第 二 周 模 板 总 结

2015.7.24

By Mango Gao

0. Dijkstra - 邻接矩阵
1. Dijkstra - 邻接表(数组)
2. Dijkstra - 邻接表(向量)
3. Dijkstra - 优先队列
4. Bellman-Ford
5. SPFA
6. Floyd
7. 并查集
8. 最小生成树
9. 拓扑排序 - 邻接表
A. 拓扑排序 - 邻接矩阵

0. Dijkstra - 邻接矩阵

```
const int MAXN = "Edit";// 点数最大值
const int INF = 0x3F3F3F3F;
int G[MAXN][MAXN];     // G[x][y] ＝ x 到 y 的距离
bool vis[MAXN];            // vis[x] = 点 x 是否访问过
int dis[MAXN];       // dis[x] = 出发点到点 x 的最短距离
// 针对存图前的初始化
void init(int n) {
    memset(G, 0x3F, sizeof(G));
}
// 添加边: (u)>---w--->(v)
void add_edge(int u, int v, int w) {
    G[u][v] = min(G[u][v], w);
}
// 求最短路: (s)>--->...>--->(n)
void Dijkstra(int s, int n) {
    // 针对遍历前的初始化
    memset(vis, 0, sizeof(vis));
    memset(dis, 0x3F, sizeof(dis));
    dis[s] = 0;
    // 开始遍历整张图
    for (int i = 0; i < n; i++) {
        int x; // 即将遍历的点 x
        int min_dis = INF; // 未访问点中的最短距离
        // 在剩余点中寻找路程最短的
        for (int j = 0; j < n; j++) {
            if (!vis[j] && dis[j] <= min_dis) {
                x = j;
                min_dis = dis[j];
            }
        }
        vis[x] = 1; // 访问找到的最短点
        // 对于所有 x 可以到达的点 j, 更新 dis[j]
        // 如果不能到达, G[x][j]=INF, 不会更新
        for (int j = 0; j < n; j++)
            dis[j] = min(dis[j], dis[x] + G[x][j]);
    }
    // 此时 dis[x] = 出发点到点 x 的最短距离 x:[0,n)
    // 若 dis[x] == INF, 说明无法从出发点走到
}
```

1. Dijkstra - 邻接表(数组)

```
const int MAX_N = "Edit";      // 点数最大值
const int MAX_E = "Edit";      // 边数最大值
const int INF = 0x3F3F3F3F;
// 为了节约空间，将边的信息无序地储存在数组中，并依靠 Head[]和 Next[]联系起来
int tot;                  // 把每条边的信息按输入顺序存在各个数组中
int Head[MAX_N];  // Head[x] = 点 x 最后添加的边 t 的所有信息，储存在这个位置, -1 为没有边
int Next[MAX_E];   // Next[t] = 每条边 t 的下一条同起点的边的信息，储存在这个位置, -1 为没有下一条边
int To[MAX_E];            // To[t] = 每条边 t 指向的点的编号
int W[MAX_E];             // W[t] = 每条边 t 的长度
int vis[MAX_N];           // vis[x] = 点 x 是否访问过
int dis[MAX_N];           // dis[x] = 出发点到点 x 的最短距离
// 针对存图前的初始化
void init() {
    tot = 0;          // 当前储存了 tot 条边
    memset(Head, -1, sizeof(Head));   // 每个点初始没有边
}
// 添加边: (u)>---w--->(v)
void add_edge(int u, int v, int d) {
    // 现在添加第 tot 条边的信息
    W[tot] = d;           // 长度为 d
    To[tot] = v;      // 指向点 v
    // 目前在点 u 出发的边中，最后一条的储存位置是 Head[u]
    // 添加第 tot 条后，第 tot 条成为新的 Head[u]
    // 第 tot 条的下一条边指向原来的 Head[u]
    Next[tot] = Head[u];
    Head[u] = tot; // 第 tot 条成为新的 Head[u]
    tot++;      // 下次添加第 tot+1 条边
}
// 求最短路: (s)>--->...>--->(n)
void Dijkstra(int s, int n) {
    // 针对遍历前的初始化
    memset(vis, 0, sizeof(vis));
    memset(dis, 0x3F, sizeof(dis));
    dis[s] = 0;
    // 开始遍历整张图
    for (int i = 0; i < n; i++) {
        int x; // 即将遍历的点 x
        int min_dis = INF; // 未访问点中的最短距离
        // 在剩余点中寻找路程最短的
```

```
            for (int j = 0; j < n; j++) {
                if (!vis[j] && dis[j] <= min_dis) {
                    x = j;
                    min_dis = dis[j];
                }
            }
            vis[x] = 1; // 访问找到的最短点
            // 点 x 最后添加的一条边的信息，在别的数组中储存位置是 j = Head[x]
            // 这条边的下一条边，储存位置是 Next[j]
            // Next[j] == -1 时，表示下面没有更多的边，停止索引
            for (int j = Head[x]; j != -1; j = Next[j]) {
                // 正在访问储存在 j 位置的边的信息
                int y = To[j];    // 这条边指向点 y: (x)>---W[j]--->(y)
                // 更新到点 y 的最短路 dis[y]
                dis[y] = min(dis[y], dis[x] + W[j]);
            }
        }
        // 此时 dis[x] = 出发点到点 x 的最短距离 x:[0,n)
        // 若 dis[x] == INF，说明无法从出发点走到
}
```

2. Dijkstra - 邻接表(向量)

```
const int MAXN = "Edit"; // 点数最大值
const int INF = 0x3F3F3F3F;
vector<int> G[MAXN];    // G[x] = 点 x 所指向的点集
vector<int> GW[MAXN]; // GW[x] = 对应于 G[x]中的每个点的距离
bool vis[MAXN];             // vis[x] = 点 x 是否访问过
int dis[MAXN];        // dis[x] = 出发点到点 x 的最短距离
// 针对存图前的初始化
void init(int n) {
    for (int i = 0; i < n; i++) {
        G[i].clear();
        GW[i].clear();
    }
}
// 添加边: (u)>---w--->(v)
void add_edge(int u, int v, int w) {
    G[u].push_back(v);
    GW[u].push_back(w);
}
// 求最短路: (s)>--->...>--->(n)
void Dijkstra(int s, int n) {
    // 针对遍历前的初始化
```

```
    memset(vis, false, sizeof(vis));
    memset(dis, 0x3F, sizeof(dis));
    dis[s] = 0;
    // 开始遍历整张图
    for (int i = 0; i < n; i++) {
        int x; // 即将遍历的点 x
        int min_dis = INF; // 未访问点中的最短距离
        // 在剩余点中寻找路程最短的
        for (int j = 0; j < n; j++) {
            if (!vis[j] && dis[j] <= min_dis) {
                x = j;
                min_dis = dis[j];
            }
        }
        vis[x] = true; // 访问找到的最短点
        // G[x] 为 x 点出发的所有边可以到达的点集, GW[x] 为对应的长度集合
        for (int j = 0; j < (int)G[x].size(); j++) {
            int y = G[x][j]; // 点集中第 j 个点为 y
            int w = GW[x][j];    // x 到 y 的距离为 w
            dis[y] = min(dis[y], dis[x] + w);
        }

    }
    // 此时 dis[x] = 出发点到点 x 的最短距离 x:[0,n)
    // 若 dis[x] == INF, 说明无法从出发点走到
}
```

3. Dijkstra - 优先队列

```
const int MAXN = "Edit";// 点数最大值
const int INF = 0x3F3F3F3F;
typedef pair<int, int> PII;
typedef vector<PII> VII;
VII G[MAXN];        // G[x] = 点 x 所指向的点集, 包含长度信息
int vis[MAXN];      // vis[x] = 点 x 是否访问过
int dis[MAXN];      // dis[x] = 出发点到点 x 的最短距离
// 针对存图前的初始化
void init(int n) {
    for (int i = 0; i < n; i++)
        G[i].clear();
}
```

```
// 添加边: (u)>---w--->(v)
void add_edge(int u, int v, int w) {
    // pair 优先比较第一个数据的大小, 因此距离放第一个
    G[u].push_back(make_pair(w, v));
}
// 求最短路: (s)>--->...>--->(n)
void Dijkstra(int s, int n) {
    // 针对遍历前的初始化
    memset(vis, 0, sizeof(vis));
    memset(dis, 0x3F, sizeof(dis));
    dis[s] = 0;
    // 优先队列(小), 每次优先弹出距离最短的点
    // 声明格式: priority_queue<数据类型,容器类型,优先大/小> 变量名称;
    priority_queue<PII, VII, greater<PII> > PQ;
    PQ.push(make_pair(dis[s], s));      // 起始点入队(距离放前面)
    while (!PQ.empty()) {
        PII t = PQ.top();      // 优先队列 PQ.top()  普通队列 Q.front()
        int x = t.second;      // 每次取最小的点的编号
        PQ.pop();              // 最小点出列
        if (vis[x]) continue;
        vis[x] = 1;
        // 下面遍历点 x 所指向的点集
        for (int i = 0; i < (int)G[x].size(); i++) {
            int y = G[x][i].second;     // 取出一点 y
            int w = G[x][i].first;  // (x)>---w--->(y)
            // 未访问过 y, 且需要更新最短距离时
            if (!vis[y] && dis[y] > dis[x] + w) {
                dis[y] = dis[x] + w;
                PQ.push(make_pair(dis[y], y));      // 新的点入队
            }
        }
    }
    // 此时  dis[x] =  出发点到点 x 的最短距离  x:[0,n)
    // 若  dis[x] == INF, 说明无法从出发点走到
}
```

4. Bellman-Ford

```
const int MAX_N = "Edit";      // 点数最大值
const int MAX_E = "Edit";      // 边数最大值
const int INF = 0x3F3F3F3F;
int From[MAX_E];   //
int To[MAX_E];              // 第 i 条边: From[i]>--->W[i]>--->To[i]
int W[MAX_E];               //
```

```
int dis[MAX_N];            // dis[x] =  出发点到点 x 的最短距离
int tot;                   // 一共储存了 tot 条边的信息
bool Exist;                // 存在负权回路时 Exist = 1  不存在为 0
// 针对存图前的初始化
void init() {
    tot = 0;
}
// 添加边: (u)>---d--->(v)
void add_edge(int u, int v, int d) {
    // 现在添加第 tot 条边
    From[tot] = u;
    To[tot] = v;
    W[tot++] = d; // 最后 tot+1
}
// 求最短路: (s)>--->...>--->(n)
void Bellman_Ford(int s, int n) {
    // 针对遍历前的初始化
    memset(dis, 0x3F, sizeof(dis));
    dis[s] = 0;
    // dis[s] = 0;
    // 进行最多 n-1 次松弛操作
    for (int k = 0; k < n - 1; k++) {
        bool relaxed = false;      // 假定这次没有进行松弛操作
        // 对于每条边依次进行松弛操作
        for (int i = 0; i < tot; i++) {
            int x = From[i];//
            int y = To[i];         // (x)>--->W[i]>--->(y)
            // 需要松弛时，松弛并标记
            if (dis[y] > dis[x] + W[i]) {
                dis[y] = dis[x] + W[i];
                relaxed = true;
            }
        }
        if (!relaxed) break;  // 这次没有进行松弛时，后续无需继续松弛
    }
    // 判断是否存在负权回路
    Exist = 0;
    for (int i = 0; i < tot && !Exist; i++)
        if (dis[To[i]] > dis[From[i]] + W[i])
            Exist = 1;  // 如果还可以松弛，说明存在负权回路
}
```

5. SPFA

```
const int MAXN = "Edit";
const int INF = 0x3F3F3F3F;
vector<pair<int, int> > G[MAXN];  // G[x] =  点 x 所指向的点集，包含长度信息
bool vis[MAXN];           // vis[x] =  点 x 是否访问过
int dis[MAXN];       // dis[x] =  出发点到点 x 的最短距离
int inqueue[MAXN];       // inqueue[x] =  点 x 入队次数，超过 n 次为存在负权回路
bool Exist;                  //  存在负权回路时 Exist = 1  不存在为 0
//  针对存图前的初始化
void init(int n) {
      for (int i = 0; i < n; i++)
            G[i].clear();
      Exist = 0;
}
//  添加边: (u)>---w--->(v)
void add_edge(int u, int v, int w) {
      G[u].push_back(make_pair(v, w));
}
//  求最短路: (s)>--->...>--->(n)
void SPFA(int s, int n) {
      //  针对遍历前的初始化
      memset(vis, 0, sizeof(vis));
      memset(dis, 0x3F, sizeof(dis));
      memset(inqueue, 0, sizeof(inqueue));
      dis[s] = 0;
      //  待优化的节点入队
      queue<int> q;
      q.push(s);
      //  不断处理更新待优化节点的最短路
      while (!q.empty()) {
            int x = q.front();
            q.pop();
            vis[x] = false;   //  点 x 此时不在队列中
            //  遍历所有 x 指向的点
            for (int i = 0; i < G[x].size(); i++) {
                  int y = G[x][i].first;
                  int w = G[x][i].second;
                  //  如果需要更新最短路
                  if (dis[y] > dis[x] + w) {
                        //  更新最短路
                        dis[y] = dis[x] + w;
                        //  更新后的节点如果不在队列中
```

```
                    if (!vis[y]) {
                        // 加入队列并标记、计数
                        q.push(y);
                        vis[y] = true;
                        // 每个点最多更新 n-1 次，超过则说明存在负权回路
                        if (++inqueue[y] >= n) Exist = 1;
                    }
                }
            }
            if (Exist) break;
        }
}
```

6. Floyd

```
const int MAXN = "Edit";
const int INF = 0x3F3F3F3F;
int G[MAXN][MAXN];    // G[i][j] = i 到 j 的最短路
// 针对存图前的初始化
void init(int n) {
    memset(G, 0x3F, sizeof(G));
    for (int i = 0; i < n; i++)G[i][i] = 0;
}
// 添加边: (u)>---w--->(v)
void add_edge(int u, int v, int w) {
    G[u][v] = min(G[u][v], w);
}
// 递推求任意两点间最短路
void Floyd(int n) {
    // 以 k 为中间点，判断 i 从 k 到 j 是否更短
    for (int k = 0; k < n; k++)
        // 下面遍历所有的两点的组合
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                // i->j 和 i->k->j 之中取最短
                G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
}
```

7. 并查集

```
const int MAXN = "Edit";
int Father[MAXN];
int Rank[MAXN];
void init() {
	for (int i = 0; i <= n; i++) {
		Father[i] = i;	// 每个节点的根节点都是自己
		Rank[i] = 0;	// 每个节点自己的集合高度为 0
	}
}
int find(int x) {
	// 递归过程中所有的 Father[i]
	// 最终都为最后 return 的 Father[x]
	if (Father[x] != x) Father[x] = find(Father[x]);
	return Father[x];
}
int unite(int x, int y) {
	// 找到 x 和 y 各自的根节点
	x = find(x);
	y = find(y);
	if (x == y) return 0; // 相同则无需并集
	if (Rank[x] < Rank[y]) {
		Father[x] = y;
	} else {
		Father[y] = x;
		if (Rank[x] == Rank[y]) Rank[x]++;
	}
	return 1;
}
bool same(int x, int y) {
	return find(x) == find(y);
}
```

8. 最小生成树

(此处应有并查集模板)

```
typedef pair<int, int> PII;
vector<pair<int, PII> > G;
void init_G() {
	G.clear();
}
void add_edge(int u, int v, int d) {
	G.push_back(make_pair(d, make_pair(u, v)));
}
int Kruskal(int n) {
	init(n);
	sort(G.begin(), G.end());
	int m = G.size();
	int num = 0, ret = 0;
	for (int i = 0; i < m; i++) {
		pair<int, PII> p = G[i];
		int x = p.second.first;
		int y = p.second.second;
		int d = p.first;
		if (unite(x, y)) {
			num++;
			ret += d;
		}
		if (num == n - 1) break;
	}
	return ret;
}
```

9. 拓扑排序 - 邻接矩阵

```
const int MAXN = "Edit";
int Ans[MAXN];          // 存放拓扑排序结果
int G[MAXN][MAXN];      // 存放图信息
int deg[MAXN];          // 存放点入度信息
// 存图前初始化
void init() {
    memset(G, 0, sizeof(G));
    memset(deg, 0, sizeof(deg));
    memset(Ans, 0, sizeof(Ans));
}
// 添加 U->V 的有向边
void add_edge(int u, int v) {
    if (G[u][v]) return;
    G[u][v] = 1;
    deg[v]++;
}
// 直接调用，如需判断有环, 0 有 1 没有
bool Toposort(int n) {
    int tot = 0;
    queue<int> que;
    for (int i = 0; i < n; ++i)
        if (deg[i] == 0) que.push(i);
    while (!que.empty()) {
        int v = que.front(); que.pop();
        Ans[tot++] = v;
        for (int i = 0; i < n; ++i)
            if (G[v][i] == 1)
                if (--deg[t] == 0) que.push(t);
    }
    if (tot < n) return false;
    return true;
}
```

A. 拓扑排序 - 邻接表

```
const int MAXN = "Edit";
typedef pair<int, int> PII;
int Ans[MAXN];                  // 存放拓扑排序结果
vector<int> G[MAXN];   // 邻接表
int deg[MAXN];                  // 记录点的入度
map<PII, bool> S;          // 用于判断重边
// 存图前初始化
void init(int n) {
    S.clear();
    for (int i = 0; i < n; i++)G[i].clear();
    memset(deg, 0, sizeof(deg));
    memset(Ans, 0, sizeof(Ans));
}
// 添加 U->V 的有向边
void add_edge(int u, int v) {
    // 判断重边
    if (S[make_pair(u, v)]) return;
    G[u].push_back(v);
    S[make_pair(u, v)] = 1;
    deg[v]++;
}
// 直接调用，如需判断有环, 0 有 1 没有
bool Toposort(int n) {
    int tot = 0;
    queue<int> que;
    // 入度为 0 的点入队
    for (int i = 0; i < n; ++i)
        if (deg[i] == 0) que.push(i);
    while (!que.empty()) {
        int v = que.front(); que.pop();
        Ans[tot++] = v;
        // 更新与该点相关的点的入度
        for (int i = 0; i < G[v].size(); ++i) {
            int t = G[v][i];
            // 若发现新的入度为 0 的点，入队
            if (--deg[t] == 0) que.push(t);
        }
    }
    if (tot < n) return false;
    return true;
}
```