(foreword: Maybe there are three typos in the note. First one is in the pseudo code of Contract. A statement of a<-c at the bottom of the situation of new cycle formed is omitted, because the super-node c should be the next vertex to process in the main while. Next one is in the Figure6, in[θ] should be (ζ , θ), because the only entering edge of θ starts at ζ instead of ι . The last one is in Figure5, in the pseudo code of weight, its return value should be w instead of vertex u.)

P1. Before writing, think how the lecture notes present the algorithm, e.g., why 5.1 is needed before 5.2 and 5.3.

A:

First, it reviews the idea of the Zhu Liu's algorithm which is maintaining a set F of cheapest entering edges, and checking if F is acyclic. That's quite time-consuming, which need Ω(n) times contractions.

So, how do we speed up contractions? Here, Tarjan spotted that the set F of cheapest entering edges always forms a path. Can we use this property to design a faster algorithm to do contractions? The answer is yes. Tarjan focus on this path and comes out an algorithm to contract vertices within O(mlogn).

Then, how do we implement it? Section 5.1 gives a brief introduction. We assume, for simplicity, that G is strongly connected.(If not, we use O(n) to add edges making it connected). And then for generality, we ignore the identity of root because it can allow us keep contracting all of the vertices into a single one and allow us to find MDST from any root. It means this algorithm gives a general solution to find MDST, and in practice, you can modify it according to your intention.

What data structures do we need and what skills should we master? It will include union-find and priority queue, and use the trick of keeping a contraction tree, meld priority queue and so on.

Then we can use O(n) to expand the contraction, thus a faster algorithm to get the MDST within O(mlogn) is designed.

What above is the mission of section 5.1. Based on section 5.1, section 5.2 focuses on the details how the contraction works and section 5.3 focuses on the details how the expansion works.

That's the way how the lecture notes present the algorithm.

(Detailed description of the implementation of contraction and expansion will be introduced in problem2)

P2. Describe the algorithm implementation in your own words. Please do not copy-paste those from the lecture notes, because then you will learn nothing. Here the goal is to beat the lecture notes and explain the algorithm in a more clear way.

A:

The whole implementation of the algorithm to find MDST is shown below.

1. If P[u] of the arbitrary vertex u is null, that means u is an isolated vertex in the G. So, we can't find a MDST.
2. If P[u] of all of the arbitrary vertex u is not null, for each arbitrary vertex u chosen
    a. we first use extract-min(P[u]) to get the cheapest non-self-loop entering edge of u (P[u] is a leftist tree to maintain all of the in- edge of the u, and we need it to give us the top element in the heap through extract-min(P[u])).
    b.then we use find(u) to its ancestor b (who is a greatest super-node containing u):
        I.here we consider the situation a≠b (once a=b it means , (u,v) is a self-loop). we update the in[a] with (u,v) , then set the prev[a] with b.
        II.If in[u]=null (equal to in[b]=null, it means u is not in the the path so we extend the path to b. Else, it means b is already in the path we constructed before, so there is a cycle in the path which we need to contract it. The way how we contract the cycle is shown below:
            1. we allocate and initialize a new vertex c by init-vertex(c).
            2. then we start from the vertex a , traverse along the prev[a] and terminate back at a, update the set parent[a] as c, const[a] with -w[in[a]], insert a into the children[c], meld P[c] and P[a], it means we let all of the unpicked entering edge of vertices contracted to c terminate at c.
            3.let a updated by c.
3. Then it goes back to the while loop, until all of the vertices are contracted into one single node. It's done.
4. After the work of contracting V into a single vertex, then we need to expand it.
    a. It starts at root to set parent of siblings including it self to be null and move upward to the root of contraction tree. Along the path, it will add non-leaf vertices into R, which means this vertex is a super node and need unpacking later.
    b. expand all of the super node in R and set the chosen entering edge back to the corresponding v, and record it.
5. Continuous work to process R until it's empty, then we get a set of edges which compose the MDST rooted at r.

P3.Please fill out the missing details about the underlying data structures, e.g., how to meld heaps?

A:

1. How to implement insert(P[v], (u,v))?

We use leftist tree to implement P[v], assuming n=size of P[v], then we use the insertion of leftist tree to implement inset(P[v], (u,v)) within O(logn).

2. How to implement insert(R ,v) and extract(R)?

We use stack to implement R, then implement insert(R, v) by stack.push() and implement extract(R) by stack.pop(). Queue can also be used to construct R.

3. How to implement extract-min(P[a])?

Assuming n=size of P[a], for one step, we use the delete-min of leftist tree to implement it within O(logn) to get the top edge of P[a], and check whether it's a self loop, if it's a self loop then do delete-min again until we get a non-self-loop edge.

4. How to implement meld(P[c],P[a])?

Assuming n= max{size of P[c], size of P[a]}, we use the meld of leftist tree to implement it within O(logn).

5. How to implement vertex()?

Allocate a new vertex and run init-vertex(u) to initialize it then return this vertex.

6. How to implement inserti(Children[c], a)?

Using arraylist is ok.

7. What's the purpose of function weight(u,v)?

To render you using this function and the MDST we get to calculate its minimum sum of weight of each edge in MDST.

(The proof of the time complexity of operations of leftist tree referred above will be given at the bottom of the article)

P4. 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.

A: DMST(G(V, E), r)

```
// initialize
foreach u∈V do
        in[u]←null
        const[u]←0
        prev[u]←null
        parent[u]←null
        children[u]←null
        P[u]←priority-queue()
foreach (u,v)∈E do
        insert(P[v], (u,v))

// contraction
a ← arbitrary vertex
while P[a] ≠ ∅ do
    (u, v) ← extract-min(P[a])
    b ← find(u)
    if a ≠ b then
        in[a] ← (u, v)
        prev[a] ← b
        if in[u] = null then // Path extended
        a ← b
```

```
                else // New cycle formed
                    c ← vertex()
                    while parent[a] = null do
                    parent[a] ← c
                    const[a] ← −w[in[a]]
                    insert(children[c], a)
                    P[c] ← meld(P[c], P[a])
                    a ← prev[a]
                a←c // revision of the typo

        // expand
        R ← ∅
        dismantle(r)
        while R ≠ ∅ do
            c ← extract(R)
            (u, v) ← in[c]
            in[v] ← (u, v)
            dismantle(v)

return {in[u] | u ∈ V \ {r}}

dismantle(u)
    while parent[u] 6 = null do
        u ← parent[u]
        foreach v ∈ children[u] do
            parent[v] ← null
            if children[v] 6 = null then
            insert(R, v)


find(u)
    while parent[u] 6 = null do
            u ← parent[u]
    return u
```

Time complexity: Assuming n=|V| and m=|E|. Suppose there is no duplicate edges and thus $m<=n+2*C(n,2)=n^2$ (n for each vertex containing a self loop and 2*C(n,2) for any pairs of vertices contain two opposite directed edges) .The initialization with two loop costs O(max{n,m}). Then in the function extract-min, it should find the cheapest non-self-loop entering edge, so in the worst case, last step is contracting n vertices into a super node, and $m=2*C(n,2)=n^2-n$, which means any pairs of vertices contain two opposite directed edges. In contraction, there will form m-n self loops to new super node.(n edges used to form a loop of the n vertices and then m-n edges left form self loops to the new super node).Thus, in the next contraction of super node. It will

have to delete m-n times top edge of the heap, which costs $\sum_1^{m-n} O(log(m-n-i)) = O(mlogm)$. Next, in the contraction phase, the main cost is at the operation of melding leftist trees. As the times of contracting vertices will not exceed n-1, suppose every time there exists a big heap of size m (impossible!!!) needing to be meld, then the time cost will not bigger than O(nlogm). In the expansion phase, we will walk along the contraction tree and unpack the super node, each time we will acquire an edge (u,v) to be added into the set of return value. So, it costs O(n). Then above all, the total time complexity is O(mlogm). (b.t.w. as $m<=n^2$, so O(m) = O(n), the result of time complexity can also be rewritten into O(mlogn))

Proof of the time complexity of operations of leftist tree mentioned in P3:

Some properties of leftist tree:
a. It's a min-heap
b. dist(left son)>=dist(right son) (dist(u) is the minimum distance between node u with two children and another node with a single child or none)
c. a leftist tree with n nodes, the max distance within this tree will not exceed log(n+1)-1

Proof of property c:
When the distance of leftist tree is a fixed value k, it has the minimum number of vertices when dist[left son]=dist[right son] for each vertex. Actually, it will be a complete full binary tree. So, the number of vertices n is less than 2^(k+1）-1， then we get k<=log(n+1)-1.

1. meld:
   The process of merging leftist tree x and y operations is as follows:
   (1) Take the smaller value of x and y as the root after merging. Assuming that the value of x is smaller, the root of x after merging is the root of the new tree, and the left sub-tree of x is the left sub-tree of the new tree
   (2) Recursively merge the right sub-trees of x and y, and if the merge does not satisfy the leftist property (dist (left son)<dist [right son]), swap the left and right sons, that is, swap (left son, right son);
   (3) When the right sub-tree of x or y is empty, the merge algorithm ends.

   Time complexity:
   For each level of recursion, the distance of a heap will be reduced by 1. For each recursion, the decomposed right sub-tree will be merged. As the max distance of a tree with n nodes will not exceed log(n+1)-1, so merging two trees with size n and m respectively will cost O(max(logn,logm))

2. extract-min:

After deleting the root node, merge the left and right sub-trees to obtain a new tree.

Time complexity:

After deleting the root, the max distance of left tree less than log(n+1)-1, the max distance of right tree is log(n+1)-2, so according to the meld operation above, the time complexity is O(logn)

3. insert:

we meld the origin tree with n nodes and a tree of only one node to implement inserting a node.So, according to the meld operation above, the time complexity is O(logn).