Nanjing University

ACM-ICPC Codebook 1

Graph Theory

October 14, 2017

Contents

1 Shortest Paths

1.1 Single-source shortest paths

1.1.1 Dijkstra

Dijkstra's algorithm with binary heap.

✘ Can't be performed on graphs with negative weights.

Usage:

`V`	Number of vertices.
`add_edge(e)`	Add edge e to the graph.
`dijkstra(src)`	Calculate SSSP from src.
`d[x]`	distance to x
`p[x]`	last edge to x in SSSP

Time complexity: $O(E \log V)$

```
const int INF = 0x7f7f7f7f;
const int MAXV = 10005;
const int MAXE = 500005;
struct edge{
    int u, v, w;
};

struct graph{
    int V;
    vector<edge> adj[MAXV];
    int d[MAXV];
    edge* p[MAXV];

    void add_edge(int u, int v, int w){
        edge e;
        e.u = u; e.v = v; e.w = w;
        adj[u].push_back(e);
    }

    bool done[MAXV];
    void dijkstra(int src){
        typedef pair<int,int> pii;
        priority_queue<pii, vector<pii>, greater<pii> > q;

        fill(d, d + V + 1, INF);
        d[src] = 0;
        fill(done, done + V + 1, false);
```

```
        q.push(make_pair(0, src));
        while (!q.empty()){
            int u = q.top().second; q.pop();
            if (done[u]) continue;
            done[u] = true;
            rep (i, adj[u].size()){
                edge e = adj[u][i];
                if (d[e.v] > d[u] + e.w){
                    d[e.v] = d[u] + e.w;
                    p[e.v] = &adj[u][i];
                    q.push(make_pair(d[e.v], e.v));
                }
            }
        }
    }
};
```

1.1.2 SPFA

Shortest path faster algorithm. (Improved version of Bellman-Ford algorithm)

This code is used to replace `void dijkstra(int src)`.

✓ Can be performed on graphs with negative weights.
⚠ For some specially constructed graphs, this algorithm is very slow.

Usage:

`spfa(src)`	Calculate SSSP from src.

Requirement:
1.1.1 Dijkstra

Time complexity: $O(kE)$, for most graphs, $k < 2$

```
    // ! This procedure is to replace `dijkstra', and cannot be used alone.
    bool inq[MAXV];
    void spfa(int src){
        queue<int> q;
        fill(d, d + V + 1, INF);
        d[src] = 0;
        fill(inq, inq + V + 1, false);
        q.push(src); inq[src] = true;
        while (!q.empty()){
            int u = q.front(); q.pop(); inq[u] = false;
            rep (i, adj[u].size()){
                edge e = adj[u][i];
                if (d[e.v] > d[u] + e.w){
```

```
                    d[e.v] = d[u] + e.w;
                    p[e.v] = &adj[u][i];
                    if (!inq[e.v])
                        q.push(e.v), inq[e.v] = true;
                }
            }
        }
    }
```

1.2 All-pairs shortest paths (Floyd-Warshall)

Floyd-Warshall algorithm.

✓ Can be performed on graphs with negative weights. To detect negative cycle, one can inspect the diagonal, and the presence of a negative number indicates that the corresponding vertex lies on some negative cycle.
⚠ **Self-loops** and **multiple edges** must be specially judged.
⚠ If the weights of edges might exceed `LLONG_MAX / 2`, the line `(*)` should be added.

Usage:

`init()`	Initialize the distances of the edges from 0 to V.
`floyd()`	Calculate APSP.
`d[i][j]`	distance from i to j

Time complexity: $O(V^3)$

```
const LL INF = LLONG_MAX / 2;
const int MAXV = 1005;
int V;
LL d[MAXV][MAXV];

void init(){
    Rep (i, V){
        Rep (j, V) d[i][j] = INF;
        d[i][i] = 0;
    }
}

void floyd(){
    Rep (k, V)
        Rep (i, V)
            Rep (j, V)
                // ! (*) if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

2 Spanning Tree

2.1 Minimum spanning tree

2.1.1 Kruskal's algorithm

Usage:

`n, m`	The number of vertices and edges, resp.
`edges[]`	Edges of the graph, numbered from 0.
`kruskal()`	Run Kruskal's algorihtm.

Time complexity: $O(E \log E)$

```
const int MAXV = 100005;
const int MAXE = 300005;

int n, m;
struct edge{
    int u, v, w;
    bool operator < (const edge& e) const {
        return w < e.w;
    }
} edges[MAXE];

int p[MAXV];
void init(int num){
    for (int i=1; i<=num; i++) p[i] = i;
}

int parent(int x){
    if (p[x] == x) return x;
    return p[x] = parent(p[x]);
}

bool unite(int u, int v){
    u = parent(u); v = parent(v);
    p[u] = v; return u != v;
}

void kruskal(){
    init(n);
    sort(edges, edges + m);
    int curn = 1;
    for (int i = 0; curn < n; i++){
        if (unite(edges[i].u, edges[i].v)){
            // choose the i-th edge
```

```
            curn++;
        }
    }
}
```

2.1.2 Prim's algorithm, adjacency matrix representation

Calculate minimum spanning tree. The result is represented as a tree rootes at `src`.

Usage:

`adj[i][j]`	Adjacency matrix, indexed from 1.
`prim(src)`	Run Prim's algorihtm from `src`.

Time complexity: $O(V^2)$

```
const int MAXN = 108;

int n;
LL adj[MAXN][MAXN]; // indexed from 1

int prev[MAXN]; // note that, prev[src] = 0
bool done[MAXN];
LL key[MAXN]; // key[v] = adj[prev[v]][v] for v != src when done
void prim(int src){
    Rep (i, n)
        key[i] = LLONG_MAX, done[i] = false;
    key[src] = 0, prev[src] = 0;
    rep (cnt, n){
        LL u, k = LLONG_MAX;
        Rep (i, n)
            if (!done[i] && key[i] < k)
                u = i, k = key[i];
        done[u] = true;
        Rep (v, n)
            if (!done[v] && adj[u][v] < key[v])
                prev[v] = u, key[v] = adj[u][v];
    }
}
```

2.2 Minimum ratio spanning tree

Minimize $\frac{\sum_{e \in ST} cost[e]}{\sum_{e \in ST} dist[e]}$ where ST is a spanning tree.

Usage:

First, build the edges of the graph as the structure shows; then, implement a usual MST algorithm; finally, call `solve()` to get the answer.

```
double k;
struct edge{
    int u, v;
    double cost, dist;
    double w(return cost - dist * w);
    bool operator < (const edge& rhs) const {
        return w() < rhs.w();
    }
};

double mst(){
    // return sum(dist[e])/sum(cost[e]) for all e in mst
}

double solve(){
    k = 1e5; // initial k estimate
    double nxt;
    while (fabs((nxt = mst()) - k)) > 1e-8){ // admissible error
        k = nxt;
    }
    return k;
}
```

2.3 Manhattan distance minimum spanning tree

Usgae:

`add_point(x, y)`	Add point (x, y).
`Manhattan_MST()`	Calculate Manhattan distance minimum spanning tree.

Time complexity: $O(n \log n)$, but constant factor may be large.

```
int V = 0;
struct pt{int id, x, y;};
typedef vector<pt>::iterator vit;
vector<pt> pts;

struct edge{
    int u, v, w;
    bool operator < (const edge& e) const {return w < e.w;}
};
vector<edge> edges;

struct BIT{
```

```
    inline int lowbit(int x) {return x&-x;}
    int N;
    vector<int> tr;
    vector<int> minv;

    BIT(int n){
        tr.resize(N = n + 5);
        minv.resize(N);
        fill(range(tr), INT_MAX);
    }

    int prefmin(int n, int& x){
        LL ans = INT_MAX;
        int v = 0;
        while (n){
            if (tr[n] < ans) ans = tr[n]; v = minv[n];
            n -= lowbit(n);
        }
        x = ans;
        return v;
    }

    void insert(int n, int v, int x){
        while (n < N){
            if (tr[n] > x) tr[n] = x, minv[n] = v;
            n += lowbit(n);
        }
    }
};

struct CMP{
    inline bool operator ()(const pt& lhs, const pt& rhs){
        if (lhs.x == rhs.x) return lhs.y > rhs.y;
        return lhs.x > rhs.x;
    }
} cmp;

const int DIFF = 1020; // ! DIFF > max(x_i, y_i); discretize when necessary
void make_edge(){
    sort(range(pts), cmp);
    BIT bit(DIFF * 2);
    for (vit it = pts.begin(); it != pts.end(); it++){
        int vxy;
        int v = bit.prefmin(it->x - it->y + DIFF, vxy);
        if (v) edges.push_back(edge{it->id, v, vxy - it->x - it->y});
        bit.insert(it->x - it->y + DIFF, it->id, it->x + it->y);
    }
```

```
}

struct UFS{
    int p[10005];
    void init(int num){
        for (int i=1; i<=num; i++) p[i] = i;
    }

    int parent(int x){
        if (p[x] == x) return x;
        return p[x] = parent(p[x]);
    }

    bool unite(int u, int v){
        u = parent(u); v = parent(v);
        p[u] = v; return u != v;
    }
} ufs;

void kruskal(){
    ufs.init(V);
    sort(range(edges));
    int curn = 1;
    for (int i = 0; curn < V; i++){
        if (ufs.unite(edges[i].u, edges[i].v)){
            // choose the i-th edge
            curn++;
        }
    }
}

inline void add_point(int x, int y){pts.push_back(pt {++V, x, y});}

void Manhattan_MST(){
    make_edge();
    for (vit it = pts.begin(); it != pts.end(); it++) swap(it->x, it->y);
    make_edge();
    for (vit it = pts.begin(); it != pts.end(); it++) it->x = -it->x;
    make_edge();
    for (vit it = pts.begin(); it != pts.end(); it++) swap(it->x, it->y);
    make_edge();
    // restore original coordinates
    // for (vit it = pts.begin(); it != pts.end(); it++) it->y = -it->y;
    kruskal();
}
```

3 Depth-first Search

3.1 Strongly connected components, condensation (Tarjan)

Find strongly connected components and compute the component graph.

⚠ The component graph may contain **multiple edges**.

Usage:

`V`	number of vertices
`scc[i]`	the SCC that i belongs to, numbered from 1.
`sccn`	number of SCCs
`find_scc()`	Find all SCCs.
`contract()`	Compute component graph.

Time complexity: $O(V + E)$

```
const int MAXV = 100005;

struct graph{
    vector<int> adj[MAXV];
    stack<int> s;
    int V; // number of vertices
    int pre[MAXV], lnk[MAXV], scc[MAXV];
    int time, sccn;

    void add_edge(int u, int v){
        adj[u].push_back(v);
    }

    void dfs(int u){
        pre[u] = lnk[u] = ++time;
        s.push(u);
        rep (i, adj[u].size()){
            int v = adj[u][i];
            if (!pre[v]){
                dfs(v);
                lnk[u] = min(lnk[u], lnk[v]);
            } else if (!scc[v]){
                lnk[u] = min(lnk[u], pre[v]);
            }
        }
        if (lnk[u] == pre[u]){
            sccn++;
            int x;
            do {
                x = s.top(); s.pop();
```

```
                scc[x] = sccn;
            } while (x != u);
        }
    }

    void find_scc(){
        time = sccn = 0;
        memset(scc, 0, sizeof(scc));
        memset(pre, 0, sizeof(pre));
        Rep (i, V){
            if (!pre[i]) dfs(i);
        }
    }

    vector<int> adjc[MAXV];
    void contract(){
        Rep (i, V)
            rep (j, adj[i].size()){
                if (scc[i] != scc[adj[i][j]])
                    adjc[scc[i]].push_back(scc[adj[i][j]]);
            }
    }
};
```

4 Flow Network

4.1 Maximum flow, Dinic's algorithm

✓ Can be performed on networks with parallel and antiparallel edges.

Usage:

`add_edge(u, v, c)`	Add an edge from u to v with capacity c.
`max_flow(s, t)`	Compute maximum flow from s to t.

Time complexity: For general graph, $O(V^2E)$; for network with unit capacities, $O(\min\{V^{2/3}, E^{1/2}\}E)$; for bipartite network, $O(\sqrt{V}E)$.

```
struct edge{
    int from, to;
    LL cap, flow;
};

const int MAXN = 1005;
struct Dinic {
```

```
    int n, m, s, t;
    vector<edge> edges;
    vector<int> G[MAXN];
    bool vis[MAXN];
    int d[MAXN];
    int cur[MAXN];

    void add_edge(int from, int to, LL cap) {
        edges.push_back(edge{from, to, cap, 0});
        edges.push_back(edge{to, from, 0, 0});
        m = edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }

    bool bfs() {
        memset(vis, 0, sizeof(vis));
        queue<int> q;
        q.push(s);
        vis[s] = 1;
        d[s] = 0;
        while (!q.empty()) {
            int x = q.front(); q.pop();
            for (int i = 0; i < G[x].size(); i++) {
                edge& e = edges[G[x][i]];
                if (!vis[e.to] && e.cap > e.flow) {
                    vis[e.to] = 1;
                    d[e.to] = d[x] + 1;
                    q.push(e.to);
                }
            }
        }
        return vis[t];
    }

    LL dfs(int x, LL a) {
        if (x == t || a == 0) return a;
        LL flow = 0, f;
        for (int& i = cur[x]; i < G[x].size(); i++) {
            edge& e = edges[G[x][i]];
            if(d[x] + 1 == d[e.to] && (f = dfs(e.to, min(a, e.cap-e.flow))) > 0)
               {
                e.flow += f;
                edges[G[x][i]^1].flow -= f;
                flow += f;
                a -= f;
                if(a == 0) break;
```

```
            }
        }
        return flow;
    }

    LL max_flow(int s, int t) {
        this->s = s; this->t = t;
        LL flow = 0;
        while (bfs()) {
            memset(cur, 0, sizeof(cur));
            flow += dfs(s, LLONG_MAX);
        }
        return flow;
    }

    vector<int> min_cut() { // call this after maxflow
        vector<int> ans;
        for (int i = 0; i < edges.size(); i++) {
            edge& e = edges[i];
            if(vis[e.from] && !vis[e.to] && e.cap > 0) ans.push_back(i);
        }
        return ans;
    }
};
```

4.2 MCMF, Ford-Fulkerson with SPFA

✓ Can be performed on networks with parallel and antiparallel edges.

Usage:

`add_edge(u, v, c)`	Add an edge from u to v with capacity c.
`min_cost(s, t, &cost)`	Compute MCMF from s to t. Return the maximum flow, and set `cost` as the minimum cost.
`min_cost(s, t, f, &cost)`	Compute minimum cost flow from s to t with capacity f. Return whether such flow exists, and set `cost` as the minimum cost, if exists.

Time complexity: $O(|f|kE)$.

```
struct edge{
    int from, to;
    int cap, flow;
    LL cost;
};

```

```cpp
const LL INF = LLONG_MAX / 2;
const int MAXN = 5005;
struct MCMF {
    int s, t, n, m;
    vector<edge> edges;
    vector<int> G[MAXN];
    bool inq[MAXN]; // queue
    LL d[MAXN];     // distance
    int p[MAXN];    // previous
    int a[MAXN];    // improvement

    void add_edge(int from, int to, int cap, LL cost) {
        edges.push_back(edge{from, to, cap, 0, cost});
        edges.push_back(edge{to, from, 0, 0, -cost});
        m = edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }

    bool spfa(){
        queue<int> q;
        fill(d, d + MAXN, INF); d[s] = 0;
        memset(inq, 0, sizeof(inq));
        q.push(s); inq[s] = true;
        p[s] = 0; a[s] = INT_MAX;
        while (!q.empty()){
            int u = q.front(); q.pop(); inq[u] = false;
            rep (i, G[u].size()){
                edge& e = edges[G[u][i]];
                if (e.cap > e.flow && d[e.to] > d[u] + e.cost){
                    d[e.to] = d[u] + e.cost;
                    p[e.to] = G[u][i];
                    a[e.to] = min(a[u], e.cap - e.flow);
                    if (!inq[e.to]) q.push(e.to), inq[e.to] = true;
                }
            }
        }
        return d[t] != INF;
    }

    void augment(){
        int u = t;
        while (u != s){
            edges[p[u]].flow += a[t];
            edges[p[u]^1].flow -= a[t];
            u = edges[p[u]].from;
        }
```

```
    }

#ifdef GIVEN_FLOW
    bool min_cost(int s, int t, int f, LL& cost) {
        this->s = s; this->t = t;
        int flow = 0;
        cost = 0;
        while (spfa()) {
            augment();
            if (flow + a[t] >= f){
                cost += (f - flow) * a[t]; flow = f;
                return true;
            } else {
                flow += a[t]; cost += a[t] * d[t];
            }
        }
        return false;
    }
#else
    int min_cost(int s, int t, LL& cost) {
        this->s = s; this->t = t;
        int flow = 0;
        cost = 0;
        while (spfa()) {
            augment();
            flow += a[t]; cost += a[t] * d[t];
        }
        return flow;
    }
#endif
};
```

5 Matching

5.1 Maximum cardinality bipartite matching, Hungarian

Usage:

`init(nx, ny)`	Initialize the algorithm with two parts containing `nx`, `ny` vertices, respectively. The vertices are numbered from 0.
`add(a, b)`	Add an edge from a to b.
`match()`	Compute and return maximum cardinality bipartite matching,.
`mx[], my[]`	The index of the other vertex if matched; otherwise -1.

Time complexity: $O(VE)$.

```
#include <bits/stdc++.h>
using namespace std;

#define rep(i, n) for (int i = 0; i < (n); i++)
#define Rep(i, n) for (int i = 1; i <= (n); i++)
#define range(x) (x).begin(), (x).end()
typedef long long LL;

struct Hungarian{
    int nx, ny;
    vector<int> mx, my;
    vector<vector<int> > e;
    vector<bool> mark;

    void init(int nx, int ny){
        this->nx = nx;
        this->ny = ny;
        mx.resize(nx); my.resize(ny);
        e.clear(); e.resize(nx);
        mark.resize(nx);
    }

    inline void add(int a, int b){
        e[a].push_back(b);
    }

    bool augment(int i){
        if (!mark[i]) {
            mark[i] = true;
            for (int j : e[i]){
                if (my[j] == -1 || augment(my[j])){
                    mx[i] = j; my[j] = i;
                    return true;
                }
            }
        }
        return false;
    }

    int match(){
        int ret = 0;
        fill(range(mx), -1);
        fill(range(my), -1);
        rep (i, nx){
            fill(range(mark), false);
            if (augment(i)) ret++;
```

```
        }
        return ret;
    }
};
```