

## Stanford University ICPC Team Notebook (2015-16)

## Contents

## 1 Combinatorial optimization

1.1	Sparse max-flow . . . . .	1
1.2	Min-cost max-flow . . . . .	2
1.3	Push-relabel max-flow . . . . .	2
1.4	Min-cost matching . . . . .	3
1.5	Max bipartite matching . . . . .	4
1.6	Global min-cut . . . . .	5
1.7	Graph cut inference . . . . .	5

## 1 Combinatorial optimization

## 1.1 Sparse max-flow

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//      O(|V|^2 |E|)
//
// INPUT:
//      - graph, constructed using AddEdge()
//      - source and sink
//
// OUTPUT:
//      - maximum flow value
//      - To obtain actual flow values, look at edges with capacity > 0
//      (zero capacity edges are residual edges).
```

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

using namespace std;
typedef long long LL;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
    LL rcap() { return cap - flow; }
};

struct Dinic {
    int N;
    vector<vector<Edge>> G;
    vector<vector<Edge*>> Lf;
    vector<int> layer;
    vector<int> Q;

    Dinic(int N) : N(N), G(N), Q(N) {}

    void AddEdge(int from, int to, int cap) {
        if (from == to) return;
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    LL BlockingFlow(int s, int t) {
```

```
        layer.clear(); layer.resize(N, -1);
        layer[s] = 0;
        Lf.clear(); Lf.resize(N);

        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail) {
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++) {
                Edge &e = G[x][i]; if (e.rcap() <= 0) continue;
                if (layer[e.to] == -1) {
                    layer[e.to] = layer[e.from] + 1;
                    Q[tail++] = e.to;
                }
                if (layer[e.to] > layer[e.from]) {
                    Lf[e.from].push_back(&e);
                }
            }
        }
        if (layer[t] == -1) return 0;

        LL totflow = 0;
        vector<Edge*> P;
        while (!Lf[s].empty()) {
            int curr = P.empty() ? s : P.back()->to;
            if (curr == t) { // Augment
                LL amt = P.front()->rcap();
                for (int i = 0; i < P.size(); ++i) {
                    amt = min(amt, P[i]->rcap());
                }
                totflow += amt;
                for (int i = P.size() - 1; i >= 0; --i) {
                    P[i]->flow += amt;
                    G[P[i]->to][P[i]->index].flow -= amt;
                    if (P[i]->rcap() <= 0) {
                        Lf[P[i]->from].pop_back();
                        P.resize(i);
                    }
                }
            } else if (Lf[curr].empty()) { // Retreat
                P.pop_back();
                for (int i = 0; i < N; ++i)
                    for (int j = 0; j < Lf[i].size(); ++j)
                        if (Lf[i][j]->to == curr)
                            Lf[i].erase(Lf[i].begin() + j);
            } else { // Advance
                P.push_back(Lf[curr].back());
            }
        }
        return totflow;
    }

    LL GetMaxFlow(int s, int t) {
        LL totflow = 0;
        while (LL flow = BlockingFlow(s, t))
            totflow += flow;
        return totflow;
    }
};
```

```
// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    Dinic flow(n);
    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        if (a == b) continue;
        flow.AddEdge(a-1, b-1, c);
        flow.AddEdge(b-1, a-1, c);
    }
    printf("%Ld\n", flow.GetMaxFlow(0, n-1));
    return 0;
}
```

```

}

// END CUT

```

## 1.2 Min-cost max-flow

```

// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972). This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]). For a regular max flow, set all edge costs to 0.
//
// Running time,  $O(|V|^2)$  cost per augmentation
// max flow:  $O(|V|^3)$  augmentations
// min cost max flow:  $O(|V|^4 * \text{MAX\_EDGE\_COST})$  augmentations
//
// INPUT:
// - graph, constructed using AddEdge()
// - source
// - sink
//
// OUTPUT:
// - (maximum flow value, minimum cost value)
// - To obtain the actual flow, look at positive values only.

```

```

#include <cmath>
#include <vector>
#include <iostream>

```

```

using namespace std;

```

```

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

```

```

const L INF = numeric_limits<L>::max() / 4;

```

```

struct MinCostMaxFlow {
    int N;
    VVL cap, flow, cost;
    VI found;
    VL dist, pi, width;
    VPII dad;

```

```

    MinCostMaxFlow(int N) :
        N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
        found(N), dist(N), pi(N), width(N), dad(N) {}

```

```

    void AddEdge(int from, int to, L cap, L cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }

```

```

    void Relax(int s, int k, L cap, L cost, int dir) {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k]) {
            dist[k] = val;
            dad[k] = make_pair(s, dir);
            width[k] = min(cap, width[s]);
        }
    }

```

```

    L Dijkstra(int s, int t) {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;

```

```

        while (s != -1) {
            int best = -1;
            found[s] = true;

```

```

            for (int k = 0; k < N; k++) {
                if (found[k]) continue;
                Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                Relax(s, k, flow[k][s], -cost[k][s], -1);
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
            s = best;
        }
    }

```

```

    for (int k = 0; k < N; k++)
        pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
}

```

```

pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
        totflow += amt;
        for (int x = t; x != s; x = dad[x].first) {
            if (dad[x].second == 1) {
                flow[dad[x].first][x] += amt;
                totcost += amt * cost[dad[x].first][x];
            } else {
                flow[x][dad[x].first] -= amt;
                totcost -= amt * cost[x][dad[x].first];
            }
        }
    }
    return make_pair(totflow, totcost);
}
};

```

```

// BEGIN CUT
// The following code solves UVA problem #10594: Data Flow

```

```

int main() {
    int N, M;

    while (scanf("%d%d", &N, &M) == 2) {
        VVL v(M, VL(3));
        for (int i = 0; i < M; i++)
            scanf("%d%d%d", &v[i][0], &v[i][1], &v[i][2]);
        L D, K;
        scanf("%d%d", &D, &K);

        MinCostMaxFlow mcmf(N+1);
        for (int i = 0; i < M; i++) {
            mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i][2]);
            mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i][2]);
        }
        mcmf.AddEdge(0, 1, D, 0);

        pair<L, L> res = mcmf.GetMaxFlow(0, N);

        if (res.first == D) {
            printf("%d\n", res.second);
        } else {
            printf("Impossible.\n");
        }
    }

    return 0;
}

```

```

// END CUT

```

## 1.3 Push-relabel max-flow

```

// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic. This implementation is
// significantly faster than straight Ford-Fulkerson. It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//

```

```

// Running time:
//      O(|V|3)
//
// INPUT:
//      - graph, constructed using AddEdge()
//      - source
//      - sink
//
// OUTPUT:
//      - maximum flow value
//      - To obtain the actual flow values, look at all edges with
//        capacity > 0 (zero capacity edges are residual edges).

#include <cmath>
#include <vector>
#include <iostream>
#include <queue>

using namespace std;

typedef long long LL;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel {
    int N;
    vector<vector<Edge>> > G;
    vector<LL> excess;
    vector<int> dist, active, count;
    queue<int> Q;

    PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2*N) {}

    void AddEdge(int from, int to, int cap) {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    void Enqueue(int v) {
        if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }
    }

    void Push(Edge &e) {
        int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
        if (dist[e.from] <= dist[e.to] || amt == 0) return;
        e.flow += amt;
        G[e.to][e.index].flow -= amt;
        excess[e.to] += amt;
        excess[e.from] -= amt;
        Enqueue(e.to);
    }

    void Gap(int k) {
        for (int v = 0; v < N; v++) {
            if (dist[v] < k) continue;
            count[dist[v]]--;
            dist[v] = max(dist[v], N+1);
            count[dist[v]]++;
            Enqueue(v);
        }
    }

    void Relabel(int v) {
        count[dist[v]]--;
        dist[v] = 2*N;
        for (int i = 0; i < G[v].size(); i++)
            if (G[v][i].cap - G[v][i].flow > 0)
                dist[v] = min(dist[v], dist[G[v][i].to] + 1);
        count[dist[v]]++;
        Enqueue(v);
    }
};

```

```

void Discharge(int v) {
    for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
    if (excess[v] > 0) {
        if (count[dist[v]] == 1)
            Gap(dist[v]);
        else
            Relabel(v);
    }
}

LL GetMaxFlow(int s, int t) {
    count[0] = N-1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
        excess[s] += G[s][i].cap;
        Push(G[s][i]);
    }

    while (!Q.empty()) {
        int v = Q.front();
        Q.pop();
        active[v] = false;
        Discharge(v);
    }

    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
}

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    PushRelabel pr(n);
    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        if (a == b) continue;
        pr.AddEdge(a-1, b-1, c);
        pr.AddEdge(b-1, a-1, c);
    }
    printf("%d\n", pr.GetMaxFlow(0, n-1));
    return 0;
}

// END CUT

```

## 1.4 Min-cost matching

```

////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs. In practice, it solves 1000x1000 problems in around 1
// second.
//
// cost[i][j] = cost for pairing left node i with right node j
// Lmate[i] = index of right node that left node i pairs with
// Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative. To perform
// maximization, simply negate the cost[][] matrix.
////////////////////////////////////

#include <algorithm>
#include <cstdio>
#include <cmath>

```

```

#include <vector>

using namespace std;

typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
    int n = int(cost.size());

    // construct dual feasible solution
    VD u(n);
    VD v(n);
    for (int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
    }
    for (int j = 0; j < n; j++) {
        v[j] = cost[0][j] - u[0];
        for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
    }

    // construct primal solution satisfying complementary slackness
    Lmate = VI(n, -1);
    Rmate = VI(n, -1);
    int mated = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (Rmate[j] != -1) continue;
            if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
                Lmate[i] = j;
                Rmate[j] = i;
                mated++;
                break;
            }
        }
    }

    VD dist(n);
    VI dad(n);
    VI seen(n);

    // repeat until primal solution is feasible
    while (mated < n) {

        // find an unmatched left node
        int s = 0;
        while (Lmate[s] != -1) s++;

        // initialize Dijkstra
        fill(dad.begin(), dad.end(), -1);
        fill(seen.begin(), seen.end(), 0);
        for (int k = 0; k < n; k++)
            dist[k] = cost[s][k] - u[s] - v[k];

        int j = 0;
        while (true) {

            // find closest
            j = -1;
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                if (j == -1 || dist[k] < dist[j]) j = k;
            }
            seen[j] = 1;

            // termination condition
            if (Rmate[j] == -1) break;

            // relax neighbors
            const int i = Rmate[j];
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
                if (dist[k] > new_dist) {
                    dist[k] = new_dist;

```

```

                    dad[k] = j;
                }
            }

            // update dual variables
            for (int k = 0; k < n; k++) {
                if (k == j || !seen[k]) continue;
                const int i = Rmate[k];
                v[k] += dist[k] - dist[j];
                u[i] -= dist[k] - dist[j];
            }
            u[s] += dist[j];

            // augment along path
            while (dad[j] >= 0) {
                const int d = dad[j];
                Rmate[j] = Rmate[d];
                Lmate[Rmate[j]] = j;
                j = d;
            }
            Rmate[j] = s;
            Lmate[s] = j;

            mated++;
        }

        double value = 0;
        for (int i = 0; i < n; i++)
            value += cost[i][Lmate[i]];

        return value;
    }
}

```

## 1.5 Max bipartite matching

```

// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
//
// INPUT: w[i][j] = edge between row node i and column node j
// OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//         mc[j] = assignment for column node j, -1 if unassigned
//         function returns number of matches made

#include <vector>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
    for (int j = 0; j < w[i].size(); j++) {
        if (w[i][j] && !seen[j]) {
            seen[j] = true;
            if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
                mr[i] = j;
                mc[j] = i;
                return true;
            }
        }
    }
    return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);

    int ct = 0;
    for (int i = 0; i < w.size(); i++) {
        VI seen(w[0].size());
        if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
}

```

```

    return ct;
}

```

## 1.6 Global min-cut

```

// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
// O(|V|^3)
//
// INPUT:
// - graph, constructed using AddEdge()
//
// OUTPUT:
// - (min cut value, nodes in half of min cut)

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
    int N = weights.size();
    VI used(N), cut, best_cut;
    int best_weight = -1;

    for (int phase = N-1; phase >= 0; phase--) {
        VI w = weights[0];
        VI added = used;
        int prev, last = 0;
        for (int i = 0; i < phase; i++) {
            prev = last;
            last = -1;
            for (int j = 1; j < N; j++)
                if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
            if (i == phase-1) {
                for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
                for (int j = 0; j < N; j++) weights[j][prev] = weights[j][last];
                used[last] = true;
                cut.push_back(last);
                if (best_weight == -1 || w[last] < best_weight) {
                    best_cut = cut;
                    best_weight = w[last];
                }
            } else {
                for (int j = 0; j < N; j++)
                    w[j] += weights[last][j];
                added[last] = true;
            }
        }
    }
    return make_pair(best_weight, best_cut);
}

// BEGIN CUT
// The following code solves UVA problem #10989: Bomb, Divide and Conquer
int main() {
    int N;
    cin >> N;
    for (int i = 0; i < N; i++) {
        int n, m;
        cin >> n >> m;
        VVI weights(n, VI(n));
        for (int j = 0; j < m; j++) {
            int a, b, c;
            cin >> a >> b >> c;
            weights[a-1][b-1] = weights[b-1][a-1] = c;
        }
        pair<int, VI> res = GetMinCut(weights);
    }
}

```

```

    cout << "Case #" << i+1 << ": " << res.first << endl;
}
// END CUT

```

## 1.7 Graph cut inference

```

// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
// minimize          sum_i psi_i(x[i])
// x[1]...x[n] in {0,1} + sum_{i < j} phi_{ij}(x[i], x[j])
//
// where
// psi_i : {0, 1} --> R
// phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
// phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)  (*)
//
// This can also be used to solve maximization problems where the
// direction of the inequality in (*) is reversed.
//
// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi_{ij}(u, v)
// psi -- a matrix such that psi[i][u] = psi_i(u)
// x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution
//
// To use this code, create a GraphCutInference object, and call the
// DoInference() method. To perform maximization instead of minimization,
// ensure that #define MAXIMIZATION is enabled.

#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef vector<VVI> VVVI;
typedef vector<VVVI> VVVVI;

const int INF = 1000000000;

// comment out following line for minimization
#define MAXIMIZATION

struct GraphCutInference {
    int N;
    VVI cap, flow;
    VI reached;

    int Augment(int s, int t, int a) {
        reached[s] = 1;
        if (s == t) return a;
        for (int k = 0; k < N; k++) {
            if (reached[k]) continue;
            if (int aa = min(a, cap[s][k] - flow[s][k])) {
                if (int b = Augment(k, t, aa)) {
                    flow[s][k] += b;
                    flow[k][s] -= b;
                    return b;
                }
            }
        }
        return 0;
    }

    int GetMaxFlow(int s, int t) {
        N = cap.size();
        flow = VVI(N, VI(N));
        reached = VI(N);

        int totflow = 0;
    }
}

```

```

while (int amt = Augment(s, t, INF)) {
    totflow += amt;
    fill(reached.begin(), reached.end(), 0);
}
return totflow;
}

int DoInference(const VVVVI &phi, const VVI &psi, VI &x) {
    int M = phi.size();
    cap = VVI(M+2, VI(M+2));
    VI b(M);
    int c = 0;

    for (int i = 0; i < M; i++) {
        b[i] += psi[i][1] - psi[i][0];
        c += psi[i][0];
        for (int j = 0; j < i; j++)
            b[i] += phi[i][j][1][1] - phi[i][j][0][1];
        for (int j = i+1; j < M; j++) {
            cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j][1][1];
            b[i] += phi[i][j][1][0] - phi[i][j][0][0];
            c += phi[i][j][0][0];
        }
    }

#ifdef MAXIMIZATION
    for (int i = 0; i < M; i++) {
        for (int j = i+1; j < M; j++)
            cap[i][j] *= -1;
        b[i] *= -1;
    }
    c *= -1;
#endif

    for (int i = 0; i < M; i++) {
        if (b[i] >= 0) {
            cap[M][i] = b[i];
        } else {
            cap[i][M+1] = -b[i];
            c += b[i];
        }
    }

    int score = GetMaxFlow(M, M+1);
    fill(reached.begin(), reached.end(), 0);
    Augment(M, M+1, INF);
}

```

```

x = VI(M);
for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
score += c;
#ifdef MAXIMIZATION
    score *= -1;
#endif

return score;
}

};

int main() {

    // solver for "Cat vs. Dog" from NWERC 2008

    int numcases;
    cin >> numcases;
    for (int caseno = 0; caseno < numcases; caseno++) {
        int c, d, v;
        cin >> c >> d >> v;

        VVVVI phi(c+d, VVVVI(c+d, VVI(2, VI(2))));
        VVI psi(c+d, VI(2));
        for (int i = 0; i < v; i++) {
            char p, q;
            int u, v;
            cin >> p >> u >> q >> v;
            u--; v--;
            if (p == 'C') {
                phi[u][c+v][0][0]++;
                phi[c+v][u][0][0]++;
            } else {
                phi[v][c+u][1][1]++;
                phi[c+u][v][1][1]++;
            }
        }

        GraphCutInference graph;
        VI x;
        cout << graph.DoInference(phi, psi, x) << endl;
    }

    return 0;
}

```