Contents

1 Setting

1.1 vimrc

2 Math

3 Data Structure

4 Graph

4.1 Min-cost Maximum Flow

```
#include <functional>
#include <queue>
#include <limits>
#include <vector>
#include <algorithm>

using namespace std;
// from KCM1700/algorithms

// precondition: there is no negative cycle.
// usage:
//  MinCostFlow mcf(n);
//  for(each edges) mcf.addEdge(from, to, cost, capacity);
//  mcf.solve(source, sink); // min cost max flow
//  mcf.solve(source, sink, 0); // min cost flow
//  mcf.solve(source, sink, goal_flow); // min cost flow with total_flow >=
  goal_flow if possible
struct MinCostFlow
{
    typedef int cap_t;
    typedef int cost_t;

    bool iszerocap(cap_t cap) { return cap == 0; }

    struct edge {
        int target;
        cost_t cost;
        cap_t residual_capacity;
        cap_t orig_capacity;
        size_t revid;
    };

    int n;
    vector<vector<edge>> graph;
    vector<cost_t> pi;
    bool needNormalize, ranbefore;
    int lastStart;

    MinCostFlow(int n) : graph(n), n(n), pi(n, 0), needNormalize(false),
      ranbefore(false) {}
    void addEdge(int s, int e, cost_t cost, cap_t cap)
    {
        if (s == e) return;
        edge forward={e, cost, cap, cap, graph[e].size()};
        edge backward={s, -cost, 0, 0, graph[s].size()};
        if (cost < 0 || ranbefore) needNormalize = true;
        graph[s].emplace_back(forward);
        graph[e].emplace_back(backward);
    }
    bool normalize(int s) {
        auto infinite_cost = numeric_limits<cost_t>::max();
        vector<cost_t> dist(n, infinite_cost);
        dist[s] = 0;
        queue<int> q;
        vector<int> v(n), relax_count(n);
        v[s] = 1; q.push(s);
        while(!q.empty()) {
            int cur = q.front();
            v[cur] = 0; q.pop();
            if (++relax_count[cur] >= n) return false;
            for (const auto &e : graph[cur]) {
                if (iszerocap(e.residual_capacity)) continue;
                auto next = e.target;
                auto ncost = dist[cur] + e.cost;
                if (dist[next] > ncost) {
```

```
                dist[next] = ncost;
                if (v[next]) continue;
                v[next] = 1; q.push(next);
            }
        }
    }
    for (int i = 0; i < n; i++) pi[i] = dist[i];
    return true;
}

pair<cost_t, cap_t> AugmentShortest(int s, int e, cap_t flow_limit) {
    auto infinite_cost = numeric_limits<cost_t>::max();
    auto infinite_flow = numeric_limits<cap_t>::max();
    typedef pair<cost_t, int> pq_t;
    priority_queue<pq_t, vector<pq_t>, greater<pq_t>> pq;
    vector<pair<cost_t, cap_t>> dist(n, make_pair(infinite_cost, 0));
    vector<int> from(n, -1), v(n);

    if (needNormalize || (ranbefore && lastStart != s))
        normalize(s);
    ranbefore = true;
    lastStart = s;

    dist[s] = pair<cost_t, cap_t>(0, infinite_flow);
    pq.emplace(dist[s].first, s);
    while(!pq.empty()) {
        auto cur = pq.top().second; pq.pop();
        if (v[cur]) continue;
        v[cur] = 1;
        if (cur == e) continue;
        for (const auto &e : graph[cur]) {
            auto next = e.target;
            if (v[next]) continue;
            if (iszerocap(e.residual_capacity)) continue;
            auto ncost = dist[cur].first + e.cost - pi[next] + pi[cur];
            auto nflow = min(dist[cur].second, e.residual_capacity);
            if (dist[next].first <= ncost) continue;
            dist[next] = make_pair(ncost, nflow);
            from[next] = e.revid;
            pq.emplace(dist[next].first, next);
        }
    }
    /** augment the shortest path **/
    auto p = e;
    auto pathcost = dist[p].first + pi[p] - pi[s];
    auto flow = dist[p].second;
    if (iszerocap(flow)|| (flow_limit <= 0 && pathcost >= 0)) return pair<
      cost_t, cap_t>(0, 0);
    if (flow_limit > 0) flow = min(flow, flow_limit);
    /* update potential */
    for (int i = 0; i < n; i++) {
        if (iszerocap(dist[i].second)) continue;
        pi[i] += dist[i].first;
    }
    while (from[p] != -1) {
        auto nedge = from[p];
        auto np = graph[p][nedge].target;
        auto fedge = graph[p][nedge].revid;
        graph[p][nedge].residual_capacity += flow;
        graph[np][fedge].residual_capacity -= flow;
        p = np;
    }
    return make_pair(pathcost * flow, flow);
}

pair<cost_t,cap_t> solve(int s, int e, cap_t flow_minimum = numeric_limits
  <cap_t>::max()) {
    cost_t total_cost = 0;
    cap_t total_flow = 0;
    for(;;) {
        auto res = AugmentShortest(s, e, flow_minimum - total_flow);
        if (res.second <= 0) break;
        total_cost += res.first;
        total_flow += res.second;
    }
    return make_pair(total_cost, total_flow);
}
};
```

5 Geometry

5.1 Operations

```
#include <stdio.h>

typedef struct point{
    int x,y;
}point;

int ccw(const point &a, const point &b, const point &c){
    return a.x*b.y+b.x*c.y+c.x*a.y - a.y*b.x-b.y*c.x-c.y*a.x;
}
```

6 String

6.1 KMP

```
#include <stdio.h>
#include <string.h>

void kmp(char *t, char *p, int *r, int *ff){
    int l = strlen(t)
    for(int i=0,j=0;i<l;i++){
        if(t[i]!=p[j]){
            if(j==0)r[i]=0;
            else{
```

```
                i--;
                j=ff[j-1];
            }
        }
        else r[i]=++j;
    }
}

int main(){
    int n;
    char a[1000], b[1000]; // a: 찾을문자열 , b: 대상문자열.
    int ff[1000], d[1000]; // ff: 실패함수배열 , d: 결과배열 .입력
    //
    kmp(a+1,a,ff+1,ff); // 실패함수생성
    kmp(b,a,d,ff);

}
```

7 Miscellaneous

7.1 Magic Numbers

ì ì : 10 007 , 10 009 , 10 111 , 31 567 , 70 001 , 1 000 003 , 1 000 033 , 4 000 037 , 1 000 000 007 , 1 000 000 009