Nanjing University

ACM-ICPC Codebook 1

# Graph Theory

June 15, 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:** $O(k|E|)$, generally $k < 2$

```
    // ! 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.

| | |
|---|---|
| `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

### 2.1.2 Prim's algorithm

# 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**.

| | |
|---|---|
| `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