



NANJING UNIVERSITY

ACM-ICPC Codebook 1

Graph Theory

October 8, 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, adjacency matrix representation	8
2.2	Minimum ratio spanning tree	8
2.3	Manhattan distance minimum spanning tree	9
3	Depth-first Search	12
3.1	Strongly connected components, condensation (Tarjan)	12
4	Flow Network	13
4.1	Maximum flow, Dinic's algorithm	13
4.2	MCMF, Ford-Fulkerson with SPFA	15

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

```

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:

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

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, 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 algoirhtm from `src`.

Time complexity: $O(V^2)$

```

1  const int MAXN = 108;
2
3  int n;
4  LL adj[MAXN][MAXN]; // indexed from 1
5
6  int prev[MAXN]; // note that, prev[src] = 0
7  bool done[MAXN];
8  LL key[MAXN]; // key[v] = adj[prev[v]][v] for v != src when done
9  void prim(int src){
10     Rep (i, n)
11         key[i] = LLONG_MAX, done[i] = false;
12     key[src] = 0, prev[src] = 0;
13     rep (cnt, n){
14         LL u, k = LLONG_MAX;
15         Rep (i, n)
16             if (!done[i] && key[i] < k)
17                 u = i, k = key[i];
18         done[u] = true;
19         Rep (v, n)
20             if (!done[v] && adj[u][v] < key[v])
21                 prev[v] = u, key[v] = adj[u][v];
22     }
23 }

```

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.

```

1  double k;
2  struct edge{
3      int u, v;
4      double cost, dist;
5      double w{return cost - dist * w};
6      bool operator < (const edge& rhs) const {
7          return w() < rhs.w();
8      }
9  };
10
11 double mst(){
12     // return sum(dist[e])/sum(cost[e]) for all e in mst
13 }
14
15 double solve(){
16     k = 1e5; // initial k estimate
17     double nxt;
18     while (fabs((nxt = mst()) - k)) > 1e-8){ // admissible error
19         k = nxt;
20     }
21     return k;
22 }
```

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.

```

1  int V = 0;
2  struct pt{int id, x, y;};
3  typedef vector<pt>::iterator vit;
4  vector<pt> pts;
5
6  struct edge{
7      int u, v, w;
8      bool operator < (const edge& e) const {return w < e.w;}
9  };
10 vector<edge> edges;
11
12 struct BIT{
```

```

13 inline int lowbit(int x) {return x&-x;}
14 int N;
15 vector<int> tr;
16 vector<int> minv;
17
18 BIT(int n){
19     tr.resize(N = n + 5);
20     minv.resize(N);
21     fill(range(tr), INT_MAX);
22 }
23
24 int prefmin(int n, int& x){
25     LL ans = INT_MAX;
26     int v = 0;
27     while (n){
28         if (tr[n] < ans) ans = tr[n]; v = minv[n];
29         n -= lowbit(n);
30     }
31     x = ans;
32     return v;
33 }
34
35 void insert(int n, int v, int x){
36     while (n < N){
37         if (tr[n] > x) tr[n] = x, minv[n] = v;
38         n += lowbit(n);
39     }
40 }
41 };
42
43 struct CMP{
44     inline bool operator()(const pt& lhs, const pt& rhs){
45         if (lhs.x == rhs.x) return lhs.y > rhs.y;
46         return lhs.x > rhs.x;
47     }
48 } cmp;
49
50 const int DIFF = 1020; // ! DIFF > max(x_i, y_i); discretize when necessary
51 void make_edge(){
52     sort(range(pts), cmp);
53     BIT bit(DIFF * 2);
54     for (vit it = pts.begin(); it != pts.end(); it++){
55         int vxy;
56         int v = bit.prefmin(it->x - it->y + DIFF, vxy);
57         if (v) edges.push_back(edge{it->id, v, vxy - it->x - it->y});
58         bit.insert(it->x - it->y + DIFF, it->id, it->x + it->y);
59     }

```

```

60 }
61
62 struct UFS{
63     int p[10005];
64     void init(int num){
65         for (int i=1; i<=num; i++) p[i] = i;
66     }
67
68     int parent(int x){
69         if (p[x] == x) return x;
70         return p[x] = parent(p[x]);
71     }
72
73     bool unite(int u, int v){
74         u = parent(u); v = parent(v);
75         p[u] = v; return u != v;
76     }
77 } ufs;
78
79 void kruskal(){
80     ufs.init(V);
81     sort(range(edges));
82     int curn = 1;
83     for (int i = 0; curn < V; i++){
84         if (ufs.unite(edges[i].u, edges[i].v)){
85             // choose the i-th edge
86             curn++;
87         }
88     }
89 }
90
91 inline void add_point(int x, int y){pts.push_back(pt {++V, x, y});}
92
93 void Manhattan_MST(){
94     make_edge();
95     for (vit it = pts.begin(); it != pts.end(); it++) swap(it->x, it->y);
96     make_edge();
97     for (vit it = pts.begin(); it != pts.end(); it++) it->x = -it->x;
98     make_edge();
99     for (vit it = pts.begin(); it != pts.end(); it++) swap(it->x, it->y);
100    make_edge();
101    // restore original coordinates
102    // for (vit it = pts.begin(); it != pts.end(); it++) it->y = -it->y;
103    kruskal();
104 }

```

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.

△ The vertices should be discretized when necessary.

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

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

```

1 struct edge{
2     int from, to;
3     int cap, flow;
4 };
5

```

```
6  const int MAXN = 50;
7  struct Dinic {
8      int n, m, s, t;
9      vector<edge> edges;
10     vector<int> G[MAXN];
11     bool vis[MAXN];
12     int d[MAXN];
13     int cur[MAXN];
14
15     void add_edge(int from, int to, int cap) {
16         edges.push_back(edge{from, to, cap, 0});
17         edges.push_back(edge{to, from, 0, 0});
18         m = edges.size();
19         G[from].push_back(m-2);
20         G[to].push_back(m-1);
21     }
22
23     bool bfs() {
24         memset(vis, 0, sizeof(vis));
25         queue<int> q;
26         q.push(s);
27         vis[s] = 1;
28         d[s] = 0;
29         while(!q.empty()) {
30             int x = q.front(); q.pop();
31             for(int i = 0; i < G[x].size(); i++) {
32                 edge& e = edges[G[x][i]];
33                 if(!vis[e.to] && e.cap > e.flow) {
34                     vis[e.to] = 1;
35                     d[e.to] = d[x] + 1;
36                     q.push(e.to);
37                 }
38             }
39         }
40         return vis[t];
41     }
42
43     int dfs(int x, int a) {
44         if(x == t || a == 0) return a;
45         int flow = 0, f;
46         for(int& i = cur[x]; i < G[x].size(); i++) {
47             edge& e = edges[G[x][i]];
48             if(d[x] + 1 == d[e.to] && (f = dfs(e.to, min(a, e.cap-e.flow))) > 0)
49                 {
50                     e.flow += f;
51                     edges[G[x][i]^1].flow -= f;
52                     flow += f;
53                 }
```

```

52         a -= f;
53         if(a == 0) break;
54     }
55 }
56 return flow;
57 }
58
59 int max_flow(int s, int t) {
60     this->s = s; this->t = t;
61     int flow = 0;
62     while(bfs()) {
63         memset(cur, 0, sizeof(cur));
64         flow += dfs(s, INT_MAX);
65     }
66     return flow;
67 }
68
69 vector<int> min_cut() { // call this after maxflow
70     vector<int> ans;
71     for(int i = 0; i < edges.size(); i++) {
72         edge& e = edges[i];
73         if(vis[e.from] && !vis[e.to] && e.cap > 0) ans.push_back(i);
74     }
75     return ans;
76 }
77 };

```

4.2 MCMF, Ford-Fulkerson with SPFA

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

Usage:

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

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

```

1 struct edge{
2     int from, to;
3     int cap, flow;
4     LL cost;

```

```

5  };
6
7  const LL INF = LLONG_MAX / 2;
8  const int MAXN = 5005;
9  struct MCMF {
10     int s, t, n, m;
11     vector<edge> edges;
12     vector<int> G[MAXN];
13     bool inq[MAXN]; // queue
14     LL d[MAXN];    // distance
15     int p[MAXN];   // previous
16     int a[MAXN];   // improvement
17
18     void add_edge(int from, int to, int cap, LL cost) {
19         edges.push_back(edge{from, to, cap, 0, cost});
20         edges.push_back(edge{to, from, 0, 0, -cost});
21         m = edges.size();
22         G[from].push_back(m-2);
23         G[to].push_back(m-1);
24     }
25
26     bool spfa(){
27         queue<int> q;
28         fill(d, d + MAXN, INF); d[s] = 0;
29         memset(inq, 0, sizeof(inq));
30         q.push(s); inq[s] = true;
31         p[s] = 0; a[s] = INT_MAX;
32         while (!q.empty()){
33             int u = q.front(); q.pop(); inq[u] = false;
34             rep (i, G[u].size()){
35                 edge& e = edges[G[u][i]];
36                 if (e.cap > e.flow && d[e.to] > d[u] + e.cost){
37                     d[e.to] = d[u] + e.cost;
38                     p[e.to] = G[u][i];
39                     a[e.to] = min(a[u], e.cap - e.flow);
40                     if (!inq[e.to]) q.push(e.to), inq[e.to] = true;
41                 }
42             }
43         }
44         return d[t] != INF;
45     }
46
47     void augment(){
48         int u = t;
49         while (u != s){
50             edges[p[u]].flow += a[t];
51             edges[p[u]^1].flow -= a[t];

```



```
52         u = edges[p[u]].from;
53     }
54 }
55
56 #ifdef GIVEN_FLOW
57     bool min_cost(int s, int t, int f, LL& cost) {
58         this->s = s; this->t = t;
59         int flow = 0;
60         cost = 0;
61         while (spfa()) {
62             augment();
63             if (flow + a[t] >= f){
64                 cost += (f - flow) * a[t]; flow = f;
65                 return true;
66             } else {
67                 flow += a[t]; cost += a[t] * d[t];
68             }
69         }
70         return false;
71     }
72 #else
73     int min_cost(int s, int t, LL& cost) {
74         this->s = s; this->t = t;
75         int flow = 0;
76         cost = 0;
77         while (spfa()) {
78             augment();
79             flow += a[t]; cost += a[t] * d[t];
80         }
81         return flow;
82     }
83 #endif
84 };
```