Highways Analysis

Timothy Gao

July 2022

To reformulate the problem, we're essentially given a grid with N (10^9) rows and M ($2 \cdot 10^5$) columns. We are tasked with answering Q ($2 \cdot 10^5$) queries, each of which asks if it's possible to travel from cell row A, column B to row C, column D by staying between rows A and C. We are free to move up and down between rows, but not across columns. We are also given K roads, each of which allows us to move freely between any columns in the range $[l, r]$ on row i.

1 Initial Observations

Let's consider two overlapping roads on the same row, with ranges $[l_i, r_i]$ and $[l_j, r_j]$, $l_i \leq l_j \leq r_i \leq r_j$. From any point on $[l_i, l_j)$, we can move to any point on the non-empty range $[l_j, r_i]$, then to any point on the range $(r_i, r_j]$. This implies that we can treat them as a single range $[l_i, r_j]$, and the same argument can be applied for any overlapping ranges on the same row.

Let's focus on a single query with parameters A_i, B_i, C_i, D_i. For any two roads on r_i and r_j, as long as $A_i \leq r_i \leq C_i$ and $A_i \leq r_j \leq C_i$, we are free to move between the roads wherever they share the same column. With this, we can apply the previous argument and combine every pair of overlapping ranges into a single larger range.

Essentially, we "collapse" all roads with rows between A_i and C_i into a one-dimensional space. With this observation, the query becomes finding whether, after combining all column-wise overlapping roads between rows A_i and C_i, we are able to move from column B_i to D_i (i.e., $[B_i, D_i]$ is contained in a larger range after merging roads).

2 Arriving at the Solution

The order in which we process the queries don't matter. Let's see how we can use sweep line to answer queries offline, sorting by $max(A_i, C_i)$ for each query i.

Instead of actually merging all the roads each query, let's use the properties of sweep line to our advantage. In particular, as we iterate from row 1 to N, let's maintain for each column the most recent (highest-indexed) row that has a road containing the column. In other words, we maintain an V array of size M, and for each row $i \in 1...N$, we iterate through each road R on row i and set $V[R_l], V[R_l+1]...V[R_r-1], V[R_r]$ equal to i. Now, answering each query j on the $max(A_j, C_j)$th row becomes intuitive - we simply query whether $min(V[min(B_j, D_j)]...V[max(B_j, D_j)]) \geq min(A_j, C_j)$. With this formulation, notice that we don't actually need to iterate through all rows $1...N$. We only care about the rows that appear in either a road or a query, so we can perform coordinate compression.

Updates and queries can accomplished in $\mathcal{O}(Klog(M))$ and $\mathcal{O}(Klog(M))$ respectively with a lazy segment tree. The initial sorting takes $\mathcal{O}(Mlog(M))$, leading to a final time complexity of $\mathcal{O}((N + Q + M) \cdot log(M))$.

3 Code

```
//@timothyg

#include <bits/stdc++.h>
```

```
using namespace std;

typedef long long ll;
typedef pair<int, int> pii;

#define pb push_back

#define f first
#define s second

template <typename T>
struct segtree
{
   T none, val, lazy;
   int gL, gR, mid;
   segtree<T> *left, *right;

   T comb(T &l, T &r)
   {
      return min(l,r);
   }

   //In this code, lazy represents the value only to be propagated(this->value already
       updated)
   void compose(segtree<T> *tree, T v)
   {
      tree->lazy = v;
      tree->val = v;
   }

   void push()
   {
      if (gL != gR)
      {
         compose(left,lazy);
         compose(right,lazy);
      }
      lazy = tree->val;
   }

   segtree(int l, int r)
   { //modify arr type
      none = INT_MAX;
      lazy = -1;
      gL = l, gR = r, mid = (gL + gR) / 2;
      if (l == r)
      {
         val = -1;
      }
      else
      {
         left = new segtree<T>(l, mid);
         right = new segtree<T>(mid + 1, r);
         val = comb(
            left->val, right->val);
      }
   }

   T query(int l, int r)
   {

      if (gL > r || gR < l)
      {
         return none;
      }

      if (gL == l && gR == r)
      {
         return val;
      }
      push();
      T a = left->query(l, min(r, mid));
      T b = right->query(max(l, mid + 1), r);
      return comb(a, b);
   }

   void update(int l, int r, T updlazy)
   {
      if(gL > r || gR < l){
         return;
      }

      if(gL == l && gR == r){
         compose(this, updlazy);
      }else{
         push();
         left -> update(l, min(r, mid), updlazy);
         right -> update(max(l, mid+1), r, updlazy);
         val = comb(left->val, right->val);
      }
   }
};

int main(){
   ios_base::sync_with_stdio(false); cin.tie(0);
   int N, M; cin >> N >> M;
   vector<pii> roads(M);
   vector<array<int, 3>> bps;
   for(int i = 0; i<M; i++){
      int lane, l, r; cin >> lane >> l >> r;
      bps.pb({lane, 0, i});
      roads[i] = pii(l, r);
   }
   int Q; cin >> Q;
   vector<array<int, 4>> queries(Q);
   vector<int> res(Q);
   for(int i = 0; i<Q; i++){
      int A, B, C, D; cin >> A >> B >> C >> D;
      if(D < B) {
         swap(A, B); swap(B, D);
      }
      bps.pb({D, 1, i});
      queries[i] = {A, B, C, D};
   }
   sort(bps.begin(), bps.end());
   segtree<int> sgt(0, N);
   for(auto i : bps){
      if(i[1]){
         array<int, 4> cur = queries[i[2]];
         //collapse roads
         int mn = sgt.query(min(cur[0], cur[2]), max(cur[0], cur[2]));
         if(mn >= cur[1]){
            res[i[2]] = 1;
         }
```

```
        }else{
            sgt.update(roads[i[2]].f, roads[i[2]].s, i[0]);
        }
    }
    for(int i : res){
        cout << (i ? "YES" : "NO") << '\n';
    }
    return 0;
}
```