Discrete_Math_Project_Report

11811901　　樊顺

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 project report, I will show these 7 demo:

1. `BFS`
2. `Dijkstra's algorithm`
3. `DFS`
4. `Dinic's algorithm`
5. `Topological  order`
6. `Minimum Spanning Tree`
7. `Strongly Connected Component (SCC)`

1. Demo of BFS

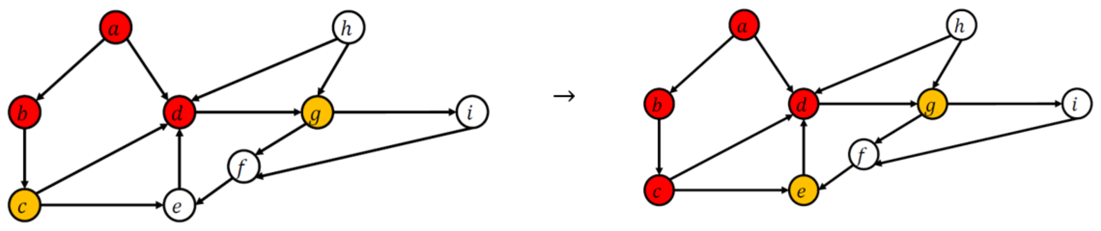
Introduction

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), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Algorithm runs steps

Given an initial point, take this initial point as the source, add this point to the queue, set the color to yellow, and then loop the queue until the queue is empty. In the loop body, remove the head element and set the color to Red, add all the subroutines of this routine, and replace the color with yellow.

- *Sample Show :*



- *`Pseudocode`*:

```
queue <- new empty Queue;
initial <- initial point;
queue.add(initial);
while(queue is not empty){
    temp <- queue.poll;
    add all son point of temp, which color is white, into queue;
}
```

Some Applications

- to find the shortest path in a Graph
- Find Topological order

Running time

$$O(\sum_{v \in V}(1 + d^{+}(v))) = O(|V| + |E|)$$

Example:

Give some Points and Edges, to find the shortest path from the first point (1) to the end point (n).

input

The first line contains 2 integers `n` and `m` means Points number and Edges number.

In each of the next `m` lines, there are 3 integers `u`, `v`(`1≤u,v≤n`) and `w`(`1≤w≤2`), which means there is a edge from `u` to `v`, and the weight of this edge is `w`.

Sample input:

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

output

Print the minimum weight in one line.

Sample output:

```
2
```

Code

First, we read in all points and edges, make an Adjacency list for this graph.

Then, input the List into BFS,

```
private static void BFS(Point m, LinkedList[] link, int n) {
    Queue queue = new Queue();
    m.colour = 1;
    Node mm = new Node(m);//Each node has current Point
    queue.enQueue(mm);
    while (!queue.isEmpty()) {
        Point point = queue.deQueue().point;
        //upadte color, means move away from queue
        point.colour = 2;
        if (point.value == 0) {
```

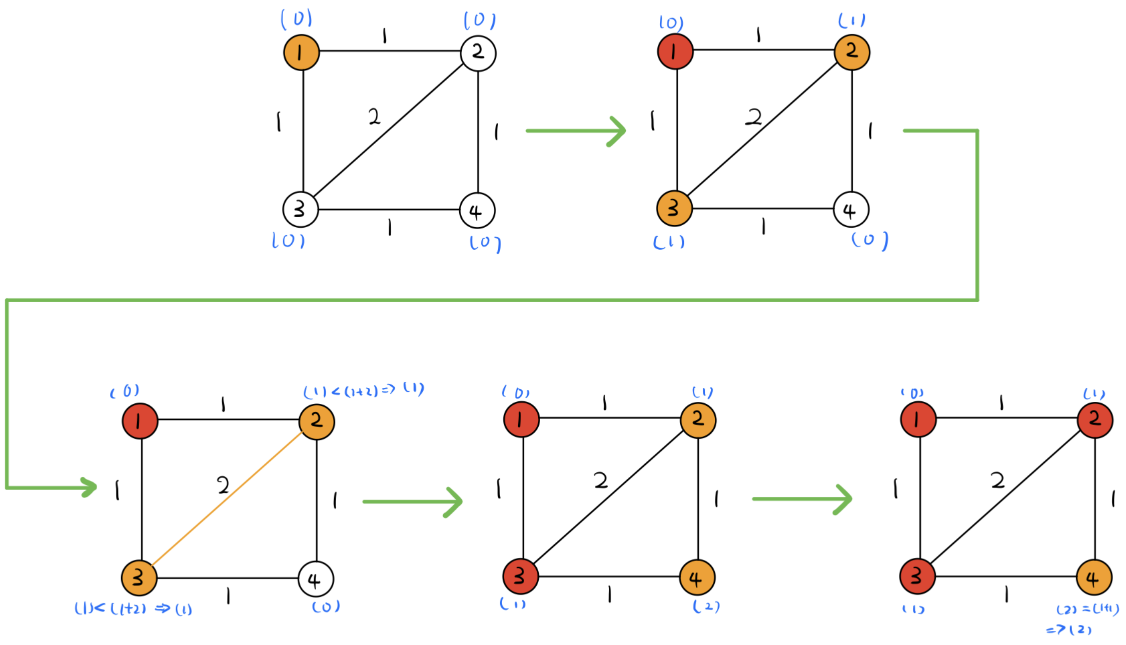
```java
            Point t = point.next;
            if (t.value == n) {
                //upadte color, means in queue
                t.colour = 1;
                t.AllWeight = point.AllWeight + 1;
                return;
            }
            if (t.colour == 0) {
                t.colour = 1;
                //update AllWeight
                t.AllWeight = point.AllWeight + 1;
                Node tt = new Node(t);
                queue.enQueue(tt);
            }
        } else {
            while (!link[point.value].isEmpty()) {
                //get the top node from queue
                Point t = link[point.value].remove().point;
                if (t.value == n) {
                    t.colour = 1;
                    //update AllWeight
                    t.AllWeight = point.AllWeight + 1;
                    return;
                }
                if (t.colour == 0) {
                    t.colour = 1;
                    //update AllWeight
                    t.AllWeight = point.AllWeight + 1;
                    Node tt = new Node(t);
                    // add new node into queue
                    queue.enQueue(tt);
                }
            }
        }
    }
}
```

After all of these, we just need to find the `destination Point`, and get the whole value of the `destination Point`

```java
System.out.println(points[n].AllWeight);
```

Steps

Some steps are shown in the figure below :



2. Demo of Dijkstra's algorithm

Introduction

`Dijkstra's algorithm` is an algorithm for finding the shortest paths between nodes in a graph.

In `Dijkstra's algorithm`, we utilizing the subsequence property, our algorithm will a shortest path tree that encodes all the shortest paths from the source vertex s.

A core operation in our algorithm is called `edge relaxation`. Given an edge (u,v), we relax it as :

```
if dist (v) < dist (u) + w(u, v):
    continue;
else
    reduce dist(v) to dist(u) + w(u, v);//dist(v) = dist(u) + w(u, v)
```

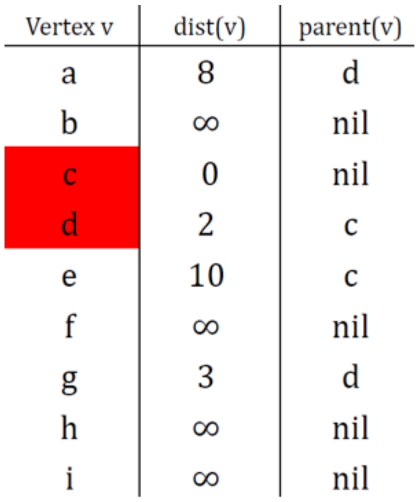
In this it is most same as the BFS, when BFS add the `edge relaxation` it is most like with `Dijkstra's algorithm`

Algorithm runs steps

Set dist (s) =0 and dist (v) = ∞for all other vertices $v \in V$.

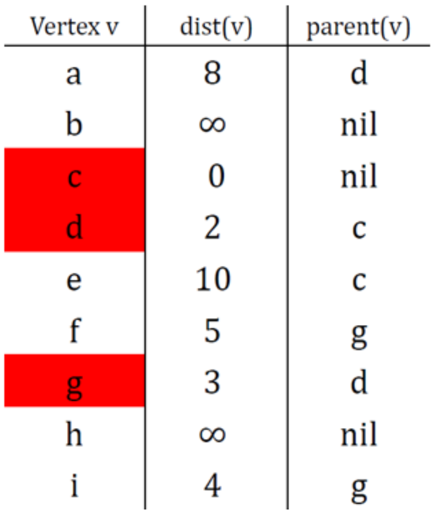
Use BFS to move by every point, and when it move away, calculate the cost to next point and compare these two value, if new one smaller then past, update the cost.

- *Simple Show:*



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

→



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

- `Pseudocode`:

```
queue <- new empty PriorityQueue
add initial point into queue
while queue is not empty:
    temp <- queue.poll()
    for all next point (next_point) of temp:
        if next_point.dist < temp.dist + w(temp, next_point):
            continue;
        else
            reduce next_point.dist to temp.dist + w(temp, next_point)
            //next_point.dist = temp.dist + w(temp, next_point)
            if next_point not in queue:
                add next_point into queue
```

Some Applications

- To find the shortest path in a Graph

Runtime

Implement `Dijkstra's algorithm` in:

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

Example

Give some Points and Edges, to find the shortest path from the first point (1) to the end point (n) and the edges have its own weight.

input

The first line contains two integers: `n` and `m` — the number of points and edges.

Each of the next `m` lines contains three space-separated integers: `u`, `v` and `w`, meaning that there is a bidirectional road from sight `u` to sight `v` with distance `w`.

The last line contains two integers: `S` and `T` — the origin and destination.

Sample input :

```
3 3
1 2 5
2 3 5
3 1 2
1 3
```

output:

Print the result — the shortest distance between point `S` and point `T`.

Sample output :

```
2
```

code

The code of `Dijkstra’s algorithm` is:

```java
public static void DiJi(int begin, Node[] point, ArrayList<ArrayList<edge>> edges) {
    Queue<Node> queue = new PriorityQueue<>((o1, o2) -> (int) (o1.length - o2.length));
    point[begin].length = 0;
    queue.add(point[begin]);
    while (!queue.isEmpty()) {
        Node temp = queue.poll();
        for (int i = 0; i < edges.get(temp.value).size(); i++) {
            int path = temp.length +  edges.get(temp.value).get(i).weight;
            int len = point[edges.get(temp.value).get(i).next.value].length;
            if (len == -1 || len > path) {
                point[edges.get(temp.value).get(i).next.value].length = path;
                point[edges.get(temp.value).get(i).next.value].parent = temp;
                queue.add(point[edges.get(temp.value).get(i).next.value]);
            }
        }

    }
}
```

Get Result to output:

```java
System.out.println(point[end].length);
```

Steps

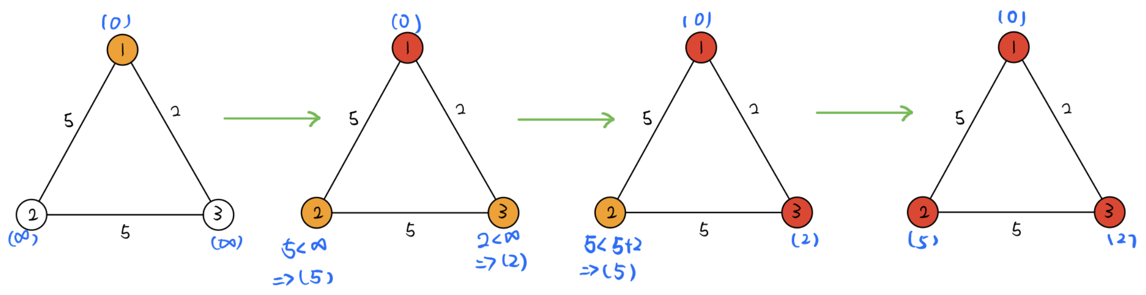
First, we read in all input, and make a `Adjacency list` for this graph.

Then, input `Adjacency list` into `DiJi` to run the `Dijkstra's algorithm`.

After `Dijkstra's algorithm` , we get the result as `System.out.println(point[end].length);`

The steps of `Dijkstra's algorithm` in the above question is showed as the picture below.

Some steps are shown in the figure below :



3. Demo of DFS

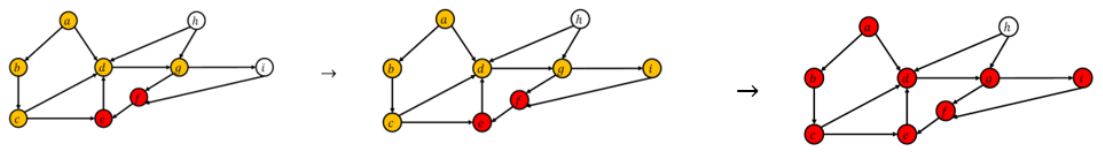
Introduction

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.

Algorithm run steps

Given an initial point, take this initial point as the source, add this point to the stack, set the color to yellow, and then loop the stack until the queue is empty. In the loop body, get the head element, add one of the subroutines of this routine, and replace the color with yellow. If this line can add edge anymore, pop it , and set the color to Red.

- *Simple Show*



- `Pseudocode`:

```
DFS(Node e){
for all next node (next_node) of e:
    if next_node.color is white:
        DFS(next_node)
        break
}
```

some Applications

- Topological sorting
- Dinic's algorithm
- Strongly Connected Component (SCC)

Running Time

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

Example

Show in next part: Dinic's algorithm

4. Demo of Dinic's algorithm

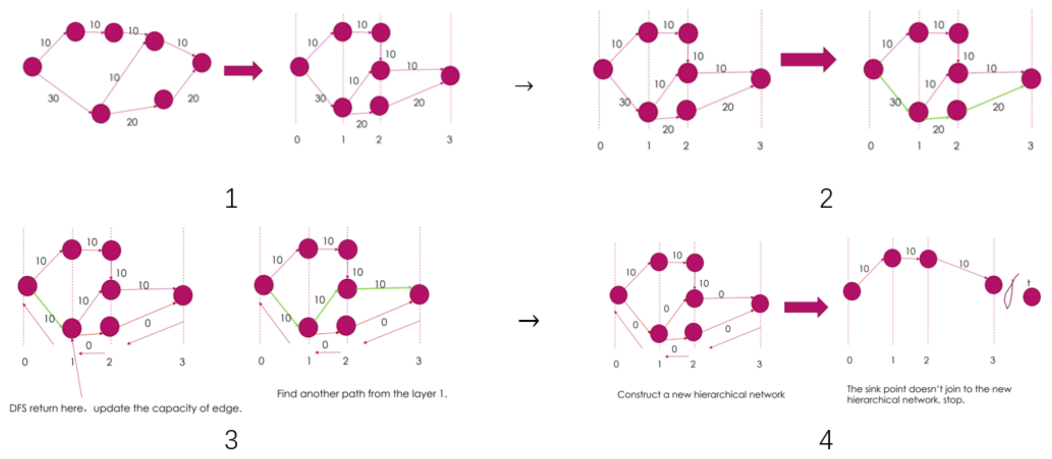
Introduction

"**Dinic's algorithm** is a strongly polynomial algorithm for computing the maximum flow in a flow network. The algorithm runs in $O(V^2E)$ time and is similar to the Edmonds–Karp algorithm, which runs in $O(VE^2)$ time, in that it uses shortest augmenting paths. "

Algorithm run steps

Initial residual graph

1. construct residual graph and hierarchical network， If the sink is not in the hierarchical network,
   stop; Otherwise, proceed to step 2.
2. Using DFS algorithm, proceed from the vertex of layer l to the vertex of layer l +1, if reach the sink point t, means find an augmented path.
3. Go back and update the capacity value. When return to layer l, and you find that you have another edge can reach layer l + 1. If the path reach t, you find another augmented path.
4. Step back, update the capacity value, and if no new augmented path can be found until step back to point s, the DFS process ends. Go back to step 1.


1
2
DFS return here，update the capacity of edge.
Find another path from the layer 1.
Construct a new hierarchical network
The sink point doesn't join to the new hierarchical network, stop.
3
4

some Applications

- Image Segmentation
- maximum Network flow

Running Time

According to Wiki:

It can be shown that the number of layers in each blocking flow increases by at least 1 each time and thus there are at most $|V|-1$ blocking flows in the algorithm. For each of them:

- the level graph G_L can be constructed by breadth-first search in $O(E)$ time
- a blocking flow in the level graph G_L can be found in $O(VE)$ time

with total running time $O(E + VE) = O(VE)$ for each layer. As a consequence, the running time of Dinic's algorithm is $O(V^2E)$.

Example

Input

The first line contains 2 integers `N` and `M`.

Each of the next `M` lines contains two integers `i` and `j` which mean there is an edge connect the i-th node and the j-th node.

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

Output

Output the maximum flow of the graph.

```
2
```

Code

The Algorithm Code of `Dinic's algorithm` is :

```
static int Dinic(){
        int re = 0;
        while (bfs()) {
            while (dfs(node[0])) {
                re += 1;
            }
        }
        return re;
    }
```

```
static boolean bfs() {
        for (int i = 0; i < n + 2; i++) {
            node[i].rank = -1;
        }
        boolean result = false;
        Queue<Node> queue = new LinkedList<>();
        queue.add(node[s]);
        node[s].rank = 0;
        while (!queue.isEmpty()) {
            temp = queue.poll();
            if (temp.index == t)
                result = true;
            for (int i = 0; i < edge.get(temp.index).size(); i++) {
                if (edge.get(temp.index).get(i).rank == -1) {
                    edge.get(temp.index).get(i).rank = temp.rank + 1;
                    queue.add(edge.get(temp.index).get(i));
```

```java
                }
            }
        }

        return result;
    }
```

```java
static boolean dfs(Node tt) {
        for (int i = 0; i < edge.get(tt.index).size(); i++) {
            if (edge.get(tt.index).get(i).index == t) {
                edge.get(tt.index).remove(node[t]);
                return true;
            }
            if (edge.get(tt.index).get(i).rank == tt.rank + 1) {
                if (dfs(edge.get(tt.index).get(i))) {
                    edge.get(tt.index).remove(i);
                    return true;
                }
            }
        }
        return false;
    }
```

Steps

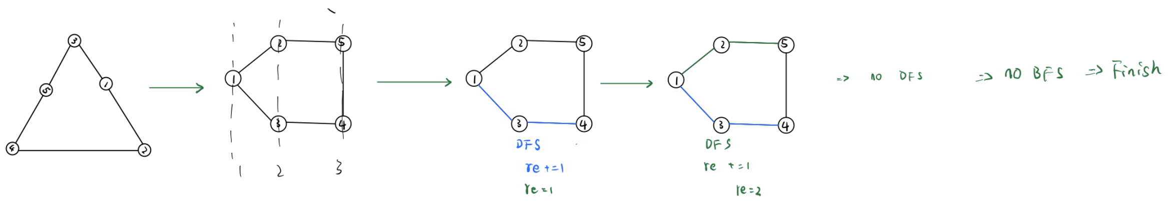
First of all, we run BFS, give every node a rank;

Then run DFS, which only can move to next rank node.

After DFS, we run BFS again.

Repeat above until can't run BFS, algorithm finish.

Some steps are shown in the figure below :



5. Demo of Topological order

Introduction

According to `wiki`, we have

"In physics, topological order is a kind of order in the zero-temperature phase of matter (also known as quantum matter). Macroscopically, topological order is defined and described by robust ground state degeneracy and quantized non-Abelian geometric phases of degenerate ground states. Microscopically, topological orders correspond to patterns of long-range quantum entanglement. States with different topological orders (or different patterns of long range entanglements) cannot change into each other without a phase transition.

Topologically ordered states have some interesting properties, such as

(1) topological degeneracy and fractional statistics or non-abelian statistics that can be used to realize topological quantum computer;

(2) perfect conducting edge states that may have important device applications;

(3) emergent gauge field and Fermi statistics that suggest a quantum information origin of elementary particles;

(4) topological entanglement entropy that reveals the entanglement origin of topological order, etc. Topological order is important in the study of several physical systems such as spin liquids and the quantum Hall effect, along with potential applications to fault-tolerant quantum computation.

Topological insulators and topological superconductors (beyond 1D) do not have topological order as defined above, their entanglements being only short-ranged."

And 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. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time."

Algorithm run steps

In the Topological sorting, we use the BFS to find it.

When run the BFS, we find a node, which has 0 in-degree, and begin from this run BFS.

Then run one node, make the in-degree of that sub 1.

If in-degree of all node v ($v \in V$) is 0, algorithm finish.

If initial can't find 0 in-degree node, means this graph haven't topological order.

- `Pseudocode`:

```
topplogical(){
    queue <- new empty Queue;//sorted with in_degree
    initial <- initial point;//which in-degree is 0
    queue.add(initial);
    while(queue is not empty)
        temp <- queue.poll;
        if temp.in_degree != 0
            return false
        for all node v in temp's next edge:
            add v in queue
            v.in_degree -= 1
}
```

some Applications

- shortest path finding
- speech recognition system
- deep learning

Running Time

$$O(\sum_{v\in V}(1+d^+(v)))=O(|V|+|E|)$$

Example

Input

The first line contains 2 integers `n` and `m` means Points number and Edges number.

In each of the next `m` lines, there are 3 integers `u`, `v`(`1≤u,v≤n`), which means there is a edge from `u` to `v`.

Sample input:

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

Output

Print the a topological order in one line.

```
1 -> 3 -> 4 -> 2 -> 5
```

Code

The core code of topological sorting is :

```
private static void topological(Node begin) {
        Queue<Node> queue = new LinkedList<>();
        queue.add(begin);
        begin.color = 1;
        while (!queue.isEmpty()){
            Node temp = queue.poll();
            System.out.print(temp.i + " -> ");
            temp.color = 2;
            for (int i = 0;i < temp.next.size();i++){
                Node t = temp.next.get(i);
                t.color = 1;
                t.in -= 1;
                if (t.in == 0)
                    queue.add(t);
            }
        }
    }
```

Steps

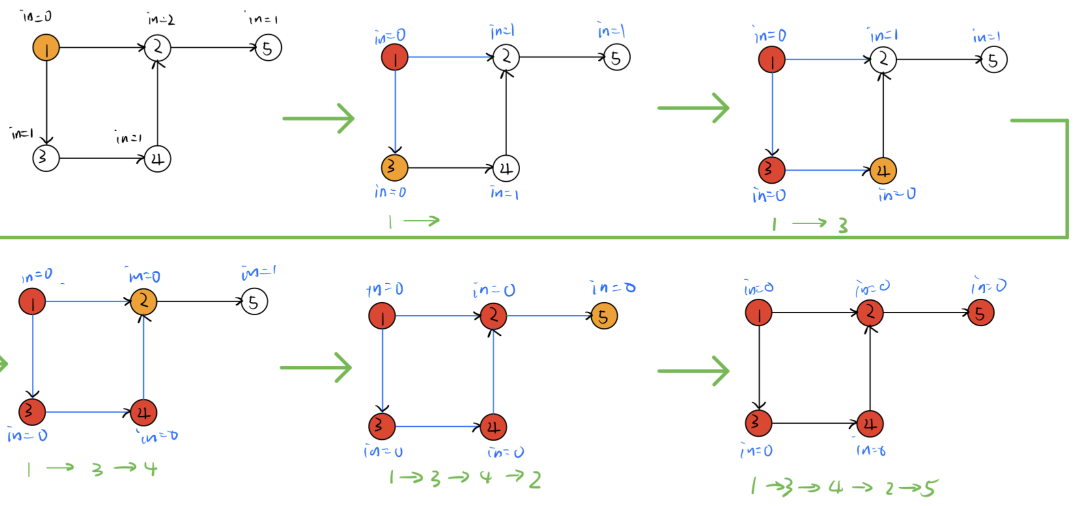
First, find the node, which in_degree is 0, if has lot of it 0 in_degree take one at a time.

Then, input the in_degree into Topological method. Then run Topological, each popped node make all next node's in_degree sub 1, if one of them is equal to 0, add it into queue.

when all node's in_degree is 0, the topological sorting is finished.

If all remain node's in-degree is 1,Topological sorting failed.

Some steps are shown in the figure below :



6. Demo of Minimum Spanning Tree

Introduction

According to `wiki`, we have

“A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the **minimum possible total edge weight**. That is, it is a spanning tree whose sum of edge weights is **as small as possible**. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.”

Algorithm run steps

When initial the data, we read in the data,and make them to an Adjacency list.

Then, input the point into prime method, which is the core method of MST.

In Prime, we use the `Priority Queue`, and sorted by the weight of each edge.

Get the peak edge in queue, then input all next_edge of current point if next point not used.

when all points used, Prime finished. Output the sum of weight.

- *`Pseudocode`*:

```
prime(points){
queue <- new PriorityQueue compared with weight
points[0].used <- -1
add points[0].next into queue
while queue is not empty:
    temp <- queue.poll
    if temp.next.used == -1:
        continue
    sum += temp.weight
    for e in temp.next.next:
        if e.next.used != -1
            add e into queue
}
```

some Applications

- Install the communication line network line
- Optimal routing problem
- The minimum cost of connecting n cities

Running Time

data Structure	Time
Adjacency_matrix, search	$O(\|V\|^2)$
Binary_heap and Adjacency list	$O((\|V\|+\|E\|)\log\|V\|)=O(\|E\|\log\|V\|)$
Fibonacci heap and Adjacency_list	$O(\|E\|+\|V\|\log\|V\|)$

Example

Input

The first line contains 2 integers `n` and `m` means Points number and Edges number.

In each of the next `m` lines, there are 3 integers `u`, `v`(`1≤u,v≤n`) and `w`(`1≤w≤2`), which means there is a edge from `u` to `v`, and the weight of this edge is `w`.

```
5 7
1 5 3
1 4 3
1 2 1
2 3 1
3 5 4
3 4 2
4 5 2
```

Output

Print the minimum weight sum in this graph when connect all point.

```
6
```

Code

The core method of MST is Prime Method:

```java
public static void prim(Node[] point){
    Queue<Edge> queue = new PriorityQueue<>(Comparator.comparingInt(o ->
o.weight));
    point[0].v = -1;
    queue.addAll(point[0].next);

    while (!queue.isEmpty()){
        ee = queue.poll();
        if (point[ee.next].v == -1)
            continue;
        sum += ee.weight;
        point[ee.next].v = -1;
        Node temp = point[ee.next];
        for (int i = 0;i < temp.next.size();i++){
            if (point[temp.next.get(i).next].v != -1){
                queue.add(temp.next.get(i));
            }
        }
    }
}
```

```java
System.out.println(sum);
```

Steps

First, Read in all data, and make a Graph.

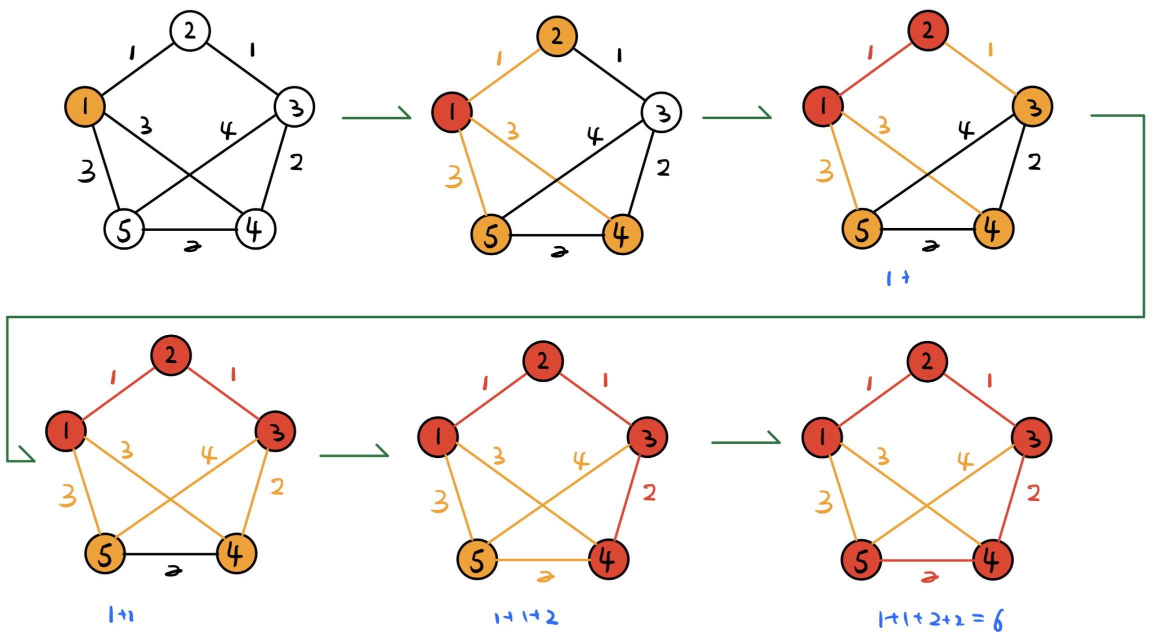
Input the Adjacency list into Prime method.

In prime method, we add all edges of this point, which next point not used.

And in loop, each step we pop the smallest weight edge, and the next point marked as used. And begin from this marked point, add all next edges, which next point not used, again.

Repeat above steps, until all points are used, then output the sum.

Some steps are shown in the figure below :



7. Demo of Strongly Connected Component (SCC)

Introduction

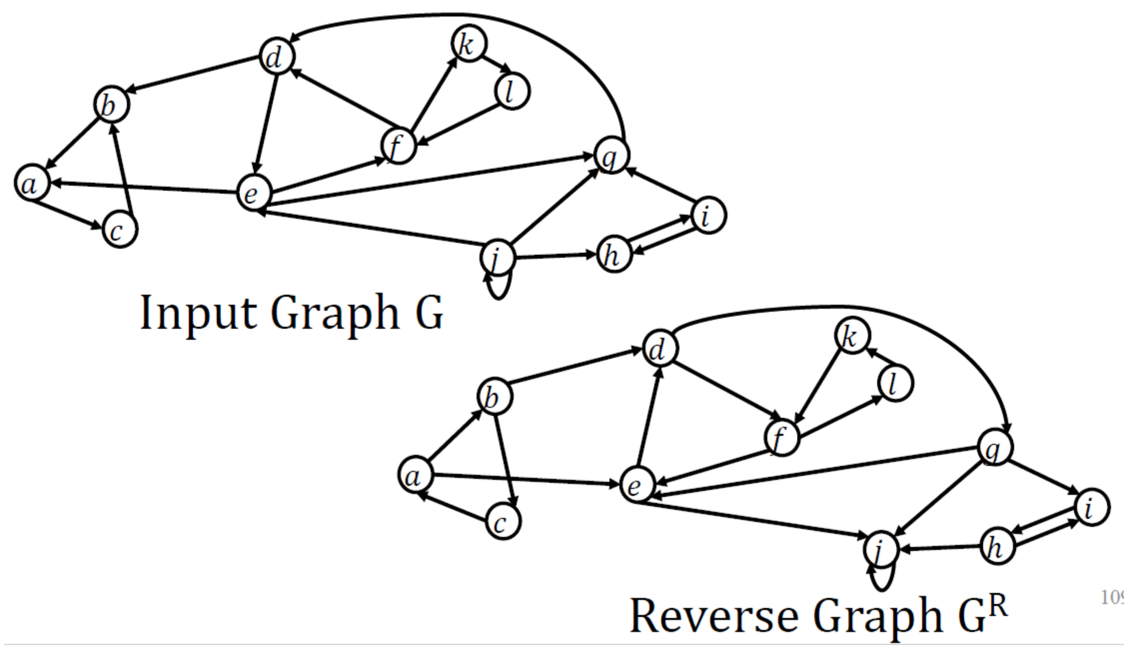
According to `wiki`, we have :

"In the mathematical theory of `directed graphs`, a graph is said to be strongly connected if **every vertex is reachable from every other vertex**. The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time (that is, `Θ(V+E)`)."

Algorithm run steps

1. obtain the reverse graph G^R by reversing the directions of all the edges in G.
2. Perform DFS on G^R , and obtain the sequence L^R that the vertices in G^R turn red (i.e., whenever a vertex is popped out of the stack, append it to L^R.
3. Perform DFS on the original graph G by obeying the following rules:
    - start the DFS at the first vertex of L
    - whenever a restart is needed, start from the first vertex of L that is still white.

- *Sample Show :*



Input Graph G

Reverse Graph G^R

- `Pseudocode :`

```
LR <- new empty ArrayList
DFS(LR,begin,reverseEdge,point)
reset all point color into 0
L <- reverseArrayList(LR);
secondDFS(1,inWhichSet,L,L.get(0),edges,reverseEdge,point);
```

some Applications

- a classification of the edges of a bipartite graph
- compute the `Dulmage–Mendelsohn decomposition`

Running Time

Steps 1 and 2 obviously require only O(|V|+|E|) time.

Regarding Step 3, the DFS itself takes O(|V|+|E|).

The cost of implement Rule 2 can be done in O(|V|) total time.

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

Example

Input

The first line contains 2 integers `n` and `m` means Points number and Edges number.

In each of the next `m` lines, there are 3 integers `u`, `v`(`1≤u,v≤n`), which means there is a edge from `u` to `v`.

```
6 8
1 2
2 3
3 1
5 3
4 1
5 4
4 6
6 5
```

Output

Output all `SCCs` in this Graph.

```
Number of SCC is : 1
3 2 1
Number of SCC is : 2
6 4 5
```

Code

The code below is the core code of the `SCC` algorithm:

In main we can call the core method of `SCC`.

```java
private static void SCC(){
    Node begin = point[1];
    ArrayList<Node> LR = new ArrayList<>();
    DFS(LR,begin,reverseEdge,point);
    for (Node node : point) {
        node.color = 0;
    }
    ArrayList<Node> L = reverseArrayList(LR);
    secondDFS(1,inWhichSet,L,L.get(0),edges,point);
}
```

```java
public static void secondDFS(int setIndex,
                                 int[] inWhichSet,
                                 ArrayList<Node> L,
                                 Node begin,
                                 ArrayList<ArrayList<Node>> edge,
                                 Node[] point) {
    Stack<Node> s = new Stack<>();
    ArrayList<Node> SSC = new ArrayList<>();
    s.push(begin);
    begin.color = 1;
    while (!s.isEmpty()) {
        Node temp = s.peek();
        boolean pushed = false;
        for (Node t : edge.get(temp.value)) {
            if (!pushed && t.color == 0) {
                s.push(t);
                t.color = 1;
```

```java
                    pushed = true;
                }
            }
            if (!pushed) {
                Node t = s.pop();
                t.color = 2;
                SSC.add(t);
                inWhichSet[t.value] = setIndex;
            }
        }
        System.out.println("Number of SCC is : " + count);
        count += 1;
        for (Node node : SSC) {
            System.out.print(point[node.value].value + " ");
        }
        System.out.println();
        boolean hasWhite = false;
        Node beginning = null;
        for (int i = 1; i < L.size(); i++) {
            if (L.get(i).color == 0) {
                hasWhite = true;
                beginning = L.get(i);
                break;
            }
        }
        if (hasWhite)
            secondDFS(setIndex + 1, inWhichSet, L, beginning, edge, point);
}
```

```java
public static void DFS(ArrayList<Node> L, Node begin,
ArrayList<ArrayList<Node>> edge, Node[] point) {
    Stack<Node> s = new Stack<>();
    s.push(begin);
    begin.color = 1;
    while (!s.isEmpty()) {
        Node temp = s.peek();
        boolean pushed = false;
        for (Node t : edge.get(temp.value)) {
            if (!pushed && t.color == 0) {
                s.push(t);
                t.color = 1;
                pushed = true;
            }
        }
        if (!pushed) {
            Node t = s.pop();
            t.color = 2;
            L.add(t);
        }
    }
    boolean hasWhite = false;
    Node beginning = null;
    for (int i = 1; i < point.length; i++) {
        if (point[i].color == 0) {
            hasWhite = true;
            beginning = point[i];
        }
    }
    if (hasWhite)
        DFS(L, beginning, edge, point);
}
```

```java
public static ArrayList<Node> reverseArrayList(ArrayList<Node> old) {
    ArrayList<Node> newOne = new ArrayList<>();
    for (int i = old.size() - 1; i >= 0; i--) {
        newOne.add(old.get(i));
    }
    return newOne;
}
```

Steps

First, read in the data, and build the Graph G. The reverse the graph and get the G^R.

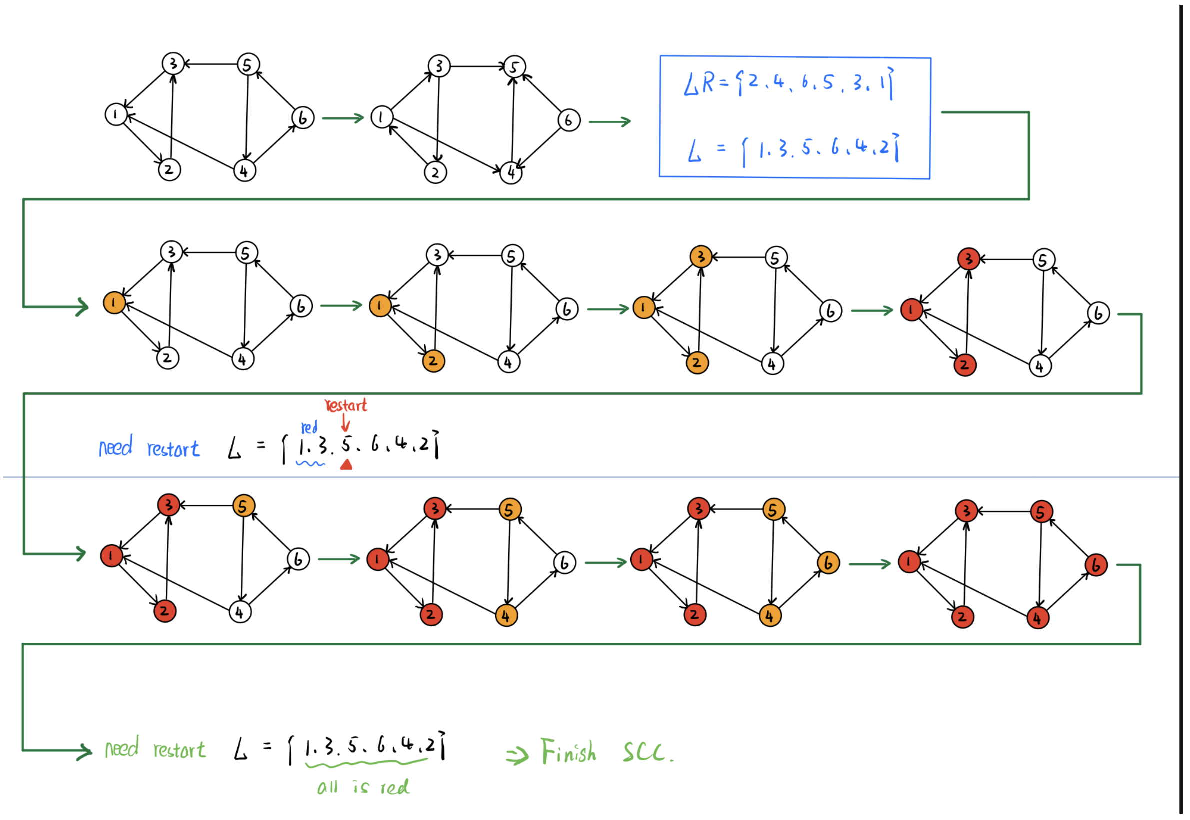
Run the `DFS` on the G^R and get the reverse list L^R.

Then, reverse the list L^R to get the list L.

After these, run `second DFS` on the original graph G. When `DFS` can't run, output the stack and change the root, as the first white point in list L.

If `second DFS` can't run. This means `SCC` algorithm is finished.

Some steps are shown in the figure below :



Reference

1. Some example from SUSTech Online Judge
2. Taken some introduction and picture from `Professor Tang Bo‘ s PPT`
3. wiki,Dijkstra's algorithm
4. wiki,BFS
5. wiki,DFS
6. wiki,Dinic
7. Taken some introduction and picture from `Professor Zhao Yao‘ s PPT`
8. wiki,Topological sorting
9. wiki,Topological order
10. wiki,Minimum Spanning Tree
11. wiki,Strongly Connected Component