typedef enum { UNDISCOVERED, DISCOVERED, VISITED } VStatus; //顶点状态

typedef enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD } EType; //边在遍历树中所属的类型

template <typename Tv, typename Te> //顶点类型、边类型

class Graph { //图Graph模板类

private:

void reset() { //所有顶点、边的辅助信息复位

for ( int i = 0; i < n; i++ ) { //所有顶点的

status ( i ) = UNDISCOVERED; dTime ( i ) = fTime ( i ) = -1; //状态，时间标签

parent ( i ) = -1; priority ( i ) = INT\_MAX; //（在遍历树中的）父节点，优先级数

for ( int j = 0; j < n; j++ ) //所有边的

if ( exists ( i, j ) ) type ( i, j ) = UNDETERMINED; //类型

}

}

void BFS ( int, int& ); //（连通域）广度优先搜索算法

void DFS ( int, int& ); //（连通域）深度优先搜索算法

void BCC ( int, int&, Stack<int>& ); //（连通域）基于DFS的双连通分量分解算法

bool TSort ( int, int&, Stack<Tv>\* ); //（连通域）基于DFS的拓扑排序算法

template <typename PU> void PFS ( int, PU ); //（连通域）优先级搜索框架

public:

// 顶点

int n; //顶点总数

virtual int insert ( Tv const& ) = 0; //插入顶点，返回编号

virtual Tv remove ( int ) = 0; //删除顶点及其关联边，返回该顶点信息

virtual Tv& vertex ( int ) = 0; //顶点v的数据（该顶点的确存在）

virtual int inDegree ( int ) = 0; //顶点v的入度（该顶点的确存在）

virtual int outDegree ( int ) = 0; //顶点v的出度（该顶点的确存在）

virtual int firstNbr ( int ) = 0; //顶点v的首个邻接顶点

virtual int nextNbr ( int, int ) = 0; //顶点v的（相对于顶点j的）下一邻接顶点

virtual VStatus& status ( int ) = 0; //顶点v的状态

virtual int& dTime ( int ) = 0; //顶点v的时间标签dTime

virtual int& fTime ( int ) = 0; //顶点v的时间标签fTime

virtual int& parent ( int ) = 0; //顶点v在遍历树中的父亲

virtual int& priority ( int ) = 0; //顶点v在遍历树中的优先级数

// 边：这里约定，无向边均统一转化为方向互逆的一对有向边，从而将无向图视作有向图的特例

int e; //边总数

virtual bool exists ( int, int ) = 0; //边(v, u)是否存在

virtual void insert ( Te const&, int, int, int ) = 0; //在顶点v和u之间插入权重为w的边e

virtual Te remove ( int, int ) = 0; //删除顶点v和u之间的边e，返回该边信息

virtual EType & type ( int, int ) = 0; //边(v, u)的类型

virtual Te& edge ( int, int ) = 0; //边(v, u)的数据（该边的确存在）

virtual int& weight ( int, int ) = 0; //边(v, u)的权重

// 算法

void bfs ( int ); //广度优先搜索算法

void dfs ( int ); //深度优先搜索算法

void bcc ( int ); //基于DFS的双连通分量分解算法

Stack<Tv>\* tSort ( int ); //基于DFS的拓扑排序算法

void prim ( int ); //最小支撑树Prim算法

void dijkstra ( int ); //最短路径Dijkstra算法

template <typename PU> void pfs ( int, PU ); //优先级搜索框架

};

#include "../Vector/Vector.h" //引入向量

#include "../Graph/Graph.h" //引入图ADT

template <typename Tv> struct Vertex { //顶点对象（为简化起见，并未严格封装）

Tv data; int inDegree, outDegree; VStatus status; //数据、出入度数、状态

int dTime, fTime; //时间标签

int parent; int priority; //在遍历树中的父节点、优先级数

Vertex ( Tv const& d = ( Tv ) 0 ) : //构造新顶点

data ( d ), inDegree ( 0 ), outDegree ( 0 ), status ( UNDISCOVERED ),

dTime ( -1 ), fTime ( -1 ), parent ( -1 ), priority ( INT\_MAX ) {} //暂不考虑权重溢出

};

template <typename Te> struct Edge { //边对象（为简化起见，并未严格封装）

Te data; int weight; EType type; //数据、权重、类型

Edge ( Te const& d, int w ) : data ( d ), weight ( w ), type ( UNDETERMINED ) {} //构造

};

template <typename Tv, typename Te> //顶点类型、边类型

class GraphMatrix : public Graph<Tv, Te> { //基于向量，以邻接矩阵形式实现的图

private:

Vector< Vertex< Tv > > V; //顶点集（向量）

Vector< Vector< Edge< Te > \* > > E; //边集（邻接矩阵）

public:

GraphMatrix() { n = e = 0; } //构造

~GraphMatrix() { //析构

for ( int j = 0; j < n; j++ ) //所有动态创建的

for ( int k = 0; k < n; k++ ) //边记录

delete E[j][k]; //逐条清除

}

// 顶点的基本操作：查询第i个顶点（0 <= i < n）

virtual Tv& vertex ( int i ) { return V[i].data; } //数据

virtual int inDegree ( int i ) { return V[i].inDegree; } //入度

virtual int outDegree ( int i ) { return V[i].outDegree; } //出度

virtual int firstNbr ( int i ) { return nextNbr ( i, n ); } //首个邻接顶点

virtual int nextNbr ( int i, int j ) //相对于顶点j的下一邻接顶点（改用邻接表可提高效率）

{ while ( ( -1 < j ) && ( !exists ( i, --j ) ) ); return j; } //逆向线性试探

virtual VStatus& status ( int i ) { return V[i].status; } //状态

virtual int& dTime ( int i ) { return V[i].dTime; } //时间标签dTime

virtual int& fTime ( int i ) { return V[i].fTime; } //时间标签fTime

virtual int& parent ( int i ) { return V[i].parent; } //在遍历树中的父亲

virtual int& priority ( int i ) { return V[i].priority; } //在遍历树中的优先级数

// 顶点的动态操作

virtual int insert ( Tv const& vertex ) { //插入顶点，返回编号

for ( int j = 0; j < n; j++ ) E[j].insert ( NULL ); n++; //各顶点预留一条潜在的关联边

E.insert ( Vector<Edge<Te>\*> ( n, n, ( Edge<Te>\* ) NULL ) ); //创建新顶点对应的边向量

return V.insert ( Vertex<Tv> ( vertex ) ); //顶点向量增加一个顶点

}

virtual Tv remove ( int i ) { //删除第i个顶点及其关联边（0 <= i < n）

for ( int j = 0; j < n; j++ ) //所有出边

if ( exists ( i, j ) ) { delete E[i][j]; V[j].inDegree--; } //逐条删除

E.remove ( i ); n--; //删除第i行

Tv vBak = vertex ( i ); V.remove ( i ); //删除顶点i

for ( int j = 0; j < n; j++ ) //所有入边

if ( Edge<Te> \* e = E[j].remove ( i ) ) { delete e; V[j].outDegree--; } //逐条删除

return vBak; //返回被删除顶点的信息

}

// 边的确认操作

virtual bool exists ( int i, int j ) //边(i, j)是否存在

{ return ( 0 <= i ) && ( i < n ) && ( 0 <= j ) && ( j < n ) && E[i][j] != NULL; }

// 边的基本操作：查询顶点i与j之间的联边（0 <= i, j < n且exists(i, j)）

virtual EType & type ( int i, int j ) { return E[i][j]->type; } //边(i, j)的类型

virtual Te& edge ( int i, int j ) { return E[i][j]->data; } //边(i, j)的数据

virtual int& weight ( int i, int j ) { return E[i][j]->weight; } //边(i, j)的权重

// 边的动态操作

virtual void insert ( Te const& edge, int w, int i, int j ) { //插入权重为w的边e = (i, j)

if ( exists ( i, j ) ) return; //确保该边尚不存在

E[i][j] = new Edge<Te> ( edge, w ); //创建新边

e++; V[i].outDegree++; V[j].inDegree++; //更新边计数与关联顶点的度数

}

virtual Te remove ( int i, int j ) { //删除顶点i和j之间的联边（exists(i, j)）

Te eBak = edge ( i, j ); delete E[i][j]; E[i][j] = NULL; //备份后删除边记录

e--; V[i].outDegree--; V[j].inDegree--; //更新边计数与关联顶点的度数

return eBak; //返回被删除边的信息

}

};

template <typename Tv, typename Te> //广度优先搜索BFS算法（全图）

void Graph<Tv, Te>::bfs ( int s ) { //assert: 0 <= s < n

reset(); int clock = 0; int v = s; //初始化

do //逐一检查所有顶点

if ( UNDISCOVERED == status ( v ) ) //一旦遇到尚未发现的顶点

BFS ( v, clock ); //即从该顶点出发启动一次BFS

while ( s != ( v = ( ++v % n ) ) ); //按序号检查，故不漏不重

}

template <typename Tv, typename Te> //广度优先搜索BFS算法（单个连通域）

void Graph<Tv, Te>::BFS ( int v, int& clock ) { //assert: 0 <= v < n

Queue<int> Q; //引入辅助队列

status ( v ) = DISCOVERED; Q.enqueue ( v ); //初始化起点

while ( !Q.empty() ) { //在Q变空之前，不断

int v = Q.dequeue(); dTime ( v ) = ++clock; //取出队首顶点v

for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u

if ( UNDISCOVERED == status ( u ) ) { //若u尚未被发现，则

status ( u ) = DISCOVERED; Q.enqueue ( u ); //发现该顶点

type ( v, u ) = TREE; parent ( u ) = v; //引入树边拓展支撑树

} else { //若u已被发现，或者甚至已访问完毕，则

type ( v, u ) = CROSS; //将(v, u)归类于跨边

}

status ( v ) = VISITED; //至此，当前顶点访问完毕

}

}

template <typename Tv, typename Te> //深度优先搜索DFS算法（全图）

void Graph<Tv, Te>::dfs ( int s ) { //assert: 0 <= s < n

reset(); int clock = 0; int v = s; //初始化

do //逐一检查所有顶点

if ( UNDISCOVERED == status ( v ) ) //一旦遇到尚未发现的顶点

DFS ( v, clock ); //即从该顶点出发启动一次DFS

while ( s != ( v = ( ++v % n ) ) ); //按序号检查，故不漏不重

}

template <typename Tv, typename Te> //深度优先搜索DFS算法（单个连通域）

void Graph<Tv, Te>::DFS ( int v, int& clock ) { //assert: 0 <= v < n

dTime ( v ) = ++clock; status ( v ) = DISCOVERED; //发现当前顶点v

for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u

switch ( status ( u ) ) { //并视其状态分别处理

case UNDISCOVERED: //u尚未发现，意味着支撑树可在此拓展

type ( v, u ) = TREE; parent ( u ) = v; DFS ( u, clock ); break;

case DISCOVERED: //u已被发现但尚未访问完毕，应属被后代指向的祖先

type ( v, u ) = BACKWARD; break;

default: //u已访问完毕（VISITED，有向图），则视承袭关系分为前向边或跨边

type ( v, u ) = ( dTime ( v ) < dTime ( u ) ) ? FORWARD : CROSS; break;

}

status ( v ) = VISITED; fTime ( v ) = ++clock; //至此，当前顶点v方告访问完毕

}

template <typename Tv, typename Te> //基于DFS的拓扑排序算法

Stack<Tv>\* Graph<Tv, Te>::tSort ( int s ) { //assert: 0 <= s < n

reset(); int clock = 0; int v = s;

Stack<Tv>\* S = new Stack<Tv>; //用栈记录排序顶点

do {

if ( UNDISCOVERED == status ( v ) )

if ( !TSort ( v, clock, S ) ) { //clock并非必需

/\*DSA\*/print ( S );

while ( !S->empty() ) //任一连通域（亦即整图）非DAG

S->pop(); break; //则不必继续计算，故直接返回

}

} while ( s != ( v = ( ++v % n ) ) );

return S; //若输入为DAG，则S内各顶点自顶向底排序；否则（不存在拓扑排序），S空

}

template <typename Tv, typename Te> //基于DFS的拓扑排序算法（单趟）

bool Graph<Tv, Te>::TSort ( int v, int& clock, Stack<Tv>\* S ) { //assert: 0 <= v < n

dTime ( v ) = ++clock; status ( v ) = DISCOVERED; //发现顶点v

for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u

switch ( status ( u ) ) { //并视u的状态分别处理

case UNDISCOVERED:

parent ( u ) = v; type ( v, u ) = TREE;

if ( !TSort ( u, clock, S ) ) //从顶点u处出发深入搜索

return false; //若u及其后代不能拓扑排序（则全图亦必如此），故返回并报告

break;

case DISCOVERED:

type ( v, u ) = BACKWARD; //一旦发现后向边（非DAG），则

return false; //不必深入，故返回并报告

default: //VISITED (digraphs only)

type ( v, u ) = ( dTime ( v ) < dTime ( u ) ) ? FORWARD : CROSS;

break;

}

status ( v ) = VISITED; S->push ( vertex ( v ) ); //顶点被标记为VISITED时，随即入栈

return true; //v及其后代可以拓扑排序

}

template <typename Tv, typename Te> void Graph<Tv, Te>::bcc ( int s ) { //基于DFS的BCC分解算法

reset(); int clock = 0; int v = s; Stack<int> S; //栈S用以记录已访问的顶点

do

if ( UNDISCOVERED == status ( v ) ) { //一旦发现未发现的顶点（新连通分量）

BCC ( v, clock, S ); //即从该顶点出发启动一次BCC

S.pop(); //遍历返回后，弹出栈中最后一个顶点——当前连通域的起点

}

while ( s != ( v = ( ++v % n ) ) );

}

#define hca(x) (fTime(x)) //利用此处闲置的fTime[]充当hca[]

template <typename Tv, typename Te> //顶点类型、边类型

void Graph<Tv, Te>::BCC ( int v, int& clock, Stack<int>& S ) { //assert: 0 <= v < n

hca ( v ) = dTime ( v ) = ++clock; status ( v ) = DISCOVERED; S.push ( v ); //v被发现并入栈

for ( int u = firstNbr ( v ); -1 < u; u = nextNbr ( v, u ) ) //枚举v的所有邻居u

switch ( status ( u ) ) { //并视u的状态分别处理

case UNDISCOVERED:

parent ( u ) = v; type ( v, u ) = TREE; BCC ( u, clock, S ); //从顶点u处深入

if ( hca ( u ) < dTime ( v ) ) //遍历返回后，若发现u（通过后向边）可指向v的真祖先

hca ( v ) = min ( hca ( v ), hca ( u ) ); //则v亦必如此

else { //否则，以v为关节点（u以下即是一个BCC，且其中顶点此时正集中于栈S的顶部）

/\*DSA\*/printf ( "BCC rooted at %c:", vertex ( v ) );

/\*DSA\*/Stack<int> temp; do { temp.push ( S.pop() ); print ( vertex ( temp.top() ) ); } while ( v != temp.top() ); while ( !temp.empty() ) S.push ( temp.pop() );

while ( v != S.pop() ); //依次弹出当前BCC中的节点，亦可根据实际需求转存至其它结构

S.push ( v ); //最后一个顶点（关节点）重新入栈——分摊不足一次

/\*DSA\*/printf ( "\n" );

}

break;

case DISCOVERED:

type ( v, u ) = BACKWARD; //标记(v, u)，并按照“越小越高”的准则

if ( u != parent ( v ) ) hca ( v ) = min ( hca ( v ), dTime ( u ) ); //更新hca[v]

break;

default: //VISITED (digraphs only)

type ( v, u ) = ( dTime ( v ) < dTime ( u ) ) ? FORWARD : CROSS;

break;

}

status ( v ) = VISITED; //对v的访问结束

}

#undef hca

template <typename Tv, typename Te> template <typename PU> //优先级搜索（全图）

void Graph<Tv, Te>::pfs ( int s, PU prioUpdater ) { //assert: 0 <= s < n

reset(); int v = s; //初始化

do //逐一检查所有顶点

if ( UNDISCOVERED == status ( v ) ) //一旦遇到尚未发现的顶点

PFS ( v, prioUpdater ); //即从该顶点出发启动一次PFS

while ( s != ( v = ( ++v % n ) ) ); //按序号检查，故不漏不重

}

template <typename Tv, typename Te> template <typename PU> //顶点类型、边类型、优先级更新器

void Graph<Tv, Te>::PFS ( int s, PU prioUpdater ) { //优先级搜索（单个连通域）

priority ( s ) = 0; status ( s ) = VISITED; parent ( s ) = -1; //初始化，起点s加至PFS树中

while ( 1 ) { //将下一顶点和边加至PFS树中

for ( int w = firstNbr ( s ); -1 < w; w = nextNbr ( s, w ) ) //枚举s的所有邻居w

prioUpdater ( this, s, w ); //更新顶点w的优先级及其父顶点

for ( int shortest = INT\_MAX, w = 0; w < n; w++ )

if ( UNDISCOVERED == status ( w ) ) //从尚未加入遍历树的顶点中

if ( shortest > priority ( w ) ) //选出下一个

{ shortest = priority ( w ); s = w; } //优先级最高的顶点s

if ( VISITED == status ( s ) ) break; //直至所有顶点均已加入

status ( s ) = VISITED; type ( parent ( s ), s ) = TREE; //将s及与其父的联边加入遍历树

}

} //通过定义具体的优先级更新策略prioUpdater，即可实现不同的算法功能

template <typename Tv, typename Te> struct PrimPU { //针对Prim算法的顶点优先级更新器

virtual void operator() ( Graph<Tv, Te>\* g, int uk, int v ) {

if ( UNDISCOVERED == g->status ( v ) ) //对于uk每一尚未被发现的邻接顶点v

if ( g->priority ( v ) > g->weight ( uk, v ) ) { //按Prim策略做松弛

g->priority ( v ) = g->weight ( uk, v ); //更新优先级（数）

g->parent ( v ) = uk; //更新父节点

}

}

};

template <typename Tv, typename Te> struct DijkstraPU { //针对Dijkstra算法的顶点优先级更新器

virtual void operator() ( Graph<Tv, Te>\* g, int uk, int v ) {

if ( UNDISCOVERED == g->status ( v ) ) //对于uk每一尚未被发现的邻接顶点v，按Dijkstra策略

if ( g->priority ( v ) > g->priority ( uk ) + g->weight ( uk, v ) ) { //做松弛

g->priority ( v ) = g->priority ( uk ) + g->weight ( uk, v ); //更新优先级（数）

g->parent ( v ) = uk; //并同时更新父节点

}

}

};

template <typename Tv, typename Te> struct BfsPU { //针对BFS算法的顶点优先级更新器

virtual void operator() ( Graph<Tv, Te>\* g, int uk, int v ) {

if ( g->status ( v ) == UNDISCOVERED ) //对于uk每一尚未被发现的邻接顶点v

if ( g->priority ( v ) > g->priority ( uk ) + 1 ) { //将其到起点的距离作为优先级数

g->priority ( v ) = g->priority ( uk ) + 1; //更新优先级（数）

g->parent ( v ) = uk; //更新父节点

} //如此效果等同于，先被发现者优先

}

};

template <typename Tv, typename Te> struct DfsPU { //针对DFS算法的顶点优先级更新器

virtual void operator() ( Graph<Tv, Te>\* g, int uk, int v ) {

if ( g->status ( v ) == UNDISCOVERED ) //对于uk每一尚未被发现的邻接顶点v

if ( g->priority ( v ) > g->priority ( uk ) - 1 ) { //将其到起点距离的负数作为优先级数

g->priority ( v ) = g->priority ( uk ) - 1; //更新优先级（数）

g->parent ( v ) = uk; //更新父节点

return; //注意：与BfsPU()不同，这里只要有一个邻接顶点可更新，即可立即返回

} //如此效果等同于，后被发现者优先

}

};

template <typename Tv, typename Te> //最短路径Dijkstra算法：适用于一般的有向图

void Graph<Tv, Te>::dijkstra ( int s ) { //assert: 0 <= s < n

reset(); priority ( s ) = 0;

for ( int i = 0; i < n; i++ ) { //共需引入n个顶点和n-1条边

status ( s ) = VISITED;

if ( -1 != parent ( s ) ) type ( parent ( s ), s ) = TREE; //引入当前的s

for ( int j = firstNbr ( s ); -1 < j; j = nextNbr ( s, j ) ) //枚举s的所有邻居j

if ( ( status ( j ) == UNDISCOVERED ) && ( priority ( j ) > priority ( s ) + weight ( s, j ) ) ) //对邻接顶点j做松弛

{ priority ( j ) = priority ( s ) + weight ( s, j ); parent ( j ) = s; } //与Prim算法唯一的不同之处

for ( int shortest = INT\_MAX, j = 0; j < n; j++ ) //选出下一最近顶点

if ( ( status ( j ) == UNDISCOVERED ) && ( shortest > priority ( j ) ) )

{ shortest = priority ( j ); s = j; }

}

} //对于无向连通图，假设每一条边表示为方向互逆、权重相等的一对边

template <typename Tv, typename Te> //Prim算法：无向连通图，各边表示为方向互逆、权重相等的一对边

void Graph<Tv, Te>::prim ( int s ) { //assert: 0 <= s < n

reset(); priority ( s ) = 0;

for ( int i = 0; i < n; i++ ) { //共需引入n个顶点和n-1条边

status ( s ) = VISITED;

if ( -1 != parent ( s ) ) type ( parent ( s ), s ) = TREE; //引入当前的s

for ( int j = firstNbr ( s ); -1 < j; j = nextNbr ( s, j ) ) //枚举s的所有邻居j

if ( ( status ( j ) == UNDISCOVERED ) && ( priority ( j ) > weight ( s, j ) ) ) //对邻接顶点j做松弛

{ priority ( j ) = weight ( s, j ); parent ( j ) = s; } //与Dijkstra算法唯一的不同之处

for ( int shortest = INT\_MAX, j = 0; j < n; j++ ) //选出下一极短跨边

if ( ( status ( j ) == UNDISCOVERED ) && ( shortest > priority ( j ) ) )

{ shortest = priority ( j ); s = j; }

}

}