## 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 6-4 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**

**问题描述：设某一机器由n个部件组成，每一种部件都可以从m个不同的供应商处购得。设wij是从供应商j处购来的部件i的重量，cij是相应的价格。**

**试设计一个算法，给出总价格不超过c的最小重量机器设计。**

**算法设计：对于给定的机器部件重量和机器部件价格，计算总价值不超过d的最小重量机器设计。**

**（二）算法思路**（用文字简单说明）

**直接以质量最轻作为优先条件：**

可以进行剪枝。

**这样的话只需要一个优先队列，对每个零件循环三个厂家，判断是否需要剪掉枝干**

**优化1：**

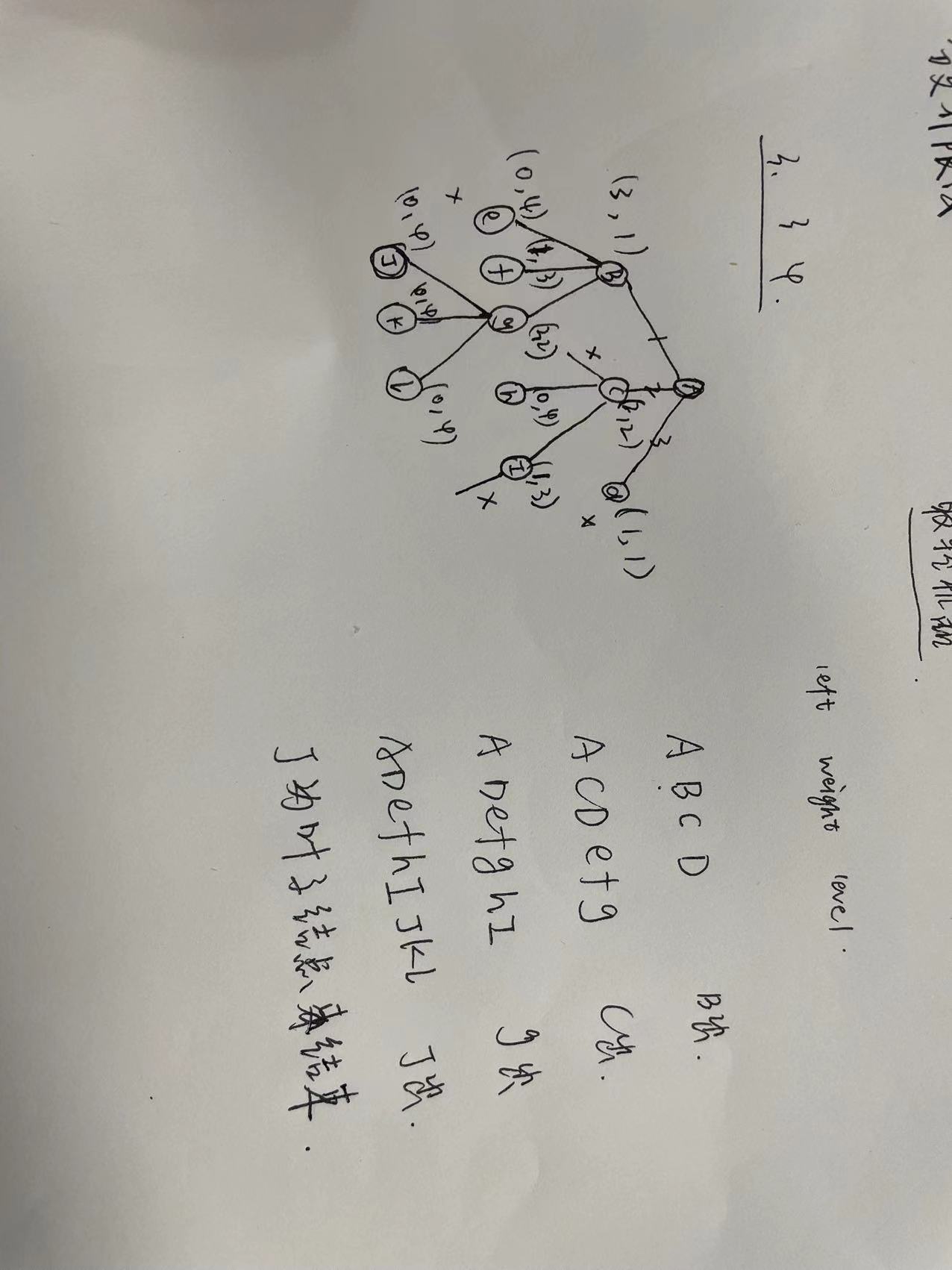
int lestcost[4] = {0,1,1,2};

特点：加入数组lestcost,这个数组的意思是每个机器至少需要多少钱，如果剩余的钱不能满足的话可以直接剪枝，这样的话可以减少节点的遍历，因为优先级和钱没有关系，所以剩余金额很少的节点依旧有可能拥有高优先级，因此，通过这样的判断就可以大大减少没有必要的节点入队来干扰。

优化2：

使用2个优先队列，因为这道题目还是以质量优先为主的，但是钱的话也是对结果有影响的，但是这两个变量很难混在一起衡量，这样的假如我设置两条优先队列，每次从第一个队列弹出来的元素，我都把优先级相同的元素全部弹出来，全部放到第二个以剩余钱的多少为优先级的队列，弹出一个最优的继续执行。这里还有一个处理的细节，我还比较了一下level

思路：弹出优先队列的首元素之后（该元素是质量最轻，剩余前最多的），队列2中剩余的元素还得放回队列1，在这里设想，质量都相同了，剩余的钱还没人家多，假如连level还小于人家的话，直接删除不用考虑了，等于也不用了，因为同一层级的已经有更优的了。（下面有例子证明这一步操作很有作用）



**（三）算法实施步骤和流程**（伪代码/流程图等方式描述）

未优化代码：

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

int cost1 = parent->nowcost + cost[level][chosen];//记录如果加入

if (cost1 <= limit) {//&&left>=leftneed

int weight1 = parent->nowweight + weight[level][chosen];

treenode\* T = new treenode;

T->chosen = chosen;

T->nowcost = cost1;

T->parent = parent;

T->nowweight = weight1;

queuenode\* Q = new queuenode;

Q->nodeontree = T;

Q->level = level;

Q->piority = weight1;//暂定优先级是质量

Q->piority2 = left;

q.push(\*Q);//入队

}

return;

}

void BBLessweight() {

int nowlevel = 0;

treenode head = { NULL,0,0 };

treenode\* parents = &head;//初始化指向头节点

queuenode qtemp;

queuenode qtemp2;

while (nowlevel < n) {

for (int i = 1; i <= m; i++) addqueue(nowlevel+1, i,parents);

if (q.empty() == 1) {

bestw = 0;

return;

}

qtemp = q.top();

q.pop();

parents = qtemp.nodeontree;

nowlevel = qtemp.level;

}

treenode\* b = parents;

bestw = b->nowweight;

for (int i = n; i > 0; i--,b = b->parent) bestlist[i] = b->chosen;//存储结果

return;

}

**优化后：**

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

int cost1 = parent->nowcost + cost[level][chosen];//记录如果加入

int left = limit - cost1;//剩余的钱

int leftneed = 0;

for (int i = level + 1; i <= n; i++) leftneed += lestcost[i];

if (cost1 <= limit && left >= leftneed) {

int weight1 = parent->nowweight + weight[level][chosen];

treenode\* T = new treenode;

T->chosen = chosen;

T->nowcost = cost1;

T->parent = parent;

T->nowweight = weight1;

queuenode\* Q = new queuenode;

Q->nodeontree = T;

Q->level = level;

Q->piority = weight1;//暂定优先级是质量

Q->piority2 = left;

q.push(\*Q);//入队

}

return;

}

void BBLessweight() {

int nowlevel = 0;

treenode head = { NULL,0,0 };

treenode\* parents = &head;//初始化指向头节点

queuenode qtemp;

queuenode qtemp2;

while (nowlevel < n) {

for (int i = 1; i <= m; i++) {

addqueue(nowlevel+1, i,parents);

}

if (q.empty() == 1) {

bestw = 0;

return;

}

qtemp = q.top();

q.pop();

int piority = qtemp.piority;

q2.push(qtemp);

while (piority == qtemp.piority && q.empty() == 0) {//把优先级相同的都拉出来

qtemp2 = q.top();

if (piority == qtemp2.piority) {

q.pop();//弹出

q2.push(qtemp2);

}

else break;

}

qtemp = q2.top();//读取二级优先队列的最优元素

q2.pop();

int levelnew = qtemp.level;

while (q2.empty() == 0) {

qtemp2 = q2.top();

if(qtemp2.level>levelnew) q.push(qtemp2);

q2.pop();

}

parents = qtemp.nodeontree;

nowlevel = qtemp.level;

}

treenode\* b = parents;

bestw = b->nowweight;

for (int i = n; i > 0; i--,b = b->parent) bestlist[i] = b->chosen;

return;

}

**（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;

int weight[4][4] = {

{0,0,0,0},

{0,1,2,3},

{0,3,2,1},

{0,5,4,3}

};

int cost[4][4] = {

{0,0,0,0},

{0,1,2,3},

{0,3,2,1},

{0,3,4,5}

};

int n = 3;//表示零件个数

int m = 3;

int limit = 10;

int bestlist[4] = { 0 };//结果序列

int lestcost[4] = {0,1,1,2};

int bestw = 0;

struct treenode {

treenode\* parent;//父亲节点

int chosen;//选择的商家

int nowweight;//目前这个定点覆盖用了多少个顶点

int nowcost;

};

struct queuenode {

int level;//记录位置

int piority;//优先级

int piority2;

treenode\* nodeontree;//树上的节点

};

struct cmp1

{

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

return a.piority > b.piority;

}

};

struct cmp2

{

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

return a.piority2 > b.piority2;

}

};

priority\_queue <queuenode, vector<queuenode>, cmp1 > q;//优先队列

priority\_queue <queuenode, vector<queuenode>, cmp2 > q2;//优先队列

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

int cost1 = parent->nowcost + cost[level][chosen];//记录如果加入

int left = limit - cost1;//剩余的钱

int leftneed = 0;

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

leftneed += lestcost[i];//计算出剩下的钱最多需要多少

}

if (cost1 <= limit && left >= leftneed) {

//开始包装

int weight1 = parent->nowweight + weight[level][chosen];

treenode\* T = new treenode;

T->chosen = chosen;

T->nowcost = cost1;

T->parent = parent;

T->nowweight = weight1;

queuenode\* Q = new queuenode;

Q->nodeontree = T;

Q->level = level;

Q->piority = weight1;//暂定优先级是质量

Q->piority2 = left;

q.push(\*Q);//入队

}

return;

}

void BBLessweight() {

int nowlevel = 0;

treenode head = { NULL,0,0 };

treenode\* parents = &head;//初始化指向头节点

queuenode qtemp;

queuenode qtemp2;

while (nowlevel < n) {

for (int i = 1; i <= m; i++) {

addqueue(nowlevel+1, i,parents);

}

if (q.empty() == 1) {

bestw = 0;

return;

}

qtemp = q.top();

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

q.pop();

int piority = qtemp.piority;

q2.push(qtemp);

while (piority == qtemp.piority && q.empty() == 0) {//把优先级相同的都拉出来

qtemp2 = q.top();

if (piority == qtemp2.piority) {

q.pop();//弹出

q2.push(qtemp2);

}

else break;

}

qtemp = q2.top();//读取二级优先队列的最优元素

q2.pop();

int levelnew = qtemp.level;

while (q2.empty() == 0) {

qtemp2 = q2.top();

if(qtemp2.level>levelnew) q.push(qtemp2);

q2.pop();

}

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

parents = qtemp.nodeontree;

nowlevel = qtemp.level;

}

treenode\* b = parents;

bestw = b->nowweight;

for (int i = n; i > 0; i--,b = b->parent) {

bestlist[i] = b->chosen;//存储结果

}

return;

}

int main(void) {

int c = 0, w = 0;

int x[4] = { 0 };

BBLessweight();

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

printf("%-3d%-3d%-3d\n\n", n, m, limit);

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

for (int j = 1; j <= m; j++) {

printf("%-3d", cost[i][j]);

}

cout << endl;

}

printf("\n");

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

for (int j = 1; j <= m; j++) {

printf("%-3d", weight[i][j]);

}

cout << endl;

}

//调用函数

//BBLessweight();

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

if (bestw == 0) {

cout << "no result" << endl;

}

else {

cout << bestw << endl;

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

printf("%-3d", bestlist[i]);

}

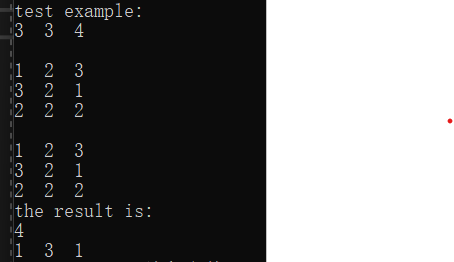
}

return 0;

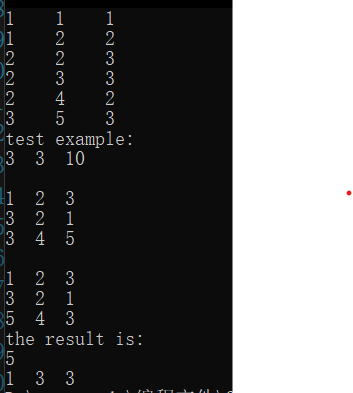
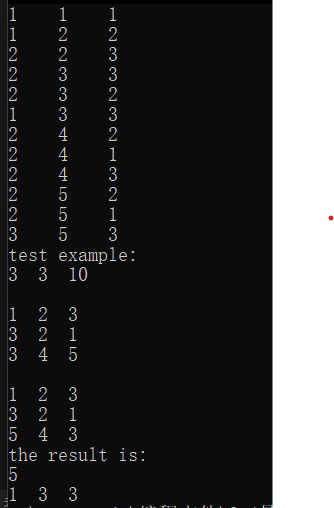
}

**（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）

测试样例1：



测试样例2：

测试样例2比较了优化过后和没有优化过后从队列里里面弹出元素的数量，在testexample可以看到这个差别，我在弹出之后做了一个输出：

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

可以看到输出的行数有明显的差异。

**（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.对于最轻机器找最优解的话，如果采用的是回溯，很显然需要遍历所有可行节点才能找到最优解，但是如果使用分支限界法的话可以尽早的结束算法尽可能的访问更少的节点从而提高算法的效率。

**暨南大学本科实验报告专用纸(附页)**