University of Tartu ICPC Team Notebook (2017-2018)

October 28, 2017

Contents

1 gcc ordered set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update>;
int main(){
  ordered_set<int>  cur;
  cur.insert(1);
  cur.insert(3);
  cout << cur.order_of_key(2) << endl; // the number of elements in the set less than 2
  cout << *cur.find_by_order(0) << endl; // the 0-th smallest number in the set(0-based)
  cout << *cur.find_by_order(1) << endl; // the 1-th smallest number in the set(0-based)
}
```

2 Triangle centers

```
const double min_delta = 1e-13;
const double coord_max = 1e6;
typedef complex < double > point;
point A, B, C; // vertixes of the triangle
bool collinear(){
  double min_diff = min(abs(A - B), min(abs(A - C), abs(B - C)));
  if(min_diff < coord_max * min_delta)
    return true;
  point sp = (B - A) / (C - A);
  double ang = M_PI/2-abs(abs(arg(sp))-M_PI/2); //positive angle with the real line
  return ang < min_delta;
}
point circum_center(){
  if(collinear())
    return point(NAN,NAN);
  //squared lengths of sides
  double a2, b2, c2;
  a2 = norm(B - C);
  b2 = norm(A - C);
  c2 = norm(A - B);
  //barycentric coordinates of the circumcenter
  double c_A, c_B, c_C;
  c_A = a2 * (b2 + c2 - a2);//sin(2 * alpha) may be used as well
```

```
  c_B = b2 * (a2 + c2 - b2);
  c_C = c2 * (a2 + b2 - c2);
  double sum = c_A + c_B + c_C;
  c_A /= sum;
  c_B /= sum;
  c_C /= sum;
  // cartesian coordinates of the circumcenter
  return c_A * A + c_B * B + c_C * C;
}
point centroid(){ //center of mass
  return (A + B + C) / 3.0;
}
point ortho_center(){ //euler line
  point O = circum_center();
  return O + 3.0 * (centroid() - O);
};
point nine_point_circle_center(){ //euler line
  point O = circum_center();
  return O + 1.5 * (centroid() - O);
};
point in_center(){
  if(collinear())
    return point(NAN,NAN);
  double a, b, c; //side lengths
  a = abs(B - C);
  b = abs(A - C);
  c = abs(A - B);
  //trilinear coordinates are (1,1,1)
  //barycentric coordinates
  double c_A = a, c_B = b, c_C = c;
  double sum = c_A + c_B + c_C;
  c_A /= sum;
  c_B /= sum;
  c_C /= sum;
  // cartesian coordinates of the incenter
  return c_A * A + c_B * B + c_C * C;
}
```

3 Dinic

```
struct MaxFlow{
    typedef long long ll;
    const ll INF = 1e18;
    struct Edge{
        int u,v;
        ll c,rc;
        shared_ptr<ll> flow;
        pair<int,int> id() const {
            return make_pair(min(u,v),max(u,v));
        }
        Edge(int _u, int _v, ll _c, ll _rc = 0):u(_u),v(_v),c(_c),rc(_rc){
        }
        void join(const Edge &t){
            if(u == t.u){
                c += t.c;
                rc += t.rc;
            }
            else{
                c += t.rc;
                rc += t.c;
            }
        }
    };
    struct FlowTracker{
        shared_ptr<ll> flow;
        ll cap, rcap;
        bool dir;
        FlowTracker(ll _cap, ll _rcap, shared_ptr<ll> _flow, int
         ↪  _dir):cap(_cap),rcap(_rcap),flow(_flow),dir(_dir){ }
        ll rem() const {
            if(dir == 0){
                return cap-*flow;
            }
```

```
            else{
                return rcap+*flow;
            }
        }
        void add_flow(ll f){
            if(dir == 0)
                *flow += f;
            else
                *flow -= f;
            assert(*flow <= cap);
            assert(-*flow <= rcap);
        }
        operator ll() const { return rem(); }
        void operator-=(ll x){ add_flow(x); }
        void operator+=(ll x){ add_flow(-x); }
    };
    int source,sink;
    vector<vector<int> > adj;
    vector<vector<FlowTracker> > cap;
    vector<Edge> edges;
    MaxFlow(int _source, int _sink):source(_source),sink(_sink){
        assert(source != sink);
    }
    int add_edge(Edge e){
        edges.push_back(e);
        return edges.size()-1;

    }
    int add_edge(int u, int v, ll c, ll rc = 0){
        return add_edge(Edge(u,v,c,rc));
    }
    void group_edges(){
        map<pair<int,int>,vector<Edge> > edge_groups;
        for(auto edge: edges)
            if(edge.u != edge.v)
                edge_groups[edge.id()].push_back(edge);
        vector<Edge> grouped_edges;
        for(auto group: edge_groups){
            Edge main_edge = group.second[0];
            for(int i = 1; i < group.second.size(); ++i)
                main_edge.join(group.second[i]);
            grouped_edges.push_back(main_edge);
        }
        edges = grouped_edges;
    }
    vector<int> now,lvl;
    void prep(){
        int max_id = max(source,sink);
        for(auto edge : edges)
            max_id = max(max_id,max(edge.u,edge.v));
        adj.resize(max_id+1);
        cap.resize(max_id+1);
        now.resize(max_id+1);
        lvl.resize(max_id+1);
        for(auto &edge : edges){
            auto flow = make_shared<ll>(0);
            adj[edge.u].push_back(edge.v);
            cap[edge.u].push_back(FlowTracker(edge.c,edge.rc,flow,0));
            adj[edge.v].push_back(edge.u);
            cap[edge.v].push_back(FlowTracker(edge.c,edge.rc,flow,1));
            assert(cap[edge.u].back() == edge.c);
            edge.flow = flow;
        }
    }
    bool dinic_bfs(){
        fill(now.begin(),now.end(),0);
        fill(lvl.begin(),lvl.end(),0);
        lvl[source] = 1;
        vector<int> bfs(1,source);
        for(int i = 0; i < bfs.size(); ++i){
            int u = bfs[i];
            for(int j = 0; j < adj[u].size(); ++j){
                int v = adj[u][j];
```

```
                if(cap[u][j] > 0 && lvl[v] == 0){
                    lvl[v] = lvl[u]+1;
                    bfs.push_back(v);
                }
            }
        }
        return lvl[sink] > 0;
    }
    ll dinic_dfs(int u, ll flow){
        if(u == sink)
            return flow;
        while(now[u] < adj[u].size()){
            int v = adj[u][now[u]];
            if(lvl[v] == lvl[u] + 1 && cap[u][now[u]] != 0){
                ll res = dinic_dfs(v,min(flow,(ll)cap[u][now[u]]));
                if(res > 0){
                    cap[u][now[u]] -= res;
                    return res;
                }
            }
            ++now[u];
        }
        return 0;
    }
    ll calc(){
        prep();
        ll ans = 0;
        while(dinic_bfs()){
            ll cur = 0;
            do{
                cur = dinic_dfs(source,INF);
                ans += cur;
            }while(cur > 0);
        }
        return ans;
    }
};
int main(){
    int n,m;
    cin >> n >> m;
    auto mf = MaxFlow(1,n); // arguments source and sink, memory usage O(largest node index), sink doesn't need
     ↪  to be last
    int edge_index;
    for(int i = 0; i < m; ++i){
        int a,b,c;
        cin >> a >> b >> c;
        //undirected edge is a pair of edges (a,b,c,0) and (a,b,0,c)
        edge_index = mf.add_edge(a,b,c,c); //store edge index if care about flow value
    }
    mf.group_edges(); // small auxillary to remove multiple edges, only use this if we need to know TOTAL FLOW
     ↪  ONLY
    cout << mf.calc() << '\n';
    //cout << *mf.edges[edge_index].flow << '\n'; // ONLY if group_edges() WAS NOT CALLED
}
```

4 Min cost max flow O(flow $\cdot n^2$)

```
const int nmax=1055;
const ll inf=1e14;
int t, n, v;
ll rem_flow[nmax][nmax];
ll cost[nmax][nmax];
ll min_dist[nmax];
int prev_node[nmax];
ll node_flow[nmax];
bool visited[nmax];
ll tot_cost, tot_flow;
void mincmaxf(){
  tot_cost=0;
  tot_flow=0;
  ll sink_pot=0;
  while(true){
    for(int i=0; i<=v; ++i){
```

```
      min_dist[i]=inf;
      visited[i]=false;
    }
    min_dist[0]=0;
    node_flow[0]=inf;
    int min_node;
    while(true){
      int min_node=v;
      for(int i=0; i<v; ++i){
        if((!visited[i]) && min_dist[i]<min_dist[min_node]){
          min_node=i;
        }
      }
      if(min_node==v){
        break;
      }
      visited[min_node]=true;
      for(int i=0; i<v; ++i){
        if((!visited[i]) && min_dist[min_node]+cost[min_node][i] < min_dist[i]){
          min_dist[i]=min_dist[min_node]+cost[min_node][i];
          prev_node[i]=min_node;
          node_flow[i]=min(node_flow[min_node], rem_flow[min_node][i]);
        }
      }
    }
    if(min_dist[v-1]==inf){
      break;
    }
    for(int i=0; i<v; ++i){
      for(int j=0; j<v; ++j){
        if(cost[i][j]!=inf){
          cost[i][j]+=min_dist[i];
          cost[i][j]-=min_dist[j];
        }
      }
    }
    sink_pot+=min_dist[v-1];
    tot_flow+=node_flow[v-1];
    tot_cost+=sink_pot*node_flow[v-1];
    int cur=v-1;
    while(cur!=0){
      rem_flow[prev_node[cur]][cur]-=node_flow[v-1];
      rem_flow[cur][prev_node[cur]]+=node_flow[v-1];
      cost[cur][prev_node[cur]]=0;
      if(rem_flow[prev_node[cur]][cur]==0){
        cost[prev_node[cur]][cur]=inf;
      }
      cur=prev_node[cur];
    }
  }
}
```

5 Aho Corasick O(|alpha| $\sum$ len)

```
const int alpha_size=26;
struct node{
  node *nxt[alpha_size]; //May use other structures to move in trie
  node *suffix;
  node(){
    memset(nxt, 0, alpha_size*sizeof(node *));
  }
  int cnt=0;
};
node *aho_corasick(vector<vector<char> > &dict){
  node *root= new node;
  root->suffix = 0;
  vector<pair<vector<char> *, node *> > cur_state;
  for(vector<char> &s : dict)
    cur_state.emplace_back(&s, root);
  for(int i=0; !cur_state.empty(); ++i){
    vector<pair<vector<char> *, node *> > nxt_state;
    for(auto &cur : cur_state){
      node *nxt=cur.second->nxt[(*cur.first)[i]];
```

```
      if(nxt){
        cur.second=nxt;
      }else{
        nxt = new node;
        cur.second->nxt[(*cur.first)[i]] = nxt;
        node *suf = cur.second->suffix;
        cur.second = nxt;
        nxt->suffix = root; //set correct suffix link
        while(suf){
          if(suf->nxt[(*cur.first)[i]]){
            nxt->suffix = suf->nxt[(*cur.first)[i]];
            break;
          }
          suf=suf->suffix;
        }
      }
      if(cur.first->size() > i+1)
        nxt_state.push_back(cur);
    }
    cur_state=nxt_state;
  }
  return root;
}
node *walk(node *cur, char c){
  while(true){
    if(cur->nxt[c])
      return cur->nxt[c];
    if(!cur->suffix){
      return cur;
    }
    cur = cur->suffix;
  }
}
void cnt_matches(node *root, vector<char> &match_in){
  node *cur = root;
  for(char c : match_in){
    cur = walk(cur, c);
    ++cur->cnt;
  }
}
void add_cnt(node *root){
  vector<node *> to_visit = {root};
  for(int i=0; i<to_visit.size(); ++i){
    node *cur = to_visit[i];
    for(int j=0; j<alpha_size; ++j){
      if(cur->nxt[j]){
        to_visit.push_back(cur->nxt[j]);
      }
    }
  }
  for(int i=to_visit.size()-1; i>0; --i){
    to_visit[i]->suffix->cnt += to_visit[i]->cnt;
  }
}
```

6 Suffix automaton O($(n+q)\log(|\text{alpha}|)$)

```
class AutoNode {
 private:
  map< char, AutoNode * > nxt_char;  // Map is faster than hashtable and unsorted arrays
 public:
  int len;
  AutoNode *suf;
  bool has_nxt(char c) const {
    return nxt_char.count(c);
  }
  AutoNode *nxt(char c) {
    if (!has_nxt(c))
      return NULL;
    return nxt_char[c];
  }
  void set_nxt(char c, AutoNode *node) {
    nxt_char[c] = node;
```

```
  }
  AutoNode *split(int new_len, char c) {
    AutoNode *new_n = new AutoNode;
    new_n->nxt_char = nxt_char;
    new_n->len = new_len;
    new_n->suf = suf;
    suf = new_n;
    return new_n;
  }
  // Extra functions for matching and counting
  AutoNode *lower_depth(int depth) {
    if (suf->len >= depth)
      return suf->lower_depth(depth);
    return this;
  }
  AutoNode *walk(char c, int depth, int &match_len) {
    match_len = min(match_len, len);
    if (has_nxt(c)) {
      ++match_len;
      return nxt(c)->lower_depth(depth);
    }
    if (suf)
      return suf->walk(c, depth, match_len);
    return this;
  }
  int paths_to_end = 0;
  void set_as_end() {
    paths_to_end = 1;
    if (suf) suf->set_as_end();
  }
  bool vis = false;
  void calc_paths_to_end() {
    if (!vis) {
      vis = true;
      for (auto cur : nxt_char) {
        cur.second->calc_paths_to_end();
        paths_to_end += cur.second->paths_to_end;
      }
    }
  }
};
struct SufAutomaton {
  AutoNode *last;
  AutoNode *root;
  void extend(char new_c) {
    AutoNode *new_end = new AutoNode;  // The equivalence class containing the whole new string
    new_end->len = last->len + 1;
    AutoNode *suf_w_nxt = last;                          // The whole old string class
    while (suf_w_nxt && !suf_w_nxt->has_nxt(new_c)) {    // is turned into the longest suffix which
                                                         // can be turned into a substring of old state
                                                         // by appending new_c
      suf_w_nxt->set_nxt(new_c, new_end);
      suf_w_nxt = suf_w_nxt->suf;
    }
    if (!suf_w_nxt) {  // The new character isn't part of the old string
      new_end->suf = root;
    } else {
      AutoNode *max_sbstr = suf_w_nxt->nxt(new_c);  // Equivalence class containing longest
                                                    // substring which is a suffix of the new state.
      if (suf_w_nxt->len + 1 == max_sbstr->len) { // Check whether splitting is needed
        new_end->suf = max_sbstr;
      } else {
        AutoNode *eq_sbstr = max_sbstr->split(suf_w_nxt->len + 1, new_c);
        new_end->suf = eq_sbstr;
        // Make suffixes of suf_w_nxt point to eq_sbstr instead of mox_sbstr
        AutoNode *w_edge_to_eq_sbstr = suf_w_nxt;
        while (w_edge_to_eq_sbstr != 0 && w_edge_to_eq_sbstr->nxt(new_c) == max_sbstr) {
          w_edge_to_eq_sbstr->set_nxt(new_c, eq_sbstr);
          w_edge_to_eq_sbstr = w_edge_to_eq_sbstr->suf;
        }
      }
    }
    last = new_end;
```

```
  }
  SufAutomaton(string to_suffix) {
    root = new AutoNode;
    root->len = 0;
    root->suf = NULL;
    last = root;
    for (char c : to_suffix) extend(c);
  }
};
```

7 Templated multi dimensional BIT O($\log(n)^{\text{dim}}$)

```
// Fully overloaded any dimensional BIT, use any type for coordinates, elements, return_value.
// Includes coordinate compression.
template < typename elem_t, typename coord_t, coord_t n_inf, typename ret_t >
class BIT {
  vector< coord_t > positions;
  vector< elem_t > elems;
  bool initiated = false;

 public:
  BIT() {
    positions.push_back(n_inf);
  }
  void initiate() {
    if (initiated) {
      for (elem_t &c_elem : elems)
        c_elem.initiate();
    } else {
      initiated = true;
      sort(positions.begin(), positions.end());
      positions.resize(unique(positions.begin(), positions.end()) - positions.begin());
      elems.resize(positions.size());
    }
  }
  template < typename... loc_form >
  void update(coord_t cord, loc_form... args) {
    if (initiated) {
      int pos = lower_bound(positions.begin(), positions.end(), cord) - positions.begin();
      for (; pos < positions.size(); pos += pos & -pos)
        elems[pos].update(args...);
    } else {
      positions.push_back(cord);
    }
  }
  template < typename... loc_form >
  ret_t query(coord_t cord, loc_form... args) { //sum in open interval (-inf, cord)
    ret_t res = 0;
    int pos = (lower_bound(positions.begin(), positions.end(), cord) - positions.begin())-1;
    for (; pos > 0; pos -= pos & -pos)
      res += elems[pos].query(args...);
    return res;
  }
};
template < typename internal_type >
struct wrapped {
  internal_type a = 0;
  void update(internal_type b) {
    a += b;
  }
  internal_type query() {
    return a;
  }
  // Should never be called, needed for compilation
  void initiate() {
    cerr << 'i' << endl;
  }
  void update() {
    cerr << 'u' << endl;
  }
};
int main() {
  // retun type should be same as type inside wrapped
```

```
  BIT< BIT< wrapped< ll >, int, INT_MIN, ll >, int, INT_MIN, ll > fenwick;
  int dim = 2;
  vector< tuple< int, int, ll > > to_insert;
  to_insert.emplace_back(1, 1, 1);
  // set up all positions that are to be used for update
  for (int i = 0; i < dim; ++i) {
    for (auto &cur : to_insert)
      fenwick.update(get< 0 >(cur), get< 1 >(cur));  // May include value which won't be used
    fenwick.initiate();
  }
  // actual use
  for (auto &cur : to_insert)
    fenwick.update(get< 0 >(cur), get< 1 >(cur), get< 2 >(cur));
  cout << fenwick.query(2, 2)<<'\n';
}
```