Assignment2

12110644 周思呈

1 Usage of Section 5.1

Section 5.1 describe the whole algorithm in a high level, giving readers a general idea of how the algorithm will be implemented.

The first paragraph describes the implementation of the previous section. Then the second paragraph points out that the *cheapest entering edges set F* always forms a path consisting both original vertices and super-vertices. As a result, the efficient implementation starts with an arbitrary vertex and extend the path step by step. When a cycle is detected, it will be contracted into a super-vertex. The program terminates when the whole graph is contracted into a single node. At last, in order to get the required MDST, those super-vertex should be expanded so that the edges forming a cycle in contraction phase can be replaced with their true in-edges. In this way, this big problem is divided into two smaller parts, contraction phase and expansion phase, each of which can be conquered more easily.

After that, section 5.1 explained why the identity of root r can be ignored. The first reason is that we can choose the cheapest entering edge without distinction in the contraction phase. By adding sufficiently large weight edges we can transform the original graph into a strongly connected graph, i.e., every vertex can arrive at any other vertexes. Thus, the root node also has its cheapest entering edge. Otherwise the contraction phase will terminate as soon as root is extended in the path. Secondly, in this way, this algorithm can be generally used to find MDST start from any root.

It also proposed five questions closely related to the specific implementation in the last but three paragraph. When reading the next few sections, the readers will try to answer these questions, thus having a clear outline.

At last, this section introduced two main data structures the algorithm will be using, union-find and priority queue(leftist tree), and briefly described their usages.

2 Algorithm Implementation

2.1 Preparation Phase

The function MDST(G = (V, E, w), r) requires a strongly connected graph G and a root r as its input. However, if a graph G' is not strongly connected or root r is not given, we can change G' to G by adding a vertex and some edges to it.

1. Add a dummy root and add edges from this dummy root to every other vertexes with weight sufficiently big. In the final result, there will be an edge from the dummy root to the true root.
2. Add edges connecting vertex i and vertex (i+1) with sufficiently big weight. Then the graph is strongly connetced. Also, because the added edges' weights are big, when chosing the cheapest entering edge of a vertex, they will not be preferred. If the final selected edges contain such large weight edge, that means the original graph has no fair MDST result.
   After the above two operations, the graph can be put into the function MDST.

2.2 Contraction Phase

Maintain five variables of each vertex u:

- **u.in**, the selected incoming edge of u. Initially u.in = null. In the contraction phase, it is the cheapest edge entering u. In the expansion phase, it is the edge selected for the final MDST. In *Lecture notes on "Analysis of Algorithms": Directed Minimum Spanning Trees*, the cheapest edge is selected through priority queue. More specifically, we use leftist tree here in order to decrease the complexity of the operation *meld*.
- **u.const**, the weight adjust constant. Not really implemented. In fact, use lazy lable to optimize time complexity. But the functionality is the same as decribed in the lecture notes. Initially u.const = 0. The weight of edges entering the node which has already chosen its cheapest in-edge should minus this constant. This is to calculate the relative weight difference of each edge when chosing the cheapest.
- **u.prev**, the vertex preceding u in the path constructed by the algorithm. Initially u.prev = null. In the contraction phase, it is used to extend the path and to detect cycle.
- **u.parent**, the super-vertex into which u was contracted. Initially u.parent = null.
- **u.children**, if u is a super-vertex, then u.children are the vertexes forming u.
- **u.Proot**, the root of the leftist tree. The leftist tree contains all the non-selected edges entering u. The root element is the smallest, and it allows O(logn) meld time complexity.

The contraction operations are as follows

1. First initialize all the vertexes. Add all the edges entering u to its' leftist tree. Pick an arbitrary vertex *a* to start contraction. This may lead to different MDST result at last, because when extending the path to the dummy root, there is possibility that multiple edges have the same sufficiently large edge weight. But the final weight summation will be the same.

2. Extract *a.Proot* from its' leftist tree, which is the cheapest edge entering *a*, referred to as *(u, v)*. Before path extension or cycle contraction, we should first detect whether it is an edge connecting two different (super-)vertexes. Invoke function find() to get the biggest super-vertex formed by *u*(and other vertexes), referred to as *b*. If *a* != *b*, it is not a self-loop, move to the next operations. Otherwise, extract another cheapest edge.

3. Set *a.in* as *(u, v)*. Set *a.prev* as *b*. Then decide whether to extend a path or contract a new super-vertex, i.e., decide whether there is a cycle. In the path extension, we always extend a node to its previous one. Thus, if *u* has an entering edge, that means a cycle has been formed and those nodes should be contracted to a super-vertex. If not, simply extend the path.

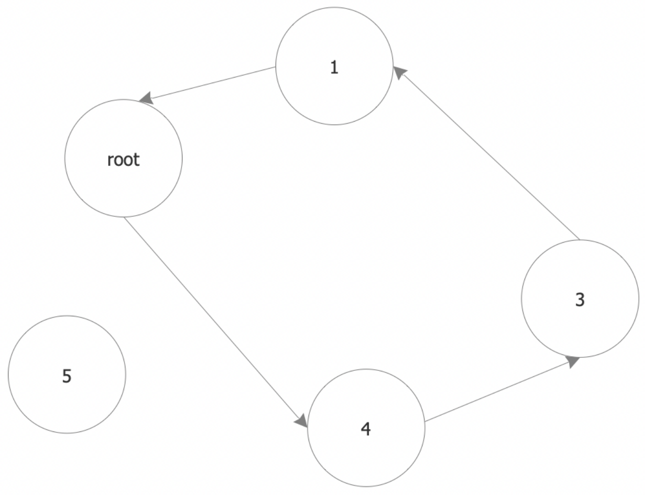

Fig2.1, extend path from root node, as long as b.in != null, a new cycle has been detected, image source: me

4. Create a super-vertex *c*. If vertex *a* has not yet been contracted, set its' parent to *c*. Record its entering weight to *a.Proot.lazy* and *a.Proot.weight*. Correspondingly add *a* to *c*'s children list. Use leftist tree to meld *c.Proot* and *a.Proot*. Contract every vertex in this cycle.

2.3 Expansion Phase

After the contraction phase, the original strongly connected graph is contracted into a single super-vertex. The variable *in* records the chosen path entering each node. However, for every super-vertex, its children's entering edges form a cycle, which is not the required final result. Meanwhile, the *in* edge of super-vertexes records the true minimum edge in MDST. Therefore, in the expansion phase, the fake *in*s should be transfered into real ones.

1. Initialize an empty set *R* which will be used to record unexpanded super-vertexes.
2. Dismantle root node. Find root's parent *u*. Traverse all the children of *u*, set their parent to null and put those who is(are) super-vertex(es) into set *R*, which means it need to be further dismantled.
3. Extract a super-vertex *c* from *R*. We know the cheapest edge entering *c* by checking variable *c.in*, however, what we need in the final MDST is the entering edges of normal vertex. Therefore we set *v.in* to *(u, v)*. Then dismantle *v*.

If we simulate the expansion phase by hand, when expanding a super-vertex *s* which has another super-vertex *c* as one of its children, we should firstly set *c.in* as *(u, v)*, then expand *c* until the original vertex's in is set as *(u, v)*. However, the vertexes the edges connecting in this algorithm implementation are all original vertexes, so we can directly pass down *(u, v)* to the original vertex.

2.3 Output Result

Currently the chosen edges in MDST are all recorded in nodes' variable *in*. Note that we added some sufficiently large weight edges, so if the original graph does not have a MDST, some of the *in*s will be those added fake edges. As a result, we can tell whether the original graph has a MDST by checking the *in* edge's value.

3 Underlying Data Structures

3.1 Lazy Label

Note that when a vertex decides its cheapest entering edge with weight *w*, every edge entering it should minus their weight by *w* in order to get their relative weight. However, update all the edge's weight is too time consumptive, so we use lazy label to promote efficiency.

> Lazy propagation is a technique used to optimize range updates in a segment tree. It works by postponing the actual updates to the nodes until they are needed, and storing the update information in a separate data structure called the lazy propagation array or lazy tag. When a range query is performed on a segment that has pending updates, the updates are applied recursively to the child nodes, and the lazy tags are propagated down the tree. This way, the updates are applied only when they are necessary, and unnecessary updates are avoided. [1]

For every edge, we have two variables *weight* and *lazy*. Every time we decide the cheapest entering edge of a vertex *a*, we should update *a.Proot.weight* and *a.Proot.lazy*, which means every edge in vertex *a*'s leftist tree should minus the chosen weight.

```
a.Proot.weight -= a.in.weight;
a.Proot.lazy += a.in.weight;
```

Now *a.Proot* has correct adjusted weight value, but other edges in *a*'s leftist tree have not been updated. But this can be effectively done by implementing function pushDown() as follow. Every time we have to use the relative weight value, simply invoke pushDown() first.

```
static void pushDown(b_edge root){
    b_edge left = root.lson;
    if (left != null){
        left.weight -= root.lazy;
        left.lazy += root.lazy;
    }
    b_edge right = root.rson;
    if (right != null){
        right.weight -= root.lazy;
        right.lazy += root.lazy;
    }
    root.lazy = 0;
}
```

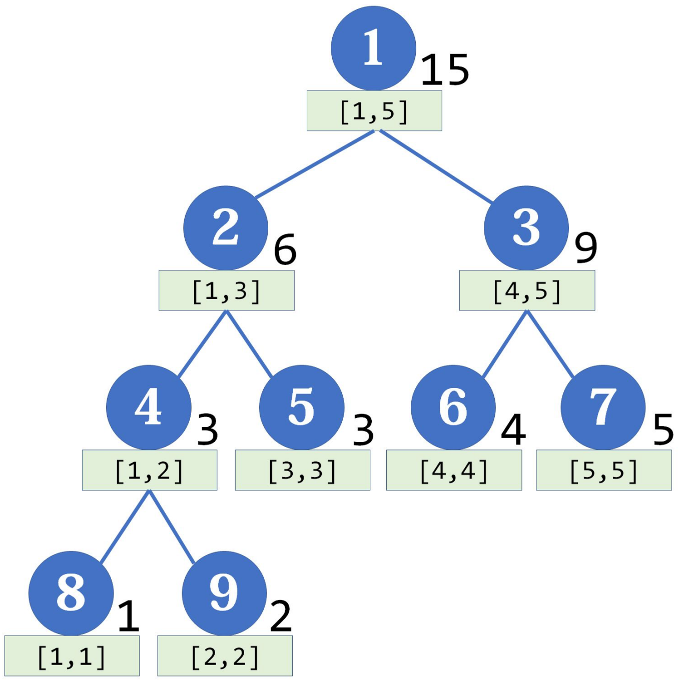

Fig 3.1, a lazy label example of segment tree, image source https://zhuanlan.zhihu.com/p/106118909. If we want to decrease every value in range[3, 4] by 2, first set the lazy label of node one to 2 then pass it down to node two, three, five and six.

3.2 Leftist Tree

Binary heap is a widely used data structure for extractMin operation. However, when it comes to the operation meld, its time complexity is O(n), because merging two binary heaps requires inserting each element of one heap into the other heap one by one, which is too costly. So we use leftist tree to optimize the time complexity.

> Leftist tree, also known as leftist heap, is a binary tree data structure that satisfies the leftist property, which states that the rank of the left child of any node is greater than or equal to the rank of its right child. Here, the rank of a node is defined as the length of the shortest path to a null node, or the height of the subtree rooted at the node.
>
> The leftist property allows for efficient merging of two leftist trees, where the root nodes of the two trees are compared based on their ranks, and the tree with the smaller rank becomes the left child of the other tree's root. This ensures that the resulting tree also satisfies the leftist property. [2]

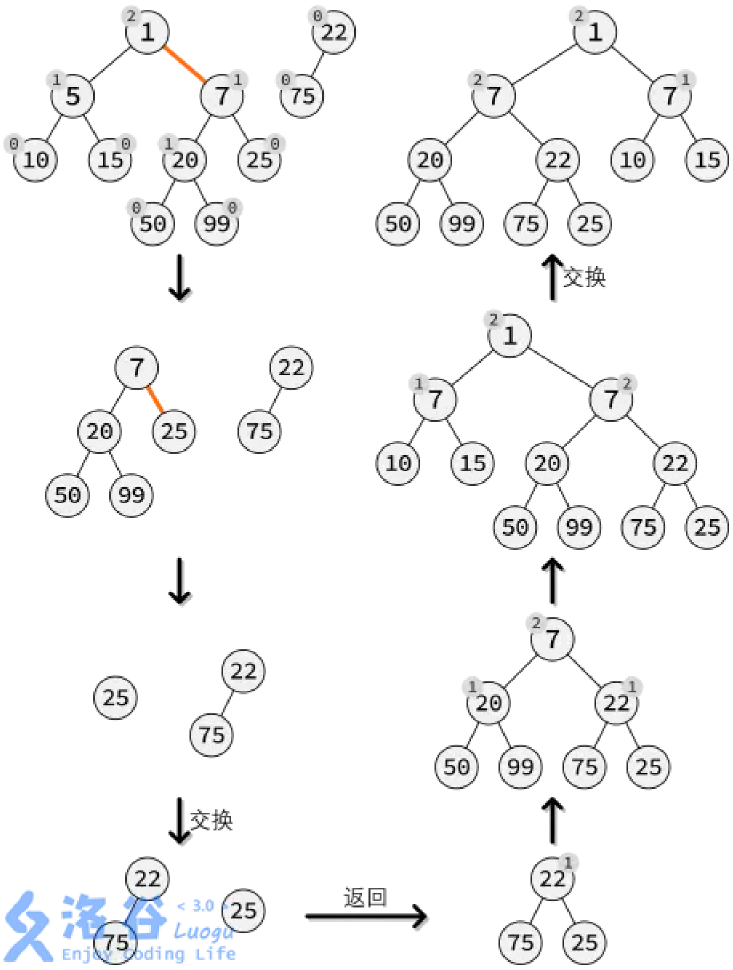

Fig 3.2, how do two leftist tree meld, image source https://www.luogu.com.cn/blog/cytus/ke-bing-dui-zhi-zuo-pian-shu.

As figure3.2 shows above, function merge(node e1, node e2) is implemented recursively. If any of e1 or e2 is null, simply return the other one. Then we compare the value of their root nodes. If e1's value is larger than e2, we merge e1.rson and e2. Otherwise, we merge e2's rson with e1. Note that before comparison, we should first call pushDown() to get the relative weight values. After merging, we should adjust their dist value in order to maintain its property of leftist.

```
static b_edge merge(b_edge e1, b_edge e2){
    if (e1 == null) return e2;
    if (e2 == null) return e1;
    if (e1.weight < e2.weight
    || (e1.weight == e2.weight && e1.inNode.index < e2.inNode.index)){
        b_edge et = e1.rson;
        pushDown(e1);
        e1.rson = merge(et, e2);
        if (e1.lson == null) {
            e1.lson = e1.rson;
            e1.rson = null;
        }else if (e1.rson.dist > e1.lson.dist){
            et = e1.rson;
            e1.rson = e1.lson;
            e1.lson = et;
        }
        if (e1.rson == null) {
            e1.dist = 0;
        }else{
            e1.dist = e1.rson.dist + 1;
        }
        return e1;
    }else{
        b_edge et = e2.rson;
        pushDown(e2);
        e2.rson = merge(e1, et);
        if (e2.lson == null) {
            e2.lson = e2.rson;
            e2.rson = null;
        }else if (e2.rson.dist > e2.lson.dist){
            et = e2.rson;
            e2.rson = e2.lson;
            e2.lson = et;
        }
        if (e2.rson == null) {
            e2.dist = 0;
        }else{
            e2.dist = e2.rson.dist + 1;
        }
        return e2;
    }
}
```

3.3 Union Find

In the contraction phase, we contract every node on cycle path into a super-vertex. In fact, this is done by setting those vertexes to a common parent vertex. After the contraction phase, we actually form a tree which every in-node is a super-vertex while every leaf node is normal vertex.

There are times when we want to find the parent of a vertex *v*. However, *v.parent* only contains its direct relative, so we have to find parent's parent until parent is null. Here we use union find and path compression to decrease time complexity from O(n^2) to O(n).

> Union-find is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It represents each subset as a tree, where each node in the tree represents an element in the subset. The tree has a root node, which represents the representative or the canonical element of the subset. Each non-root node in the tree has a parent pointer that points to its parent node, and the root node has a self-loop pointer that points to itself.
>
> The find operation finds the root of the tree to which a given element belongs by following the parent pointers until it reaches the root node. The find operation can also perform path compression, which optimizes future find operations by making the tree flat. [3]

Initially *u.ans* = *u.parent*. In path compression, every time we find node *u*'s oldest ancestor, we record it in *u.ans*. Then next time we want to find it, simply call *u.ans*.

```java
static b_node find(b_node u){
    b_node v = u;
    while (u.ans != null){
        u = u.ans;
    }
    while (v.ans != null){
        b_node t = v.ans;
        v.ans = u;
        v = t;
    }
    return u;
}
```

4 Java Program and Complexity Analysis

4. Write your pseudocode or real program for the O(m log n) algorithm and then analyze its time complexity. Please be precise, e.g., specify the time complexity for each crucial step in your code.

4.1 Description

n = number of vertexes, m = number of edges of the original graph.

- Input vertexes and edges information O(m+n).
- Preparation phase adding edges O(n). After that, #vertexes = (n + 1), #edges = (m + n + n).
- Contraction phase O((m+2n)log(m+2n)). while loop repeat (m+2n) times, each loop contains the operations below.
    - merge(). O(log(m + 2n)) because in total we have (m + 2n) edges.
    - pushDown(). O(m+2n) because in the worst case all the edge weight will be updated by pushDown.
    - find(). O(n) because in total we have (n+1) vertexes.
- Expand phase O(nlogn). while loop repeat O(nlogn) times because at most there will be ((1+n/2) * logn)/2 super-vertexes. dismantle also O(nlogn).
- Output phase O(n). Traverse vertexes.

4.2 Code

```java
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

public class B_MDST {
    public static ArrayList<b_node> R = new ArrayList<>();
    public static void main(String[] args) throws IOException{
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();
        int m = input.nextInt();
        b_node[] nodes = new b_node[n+1];
        for (int i = 0; i < n; i++){
            b_node nd = new b_node(i);
            nodes[i] = nd;
        }
        int indexCounter = n;
        for (int i = 0; i < m; i++){
            int s = input.nextInt();
            int t = input.nextInt();
            int weight = input.nextInt();
            b_edge eg = new b_edge(nodes[s], nodes[t], weight);
            nodes[t].Proot = merge(nodes[t].Proot, eg);
        }
        b_node root = new b_node(indexCounter++);
        nodes[n] = root;
        for (int i = 0; i < n; i++){
            b_edge eg = new b_edge(root, nodes[i], 1000000);
            nodes[i].Proot = merge(nodes[i].Proot, eg);
        }
        for (int i = 0; i < n; i++){
            b_edge eg = new b_edge(nodes[i], nodes[i+1], 1000000);
            nodes[i+1].Proot = merge(nodes[i+1].Proot, eg);
        }
        // contract
        b_node a = nodes[n];
        int trueRootIndex = -1;
        while (a.Proot != null){
            b_edge uv = a.Proot;
            pushDown(uv);
            a.Proot = merge(a.Proot.lson, a.Proot.rson);
            while(find(uv.inNode) == find(uv.outNode)){ // 去自环
                uv = a.Proot;
                if (uv == null) break;
                pushDown(uv);
                a.Proot = merge(a.Proot.lson, a.Proot.rson);
            }
            if (uv == null) break;
            b_node u = uv.outNode;
            b_node b = find(u);
            if (a != b){
                a.in = uv;
                a.prev = b;
                if (b == root) {
                    trueRootIndex = a.in.inNode.index;
                }
                if (b.in == null){ // path extended
                    a = b;
                }else{ // new cycle formed
                    b_node c = new b_node(indexCounter++);
                    while(a.parent == null){
                        a.parent = c; // super-vertex
                        a.ans = c;
                        if (a.Proot != null){
                            a.Proot.lazy += a.in.weight;
                            a.Proot.weight -= a.in.weight;
                        }
                        c.children.add(a);
                        // meld
                        c.Proot = merge(c.Proot, a.Proot);
                        a = a.prev;
                        if (a.parent != null) a = find(a);
                        if (a == c) break;
                    }
                }
            }
        }
        // expand
        dismantle(root);
        while(R.size() > 0){
            b_node c = R.get(R.size()-1);
            R.remove(R.size()-1);
            b_edge uv = c.in;
            b_node v = uv.inNode;
            v.in = uv;
            dismantle(v);
        }
        // calculate result
        int result = 0;
        boolean flag = false;
        for (int i = 0; i < n; i++){
            if (i == trueRootIndex) continue;
            if (nodes[i].in.originalWeight > 10000){
                System.out.println("impossible");
                flag = true;
                break;
            }
            if (nodes[i].prev != root) result += nodes[i].in.originalWeight;
        }
        if (!flag) System.out.printf("%d %d\n", result, trueRootIndex);
    }
    static b_node find(b_node u){
        b_node v = u;
        while (u.ans != null){
            u = u.ans;
        }
        while (v.ans != null){
            b_node t = v.ans;
            v.ans = u;
            v = t;
        }
        return u;
    }
    static void pushDown(b_edge root){
        b_edge left = root.lson;
        if (left != null){
            left.weight -= root.lazy; //lazy是个正数
            left.lazy += root.lazy;
        }
        b_edge right = root.rson;
        if (right != null){
            right.weight -= root.lazy;
            right.lazy += root.lazy;
        }
        root.lazy = 0;
    }
    static b_edge merge(b_edge e1, b_edge e2){
        if (e1 == null) return e2;
        if (e2 == null) return e1;
        if (e1.weight < e2.weight
        || (e1.weight == e2.weight && e1.inNode.index < e2.inNode.index)){
            b_edge et = e1.rson;
            pushDown(e1);
            e1.rson = merge(et, e2);
            if (e1.lson == null) {
                e1.lson = e1.rson;
                e1.rson = null;
            }else if (e1.rson.dist > e1.lson.dist){
                et = e1.rson;
                e1.rson = e1.lson;
                e1.lson = et;
            }
            if (e1.rson == null) {
                e1.dist = 0;
            }else{
                e1.dist = e1.rson.dist + 1;
            }
            return e1;
        }else{
            b_edge et = e2.rson;
            pushDown(e2);
            e2.rson = merge(e1, et);
            if (e2.lson == null) {
                e2.lson = e2.rson;
                e2.rson = null;
            }else if (e2.rson.dist > e2.lson.dist){
                et = e2.rson;
                e2.rson = e2.lson;
                e2.lson = et;
            }
            if (e2.rson == null) {
                e2.dist = 0;
            }else{
                e2.dist = e2.rson.dist + 1;
            }
            return e2;
        }
    }
    static void dismantle(b_node u){
        while (u.parent != null){
            b_node t = u;
            u = u.parent;
            for (int i = 0; i < u.children.size(); i++){
                b_node v = u.children.get(i);
                v.parent = null;
                if (v == t) continue;
                if (v.children.size() > 0){
                    R.add(v);
                }
            }
        }
    }
}
class b_node{
    int index;
    b_edge in; // initially null
    b_node prev; // null
    b_node parent; // null
    b_node ans;
    ArrayList<b_node> children = new ArrayList<>(); // empty
    b_edge Proot;
    b_node(int index){
        this.index = index;
        children = new ArrayList<>();
    }
}

class b_edge{
    int originalWeight;
    b_node outNode;
    b_node inNode;
    int weight;
    b_edge lson;
    b_edge rson;
    int dist;
    int lazy;
    b_edge(b_node outNode, b_node inNode, int weight){
        this.outNode = outNode;
        this.inNode = inNode;
        this.weight = weight;
        this.originalWeight = weight;
        this.dist = 0;
    }
}
```

---

1. chatGPT ↩
2. chatGPT ↩
3. chatGPT ↩