**暨南大学本科实验报告专用纸**

课程名称 算法分析与设计实验 成绩评定

实验项目名称 指导教师 李展 实验地点 N116

实验项目编号 6-2 实验项目类型 综合型

学生姓名 张瑞鹏 学号 2020101124

学院 信息科学技术学院 系 计算机 专业 计算机科学与技术

实验时间 2022 年 月 日 午～ 月 日 下 午

**（一）问题描述**

**给定一个赋权无向图 G=(V,E)，每个顶点 v∈V 都有一个权值 w(v)。如果 U⊆VU⊆V，且对任意(u,v)∈E 有 u∈U 或 v∈U，就称 U 为图 G 的一个顶点覆盖。G 的最小权顶点覆盖是指 G 中所含顶点权之和最小的顶点覆盖。**

**对于给定的无向图 G，设计一个优先队列式分支限界法，计算 G 的最小权顶点覆盖。**

**（二）算法思路**（用文字简单说明）

**关于定点覆盖问题，会先联想到刚学过的最大团问题，他们都是已经被证明的np完全问题。我们思考最大团问题其实是一个考虑顶点为主的一个问题，意思是每一个顶点都试一试能不能加入团。但是对于顶点覆盖问题就不同了，我认为这个问题是以考虑边为主，对每一条边进行考虑，也就是说，每一条边都得做节点二选一，这人就产生了分支，也就是解空间树的形成，但是对于很多孤立点，就不需要考虑了。**

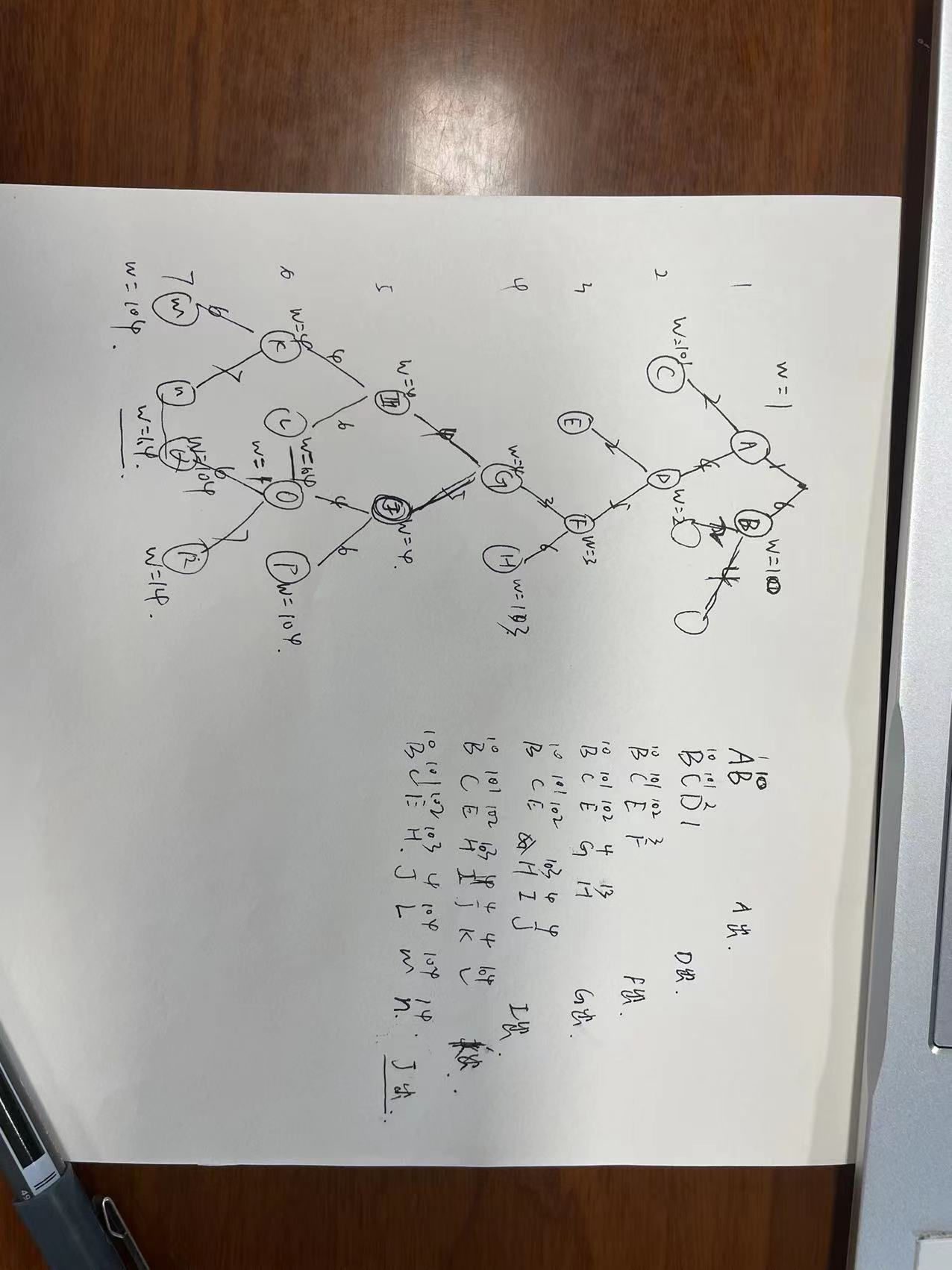
**本次实验用到很多的数据结构：**

edge sumup[7] = { {1,6},{2,4},{2,5},{3,6},{4,5},{4,6},{6,7} };

//这个是作为对每一条边进行考虑，每一条边都需要二选一

**对于优先级的设计是目前已经选了的节点权值之和，这种选法未考虑未来，但是一旦有叶子节点出队，该节点就成为最优解，不需要做其他判断或者等待队列为空。**

**当然这道题目也可以采用选点的方式来实现，根据每一个点是否选来构造二叉树，还是以权值最小为优先队列的条件，实现算法。**

****

**（三）算法实施步骤和流程**（伪代码/流程图等方式描述）

选择边的算法：

void addqueue(int level,int chosen,treenode \*parent) {

treenode\* Tnode = new treenode;

treenode\* b= Tnode;

Tnode->chosen = chosen;

Tnode->parent = parent;

Tnode->pointnum = 0;//重新计算

b = Tnode;

for (int i = 0; i < n; i++) testlist[i] = 0;//初始化为零

for (int i = level; i >= 1; i--, b = b->parent) testlist[(b->chosen) - 1] = 1;//选上

for (int i = 0; i < n; i++) if (testlist[i] == 1) Tnode->pointnum += weight[i];//保证不能重复加

int sumleft = 0;

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

if(!testlist[sumup[level].leftnode-1]&&!testlist[sumup[level].rightbode - 1])

sumleft += edgelestweight[i];//最优集的定义剩下未选边必须得选上一些

}

queuenode\* qnode = new queuenode;

qnode->nodeontree = Tnode;//节点挂载到上面去

qnode->level = level;

qnode->piority = Tnode->pointnum;// +sumleft;//优先级

q.push(\*qnode);//入队

return;

}

int main(void) {

int nowlevel = 0;

treenode head = {NULL,0,0};

treenode \*parents = &head;//初始化指向头节点

queuenode qtemp;

while (nowlevel < m) {

int chosen;

chosen = sumup[nowlevel].leftnode;

addqueue(nowlevel + 1, chosen, parents);

chosen = sumup[nowlevel].rightbode;

addqueue(nowlevel + 1, chosen, parents);

qtemp = q.top();

q.pop();

parents = qtemp.nodeontree;

nowlevel = qtemp.level;

}

return 0;

}

选择点的算法：所采用的数据结构都是不一样的，因此两种算法不能用同一个接口调用

bool VC::cover(HeapNode E)

{

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

{

if (E.x[j] == 0 && E.c[j] == 0)//如果此时j结点既不是U中的点,而且也没有U中的点与其相连,则至少这个点未被覆盖

{

return false;

}

}

return true;

}

void VC::AddLiveNode(priority\_queue<HeapNode>& H, HeapNode E, int cn, int i, bool ch)

{

HeapNode N;//创建一个新的堆结点

N.x = new int[n + 1];

N.c = new int[n + 1];

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

{

N.x[j] = E.x[j];

N.c[j] = E.c[j];

}

N.x[i] = ch;

if (ch)

{

N.cn = cn + w[i];//此时i要加入集合U,所以其权重应该加上cn

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

{

if (a[i][j])

N.c[j]++;//表明此时对于结点j来说,有一节点i属于U与其连接,表明这个点被覆盖了

}

}

else

N.cn = cn;

N.i = i;

H.push(N);

}

//MinCover完成最小覆盖计算

int MinCover(int\*\* a, int v[], int n)//v表示的是结点权重数组

{

VC Y;

Y.w = new int[n + 1];

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

Y.w[j] = v[j];

Y.a = a;

Y.n = n;

Y.bestx = v;

Y.BBVC();

return Y.bestn;

}

**（四）源代码**（通过了编译运行的正确程序）

#include<iostream>

#include<vector>

#include<queue>

using namespace std;

struct treenode {//节点

treenode\* parent;//父亲节点

int chosen;

int pointnum;//目前这个定点覆盖用了多少个顶点

};

struct edge {//边

int leftnode;//顶点编号唯一

int rightbode;//顶点编号为一

};

struct queuenode {

int level;//记录位置

int piority;//优先级

treenode \*nodeontree;//树上的节点

};

struct cmp2

{

bool operator()(const queuenode& a, const queuenode& b){

return a.piority > b.piority;

}

};

int n = 7;//顶点个数

int m = 7;//边的个数

int best = 100;

edge sumup[7] = { {1,6},{2,4},{2,5},{3,6},{4,5},{4,6},{6,7} };//记录所有边

int edgelestweight[7] = {1,1,1,1,1,1,10};

int weight[7] = { 1,100,1,1,1,100,10 };

int testlist[7] = {0};

priority\_queue <queuenode, vector<queuenode>, cmp2 > q;//优先队列

void addqueue(int level,int chosen,treenode \*parent) {

treenode\* Tnode = new treenode;

treenode\* b= Tnode;

Tnode->chosen = chosen;

Tnode->parent = parent;

Tnode->pointnum = 0;//重新计算

b = Tnode;

for (int i = 0; i < n; i++) testlist[i] = 0;//初始化为零

for (int i = level; i >= 1; i--, b = b->parent) testlist[(b->chosen) - 1] = 1;//选上

for (int i = 0; i < n; i++) if (testlist[i] == 1) Tnode->pointnum += weight[i];//保证不能重复加

int sumleft = 0;

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

if(!testlist[sumup[level].leftnode-1]&&!testlist[sumup[level].rightbode - 1])

sumleft += edgelestweight[i];//最优集的定义剩下未选边必须得选上一些

}

queuenode\* qnode = new queuenode;

qnode->nodeontree = Tnode;//节点挂载到上面去

qnode->level = level;

qnode->piority = Tnode->pointnum;// +sumleft;//优先级

q.push(\*qnode);//入队

return;

}

int main(void) {

int nowlevel = 0;

treenode head = {NULL,0,0};

treenode \*parents = &head;//初始化指向头节点

queuenode qtemp;

while (nowlevel < m) {

int chosen;

chosen = sumup[nowlevel].leftnode;

addqueue(nowlevel + 1, chosen, parents);

chosen = sumup[nowlevel].rightbode;

addqueue(nowlevel + 1, chosen, parents);

qtemp = q.top();

//printf("%-5d%-5d%-5d\n", qtemp.piority, qtemp.level, qtemp.nodeontree->chosen);

q.pop();

parents = qtemp.nodeontree;

nowlevel = qtemp.level;

}

cout << "test example:" << endl;

cout << n << " " << m << endl;

for (int i = 0; i < n; i++) cout << weight[i] << " ";

cout << endl;

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

cout << sumup[i].leftnode << " " << sumup[i].rightbode << endl;

}

treenode\* b = parents;

cout << endl << "the result is:" << endl;

cout << parents->pointnum << endl;

for (int i = 0; i < n; i++) testlist[i] = 0;//初始化为零

for (int i = m; i >= 1; i--, b = b->parent) {

//printf("%-5d%-5d\n", b->pointnum, b->chosen);

testlist[(b->chosen) - 1] = 1;

}

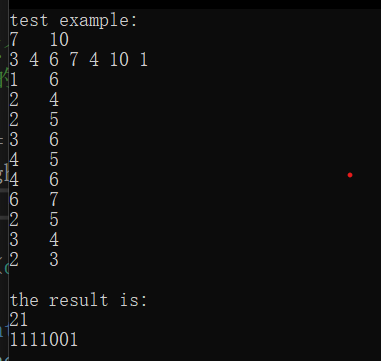
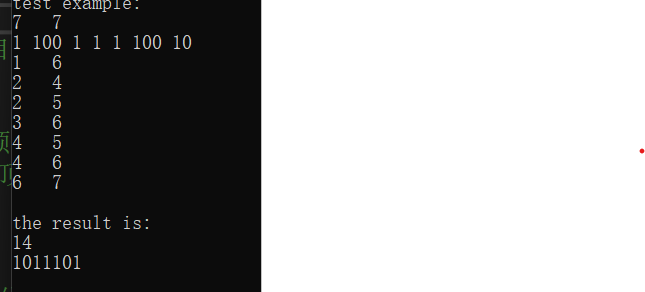
for (int i = 0; i < n; i++) cout << testlist[i];

cout << endl;

return 0;

}

**（五）测试结果**（至少有两个以上算例及程序运行结果，截图贴进实验报告）



**（六）实验总结**（至少三句话，可以写复杂度分析、遇到问题、可能的改进措施、心得体会等）

1.分支限界法和回溯都是针对解空间树的解题方法，回溯是采用了深度优先的搜索方式，而分支限界法的话采用的是广度优先或者优先队列的方式。假设是要找一个最优解，回溯法是需要把所有叶子节点都遍历到，比较到，才能够得到答案。但是对于这种题目的话，分支限界法可以通过设计一个优先队列，优先让高质量的节点先结束遍历，更早的得到最优解，更早的结束算法。对于谁才是高质量的解，只能是一种预测，不能说确实这个最优的节点遍历下去一定能得到答案，所以设计优先队列的时候也没有必要设计过于复杂，做合理的限制，这样就已经比回溯算法时间上更优了。

2.分支限界法有一个问题，它的所有节点都是实实在在存在的，它需要真实的空间，原因是它广度优先遍历的话节点会非常的多，必须通过树的方式实实在在的连到一起，这样一来，没有什么信心能够比回溯算法效率高很多的话就不需要采取分支限界法了，因为它耗费的空间比回溯大得多。

3.我认为就这道题目而言，回溯法是一定需要遍历整棵树才能得到答案的，因为求的是最优解，这么一来分支界限法就会更为合适。

4.至于到底是选边算法还是选点，也就是到底根据边来构造二叉树呢还是根据点来构造二叉树，这个就要取决于图的特点。对于一个稀疏矩阵而言，肯定是选边是更为合适的，但对于一个边比点更加多的矩阵，则考虑通过顶的是否选择来建立二叉树。

**暨南大学本科实验报告专用纸(附页)**