



NANJING UNIVERSITY

ACM-ICPC Codebook 1
Graph Theory

August 24, 2017

Contents

1	Shortest Paths	4
1.1	Single-source shortest paths	4
1.1.1	Dijkstra	4
1.1.2	SPFA	5
1.2	All-pairs shortest paths (Floyd-Warshall)	6
2	Spanning Tree	7
2.1	Minimum spanning tree	7
2.1.1	Kruskal's algorithm	7
2.1.2	Prim's algorithm	8
3	Depth-first Search	8
3.1	Strongly connected components, condensation (Tarjan)	8
4	Flow Network	9

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:

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

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

```

1  const int INF = 0x7f7f7f7f;
2  const int MAXV = 10005;
3  const int MAXE = 500005;
4  struct edge{
5      int u, v, w;
6  };
7
8  struct graph{
9      int V;
10     vector<edge> adj[MAXV];
11     int d[MAXV];
12     edge* p[MAXV];
13
14     void add_edge(int u, int v, int w){
15         edge e;
16         e.u = u; e.v = v; e.w = w;
17         adj[u].push_back(e);
18     }
19
20     bool done[MAXV];
21     void dijkstra(int src){
22         typedef pair<int,int> pii;
23         priority_queue<pii, vector<pii>, greater<pii> > q;
24
25         fill(d, d + V + 1, INF);
26         d[src] = 0;
27         fill(done, done + V + 1, false);

```

```

28     q.push(make_pair(0, src));
29     while (!q.empty()){
30         int u = q.top().second; q.pop();
31         if (done[u]) continue;
32         done[u] = true;
33         rep (i, adj[u].size()){
34             edge e = adj[u][i];
35             if (d[e.v] > d[u] + e.w){
36                 d[e.v] = d[u] + e.w;
37                 p[e.v] = &adj[u][i];
38                 q.push(make_pair(d[e.v], e.v));
39             }
40         }
41     }
42 }
43 };

```

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 $k < 2$

```

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

```

```

14         d[e.v] = d[u] + e.w;
15         p[e.v] = &adj[u][i];
16         if (!inq[e.v])
17             q.push(e.v), inq[e.v] = true;
18     }
19 }
20 }
21 }
```

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 $\text{LLONG_MAX} / 2$, 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>i</i> to <i>j</i>

Time complexity: $O(|V|^3)$

```

1  const LL INF = LLONG_MAX / 2;
2  const int MAXV = 1005;
3  int V;
4  LL d[MAXV][MAXV];
5
6  void init(){
7      Rep (i, V){
8          Rep (j, V) d[i][j] = INF;
9          d[i][i] = 0;
10     }
11 }
12
13 void floyd(){
14     Rep (k, V)
15         Rep (i, V)
16             Rep (j, V)
17                 // ! (*) if (d[i][k] < INF && d[k][j] < INF)
18                     d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
19 }
```

2 Spanning Tree

2.1 Minimum spanning tree

2.1.1 Kruskal's algorithm

Usage:

<code>n, m</code>	The number of vertices and edges, resp.
<code>edges[]</code>	Edges of the graph, numbered from 0.
<code>kruskal()</code>	Run Kruskal's algorithm.

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

```

1  const int MAXV = 100005;
2  const int MAXE = 300005;
3
4  int n, m;
5  struct edge{
6      int u, v, w;
7      bool operator < (const edge& e) const {
8          return w < e.w;
9      }
10 } edges[MAXE];
11
12 int p[MAXV];
13 void init(int num){
14     for (int i=1; i<=num; i++) p[i] = i;
15 }
16
17 int parent(int x){
18     if (p[x] == x) return x;
19     return p[x] = parent(p[x]);
20 }
21
22 bool unite(int u, int v){
23     u = parent(u); v = parent(v);
24     p[u] = v; return u != v;
25 }
26
27 void kruskal(){
28     init(n);
29     sort(edges, edges + m);
30     int curn = 1;
31     for (int i = 0; curn < n; i++){
32         if (unite(edges[i].u, edges[i].v)){
33             // choose the i-th edge

```

```

34         currn++;
35     }
36 }
37 }

```

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**.

Usage:

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

Time complexity: $O(|V| + |E|)$

```

1  const int MAXV = 100005;
2
3  struct graph{
4      vector<int> adj[MAXV];
5      stack<int> s;
6      int V; // number of vertices
7      int pre[MAXV], lnk[MAXV], scc[MAXV];
8      int time, sccn;
9
10     void add_edge(int u, int v){
11         adj[u].push_back(v);
12     }
13
14     void dfs(int u){
15         pre[u] = lnk[u] = ++time;
16         s.push(u);
17         rep (i, adj[u].size()){
18             int v = adj[u][i];
19             if (!pre[v]){
20                 dfs(v);

```



```

21         lnk[u] = min(lnk[u], lnk[v]);
22     } else if (!scc[v]){
23         lnk[u] = min(lnk[u], pre[v]);
24     }
25 }
26 if (lnk[u] == pre[u]){
27     sccn++;
28     int x;
29     do {
30         x = s.top(); s.pop();
31         scc[x] = sccn;
32     } while (x != u);
33 }
34 }
35
36 void find_scc(){
37     time = sccn = 0;
38     memset(scc, 0, sizeof(scc));
39     memset(pre, 0, sizeof(pre));
40     Rep (i, V){
41         if (!pre[i]) dfs(i);
42     }
43 }
44
45 vector<int> adjc[MAXV];
46 void contract(){
47     Rep (i, V)
48         rep (j, adj[i].size()){
49             if (scc[i] != scc[adj[i][j]])
50                 adjc[scc[i]].push_back(scc[adj[i][j]]);
51         }
52 }
53 };

```

4 Flow Network