**图论：**

**dijkstra（优先队列优化）：**

维持两个点集，每次放入dis最小的点，直到成为空集

//如果不用优先队列，则去掉优先队列，在每次循环时找到没有被visited的点中dis值最小的

int dis[maxn2];

bool vis[maxn2];

const int Inf=;

struct FSO

{

int from,dis;

FSO(int from\_,int dis\_):from(from\_),dis(dis\_) {}

bool operator<(const FSO &n)const {return dis>n.dis;}

};

int dijkstra(const int &n,int from,int to)

{

for(int i=0; i<maxn2; ++i)

dis[i]=Inf;

memset(vis,0,sizeof(vis));

priority\_queue<FSO> que;

que.push(FSO(from,0));

dis[from]=0;

int mx,now;

while(1)

{

if(que.empty()) break;

mx=que.top().from;

if(vis[mx])

{

que.pop();

continue;

}

que.pop();

for(int j=box[mx]; j; j=edge[j].next)

{

now=edge[j].to;

if(dis[now]>dis[mx]+1)

{

dis[now]=dis[mx]+1;

que.push(FSO(edge[j].to,dis[now]));

}

}

vis[mx]=1;

}

if(dis[to]!=Inf)

return dis[to];

else return -1;

}

**迭代加深搜索：（搜分解分数式）：**

注意剪枝….

bool ids(int d,double sum,int now) //深度，和，现在的值

{

if(d==maxd)

{

if(abs(sum-val)<dev)

{

return 1;

}

else return 0;

}

int k=(maxd-d)/abs(val-sum)+1; //迭代上限，还剩下maxd-d个数,到val的值为val-sum，为能到达val现在可以填的最小的数即1/k

for(int i=now;i<=k;++i)

{

if(ids(d+1,sum+1.0/i,i+1))

{

cout<<"1/"<<i<<" ";

return 1;

}

}

return 0;

}

Int main()

{

While(ids(0,0,1))

deep++;

}

**第k短路（A\*算法）：**

首先对反图做一遍dijkstra,作为预处理

（存图的时候就存反边

void add(int from,int to,int weight)

{

edge[ent].to=to;

edge[ent].from=from;

edge[ent].next=box[from];

edge[ent].last=rebox[to];

edge[ent].weight=weight;

box[from]=ent;

rebox[to]=ent++;

}

//这个是打印路径的，如果不需要打印路径，则吧关于vector的东西去掉

struct ASO

{

int from,dis,f;

vector<int> luj;

ASO(int from\_,int dis\_,int f\_,vector<int> l):from(from\_),dis(dis\_),f(f\_),luj(l){}

bool operator<(const ASO &a)const

{

return f>a.f;

}

};

int Astar(int from,int to,int k)

{

int mx,now,ndis;

int c=0;

if(from==to)

c—

//这个判断有争议，用不用都可以试下

priority\_queue<ASO> que;

vector<int> tem;

tem.push\_back(from);

que.push(ASO(from,0,dis[from],tem));

while(1)

{

if(que.empty()) break;

mx=que.top().from;

ndis=que.top().dis;

tem=que.top().luj;

que.pop();

if(mx==to)

{

c++;

printf("%d : %d\n",c,ndis);

printf("lujing:");

for(vector<int>::iterator iter=tem.begin();iter!=tem.end();++iter)

cout<<\*iter<<" ";

cout<<endl;

}

if(c==k)

return ndis;

tem.push\_back(1);

for(int i=box[mx]; i; i=edge[i].next)

{

now=edge[i].to;

tem.pop\_back();

tem.push\_back(now);

que.push(ASO(now,ndis+edge[i].weight,ndis+edge[i].weight+dis[now],tem));

}

}

return -1;

}

**最小生成树：**

**krusakal：**

O(MlogM+N)

//把所有边按长度排序，然后从头开始加边，如果存在环路则跳过，判断环路用并查集

bool used[maxn]

void kruskal()

{

sort(edge,edge+ent);

ub=0;

for(int i=0; i<ent; ++i)

{

if(find(edge[i].u)==find(edge[i].v))

continue;

Uni(edge[i].u,edge[i].v);

used[ub++]=i;

if(ub==p-1) break;

}

}

**Prim:**

O(N^2)

维持两个点集U（空）和K（全部），先随意取一个点放入U

找从U到K点的边中最小的一条，把那条连着的K中的点放入U中

循环直到K中没点了

//使用邻接表来存（内存消耗很大，但速度较快）

int low\_dis[maxn];

bool vis[maxn];

int prim()

{

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

low\_dis[i]=inf;

memset(vis,0,sizeof(vis));

vis[0]=1; //!!!!!不要漏

int res=0;

int now=0;

int l,ndis;

for(int k=1;k<n;k++)

{

ndis=inf;

for(int i=1;i<n;i++)

{

if(!vis[i]&&map[now][i]&&map[now][i]<low\_dis[i])

low\_dis[i]=map[now][i];

if(!vis[i]&&ndis>low\_dis[i])

{

ndis=low\_dis[i];

l=i;

}

}

vis[l]=1;

now=l;

res+=ndis;

}

return res;

}

**Prim优先队列优化：**

O((N+M)log(N))

空间复杂度很高，如果图非常稠密的话，可能会爆空间，但速度非常快

struct FSO //原来的版本没有dis,直接用dp[p]来判断，这导致队列不能有效更新

{

int p,dis;

FSO(int p\_,int dis\_):p(p\_),dis(dis\_) {}

bool operator<(const FSO &a)const

{

return dis>a.dis;

}

};

void prim()

{

int now;

for(int i=0; i<p; ++i)

dp[i]=Inf;

memset(vis,0,sizeof(vis));

priority\_queue<FSO> que;

que.push(FSO(0,0));

while(1)

{

if(que.empty()) break;

now=que.top().p;

if(vis[now])

{

que.pop();

continue;

}

que.pop();

for(int i=0; i<p; ++i)

{

if(i==now) continue;

if(vis[i]) continue;

if(dis[now][i]<dp[i])

{

dp[i]=dis[now][i];

que.push(FSO(i,dp[i]));

}

}

vis[now]=1;

}

}

次小生成树：

1. prim

令dp[i][j]等于MST上i到j路径的边权的最大值，然后对于所有i,j，并把MST上路径值该为inf，则

Ans=min{map[i][j]-dp[i][j]}

（即去掉i,j在MST上边权最大的值，加上i,j这条边获得的次小生成树）

dp[i][j]直接在prim中求

1. kruskal

每次加入边(a,b)时，遍历点数较小的一边的所有点，若有到另一边的点，则这个为可行解

K度最小生成树：

1. 去掉k度点，做最小生成森林
2. 连k度点和每个连通块中的最近点
3. 进行(k-p)次消圈（连一条k度点到森林的边，删掉产生的环中最长的边），得最终解

曼哈顿距离最小生成树：

O(Nlog(N))

只考虑有用的边：对于一个点来说每45度只需要考虑一条最短的边,每次坐标变换后用线段树或BIT维护求得。

然后用kruscal

**spfa(裸的):**

只要更新，放入队列中

int cnt[maxn];

bool in[maxn];

int dis[maxn];

queue<int> que;

int n,m;

bool spfa(int start)

{

memset(cnt,0,sizeof(cnt));

memset(in,0,sizeof(in));

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

dis[i]=inf;

que.push(start);

in[start]=0;

dis[start]=0;

cnt[start]++;

int tmp;

while(!que.empty())

{

tmp=que.front();

que.pop();

for(int i=box[tmp]; i!=-1; i=edge[i].next)

{

if(dis[edge[i].to]>dis[tmp]+1)

{

dis[edge[i].to]=dis[tmp]+1;

if(!in[edge[i].to])

{

que.push(edge[i].to);

in[edge[i].to]=1;

cnt[edge[i].to]++;

if(cnt[edge[i].to]>n)

{

while(!que.empty()) que.pop();

return false;

}

}

}

}

in[tmp]=0;

}

return true;

}

**Bellman-flod判断负环**

当松弛n-1次后还可以松弛（第n次还可以松弛）则有负环

bool bellman(int start)

{

memset(dis,0,sizeof(dis));

dis[start]=1;

bool flag;

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

{

flag=1;

for(int j=0;j<ctn;j++)

{

if(dis[edge[j].to]<(dis[edge[j].from])\*edge[j].c)

{

flag=0;

dis[edge[j].to]=(dis[edge[j].from])\*edge[j].c;

}

}

if(flag) break;

if(i==n-1&&!flag) return true;

}

return false;

}

**拓扑排序：**

当A<B的时候，建立A到B的有向边

找一个入度为0的点，去掉所有连此点的边

循环直到没点了

bool map[maxn][maxn];

int n,m;

bool vis[maxn];

bool nmap[maxn][maxn];

int sorted[maxn];

int TopoSort()

{

int s;

bool flag;

bool res=1;

memset(vis,0,sizeof(vis));

memcpy(nmap,map,sizeof(map));

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

{

s=-1;

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

{

if(vis[i]) continue;

flag=true;

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

{

if(i==j) continue;

if(nmap[j][i])

{

flag=false;

break;

}

}

if(flag)

{

if(s!=-1) res=0; //注意这里不能直接return 0;因为第一次判断有多个头的时候可能之后会出现错误

else s=i;

}

}

if(s==-1)

{

return -1;

}

vis[s]=1;

sorted[k]=s;

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

nmap[s][i]=0;

}

return res;

}

**匈牙利：**

每次先匹配一个点，并记录，如果有冲突，则选之前的点再匹配，如果能成功则又匹配成功

复杂度O(n3)

链表：O(nm)

bool visit[MAXN];

http://www.cppblog.com/Images/OutliningIndicators/None.gifint n,m,k,mark[MAXN];

http://www.cppblog.com/Images/OutliningIndicators/None.gifbool map[MAXN][MAXN];

http://www.cppblog.com/Images/OutliningIndicators/None.gif

bool dfs(int u){

    int i;

    for(i=1;i<=m;i++)

        if(map[u][i] && !visit[i]){

            visit[i]=true;

            if(mark[i]==-1 || dfs(mark[i])){

                mark[i]=u;

                return true;

            }

        }

    return false;

}

int hungary(){

    int i,ans=0;

    for(i=1;i<=n;i++){

        memset(visit,false,sizeof(visit));

        if(dfs(i)) ans++;

    }

    return ans;

}

**Hopcroft-Karp:**

int pa[maxn],pb[maxn];

int disa[maxn],disb[maxn];

bool bfs()

{

memset(disa,0,sizeof(disa));

memset(disb,0,sizeof(disb));

bool flag=0;

queue<int> que;

for(int i=1; i<=N; i++)

if(pa[i]==-1)

que.push(i);

int now;

while(!que.empty())

{

now=que.front();

que.pop();

for(int i=box[now]; i!=-1; i=edge[i].next)

{

if(!disb[edge[i].to])

{

disb[edge[i].to]=disa[now]+1;

if(pb[edge[i].to]==-1) flag=1;

else

{

disa[pb[edge[i].to]]=disb[edge[i].to]+1;

que.push(pb[edge[i].to]);

}

}

}

}

return flag;

}

bool dfs(int now)

{

for(int i=box[now]; i!=-1; i=edge[i].next)

{

if(disb[edge[i].to]==disa[now]+1)

{

disb[edge[i].to]=0;

if(pb[edge[i].to]==-1||dfs(pb[edge[i].to]))//

{

pb[edge[i].to]=now;

pa[now]=edge[i].to;

return 1;

}

}

}

return 0;

}

int Hopcroft\_Karp()

{

memset(pa,-1,sizeof(pa));

memset(pb,-1,sizeof(pb));

int res=0;

while(bfs())

{

for(int i=1; i<=N; i++)

if(pa[i]==-1&&dfs(i))//

res++;

}

return res;

}

**KM算法：**

解决最大权二分匹配问题，可以用费用流，这个比较快

int n,m,slack[MAXN],lx[MAXN],ly[MAXN],maty[MAXN],lenx,leny;

bool vx[MAXN],vy[MAXN];//S集合、Y集合

char map[MAXN][MAXN];

int a[MAXN][MAXN];

//网络流邻接表构图要用双向边(反向边流值初始为0，可用于增广流)，二分图单向边即可(二分图的邻接矩阵边也是单向边)

int mabs(int a)

{

return a>0?a:-a;

}

bool search(int u)

{

int i,t;

vx[u]=1;

for(i=0; i<leny; ++i)

if(!vy[i])

{

t=lx[u]+ly[i]-a[u][i];

if(t==0/\*相等子图\*/)

{

vy[i]=1;

if(maty[i]==-1||search(maty[i]))

{

maty[i]=u;

return 1;

}

}

else if(slack[i]>t)

slack[i]=t;

}

return 0;

}

int KM()

{

int i,j,ans=0;

for(i=0; i<lenx; ++i)

for(lx[i]=-INT\_MAX,j=0; j<leny; ++j)

lx[i]=max(lx[i],a[i][j]);

memset(maty,-1,sizeof(maty));

memset(ly,0,sizeof(ly));

for(i=0; i<lenx; ++i)

{

//找增广路

for(j=0; j<leny; ++j)

slack[j]=INT\_MAX;

while(1)

{

memset(vx,0,sizeof(vx)); //vx,vy表示已经访问过

memset(vy,0,sizeof(vy));

if(search(i))//找到i对应的增广路，不再找

break;

//没找到增广路，修正

int d=INT\_MAX;

for(j=0; j<leny; ++j)

if(!vy[j]&&d>slack[j])

d=slack[j];

for(j=0; j<lenx; ++j)

if(vx[j])

lx[j]-=d;

for(j=0; j<leny; ++j)

if(vy[j])

ly[j]+=d;

}

}

for(i=0; i<leny; ++i)

if(maty[i]!=-1)

ans+=a[maty[i]][i];

return ans;

}

**Tarjan缩点：**

int dfn[maxn],low[maxn];

int fl[maxn],fln;

vector<int> fls[maxn];

int stack[maxn],top;

bool in[maxn];

int ti;

void tarjan(int now)

{

dfn[now]=low[now]=ti++;

stack[++top]=now;

in[now]=1;

for(int i=box[now];i!=-1;i=edge[i].next)

{

if(!dfn[edge[i].to])

{

tarjan(edge[i].to);

low[now]=min(low[now],low[edge[i].to]);

}

else if(in[edge[i].to])

low[now]=min(low[now],dfn[edge[i].to]);

}

int nd;

if(dfn[now]==low[now])

{

fls[fln].clear();

while(top!=-1)

{

nd=stack[top];

top--;

fls[fln].push\_back(nd);

fl[nd]=fln;

in[nd]=0;

if(nd==now) break;

}

fln++;

}

}

void initTarjan()

{

memset(dfn,0,sizeof(dfn));

memset(in,0,sizeof(in));

top=-1;

fln=0;

}

**2-sat:**

构图：

边表示选了from必须选to

如果一个状态肯定不能选，连一条边到其对立状态

输出建反图用topsort

**无向图求桥：**

int dfn[maxn],low[maxn];

int res;

int dep;

void dfs(int now,int id)

{

low[now]=dfn[now]=++dep;

for(int i=box[now];i!=-1;i=edge[i].next)

{

if((i^id)==1) continue;//同边

if(dfn[edge[i].to])

{

if(dfn[edge[i].to]<low[now])

low[now]=dfn[edge[i].to];

}

else

{

dfs(edge[i].to,i);

if(low[edge[i].to]<low[now])

low[now]=low[edge[i].to];

if(low[edge[i].to]>dfn[now])//桥

res=min(res,edge[i].v);

}

}

}