USACO JAN09 Problem 'travel' Analysis

by Long Fan

First, we need to build the shortest-path tree started from root node 1. This tree has all edges used in the shortest paths from 1 to other nodes. Since the shortest path for each node is unique, this tree is also unique and we can build it with dijkstra algorithm with heap in O(m log n).

The key to solve this problem is to notice that, in the "safe shortest path", only one edge is not from this shortest-path tree. Consider the case that we need to compute the "safe shortest path" from node 1 to node x. The blocking path will separate node 1 and node x in the shortest-path tree to prevent us from choosing the shortest-path directly. In the "safe shortest path" from 1 to x, we will first need to go from 1 to some intermediate node i along the shortest-path tree. Then we use an outside edge i->j, while j is a node connected with x(actually, x is j's ancestor) in the tree. Finally, we go from j to x along the shortest-path tree.

```
    1
   / \
 ... ...
 /     \
i       y
        |
      ----- blocked
        |
        x
       /
     ...
     /
    j
```

We travel in DFS order in this shortest-path tree and try to compute the "safe-shortest path" from bottom to top. For the leaves in the tree, with the edge to his parent blocked, the only way to reach it is by some outside edges. We use a heap to store all such edges with the path length to be the key. Then to get the answer for the leaves in the tree, we just pick the minimum element in the heap.

Now consider the non-leave node case. For a node X with A and B to be his children, we can either use an outside edge to go to x directly or through A or B. Thus, we increase the keys of all elements in A's heap by the length of edge (X,A) and the keys in B's heap by the length of edge (X,B). Then we merge A's heap and B's heap together to form the heap for X. Finally, we add the the path with outside edge directly going to X to this heap. However, we should note that some path is no longer valid, so we should delete all the path that used outside edge (U, V) that U and V's least common ancestor is X from X's heap. Following this idea, we can compute the answer for each node from bottom to top.

If we use some mergeable heap that can do deletion, insertion and merge in O(log m), the complexity of this algorithm will be O(m log m). Alternatively, if we use a standard binary heap or balanced binary tree, which do not support fast merging, we can still achieve O(m(log m)^2) time by always merging smaller structures into larger ones rather than vice versa.

Below is the solution of Canada's brilliant Hanson Wang:

```
#include <iostream>
#include <queue>
#include <set>
using namespace std;
#define For(i,a,b) for (int i = a; i < b; i++)
typedef pair<int,int> pii;

struct edge {
        int w, d;
};

vector<edge> adj[100111];

//part 1: find the shortest path tree
bool vis[100111];
int dist[100111];
void dijkstra(pair<int,int> restrict = pair<int,int>())
{
        memset(vis, 0, sizeof(vis));
        memset(dist, 0x7f, sizeof(dist));
        priority_queue<pii, vector<pii>, greater<pii> > Q;
        Q.push(make_pair(0, 0));
        dist[0] = 0;
        while (Q.size())
        {
                pii x = Q.top(); Q.pop();
                int d = x.first, v = x.second;
                if (vis[v]) continue;
                vis[v] = true;

                For (i, 0, adj[v].size())
                {
                        int w = adj[v][i].w, wt = adj[v][i].d;
                        if (d + wt < dist[w] && make_pair(v<?w, v>?w) !=
restrict)
                                dist[w] = d + wt, Q.push(make_pair(d + wt, w));
                }
        }
}

//part 2: calculate the answers
int ans[100111];
int par[100111];
int num[100111];
vector<pii> child[100111];

multiset<pii> dat[100111];
int id[100111];
int find(int x) {
        while (id[x] != x)
                x = id[x] = id[id[x]];
        return x;
}
void unite(int x, int y)
{
        x = find(x), y = find(y);
        if (x == y) return;
        if (dat[x].size() < dat[y].size())
        {
                dat[y].insert(dat[x].begin(), dat[x].end());
                dat[x].clear();
                id[x] = y;
        }
        else
        {
                dat[x].insert(dat[y].begin(), dat[y].end());
                dat[y].clear();
                id[y] = x;
        }
}

int cnum = 0;
int depth[100111];
void calculate(int v, int dep = 0)
{
        depth[v] = dep;
        //printf("v=%d depth=%d par=%d\n", v, depth[v], par[v]);
        num[v] = cnum++; //dfs number

        For (i, 0, child[v].size()) {
                calculate(child[v][i].first, dep + child[v][i].second);
                unite(v, child[v][i].first);
        }

        int fv = find(v);
        //put all the edges in
        For (i, 0, adj[v].size())
        {
                int w = adj[v][i].w, d = adj[v][i].d;
                if (w == par[v]) continue;

                //add cross / back edges
                if (!vis[w] || num[w] < num[v])
                {
                        dat[fv].insert(make_pair(depth[v] + d + dist[w], w));
                        //printf("%d-%d, %d\n", v, w, depth[v] + d + dist[w]);
                }
                else
                {
                        //remove this edge
                        multiset<pii>::iterator msi =
dat[fv].find(make_pair(depth[w] + d + dist[v], v));
                        if (msi != dat[fv].end())
                                dat[fv].erase(msi);
                }
        }
        //printf("v: %d\n", v); for (typeof dat[fv].begin() si =
dat[fv].begin(); si != dat[fv].end(); ++si) printf("%d,%d ", si->first, si-
>second); puts("");

        while (dat[fv].size())
        {
                int w = dat[fv].begin()->second;
                if (!vis[w] || num[w] < num[v]) break;
                dat[fv].erase(dat[fv].begin());
        }

        ans[v] = (dat[fv].empty() ? -1 : dat[fv].begin()->first - depth[v]);

        vis[v] = true;
}

int main()
{
        freopen("travel.in", "r", stdin);
        freopen("travel.out", "w", stdout);

        int N, M;
        scanf("%d %d", &N, &M);
        For (i, 0, M)
        {
                int a, b, c; scanf("%d %d %d", &a, &b, &c); a--, b--;
                adj[a].push_back((edge) {b, c});
                adj[b].push_back((edge) {a, c});
        }

        dijkstra();

        For (i, 0, N)
                For (j, 0, adj[i].size())
                        if (dist[i] + adj[i][j].d == dist[adj[i][j].w]) {
                                child[i].push_back(make_pair(adj[i][j].w,
adj[i][j].d));
                                par[adj[i][j].w] = i;
                        }

        For (i, 0, N) id[i] = i;

        memset(vis, 0, sizeof(vis));
        memset(ans, -1, sizeof(ans));
        calculate(0);

        For (i, 1, N)
                printf("%d\n", ans[i]);
}
```