NANJING UNIVERSITY

ACM-ICPC Codebook 1

Graph Theory

May 25, 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:

`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: $O(|E| \log |V|)$

```
#include <queue>

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;
            for (int i = 0; i < adj[u].size(); i++){
                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;
            for (int i = 0; i < adj[u].size(); i++){
                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.
⚠ **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(){
    for (int i = 0; i <= V; i++){
        for (int j = 0; j <= V; j++)
            d[i][j] = INF;
        d[i][i] = 0;
    }
}

void floyd(){
    for (int k = 0; k <= V; k++)
        for (int i = 0; i <= V; i++)
            for (int j = 0; j <= V; j++)
                // ! (*) 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 Flow Network