CS216 Assignment 2

Presentation Organization

> Before writing, think how the lecture notes present the algorithm.

In the previous chapters of the lecture notes, the author has introduced Chu-Liu/Edmonds' algorithm, the basic greedy algorithm for finding a DMST in $O(mn)$. And the optimized algorithm is inspired by Chu-Liu algorithm, so it is necessary to show the connection between the original algorithm and the optimized one, in other words, the bottleneck of the basic algorithm and the key of breakthrough to obtain the better one.

Quite like an abstract for a paper, it also give a high-level description (general idea, phases etc.) about the algorithm and tells the time complexity the algorithm can achieve, and then divide the rest part of the lecture notes correspondingly. Readers can think by themselves with the hints and go to the following details top-down later, which is supposed to be more comprehensible.

Algorithm Analysis

> Describe the algorithm implementation.

General Idea

In Chu-Liu/Edmonds' algorithm, the most important steps to find the DMST are:

1. Selecting the smallest in-edge for every vertex $O(m)$
2. Contracting cycles in $O(n)$
3. Updating edge values and rebuild the graph in $O(m)$

As in the worst case we need to go through these steps for near n times, the whole time complexity is $O(m\,n)$. Review the most costly operations: select the smallest in-edge and update edge values for vertices in cycles. Both of them are in $O(m)$, and it is easy to think of using a data structure to accelerate them. Obviously, mergeable heap can fit our requirements (here we use leftist tree): getting a minimum from a set in $O(1)$, merging two set in $O(log\,m)$, deleting a element from a set in $O(log\,m)$ and with the technique of lazy tag, update the values for the whole set in $O(1)$. So now the point is how to make use of it.

Similar to the procedure of Chu-Liu/Edmonds' algorithm, we contract the cycles we find on our way into super nodes. But this time we use mergeable heaps to maintain the in-edges for each vertices. So after choosing the smallest in-edge for a node, we can use lazy tag to update the whole edge set. And when we come across a cycle, we can create a new super node to represent the original vertices on the cycle and merge up their edge sets in $O(log\,m)$. After exploring every edge, the contraction phase ends.

If we only ends up here, we can take root node as a special case and count the edge value along the way then get the answer in $O(m\,log\,m)$ for a given root. However, we notice that the process of contracting cycles do not care which one is the root. And because the cycle path is shaped via selecting the minimal in-edges, if we select an arbitrary vertex as the entry to traverse the cycle, choose the cycle path is always optimal. So if the graph is strongly connected, we can contract it into a single node. And we can expand the cycles to get the DMST for any given root in $O(n)$. To achieve this, we can add n in advance. If we select these supporting edges in expansion phase, then we can tell there is no DMST for this specific root.

Though till now the time complexity we get is literally $O(m\,log\,m)$, here is a small additional trick to lessen it further. Noticed that in a graph with n vertices, each vertex can only have $n-1$ out-edge towards all other vertices if there are no parallel edges. So if we find $m > n \times (n-1)$, we can use adjacent matrix to reduce $||E||$ to n^2, then with the diminished edge set E', we now have $O(m + 2n^2 log\,n)$, or the more widely used representation $O(m\,log\,n)$, for the whole algorithm.

Description by Phases

Initialization

1. Add n supporting edges (from each i to $i\%n+1$ with the value of INF) to ensure strong connectivity.
2. Build heaps of in-edges for each vertex. (Here we can use a technique to build heaps by merging in $O(m)$ rather than by inserting in $O(m\,log\,m)$)

Contraction

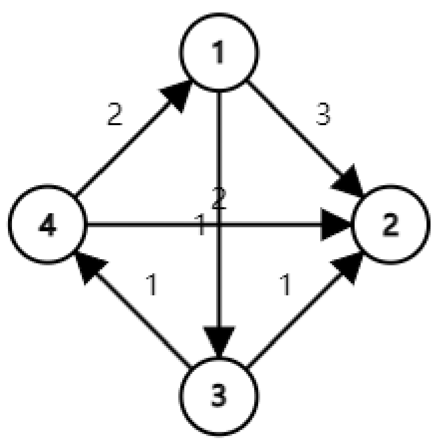
1. Select an arbitrary vertex as a starter.
2. Mark the node s to show it is visited.
3. Extract the minimal edge from its heap (omit self-loop edges).
4. Get the super node t of the node on the other side of this edge.
   1. If there is no edge left ($t == s$), the whole graph has been contracted into one single node. Contraction phase ends.
   2. If t has not been visited, no cycle is found but the path extends. Let $s = t$, and go back to 2.
   3. If t has been visited, a cycle is found. Record the cycle, construct a super node and maintain data structures. Then let $s = new\ super\ node$, and go back to 2.

Expansion

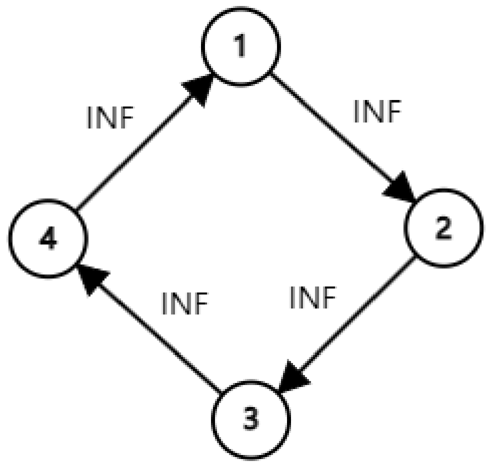
1. Provide the direction information: expand from real root to the biggest cycle node (root of the contraction tree).
2. Expand cycles along the way and calculate the answer.
3.

Example

Nothing could be better than a concrete example. Let us take this simple graph as input on the scenario ($root = 1$):



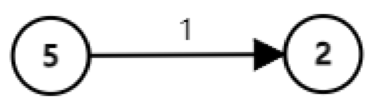
At first, we add 4 additional edges to make sure the whole graph is strongly connected:

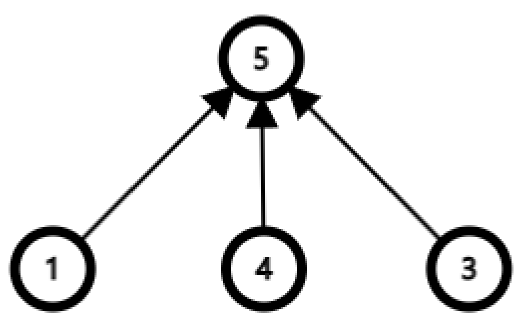


Then we start to contract the graph (from vertex 1):

1.
```
1: (4, 1, 2)
4: (3, 4, 1)
3: (1, 3, 1)
```

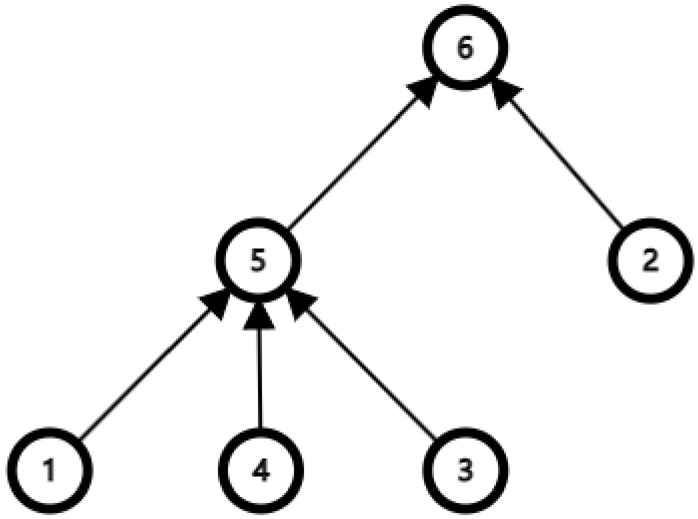
Contract 1, 4, 3 into a new super node 5.





2.
```
5: (2, 5(3), INF)
2: (5, 2, 1)
```

Contract 2, 5 into a new super node 6.



Here we let 1 as the root, and start expand from 1 to 6 on the contraction tree:

1. Expand the cycle represented by $5:(1,4,3)$, the edge (1, 3, 1), (3, 4, 1) are chosen.
2. Expand the cycle represented by $6:(5,2)$, the edge (5, 2, 1) is chosen.

Then the answer is $1+1+1 \ = \ 3$.

If we set 2 as root (Though we can directly end when meet virtual edge, but I like to show the whole process to show the iteration of expansion functions' invoking):

1. Expand the cycle represented by $6:(5,2)$, the edge (2, 5, INF) is chosen.
2. Expand the other nodes in this cycle. Only 5 here.

   Expand the cycle represented by $5:(1,4,3)$, the edge (3, 4, 1), (4, 1, 2) are chosen.

Then the answer is $INF+1+2=INF$, no such a DMST that rooted at 2.

Data Structure Elaboration

> Please fill out the missing details about the underlying data structures.

Here I use leftist tree to implement mergeable heap.

Leftist Tree

Structure

`(Comparable Element), dist, lc, rc`

Properties

- $this.element < lc.element \&\& this.element < rc.element$
- $lc.dist \geq rc.dist$
- $this.dist = rc.dist + 1$
- $leaf.dist = 0$

Operations

The most vitally important operation for Leftist is `merge()`, all other operations can be achieve by invoke `merge()` with some additional modifications.

When merging two leftist tree, we set the one with the larger element as the root, and recursively merge the other with its right child. And finally check children's `dist` and update `dist` for itself to remain "leftish". Though the depth of a leftist tree of size n varies from $log\, n$ (binary tree) to n (chain), its right child's depth is certainly be equal or less than $\lfloor \frac{n}{2} \rfloor$. When merging, we only recursively do something with the right child, so the time complexity is $O(log\, n)$.

```cpp
struct Node  // for leftist tree
{
    Edge *e;
    int dist, lazy;
    Node *lc, *rc;

    Node(Edge *e):
        e(e), dist(0), lazy(0), lc(nullptr), rc(nullptr) {};

    void push() // push down the lazy tag
    {
        if (lc)  lc->lazy += lazy;
        if (rc)  rc->lazy += lazy;
        e->w += lazy;
        lazy = 0;
    }
};

// merge two leftist tree
Node *merge(Node *x, Node *y)
{
    if (!y)  return x;
    if (!x)  return y;
    if (x->e->w+x->lazy > y->e->w+y->lazy)
        swap(x, y);
    x->push();
    x->rc = merge(x->rc, y);
    if (!x->lc || x->lc->dist < x->rc->dist)
        swap(x->lc, x->rc);
    if (x->rc)  x->dist = x->rc->dist+1;
    else x->dist = 0;
    return x;
```

```
}

// get the minimal edge and delete the root
Edge *extract(Node *&x)
{
    Edge *ret = x->e;
    x->push();
    x = merge(x->lc, x->rc);
    return ret;
}
```

$O(m)$ heap building is also worth a mention. (covered in DSAA, though will not change the general time complexity here)

```
// build leftist trees in O(m)
    for (int i = 1; i <= n; ++i)
    {
        queue<Node *>q;
        for (int j = 0; j < edge[i].size(); ++j)
            q.push(new Node(edge[i][j]));
        while (q.size() > 1)
        {
            Node *a, *b;
            a = q.front();
            q.pop();
            b = q.front();
            q.pop();
            q.push(merge(a, b));
        }
        tree[i] = q.front();
    }
```

Code Implementation

> Write your pseudocode or real program for the $O(mlogn)$ algorithm and then analyze its time complexity.

The time complexity analysis has been declared in previous sections.

Detail Demonstration

Libraries, Namespaces, Global Variables and Functions

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

```

```cpp
typedef long long ll;

const int N, M;
const ll INF = 0x3f3f3f3f3f3f3f3f;

struct Edge
{
    int u, v;
    ll w, w0;

    Edge(int u, int v, ll w);
};

struct UnionFind
{
    int fa[N<<1];

    int find(int x);
    int operator[](int x);
};

struct Node
{
    Edge *e;
    int dist, lazy;
    Node *lc, *rc;

    Node(Edge *e);
    void push();
};

Node *merge(Node *x, Node *y);
Edge *extract(Node *&x);

int n, m, fa[N<<1], nxt[N<<1];
Edge *in[N<<1];
UnionFind id;
Node *tree[N<<1];
vector<Edge *> edge[N];
bool vis[N<<1];

void contract();
ll expand(int x, int r);
ll expand_cycle(int x);
int main();
```

Initialization

```
// ensure the whole graph is strongly connected
for (int i = 1; i < n; ++i)
    edge[i+1].push_back(new Edge(i, i+1, INF));
```

```
// build leftist trees in O(m)
for (int i = 1; i <= n; ++i)
{
    queue<Node *>q;
    for (int j = 0; j < edge[i].size(); ++j)
        q.push(new Node(edge[i][j]));
    while (q.size() > 1)
    {
        Node *a, *b;
        a = q.front();
        q.pop();
        b = q.front();
        q.pop();
        q.push(merge(a, b));
    }
    tree[i] = q.front();
}
```

Contraction

```
// start from an arbitrary vertex
int s, t = 1, p;
while (tree[t])
{
    vis[t] = true;
    s = t;
    do
    {
        in[t] = extract(tree[t]);  // get the minimal edge
        t = id[in[t]->u];  // the super node u belongs to
    } while (t == s && tree[t]);  // till the path extends or no edge
    if (t == s)  break;  // the whole graph is contracted
    if (!vis[t])  continue;  // no cycle found
    // contract the cycle, update id[], fa[] and the lazy tag
    t = s;
    ++n;
    while (t != n)  // till all are merged the new super node
    {
        id.fa[t] = fa[t] = n;
        if (tree[t])  tree[t]->lazy -= in[t]->w;
        tree[n] = merge(tree[n], tree[t]);
        p = id[in[t]->u];  // the super node u belongs to
```

```
        nxt[p == n ? s : p] = t;  // record the cycle
        t = p;
    }
}
```

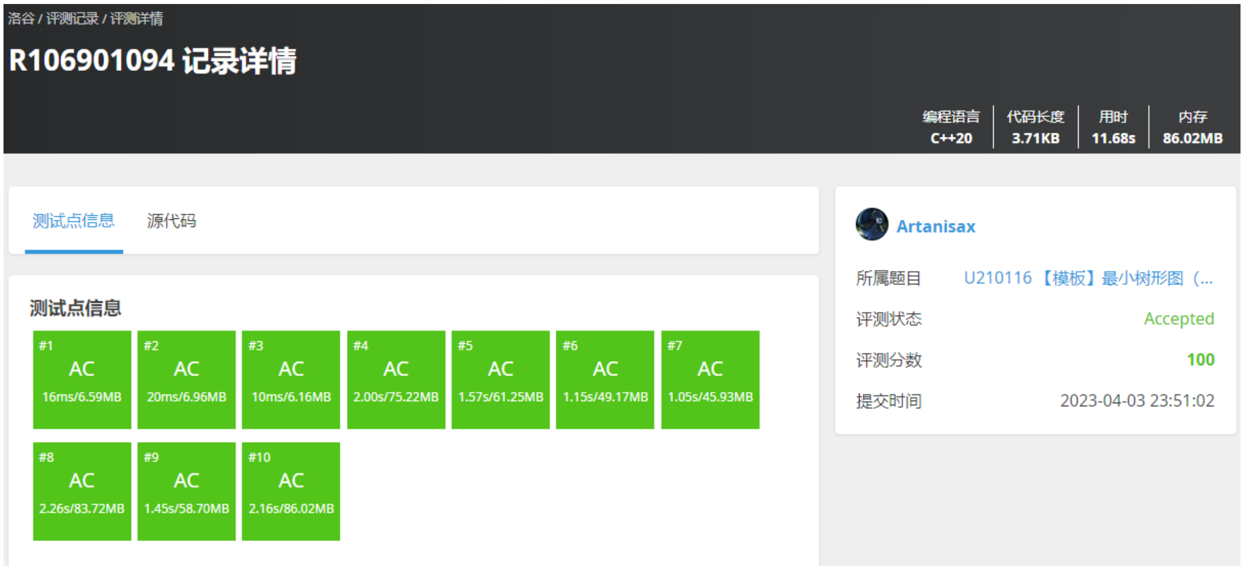
Expansion

```
// expand x as a super node
ll expand_cycle(int x)
{
    cerr << "cycle: " << x << '\n';
    ll ret = 0;
    // traverse the cycle (nodes of the same father)
    for (int t = nxt[x]; t != x; t = nxt[t])
        if (in[t]->w0 == INF)
            return INF;
        else
            ret += expand(in[t]->v, t)+in[t]->w0;  // expand down
    return ret;
}

// from x to r in the contraction tree, O(n) in total
ll expand(int x, int r)
{
    ll ret = 0;
    while (x != r)
    {
        cerr << "expand: " << x << ' ' << r << '\n';
        ret += expand_cycle(x);
        if (ret >= INF)  return INF;
        x = fa[x];  // expand up
    }
    return ret;
}

int main()
{
    ...
    ll ans = expand(r, n);  // enter from the true root
    ...
}
```

Record of Passing U210116

数据范围

对于所有数据，$1 \leq u, v \leq n \leq 10^5, 1 \leq m \leq 10^6, 1 \leq w \leq 10^9$



Full Code and Comments

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

typedef long long ll;

const int N = 1e5+5, M = 1e6+5;
const ll INF = 0x3f3f3f3f3f3f3f3f;

struct Edge
{
    int u, v;
    ll w, w0;

    Edge(int u, int v, ll w):
        u(u), v(v), w(w), w0(w) {};
};

struct UnionFind
{
    int fa[N<<1];

```

```
    int find(int x)
    { return fa[x] ? fa[x] = find(fa[x]) : x; }

    int operator[](int x)
    { return find(x); }
};

struct Node  // for leftist tree
{
    Edge *e;
    int dist, lazy;
    Node *lc, *rc;

    Node(Edge *e):
        e(e), dist(0), lazy(0), lc(nullptr), rc(nullptr) {};

    void push() // push down the lazy tag
    {
        if (lc)  lc->lazy += lazy;
        if (rc)  rc->lazy += lazy;
        e->w += lazy;
        lazy = 0;
    }
};

// merge two leftist tree
Node *merge(Node *x, Node *y)
{
    if (!y)  return x;
    if (!x)  return y;
    if (x->e->w+x->lazy > y->e->w+y->lazy)
        swap(x, y);
    x->push();
    x->rc = merge(x->rc, y);
    if (!x->lc || x->lc->dist < x->rc->dist)
        swap(x->lc, x->rc);
    if (x->rc)  x->dist = x->rc->dist+1;
    else x->dist = 0;
    return x;
}

// get the minimal edge and delete the root
Edge *extract(Node *&x)
{
    Edge *ret = x->e;
    x->push();
    x = merge(x->lc, x->rc);
    return ret;
}

// fa[] record the contraction tree
int n, m, fa[N<<1], nxt[N<<1];
```

```
Edge *in[N<<1];
UnionFind id;
Node *tree[N<<1];
vector<Edge *> edge[N];

bool vis[N<<1];
void contract()
{
    // build leftist trees in O(m)
    for (int i = 1; i <= n; ++i)
    {
        queue<Node *>q;
        for (int j = 0; j < edge[i].size(); ++j)
            q.push(new Node(edge[i][j]));
        while (q.size() > 1)
        {
            Node *a, *b;
            a = q.front();
            q.pop();
            b = q.front();
            q.pop();
            q.push(merge(a, b));
        }
        tree[i] = q.front();
    }

    // start from an arbitrary vertex, O(mlongn) in total
    int s, t = 1, p;
    while (tree[t])
    {
        vis[t] = true;
        s = t;
        do
        {
            in[t] = extract(tree[t]);  // get the minimal edge
            t = id[in[t]->u];  // the super node u belongs to
        } while (t == s && tree[t]);  // till the path extends or no edge
        if (t == s)  break;  // the whole graph is contracted
        if (!vis[t])  continue;  // no cycle found
        // contract the cycle, update id[], fa[] and the lazy tag
        t = s;
        ++n;
        while (t != n)  // till all are merged the new super node
        {
            id.fa[t] = fa[t] = n;
            if (tree[t])  tree[t]->lazy -= in[t]->w;
            tree[n] = merge(tree[n], tree[t]);
            p = id[in[t]->u];  // the super node u belongs to
            nxt[p == n ? s : p] = t;  // record the cycle
            t = p;
        }
    }
```

```
}

ll expand(int x, int r);

// expand x as a super node
ll expand_cycle(int x)
{
    cerr << "cycle: " << x << '\n';
    ll ret = 0;
    // traverse the cycle (nodes of the same father)
    for (int t = nxt[x]; t != x; t = nxt[t])
        if (in[t]->w0 == INF)
            return INF;
        else
            ret += expand(in[t]->v, t)+in[t]->w0;  // expand down
    return ret;
}

// from x to r in the contraction tree, O(n) in total
ll expand(int x, int r)
{
    ll ret = 0;
    while (x != r)
    {
        cerr << "expand: " << x << ' ' << r << '\n';
        ret += expand_cycle(x);
        if (ret >= INF)  return INF;
        x = fa[x];  // expand up
    }
    return ret;
}

int main()
{
    int r;
    cin >> n >> m >> r;
    for (int i = 1; i <= m; ++i)
    {
        int u, v, w;
        cin >> u >> v >> w;
        edge[v].push_back(new Edge(u, v, w));
    }
    // ensure the whole graph is strongly connected
    for (int i = 1; i < n; ++i)
        edge[i+1].push_back(new Edge(i, i+1, INF));
    contract();
    ll ans = expand(r, n);  // enter from the true root
    printf("%lld", ans == INF ? -1 : ans);
    return 0;
}
```

References

Lecture notes on “Analysis of Algorithms”: Directed Minimum Spanning Trees

OI WIKIL: 最小树形图