第六章算法实现题

学号：2209060322　　姓名：梁桐　　班级：计算机 2203

6-2

题目：

6-2　最小权顶点覆盖问题。

问题描述：给定一个赋权无向图 $G=(V,E)$，每个顶点 $v\in V$ 都有权值 $w(v)$。如果 $U\subseteq V$，且对任意 $(u,v)\in E$ 有 $u\in U$ 或 $v\in U$，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。

算法设计：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。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 $1,2,\cdots,n$。第 2 行有 n 个正整数表示 n 个顶点的权。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示图 G 的一条边 (u,v)。

结果输出：将计算的最小权顶点覆盖的顶点权之和以及最优解输出到文件 output.txt。文件的第 1 行是最小权顶点覆盖顶点权之和；第 2 行是最优解 x_i（$1\leqslant i\leqslant n$），$x_i=0$ 表示顶点 i 不在最小权顶点覆盖中，$x_i=1$ 表示顶点 i 在最小权顶点覆盖中。

输入文件示例	输出文件示例
input.txt	output.txt
7 7 1 100 1 1 1 100 10 1 6 2 4 2 5 3 6 4 5 4 6 6 7	13 1 0 1 1 0 0 1

算法思路：

1. 如何判断图是否被覆盖

判断图是否被覆盖是关键步骤之一。在最小权顶点覆盖问题中，一个顶点集合 U 被认为是图 G 的顶点覆盖，当且仅当对于每条边 (u, v)，至少有一个端点在集合 U 中。

为了实现这一点，我们可以使用一个辅助数组 c，其中 c[j] = 0 表示顶点 j 没有被集合 U 覆盖，且没有任何一个已选择的顶点与 j 相连。反之，如果 c[j] ≠ 0，则表示顶点 j 已被覆盖。

2. 优先级的确定

由于最小权顶点覆盖的目标是找到一个顶点集合，使得其权重之和最小，因此我们需要通过顶点的权重来设置优先级。为了实现这一点，可以使用一个最小堆（优先队列），其中优先级为顶点的权重。每次选择当前权重最小的顶点进行处理，优先队列会自动保持元素的顺序。

在分支限界法中，通常我们使用顶点的权重作为优先级来进行分支。由于权重较

小的顶点可以帮助我们更快地达到最小覆盖，所以我们优先选择这些顶点。

3. 界限函数
界限函数用于判断某个分支的潜在最优解是否有希望超越当前最优解，从而决定是否继续探索该分支。然而，在最小权顶点覆盖问题中，由于没有明确的界限函数来约束右孩子路径（即不选择当前顶点的路径），我们不能在不处理右孩子的情况下排除一些分支。因此，即使右孩子路径的点权和更小，我们也不能直接跳过该分支，必须继续计算。

4. 将活结点加入队列
在分支限界法中，我们需要根据当前的状态（即当前顶点集合、顶点的覆盖情况）生成新的活结点，并将其加入优先队列中。每次从队列中弹出一个结点进行扩展，并生成两个子结点：一个表示当前顶点被加入集合 U 中，另一个表示当前顶点没有被加入集合。

在加入新结点时，需要更新以下几个信息：
优先级：即当前结点的权重总和。若当前结点将顶点 i 加入集合 U，则新的权重总和是旧权重总和加上 i 的权重。
结果向量：记录哪些顶点已经加入集合 U。
覆盖数组 c：标记哪些顶点已经被覆盖，哪些还没有被覆盖。如果将顶点 i 加入集合 U，则需要更新与 i 相连的顶点的覆盖状态。

算法的基本思路

1. 初始化：
   设定一个优先队列 H。
   初始化一个结点 E 表示当前状态，其中包含：
      结果向量 x（记录哪些顶点已经加入到集合 U 中）。
      覆盖数组 c（记录哪些顶点已经被覆盖）。
      当前权重和 cn（当前选中的顶点集合的权重和）。
   将初始结点 E 加入优先队列。

2. 扩展结点：
   从优先队列中弹出当前权重和最小的结点 E。
   对结点 E 进行扩展，生成两个子结点：
      一个结点表示将顶点 i 加入集合 U。
      另一个结点表示不将顶点 i 加入集合。
   更新新结点的优先级、结果向量和覆盖数组，然后将这两个新结点加入优先队列。

3. 检查覆盖情况：
   每当从队列中弹出一个结点 E，需要判断图是否被完全覆盖。如果被完全覆盖且当前权重和小于最优解，则更新最优解。

4. 结束条件：

优先队列为空时，表示所有可能的顶点集合都已被处理，算法结束，输出最优解。

代码

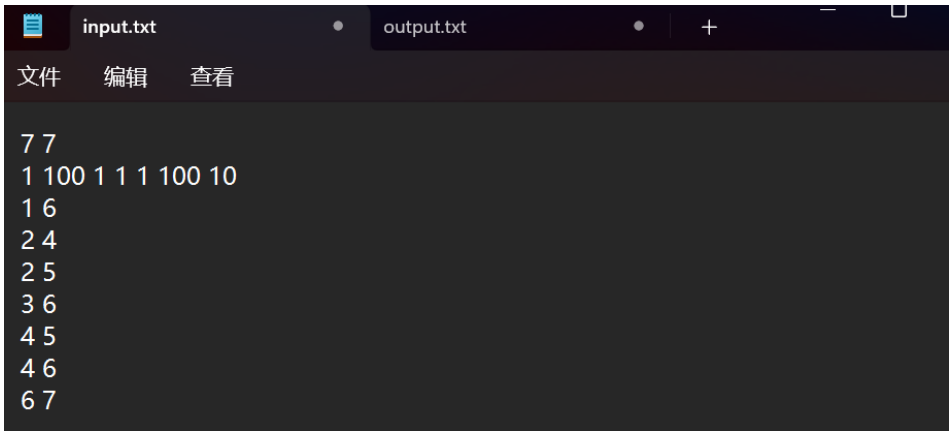
```
#include <iostream>
#include <fstream>
#include <queue>
using namespace std;

class HeapNode {
    friend class VC;  // 求解最小权覆盖问题的类，融合了所有函数和所需的参数
public:
    operator ()(int x, int y) const { return x < y; }  // 定义优先级
private:
    int i, cn, *x, *c;  // i 表示结点序号, cn 表示当前权重, x 表示结果数组, c 数组表示此时是否有一点 i 属于 U，且 i 与 j 相连，如果有，则 c[j] != 0
};

// 解最小权顶点覆盖大类
class VC {
    friend int MinCover(int **, int[], int);
private:
    void BBVC();
    bool cover(HeapNode E);  // 判断图是否已经被全部覆盖了
    void AddLiveNode(priority_queue<HeapNode>& H, HeapNode E, int cn, int i, bool ch);
    int **a, n, *w, *bestx, bestn;  // 邻接矩阵，节点数目，每个点的权重，结果向量，最优解
};

void VC::BBVC() {
    priority_queue<HeapNode> H;
    HeapNode E;  // 扩展节点
    E.x = new int[n + 1];  // 开辟结果向量
    E.c = new int[n + 1];  // 开辟一数组，用于判断图是否被完全覆盖
    for (int j = 1; j <= n; j++) {
        E.x[j] = E.c[j] = 0;
    }
    int i = 1, cn = 0;  // 初始化当前点权总和为 0
    while (true) {
```

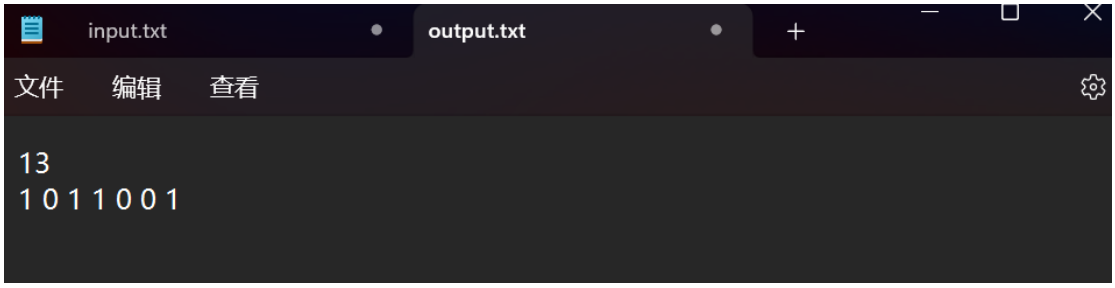
```
        if (i > n) {
            if (cover(E)) {
                for (int j = 1; j <= n; j++)
                    bestx[j] = E.x[j];
                bestn = cn;
                break;
            }
        } else {
            if (!cover(E)) // 如果当前没有完全覆盖，就将这个点加入到U集合中
                AddLiveNode(H, E, cn, i, 1);
            AddLiveNode(H, E, cn, i, 0);
        }
        if (H.size() == 0)
            break;
        E = H.top();
        H.pop();
        cn = E.cn;
        i = E.i + 1;
    }
}

// 判断图是否完全覆盖
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;
}

// 主函数
int main() {
    ifstream inputFile("input.txt");
    ofstream outputFile("output.txt");

    int n, e, u, v;  // 结点数, 边数, u, v 为结点编号
    inputFile >> n >> e;

    int **a = new int*[n + 1];
    for (int i = 0; i <= n; i++) {
        a[i] = new int[n + 1]();  // 初始化为 0
    }

    int* p = new int[n + 1];  // 定义结果向量

    for (int i = 1; i <= e; i++) {
        inputFile >> u >> v;
        a[u][v] = 1;
        a[v][u] = 1;
    }
```

```
    int* w = new int[n + 1];  // 顶点权重数组
    for (int i = 1; i <= n; i++) {
        inputFile >> w[i];
    }

    int minCoverValue = MinCover(a, w, n);

    outputFile << minCoverValue << endl;
    for (int i = 1; i <= n; i++) {
        outputFile << (p[i] ? 1 : 0) << " ";
    }
    outputFile << endl;

    inputFile.close();
    outputFile.close();

    // 释放内存
    delete[] p;
    delete[] w;
    for (int i = 0; i <= n; i++) {
        delete[] a[i];
    }
    delete[] a;

    return 0;
}
```

运行结果：





题目

6-3　无向图的最大割问题。

问题描述：给定一个无向图 $G=(V, E)$，设 $U \subseteq V$ 是 G 的顶点集。对任意$(u, v) \in E$，若 $u \in U$ 且 $v \in V-U$，就称(u, v)为关于顶点集 U 的一条割边。顶点集 U 的所有割边构成图 G 的一个割。G 的最大割是指 G 中所含边数最多的割。

算法设计：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 的最大割。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m，表示给定的图 G 有 n 个顶点和 m 条边，顶点编号为 $1, 2, \cdots, n$。接下来的 m 行中，每行有 2 个正整数 u 和 v，表示图 G 的一条边(u, v)。

结果输出：将计算的最大割的边数和顶点集 U 输出到文件 output.txt。文件的第 1 行是最大割的边数；第 2 行是表示顶点集 U 的向量 x_i（$1 \leqslant i \leqslant n$），$x_i=0$ 表示顶点 i 不在顶点集 U 中，$x_i=1$ 表示顶点 i 在顶点集 U 中。

算法分析与思路：
需求分析与算法设计

1. 问题描述

给定一个无向图 G = (V, E)，其中 V 是顶点集，E 是边集。对于任意边 (u, v) ∈ E，如果 u ∈ U 且 v ∈ V - U，则称边 (u, v) 是关于顶点集 U 的一条割边。所有割边构成图 G 的一个割。我们的目标是计算图 G 的最大割，即包含最多边的割。

输入文件示例	输出文件示例
input.txt	output.txt
7 18	12
1 4	1 1 1 0 1 0 0
1 5	
1 6	
1 7	
2 3	
2 4	
2 5	
2 6	
2 7	
3 4	
3 5	
3 6	
3 7	
4 5	
4 6	
5 6	
5 7	
6 7	

2. 问题分析

1. 解向量：解向量 (x1, x2, …, xn) 是一个二进制序列，其中 xi = 1 表示顶点 i 在割集 U 中，xi = 0 表示顶点 i 不在割集 U 中。

2. 解空间：解空间是一个完全二叉树。树的每一层代表一个顶点，每个结点表示对当前顶点的选择（是否加入到割集中）。

3. 约束条件：每个结点代表一个部分解。我们通过扩展树的结点来不断更新部分解。当考虑右子结点时，若当前割边数加上剩余边数已经不可能超过当前最优解，则可以剪枝，不再扩展此结点。

4. 优先级：每个结点的优先级根据当前的割边数来决定。割边数越多的结点优先处理。

3. 算法的基本思想

算法的核心是使用分支限界法，通过优先队列处理解空间树。

解空间树：解空间树是一个完全二叉树，树的每一层代表一个顶点，每个结点包含当前割边数、剩余的割边数以及当前的部分解。

优先队列：我们使用优先队列来存储待处理的结点。每个结点的优先级是当前割边数。每次从队列中取出当前割边数最多的结点进行处理。

扩展结点：从当前结点出发，生成两个子结点：
左子结点表示将当前顶点加入割集中；
右子结点表示将当前顶点不加入割集中。

对于左子结点，直接更新当前的割边数。对于右子结点，在扩展之前需要判断当前的割边数加剩余边数是否大于当前最优解，如果不大于，则不再扩展该结点（剪枝）。

叶子结点：当到达叶子结点时，更新最优解。

4. 具体算法设计

1. 输入：首先输入图的顶点数 n 和边数 e，然后输入每条边的两个端点。

2. 计算最大割：
初始化当前结点，起始为根结点，割边数为 0，剩余边数为所有边数。
通过优先队列处理每个结点，计算并更新当前割边数。
每次扩展结点时，如果当前割边数加上剩余边数超过当前最优解，则继续扩展右子结点，否则剪枝。

3. 输出：输出最大割的边数和对应的顶点集 U。

代码：

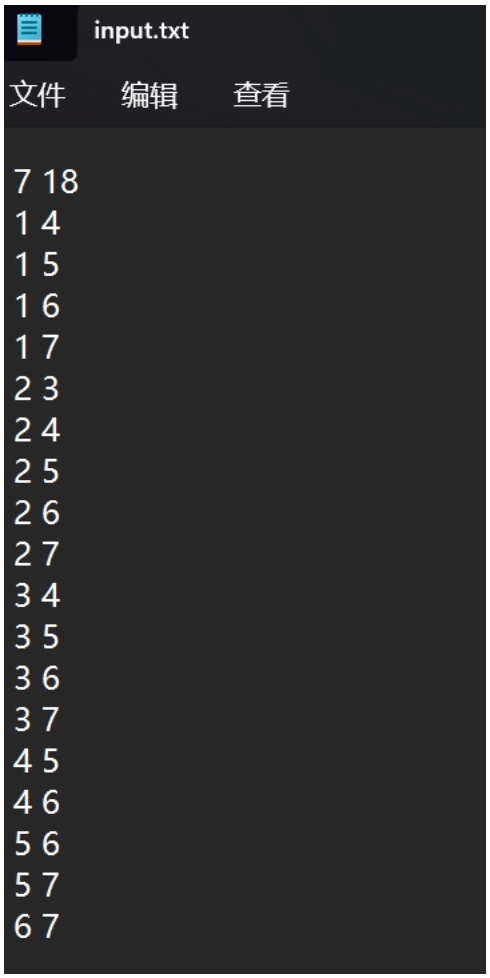
```
#include <bits/stdc++.h>
#include <fstream>
#include <iostream>
```

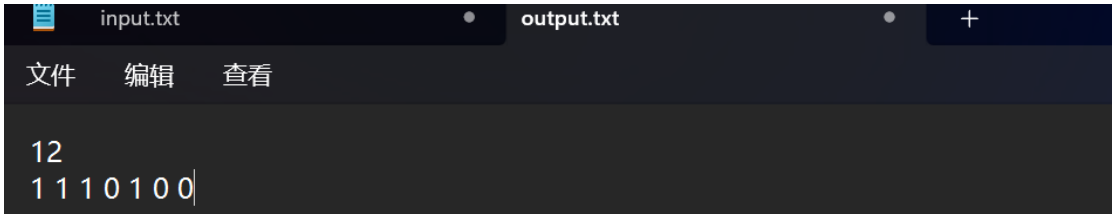
```
using namespace std;

const int MAX = 1000;
int G[MAX][MAX];  // 存储边信息
int bestx[MAX];   // 最优值的割集
int w[MAX];       // 顶点的权
int n, e;         // 顶点数，边数
ifstream in("input.txt");
ofstream out("output.txt");

// 结点类
class Node {
public:
    int dep;  // 当前层
    int cut;  // 割边数量
    int e;    // 剩余边的数量
    int *x;   // 解向量

    Node(int d, int c, int ee) {
        x = new int[n + 1];
        dep = d;
        cut = c;
        e = ee;
    }

    // 割边数大的先出队列
    bool operator < (const Node &node) const {
        return cut < node.cut;
    }
};

// 存储待解决的结点的优先队列
priority_queue<Node> q;

// 添加结点
void addNode(Node enode, bool isLeft) {
    Node now(enode.dep + 1, enode.cut, enode.e);
    copy(enode.x, enode.x + n + 1, now.x);

    // 是左结点
    if (isLeft) {
        now.x[now.dep] = 1;  // 标记是割集元素
        for (int j = 1; j <= n; j++)  // 统计割边的增量
            if (G[now.dep][j])
```

```
                if (now.x[j] == 0)  // 如果当前顶点在割集中，但边的另一个顶点不在割集
                {
                    now.cut++;  // 割边的数量增加
                    now.e--;    // 剩余边的数量减少
                }
                else
                    now.cut--;  // 否则割边数量减少
    }
    q.push(now);  // 加入优先队列
}

int search() {
    // 初始化
    Node enode(0, 0, e);
    for (int j = 1; j <= n; j++)
        enode.x[j] = 0;
    int best = 0;
    // 分支限界求解
    while (true) {
        if (enode.dep > n) {  // 到达叶子节点，如果比当前最优解更优，更新
            if (enode.cut > best) {
                best = enode.cut;
                copy(enode.x, enode.x + n + 1, bestx);
                break;
            }
        }
        else {  // 没有到达叶子节点
            addNode(enode, true);  // 加入左子结点
            if (enode.cut + enode.e > best)  // 满足约束条件，加入右子结点
                addNode(enode, false);
        }
        if (q.empty())
            break;
        else {  // 取出队首元素
            enode = q.top();
            q.pop();
        }
    }
    return best;
}

int main() {
    int a, b, i;
```

```
    in >> n >> e;
    memset(G, 0, sizeof(G));

    // 读取边的信息
    for (i = 1; i <= e; i++) {
        in >> a >> b;
        G[a][b] = G[b][a] = 1;
    }

    // 计算最大割
    int max_cut = search();
    out << max_cut << '\n';  // 输出最大割的边数

    // 输出顶点集 U 的向量
    for (i = 1; i <= n; i++) {
        out << bestx[i];
        if (i != n) out << " ";
    }
    out << '\n';

    in.close();
    out.close();
    return 0;
}
```

运行结果：


input.txt
文件 编辑 查看
7 18
1 4
1 5
1 6
1 7
2 3
2 4
2 5
2 6
2 7
3 4
3 5
3 6
3 7
4 5
4 6
5 6
5 7
6 7


input.txt
output.txt
文件 编辑 查看
12
1 1 1 0 1 0 0

题目：

6-4　最小重量机器设计问题。

问题描述：设某一机器由 n 个部件组成，每种部件都可以从 m 个不同的供应商处购得。设 w_{ij} 是从供应商 j 处购得的部件 i 的重量，c_{ij} 是相应的价格。设计一个优先队列式分支限界法，给出总价格不超过 d 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，设计一个优先队列式分支限界法，计算总价格不超过 d 的最小重量机器设计。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 3 个正整数 n、m 和 d。接下来的 $2n$ 行，每行 n 个数。前 n 行是 c，后 n 行是 w。

结果输出：将计算的最小重量，以及每个部件的供应商输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 4	4
1 2 3	1 3 1
3 2 1	
2 2 2	
1 2 3	
3 2 1	
2 2 2	

思路分析：

解题思路分析

本问题是一个典型的分支限界法问题，要求在给定成本限制的情况下，选择每个零件的供应商，使得总重量最小。具体来说，我们可以把问题建模为一个解空间树，其中每个节点代表一个部分解，每一层的节点表示选择不同零件的供应商，叶节点则表示一个完整的解。在这个过程中，我们需要设计一个剪枝策略来避免无效的计算，从而提高搜索效率。

1. 问题建模

我们将每个零件的选择看作是一个决策，每个零件有多个供应商可供选择，每个供应商的选择对应着不同的重量和价格。目标是选择零件和供应商，使得：

总成本不超过给定的限制 d。

总重量最小。

我们将整个选择过程建模为一棵解空间树：

根节点表示一个未选择零件的空解。

每个非叶子节点表示已经选择了一部分零件的解。

叶子节点表示对所有零件完成选择的解。

2. 分支限界法的思想

分支限界法的核心思想是通过优先队列和剪枝策略来减少搜索空间。其主要流程如下：

优先队列：每个节点有一个优先级，表示当前解的优劣。在这个问题中，我们的优先级是零件的总重量，因为我们要求的是最小重量。优先队列中的节点会根据当前重量来排序，最轻的方案优先扩展。

剪枝：在选择某个节点的子节点时，除了检查当前节点是否满足约束条件（即总成本不超过 d），我们还需要计算该节点的下界。如果某个节点的下界（即它能得到的最小重量）已经大于当前最优解，那么这个节点及其子树就不再继续扩展，从而进行剪枝。

具体来说，下界函数的计算方式是：

对于每个已经选择的零件，累计当前的重量和成本。

对于未选择的零件，假设每个零件都选取它的最小重量，这样得到的重量之和就是当前解的下界。

选择零件的顺序：我们按照零件的顺序进行选择，在每个节点上有 m 种选择（每个零件有 m 个供应商）。每次选择一个供应商后，我们进入下一层，直到所有零件都被选择。

3. 剪枝函数设计

剪枝函数是分支限界法的核心之一，决定了哪些分支会被丢弃。我们的剪枝函数依赖于当前重量和成本，以及下界。具体步骤如下：

当前重量：从根节点到当前节点的路径表示了已经选择的零件和供应商，累计这些选择的重量得到当前重量。

下界：在当前节点的基础上，我们假设剩下的零件都选择它们的最小重量，得到该节点的下界。通过下界可以判断某个分支是否值得进一步扩展。如果某个分支的下界大于当前最优解，则不再考虑该分支。

剪枝函数的计算可以通过预先计算每个零件的最小重量来加速。这样，当我们计算某个节点的下界时，可以直接从预先计算的数组中取值，而不需要重新计算。

4. 算法步骤

1. 初始化：读取输入数据，包括零件的重量和价格、成本约束等。计算每个零件的最小重量，并建立一个下界数组。
2. 优先队列初始化：将根节点（没有选择任何零件）放入优先队列。
3. 分支限界搜索：

从优先队列中取出优先级最高（重量最小）的节点。

对于每个未选择的零件，生成 m 个子节点，表示选择不同供应商的情况。

计算每个子节点的当前重量、当前成本和下界。

若子节点的成本超过了约束 d 或下界大于当前最优解，则剪枝（忽略该子节点）。

否则，将子节点加入优先队列继续处理。

4. 更新最优解：当到达叶节点时，检查该解是否满足约束条件并且比当前最优

解更好。如果是，则更新最优解。
5. 输出结果：遍历最优解路径，输出最小重量和每个零件对应的供应商。
代码：

```
#include <iostream>
#include <vector>
#include <queue>
#include <fstream>
#include <climits>
using namespace std;

struct node
{
    int w;          //当前重量和
    int c;          //当前花费和
    int i;          //零件序号（也是层数）
    int j;          //厂商序号
    node* parent;   //父母节点

    //用于赋值
    node(int w, int c, int i, int j, node* p) : w(w), c(c), i(i), j(j), parent(p)
{}

    //重载运算符<，用于优先队列中排序优先级
    bool operator<(const node& a) const {
        if (w != a.w)
            return w > a.w;  //w大，则优先级高
        else if (i != a.i)
            return a.i > i;  //w相同则比较层级，倾向于层数更深的节点，这样可以
更快到达叶节点
        else
            return j > a.j;  //前二者均相同，按厂商序号排
    }
};

class MinWeight
{
private:
    int n;                      //零件数
    int m;                      //厂商数
    int d;                      //成本约束
    int bestw;                  //最优重量
    vector<int> lb;             //下界数组，lb[i]表示第i到n零件最小的重量和
    vector<int> bestx;          //解
    vector<vector<int>> w, c;  //重量价值数组
```

```
public:
    MinWeight()
    {
        ifstream input("input.txt");  // 从文件读取输入
        input >> n >> m >> d;

        bestx.resize(n);
        w.resize(n);
        c.resize(n);
        for (int i = 0; i < n; i++) {
            w[i].resize(m);
            c[i].resize(m);
        }

        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                input >> c[i][j];

        //lb 的初始化，输入 w 时顺便计算
        lb.resize(n);
        for (int i = 0; i < n; i++) {
            int minw = INT_MAX;
            for (int j = 0; j < m; j++) {
                input >> w[i][j];
                //记录下第 i 个零件的最小重量
                if (w[i][j] < minw)
                    minw = w[i][j];
            }
            //初始化 lb 数组
            lb[i] = minw;
        }

        //计算 lb 数组
        for (int i = n - 2; i >= 0; i--)
            lb[i] = lb[i + 1] + lb[i];

        bestw = INT_MAX;  // 初始化最小重量
        input.close();
    }

    void BranchBound()
    {
        priority_queue<node> q;  // 小根堆
```

```
        node root(0, 0, -1, 0, NULL); // 优先队列初始化，放入根节点
        q.push(root);

        // 分枝限界的最小代价搜索
        while (!q.empty()) {
            node* curNode = new node(q.top()); // 为每个访问过的节点申请内存，
以便后续求解路径
            q.pop();

            // 到达叶子节点
            if (curNode->i == n - 1)
            {
                // 满足剪枝函数和约束条件
                if (curNode->w < bestw && curNode->c <= d)
                {
                    // 最小重量更新
                    bestw = curNode->w;
                    // 最优路径更新
                    for (node* t = curNode; t->i != -1; t = t->parent)
                        bestx[t->i] = t->j + 1; // +1是因为j从0开始计数
                }
            }

            // 未到达叶节点
            else
            {
                // 子节点加入优先队列
                for (int j = 0; j < m; j++) {
                    // 临时变量不存储
                    node child(curNode->w + w[curNode->i + 1][j], curNode->c +
c[curNode->i + 1][j], curNode->i + 1, j, curNode);
                    // 约束条件与限界条件，不满足的被忽略剪枝
                    if (child.c <= d && LB(child) < bestw)
                        q.push(child);
                }
            }
        }

        // 输出结果到文件
        ofstream output("output.txt");
        if (bestw == INT_MAX) {
            output << "无法满足成本约束条件" << endl;
            return;
        }
```

```
        output << bestw << endl;
        for (int i = 0; i < n; i++)
            output << bestx[i] << " ";
        output.close();
    }

private:
    // 计算节点分枝的重量下界
    int LB(node x)
    {
        if (x.i == n - 1)
            return x.w;
        return x.w + lb[x.i + 1];
    }
};

int main()
{
    MinWeight ex;
    ex.BranchBound();
}
```

运行结果：