图论

1.1链式前向星

/\*

可以保存任意形式的图，常用于网络流

使用链表存储有向图的边

info[i]为节点i的边集所对应的链表的头指针，next[j]为第j条边的指向下一条边的指针，to[j]表示第j条边所指向的节点编号。

\*/

struct graph {

typedef vector<int> VI;

VI info, next, to;

//初始化图为n个点，m条边

graph(int n = 0, int m = 0) : to(0), next(0) {

info.resize(n);

next.resize(m);

to.resize(m);

}

//返回边的数量

int edge\_size() {

return to.size();

}

//返回值为最大点的编号+1

int vertex\_size() {

return info.size();

}

void expand(int i) {

if (info.size() < i + 1) info.resize(i + 1);

}

//添加一条i到j的边

void add(int i, int j) {

expand(i), expand(j);

to.push\_back(j);

next.push\_back(info[i]);

info[i] = to.size() - 1;

}

//删除最后一次添加的边

void del\_back() {

int i;

for (i = 0; i < info.size(); i ++) {

if (info[i] == to.size() - 1) {

info[i] = next.back();

break;

}

to.pop\_back();

next.pop\_back();

}

}

void clear() {

info.clear();

next.resize(0);

to.resize(0);

}

};

1.2割点与桥

1.3强连通分量

1.4tarjan求强连通分量

1.5拓扑排序

1.6 2sat

2.1 Dijkstra

const int maxn = 100010;

const int INF = 0x3f3f3f3f;

int cost[maxn][maxn]; //e[u][v]的权值

int d[maxn]; //顶点s出发的最短距离

bool used[maxn]; //已经使用过的图

int n; //顶点数

//求从起点s出发到各个顶点的最短距离

//邻接矩阵实现dijkstra，复杂度o(n^2)

//顶点从1到n

void dijkstra(int s) {

fill(d, d + n + 1, INF);

fill(used, used + n + 1, false);

d[s] = 0;

while (true) {

int v = -1;

for (int u = 0; u < n; u ++) {

if (!used[u] && (v == -1 || d[u] < d[v])) v = u;

}

if (v == -1) break;

used[v] = true;

for (int u = 0; u < n; u ++) {

d[u] = min(d[u], d[v] + cost[v][u]);

}

}

}

//邻接表实现实用数据结构优化（优先队列）,o(m \* logn)

struct edge { int to, cost; };

typedef pair <int, int> P; //first是最短距离，second是顶点的编号

vector <edge> G[maxn];

void dijkstra\_queue(int s) {

//通过指定参数，优先队列按照first从小到大的顺序

priority\_queue <P, vector <P>, greater <P> > que;

fill(d, d + n + 1, INF);

d[s] = 0;

que.push(P(0, s));

while (!que.empty()) {

P p = que.top(); que.pop();

int v = p.second;

if (d[v] < p.first) continue;

for (int i = 0; i < G[v].size(); i ++) {

edge e = G[v][i];

if (d[e.to] > d[v] + e.cost) {

d[e.to] = d[v] + e.cost;

que.push(P(d[e.to], e.to));

}

}

}

}

2.2 SPFA

2.3 Ford算法

const int maxm = 10010;

const int maxn = 10010;

const int INF = 0x3f3f3f3f;

struct edge { int from, to, cost; }; //form到to的边，权值为cost

edge es[maxm]; //边

int d[maxn]; //最短距离

int n, m; //顶点数，边数

//求解从s出发到所有点的最短距离,O(m \* n),无负圈

void shortest\_path(int s) {

for (int i = 0; i < n; i ++) d[i] = INF;

d[s] = 0;

while (true) {

bool update = false;

for (int i = 0; i < m; i ++) {

edge e = es[i];

if (d[e.from] != INF && d[e.to] > d[e.from] + e.cost) {

d[e.to] = d[e.from] + e.cost;

update = true;

}

}

if (!update) break;

}

}

2.4 floyd-warshall

const int maxn = 1001;

const int INF = 0x3f3f3f3f;

int d[maxn][maxn]; //d[u][v]表示权值

int n; //顶点数

//o(v^3),使用dp

void warshall\_froyd() {

for (int k = 0; k < n; k ++) {

for (int i = 0; i < n; i ++) {

for (int j = 0; j < n; j ++) {

d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

}

}

}

}

2.5无环图最短路

2.6第k短路

2.7欧拉回路

2.8混合欧拉回路

3.1匹配问题匈牙利算法

3.2 hopcroft-Karp算法

3.3 KM算法

3.4一般图最大匹配

4.1 LCA

4.2 Prim

4.3 kruskal

//无向图,o(m \* logn)

const int maxm = 100010;

const int maxn = 100010;

int n, m;

int u[maxm], v[maxm], w[maxm];

int r[maxm];

int p[maxn];

int cmp(const int i, const int j) { return w[i] < w[j]; }

int find(int x) { return p[x] == x ? x : p[x] = find(p[x]); } //并查集的查找

int kruskal() {

int ans = 0;

for (int i = 0; i < n; i ++) p[i] = i; //初始化并查集

for (int i = 0; i < m; i ++) r[i] = i; //初始化边序号

sort(r, r + m, cmp); //边排序

for (int i = 0; i < m; i ++) {

int e = r[i];

int x = find(u[e]);

int y = find(v[e]);

if (x != y) { ans += w[e]; p[x] = y; }

}

return ans;

}

4.4单度限制最小生成树

4.5最小树形图

4.6最优比例生成树

4.7树的直径

5.1最大流Dinic算法

5.2最小割

5.3无向图最小割

5.4有上下界的网络流

5.5费用流

6.1