0
0 0  0 9  0  10 0 0  0
0 0  4 14 10 0  2 0  0
0 0  0 0  0  2  0 1  6
8 11 0 0  0  0  1 0  7
0 0  2 0  0  0  6 7  0
```

output

sample output:

```
Distance from Source 0 to Vertex 0 is 0
Distance from Source 0 to Vertex 1 is 4
Distance from Source 0 to Vertex 2 is 12
Distance from Source 0 to Vertex 3 is 19
Distance from Source 0 to Vertex 4 is 21
Distance from Source 0 to Vertex 5 is 11
Distance from Source 0 to Vertex 6 is 9
Distance from Source 0 to Vertex 7 is 8
Distance from Source 0 to Vertex 8 is 14
```

code

```
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath {
    // A utility function to find the vertex with minimum distance value,
    // from the set of vertices not yet included in shortest path tree
    static int V;

    int minDistance(int dist[], Boolean sptSet[]) {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min) {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed distance array
    void printSolution(int dist[],int source) {
        for (int i = 0; i < V; i++)
//            System.out.println(i + " \t\t " + dist[i]);
            System.out.println("Distance from Source "+source +" to Vertex " +i +" is "+ dist[i]);
    }


    // Function that implements Dijkstra's single source shortest path
    // algorithm for a graph represented using adjacency matrix
    // representation
    void dijkstra(int graph[][], int src) {
        int dist[] = new int[V]; // The output array. dist[i] will hold
        // the shortest distance from src to i

        // sptSet[i] will true if vertex i is included in shortest
        // path tree or shortest distance from src to i is finalized
        Boolean sptSet[] = new Boolean[V];

        // Initialize all distances as INFINITE and stpSet[] as false
        for (int i = 0; i < V; i++) {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

        // Distance of source vertex from itself is always 0
        dist[src] = 0;

        // Find shortest path for all vertices
```

```
        for (int count = 0; count < V - 1; count++) {
            // Pick the minimum distance vertex from the set of vertices
            // not yet processed. u is always equal to src in first
            // iteration.
            int u = minDistance(dist, sptSet);

            // Mark the picked vertex as processed
            sptSet[u] = true;

            // Update dist value of the adjacent vertices of the
            // picked vertex.
            for (int v = 0; v < V; v++)

                // Update dist[v] only if is not in sptSet, there is an
                // edge from u to v, and total weight of path from src to
                // v through u is smaller than current value of dist[v]
                if (!sptSet[v] && graph[u][v] != 0 && dist[u] != Integer.MAX_VALUE && dist[u] + graph[u][v] < dis
                    dist[v] = dist[u] + graph[u][v];
        }

        // print the constructed distance array
        printSolution(dist,src);
    }

    // Driver method
    public static void main(String[] args) {
        /* Let us create the example graph discussed above */
        Scanner in = new Scanner(System.in);
        int source = in.nextInt();
        int length = in.nextInt();
        V = length;
        int graph[][] = new int[length][length];
        for(int i =0;i<length;i++){
            for(int j =0;j<length;j++){
                graph[i][j]=in.nextInt();
            }
        }

        ShortestPath t = new ShortestPath();
        t.dijkstra(graph, source);
    }
}
```

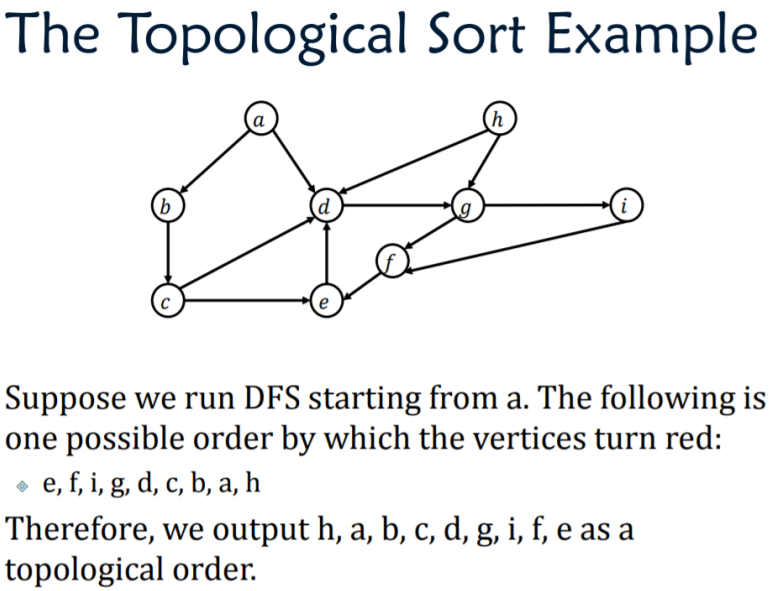
4.Topological sorting

Introduction

In computer science, a **topological sort** or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

Step

Every step we put those vertex which degree is 0 in the queue.
then, deque it ,reflesh the degree, if degree is 0 ,put it in queue.
Repeat until all vertex is visited.



Time complexity

$$O(|V| + |E|)$$

Application

Example:

One day, a group student in SUSTech want to combine some paragraphs into an essay. It is a terrible job because the paragraphs are written by different people. Each paragraph has a index number.
As a result, they come up with an idea - put these paragraphs into a queue and pop them out sequently and they just need to consider how to combine the paragraph from the front of the queue into the essay in every iteration.
However, there are n special regulations about the order of the paragraphs in the queue. Each regulation describes something like "paragraph a must be in front of paragraph b" using the notation "a b".
So it is your job to help them determine the order of the paragraphs in the queue.

input

Line 1: Two integers n(2≤n≤105) and m(1≤m≤105), where n indicates the number of paragraphs and m indicates the number of regulations.
There are m lines following:
Line 2~(m+1): Each line contains two integers a(1≤a≤n) and b(1≤b≤n), which means the paragraph amust be in front of paragraph b in the queue.
simple input:

```
3 3
1 2
2 3
1 3
```

output

Print n integers in range [1,n] indicating the order of these n paragraphs in queue.
If there are multiple answers, please output the one with the smallest lexicographic order.
simple output:

```
1 2 3
```

code

```
import java.io.*;
import java.util.*;

public class topologicalSort {
    static class PQueue {
        int value;
        PQueue parentNode;
        PQueue left;
        PQueue right;

        PQueue(int value) {
            this.value = value;
            parentNode = null;
            left = null;
            right = null;
        }
    }

    public static void main(String[] args) {
        OutputStream outputStream = System.out;
        Scanner in = new Scanner(System.in);
        PrintWriter out = new PrintWriter(outputStream);

        int numOfElement = in.nextInt();
        int numOfEdge = in.nextInt();

        PQueue root = new PQueue(0);
        int[] inDegree = new int[numOfElement + 1]; //入度
        ArrayList<Integer>[] graph = new ArrayList[numOfElement + 1];

        for (int i = 0; i < numOfElement + 1; i++) {
            graph[i] = new ArrayList<>();
        }

        for (int i = 0; i < numOfEdge; i++) {
            int dad = in.nextInt();
            int son = in.nextInt();
            inDegree[son]++;
            graph[dad].add(son);
        }

        int currentIndex = 0;
        for (int i = 1; i < numOfElement + 1; i++) {
            if (inDegree[i] == 0) {
                insert(root, new PQueue(i), currentIndex);
                currentIndex++;
            }
        }

        while (root.value != 0) {
            int now = root.value;
```

```
        out.print(root.value + " ");
        deleteMin(root, currentIndex);
        currentIndex--;

        for (int i = 0; i < graph[now].size(); i++) {
            inDegree[graph[now].get(i)]--;
            if (inDegree[graph[now].get(i)] == 0) {
                insert(root, new PQueue(graph[now].get(i)), currentIndex);
                currentIndex++;
            }
        }
    }

    out.close();
}

public static void deleteMin(PQueue root, int currentCount) {
    if (currentCount == 1) {
        root.value = 0;
        return;
    }

    PQueue keyNode = findCorrectPosition(root, currentCount - 1);
    PQueue deleteNode;

    if (keyNode.right == null) {
        deleteNode = keyNode.left;
        keyNode.left = null;
    } else {
        deleteNode = keyNode.right;
        keyNode.right = null;
    }
    deleteNode.parentNode = null;
    root.value = deleteNode.value;
    PQueue temp = root;

    while (temp.left != null && (temp.value > temp.left.value || temp.right == null || temp.value > temp.righ
        if (temp.value <= temp.left.value && temp.right == null) break;
        else {
            int tempValue = temp.value;
            if (temp.right == null || temp.left.value < temp.right.value) {
                temp.value = temp.left.value;
                temp.left.value = tempValue;
                temp = temp.left;
            } else {
                temp.value = temp.right.value;
                temp.right.value = tempValue;
                temp = temp.right;
            }
        }
    }
```

```
    }

    public static void insert(PQueue root, PQueue newNode, int currentIndex) {
        if (root.value == 0) {
            root.value = newNode.value;
            return;
        }

        PQueue keyNode = findCorrectPosition(root, currentIndex);
        newNode.parentNode = keyNode;

        if (keyNode.left == null) keyNode.left = newNode;
        else keyNode.right = newNode;

        PQueue temp = newNode;
        while (temp != root && temp.parentNode.value > temp.value) {
            int tempValue = temp.parentNode.value;
            temp.parentNode.value = temp.value;
            temp.value = tempValue;
            temp = temp.parentNode;
        }
    }

    public static PQueue findCorrectPosition(PQueue root, int currentNode) {
        PQueue result = root;
        String temp = getBinaryForm(currentNode + 1);
        for (int i = 1; i < temp.length() - 1; i++) {
            if (temp.charAt(i) == '1') result = result.right;
            else result = result.left;
        }
        return result;
    }

    public static String getBinaryForm(int currentNode) {
        String binaryForm = "";
        while (currentNode != 0) {
            binaryForm = currentNode % 2 + binaryForm;
            currentNode >>= 1;
        }
        return binaryForm;
    }

}
```

5.Tree traversa algorithm

Introduction

In computer science, **tree traversal** (also known as tree search and walking the tree) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

Step

Pre-order traversa

1.Access the data part of the current node.
2.Traverse the left subtree by recursively calling the pre-order function.
3.Traverse the right subtree by recursively calling the pre-order function.
The pre-order traversal is a topologically sorted one, because a parent node is processed before any of its child nodes is done.

In-order traversa

1.Traverse the left subtree by recursively calling the in-order function.
3.Access the data part of the current node.
3.Traverse the right subtree by recursively calling the in-order function.
In a binary search tree ordered such that in each node the key is greater than all keys in its left subtree and less than all keys in its right subtree, in-order traversal retrieves the keys in ascending sorted order.

Post-order traversa

1.Traverse the left subtree by recursively calling the post-order function.
2.Traverse the right subtree by recursively calling the post-order function.
3.Access the data part of the current node.

Time complexity

$$O(n)$$

(n is the number of the nodes)

Application

Pre-order traversal can be used to make a prefix expression (Polish notation) from expression trees.
Post-order traversal can generate a postfix representation (Reverse Polish notation) of a binary tree.

Post-order traversal while deleting or freeing nodes and values can delete or free an entire binary tree. In-order traversal is very commonly used on binary search trees.

Example:

input

first line is the number of the nodes, the root node and the opreate numbers
second line is the seven node value
follow 6 lines is the relation between nodes
sample input:

```
7 26 6
26 12 32 4 18 14 24
26 12
26 32
12 4
12 18
18 14
18 24
```

output

first line is preorder traversal
second line is inorder traversal
third line is postorder traversal
sample output:

```
26 12 4 18 14 24 32
4 12 14 18 24 26 32
4 14 24 18 12 32 26
```

code

```
import java.util.Scanner;

public class Main {
    static class node{
        int value;
        node left;
        node right;
        node(int value){
            this.value=value;
            left=null;
            right = null;
        }
    }
    static node ROOT;

    public static void main(String[] args) {
        Scanner in =new Scanner(System.in);
        int length = in.nextInt();
        int root1 = in.nextInt();
        int times =in.nextInt();
        node[] list = new node[length];
        for(int i = 0; i<length;i++){
            list[i] = new node(in.nextInt());
        }

        for(int time =0;time<times;time++){
            int a = in.nextInt();
            int b = in.nextInt();
            node root = null;
            node child =null;
            for(int i=0;i<length;i++){
                if(a == list[i].value){
                    root = list[i];
                }
                if(b == list[i].value){
                    child = list[i];
                }
            }
            if(root.left==null){
                root.left=child;
            }else {
                root.right=child;
            }
        }
        for(int i =0; i<length;i++){
            if(root1==list[i].value){
                ROOT = list[i];
            }
        }
        preOrderPrint(ROOT);
        System.out.println();
```

```
        inOrderPrint(ROOT);
        System.out.println();
        postOrderPrint(ROOT);
    }

    static void preOrderPrint(node root){
        System.out.print(root.value+" ");
        if(root.left!=null){
            preOrderPrint(root.left);
        }
        if(root.right!=null){
            preOrderPrint(root.right);
        }

    }
    static void inOrderPrint(node root){
        if(root.left!=null){
            inOrderPrint(root.left);
        }
        System.out.print(root.value+" ");
        if(root.right!=null){
            inOrderPrint(root.right);
        }

    }
    static void postOrderPrint(node root){
        if(root.left!=null){
            postOrderPrint(root.left);
        }
        if(root.right!=null){
            postOrderPrint(root.right);
        }
        System.out.print(root.value+" ");

    }
}
```

Reference

pictures from TangBo's dsaa(CS203) ppt
example from SUSTech OJ ,LeetCode and TangBo's dsaa(CS203) ppt
BFS and DFS's key code from github
https://en.wikipedia.org/wiki/Breadth-first_search
https://en.wikipedia.org/wiki/Depth-first_search

https://en.wikipedia.org/wiki/Dijkstra's_algorithm
https://en.wikipedia.org/wiki/Topological_sorting
https://en.wikipedia.org/wiki/Tree_traversal