# 7. 图论

## 7.1. 图的存储

### 7.1.1 邻接矩阵

Description:  
邻接矩阵就是一个二维数组 数组下表表示点的编号 数组数值为点之间的距离 可以存有向图，也可以存无向图 空间复杂度高，视情况使用 矩阵初始化为无穷大

//初始化为-1  
int mp[m][n];  
memset(mp, -1, sizeof(mp));

### 7.1.2 邻接表

Description:  
使用多个链表表示点和链表内的点相连 所占空间少

map<int,list<int>>mp;  
//不如stl的存图方式

### 7.1.3 链式前向星

Description:广泛使用的存图方法,常数优秀

struct EDGE{int to,next,val;}e[M];  
int head[N],cur;  
void init(void)  
{  
 memset(head,-1,sizeof head);  
 cur=0;return;  
}  
void add(int u,int v,int w)  
{  
 e[cur]={v,head[u],w};//lamda表达式  
 head[u]=cur++;return;  
}  
//遍历  
for(int i=head[now],to,:~i;i=e[i].next){to=e[i].to;}

### 7.1.4 STL存图

Description:  
最方便的，没有之一  
可以使用unordered优化复杂度 因为map不会具体的保存所有的数，所以占用空间少 但是要注意，如果存在删除操作，他还是会占用出乎意料的空间 其次，由于使用了STL模板，程序的常数会略大

map<int,map<int,int>>mp;  
for(auto x:mp)  
{  
 for(auto y:x.second){}  
}

## 7.2. 图的遍历

### 7.2.1 深度优先搜索

Description:  
优先对子节点进行访问，同级节点先后访问

map<int,map<int,int>>mp;  
map<int,bool>visited;  
  
int dfs(int p)  
{  
 visited[p]=true;  
 for(auto i:mp[p])  
 {  
 if(!visited[i.first])  
 {  
 dfs(i.first);  
 }  
 }  
}

### 7.2.2 广度优先搜索

map<int, map<int,int>> graph;  
queue<int> q;  
map<int,bool> visited;  
int bfs()  
{  
 q.push(s);  
 visited[s]=true;  
 while(!q.empty())  
 {  
 int p=q.front();  
 q.pop();  
 for(auto i:graph[p])  
 {  
 if(!visited[i.first])  
 {  
 visited[i.first]=true;  
 q.push(i.first);  
 }  
 }  
 }  
}

### 7.2.3.优先队列广搜

description:  
使用优先队列优化搜索次序  
和广搜一样,注意重载运算符

## 7.3. 最短路

### 7.3.1 朴素Dijkstra

Description:  
走当前可到达但未到达最近的点或该点距离会被更新为更小

map<int,map<int,int>>graph;  
queue<int>q;  
map<int,int>dist;  
map<int,bool>visited;  
int dijkstra(int s) {  
 q.push(s);  
 dist[s] = 0;  
 visited[s] = true;  
 while (!q.empty()) {  
 int u = q.front();  
 q.pop();  
 for (int v : graph[u]) {  
 if (!visited[v]) {  
 q.push(v);  
 visited[v] = true;  
 dist[v] = dist[u] + graph[u][v];  
 }  
 }  
 }  
}

### 7.3.2 Dijkstra堆优化

description:  
优先队列优化dijkstra

map<int, map<int,int>> graph;  
priority\_queue<node>q;  
map<int,bool> visited;  
struct node  
{  
 int x;  
 int wei;  
 inline bool operator<(const node& rhs) const  
 {  
 return wei > rhs.wei;  
 }  
}  
int dijkstra(int s) {  
 q.push(node{s,0});  
 visited[s] = true;  
 while (!q.empty()) {  
 node u = q.top();  
 q.pop();  
 for (int v : graph[u.x]) {  
 if (!visited[v]) {  
 q.push(node{v,dist[u.x] + graph[u.x][v]});  
 visited[v] = true;  
 }  
 }  
 }  
}

### 7.3.3 Bellman-Ford

description: 关于SPFA 它死了

map<int,map<int,int>>mp;  
map<int,int>dist;  
queue<int>q;  
map<int,bool>visited;  
q.push(s);  
visited[s] = true;  
int SPFA(int s) {  
 dist[s] = 0;  
 while (!q.empty()) {  
 int u = q.front();  
 q.pop();  
 for (auto v : mp[u]) {  
 if (dist[v.first] > dist[u]+v.second) {  
 dist[v.first] = dist[u]+v.second;  
 if (!visited[v.first]) {  
 q.push(v.first);  
 visited[v.first] = true;  
 }  
 }  
 }  
 }  
}

### 7.3.4 Floyd

description:  
多源最短路，适合邻接矩阵存图  
等效于进行n遍Bellman-Ford  
复杂度

const int INF = 0x3f3f3f3f;  
const int MAXN = 100;  
int n, m;  
int mp[MAXN][MAXN];  
int floyd() {  
 for (int k = 1; k <= n; ++k) {  
 for (int i = 1; i <= n; ++i) {  
 for (int j = 1; j <= n; ++j) {  
 if(i!=j)mp[i][j] = min(mp[i][j], mp[i][k] + mp[k][j]);//很像矩阵乘法  
 }  
 }  
 }  
}

### 7.3.5 Johnson

description:  
使用超节点去除负权边  
求负权图多源最短路等, STL版复杂度约为

//STL版本  
const int INF = 0x3f3f3f3f;  
const int MAXN = 100;  
map<int,map<int,int>>mp;  
map<int,int>dist;  
queue<int>q;  
map<int,bool>visited;  
  
int SPFA(int s) {  
 q.push(s);  
 visited[s] = true;  
 dist[s] = 0;  
 while (!q.empty()) {  
 int u = q.front();  
 q.pop();  
 for (int v : mp[u]) {  
 if (dist[v.first] > dist[u]+v.second) {  
 dist[v.first] = dist[u]+v.second;  
 if (!visited[v.first]) {  
 q.push(v.first);  
 visited[v.first] = true;  
 }  
 }  
 }  
 }  
}  
int johnson(int s) {  
 int u = n+1;  
 for (int i = 1; i <= n; ++i) {  
 mp[i][u] = 0;  
 mp[u][i] = INF;  
 }  
 for (int i = 1; i <= n; ++i) {  
 for (int j = 1; j <= n; ++j) {  
 mp[i][j] = min(mp[i][j], mp[i][u]+mp[u][j]);  
 }  
 }  
 SPFA(u);  
 for (int i = 1; i <= n; ++i) {  
 for (int j = 1; j <= n; ++j) {  
 mp[i][j] = min(mp[i][j], mp[i][u]+mp[u][j]);  
 }  
 }  
}

## 7.4. 最小生成树

### 7.4.1 Prim

description:  
过程类似于Dijkstra，

map<int, map<int,int>> graph;  
map<int,int> dis;  
map<int,bool> visited;  
//返回最小生成树的边权和  
int prim(int s) {  
 int res = 0;  
 for (int i = 1; i <= n; ++i) {  
 dis[i] = INF;  
 visited[i] = false;  
 }  
 dis[s] = 0;  
 visited[s] = true;  
 while (!visited[n]) {  
 int u = -1;  
 for (int i = 1; i <= n; ++i) {  
 if (!visited[i] && (u == -1 || dis[i] < dis[u])) {  
 u = i;  
 }  
 }  
 visited[u] = true;  
 for (int v : graph[u]) {  
 if (!visited[v.first] && v.second < dis[v.first]) {  
 dis[v.first] = v.second;  
 }  
 }  
 }  
 for (int i = 1; i <= n; ++i) {  
 res += dis[i];  
 }  
 return res;  
}

### 7.4.2 Kruskal

description:  
从所有边中从小到大选取最小生成树的边， 需要使用并查集 适合链式前向星存图 得改改代码

class UnionFind;  
UnionFind::merge(int x,int y);  
UnionFind::find(int x);  
UnionFind uf;  
struct edge  
{  
 int u,v,w;  
 inline bool operator<(const edge&e)const  
 {  
 return w<e.w;  
 }  
};  
vector<edge>edges;  
int Kruskal()  
{  
 int res=0;  
 sort(edges.begin(),edges.end());  
 for(auto e:edges)  
 {  
 if(uf.find(e.u)!=uf.find(e.v))  
 {  
 uf.merge(e.u,e.v);  
 res+=e.w;  
 }  
 }  
 return res;  
}

## 7.5. 启发式搜索

### 7.5.1 A\*

description:  
贪心算法，核心公式： g(n)为当前节点到起始节点的距离 h(n)为当前节点到目标节点的距离 比较适用于隐式建图的最短路 其他与Dijkstra相同

int g(int x);  
int h(int x);  
int f(int x);  
int Astar(int s) {  
 int res = 0;  
 priority\_queue<int, vector<int>, greater<int>>q;  
 q.push(s);  
 while (!q.empty()) {  
 int u = q.top();  
 q.pop();  
 for (int v : graph[u]) {  
 if (f(v.first) < f(u)) {  
 q.push(v.first);  
 }  
 }  
 }  
}

## 7.6. 拓扑排序

description:  
按照点的度数关系排序， 不断摘叶子

map<int,map<int,int>>mp;  
map<int,set<int>>in;  
map<int,int>deg;  
vector<int>topo;  
inline void dfs(int x)  
{  
 for(auto &y:in[x])  
 {  
 deg[y]-=1;  
 if(deg[y]==0)  
 {  
 dfs(y);  
 }  
 }  
 topo.push\_back(x);  
}  
void topo\_sort()  
{  
 for(auto &x:deg)  
 {  
 if(x.second==0)  
 {  
 dfs(x.first);  
 }  
 }  
}

## 7.7. 网络流

### 7.7.1 最大流最小割

description:  
可以解决二分图匹配问题和最大流问题 多源多汇可转化为单源单汇

map<int,map<int,int>>mp;  
map<int,map<int,int>>up;  
map<int,int>dis;  
map<int,int>pre;  
queue<int>q;  
//返回最大流  
//dinic  
int bfs(int,int);  
int dinic(int s,int t)  
{  
 int res=0;  
 while(bfs(s,t))  
 {  
 for(int i=1;i<=n;i++)  
 {  
 pre[i]=i;  
 }  
 int u=s;  
 while(u!=t)  
 {  
 int v=pre[u];  
 int f=INF;  
 for(auto &w:mp[u])  
 {  
 if(w.first==v)  
 {  
 f=min(f,w.second);  
 }  
 }  
 for(auto &w:mp[u])  
 {  
 if(w.first==v)  
 {  
 w.second-=f;  
 }  
 else  
 {  
 up[w.first][u]-=f;  
 up[u][w.first]+=f;  
 }  
 }  
 u=v;  
 }  
 res+=f;  
 }  
 return res;  
}  
int bfs(int s,int t)  
{  
 memset(dis,0,sizeof(dis));  
 dis[s]=1;  
 q.push(s);  
 while(!q.empty())  
 {  
 int u=q.front();  
 q.pop();  
 for(auto &w:mp[u])  
 {  
 if(dis[w.first]==0&&w.second>0)  
 {  
 dis[w.first]=dis[u]+1;  
 q.push(w.first);  
 }  
 }  
 }  
 return dis[t]>0;  
}

### 7.7.2 最小费用最大流

description:

## 7.8. 二分图

### 7.8.1 染色匹配法

description:  
二分图中没有奇数长度环，用染色法可找到冲突

//不会吧不会吧，不会有人看到这还不会搜索吧

### 7.8.2 匈牙利算法

description:  
增广路求解二分图匹配，可转化为多源多汇最大流

map<int,map<int,int>>mp;  
map<int,int>match;  
map<int,int>vis;  
int n;  
void init(int size)  
{  
 mp.clear();  
 match.clear();  
 vis.clear();  
 for(int i=1;i<=n;i++)  
 {  
 match[i]=-1;  
 }  
}  
int dfs(int x)  
{  
 vis[x]=1;  
 for(auto &y:mp[x])  
 {  
 if(match[y.first]==-1||(vis[match[y.first]]==0&&dfs(match[y.first])))  
 {  
 match[y.first]=x;  
 return 1;  
 }  
 }  
 return 0;  
}  
int bipartite\_matching()  
{  
 int res=0;  
 for(int i=1;i<=n;i++)  
 {  
 if(match[i]==-1)  
 {  
 vis.clear();  
 res+=dfs(i);  
 }  
 }  
 return res;  
}

## 7.9. 并查集

description:  
union-find算法， 使用路径压缩和按秩合并后复杂度为 十分高效

struct UnionFind {  
 vector<int> fa;  
 vector<int> rank;  
 int n;  
 UnionFind(int n) {  
 this->n = n;  
 fa.resize(n + 1);  
 rank.resize(n + 1);  
 for (int i = 1; i <= n; ++i) {  
 fa[i] = i;  
 rank[i] = 1;  
 }  
 }  
 ~UnionFind() {  
 fa.clear();  
 rank.clear();  
 }  
 inline int find(int x) {  
 if (fa[x] != x) {  
 fa[x] = find(fa[x]);  
 }  
 return fa[x];  
 }  
 void merge(int x, int y) {  
 int fx = find(x);  
 int fy = find(y);  
 if (fx != fy) {  
 if (rank[fx] < rank[fy]) {  
 fa[fx] = fy;  
 } else {  
 fa[fy] = fx;  
 if (rank[fx] == rank[fy]) {  
 ++rank[fx];  
 }  
 }  
 }  
 }  
};