****



**算法设计与分析作业：0/1背包问题算法求解性能比较**

**学 院： 智能与计算学部**

**班 级： 大类八班**

**姓名及学号： 覃钰源3022244285**

**陈秋澄3022244290**

**佟美萱3022244291**

**2023年12月17日**

**摘要**

本实验针对0/1背包问题，分别利用动态规划、回溯及分支限界三种算法，对Florida State University(FSU)中的数据集进行了计算，并将得到的结果与答案相比对，验证了算法的正确性，并对三种算法分别进行了时间复杂度和空间复杂度分析，归纳总结出不同算法求解0/1背包问题的优缺点。

1. **实验目的**

利用动态规划、回溯及分支限界三种算法进行求解，分析不同算法解决同一问题的复杂度，归纳不同求解算法的优缺点。

1. **实验设计流程**

我们知道，0/1背包问题是NP完全问题，若使用贪心算法则无法在多项式时间内得到最优解，我们便选择回溯算法、分支限界法和动态规划法进行求解。

1. **回溯法**

从根结点开始，以深度优先搜索（DFS）的方式进行搜索。根结点成为活结点，也是当前的扩展结点。

我们约定子树集左分支的值是1，因此沿着扩展结点的左分支进行扩展，代表放入背包。在此时需要判断约束条件是否成立，即该物品的重量能否放入背包。

如果成立，则扩展当前结点生成左子树，继续进行深度优先搜索，如此往复。

如果不成立，剪掉扩展结点的左分支，对右分支进行扩展，代表不放入背包。在此时，沿着右子树也可能生成最优解，所以我们这时候要判断限界条件（即已有的物品价值加上除该节点外所有剩余还未判断的物品价值能否比已有的最优值大），如果满足，则说明有可能生成最优解，此时右结点成为活结点，继续扩展。

如果不满足限界条件，则剪掉扩展的右边分支，向最近的父结点回溯。直到死结点。

1. **分支限界法**

首先定义了一个bound值作为子树扩展优先级，bound值大的结点优先扩展。

从根节点开始，首先检查待扩展结点的左子树，若将待取物品放入背包后未超过背包容量，则将左子树结点加入活结点优先队列中并更新bestp的值。再检查待扩展结点的右子树，因为右子树结点一定是可行结点，此时即计算右子树结点的bound值，若bound值小于bestp，则剪枝，反之则将右子树结点加入到活结点优先队列中。当扩展到叶节点时，算法结束，叶子节点对应的解即为问题的最优值。

1. **动态规划法**

满足最优化原理，可以分解成一个个子问题解决。利用优化原理，从枚举“放”和“不放”两种情况建立优化值之间的递归式，以代码为例，f(i,y)表示背包容量y时，可选择放入物品中的第i个到第n个时得到的优化效益值，根据优化原理可以得到如下递归关系：

由此可以得出最优解，并通过是否放入背包得到解空间。

1. **代码实现**
2. **回溯法**

#include <bits/stdc++.h>

using namespace std;

struct Node {

    int w, v;

    int id;

};

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    int n, m;

    cin >> n >> m;

    vector<Node> g(n);

    for(int i = 0; i < n; i ++) {

        cin >> g[i].w;

    }

    for(int i = 0; i < n; i ++) {

        cin >> g[i].v;

        g[i].id = i;

    }

    //按照单位质量的价值从达到小排序

    ranges::sort(g, [](const Node &a, const Node &b) {

        return 1.0 \* a.v / a.w > 1.0 \* b.v / b.w;

    });

    int curW = 0, curV = 0;

    auto bound = [&](int x) {

        int Wleft = m - curW;

        int upV = curV;

        //往左下角搜

        while(x < n && Wleft >= g[x].w) {

            upV += g[x].v;

            Wleft -= g[x].w;

            x ++;

        }

        if(x < n) {

            upV += g[x].v / g[x].w \* Wleft;

        }

        return upV;

    };

    int maxV = 0;

    vector<bool> curState(n), bestState(n);

    auto backTrack = [&](auto self, int x) {

        if(x >= n) {

            maxV = curV;

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

                bestState[i] = curState[i];

            }

            return;

        }

        if(curW + g[x].w <= m) {

            curState[g[x].id] = true;

            curW += g[x].w;

            curV += g[x].v;

            self(self, x + 1);

            curState[g[x].id] = false;

            curW -= g[x].w;

            curV -= g[x].v;

        }

        if(bound(x + 1) > maxV) {

            curState[g[x].id] = false;

            self(self, x + 1);

        }

    };

    cout << "回溯法解决01背包问题：\n\n";

    cout << "物品数量：" << n << "背包体积：" << m << "\n\n";

    for(int i = 0; i < n; i ++) {

        cout << "物品" << i << "的体积和价值为：" << g[i].w << ' ' << g[i].v << '\n';

    }

    cout << '\n';

    backTrack(backTrack, 0);

    cout << "背包能装下的最大物品价值为：" << maxV << '\n';

    cout << "解向量为：";

    for(auto c : bestState) cout << c << ' ';

    cout << '\n';

    return 0;

}

1. **分支限界法**

#include <bits/stdc++.h>

using namespace std;

struct Item {

    int w, v, id;

};

*//优先队列优化*

struct Node {

    int ne, totW, totV;

    double upV;

    vector<int> items;*//每个节点都选了哪些物品*

    bool operator < (const Node &*T*) const {

        return upV < *T*.upV;

    }

};

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(nullptr);

    int n, m;

    cin >> n >> m;

    vector<Item> g(n), tmp(n);

    for(int i = 0; i < n; i++) cin >> g[i].w, tmp[i].w = g[i].w;

    for(int i = 0; i < n; i++) cin >> g[i].v, g[i].id = i, tmp[i].v = g[i].v;

*//按价值密度排序*

    ranges::sort(g, [](const Item &*a*, const Item &*b*) {

        return 1.0 \* *a*.v / *a*.w > 1.0 \* *b*.v / *b*.w;

    });

*//计算上界*

    auto bound = [&](int *x*, int *curW*, int *curV*) {

        int Wleft = m - *curW*;

        double upv = *curV*;

        while(*x* < n && g[*x*].w <= Wleft) {

            Wleft -= g[*x*].w;

            upv += g[*x*].v;

*x* ++;

        }

        if(*x* < n) {

            upv += (1.0 \* g[*x*].v / g[*x*].w) \* Wleft;

        }

        return upv;

    };

    auto bfs = [&] {

        Node r = {0, 0, 0, bound(0, 0, 0), {}};

        priority\_queue<Node> q;

        q.push(r);

        Node bestNode;

        int ans = 0;

        while(q.top().upV > ans) {

            auto t = q.top();

            q.pop();

            if(t.ne == n) {

                if(t.totV > ans) {

                    ans = t.totV;

                    bestNode = t;

                }

            } else {

                Node t2 = t;

                if(t.totW + g[t.ne].w <= m) {

                    t.upV = bound(t.ne + 1, t.totW + g[t.ne].w, t.totV + g[t.ne].v);

                    if(t.upV > ans) {

                        t.totV += g[t.ne].v;

                        t.totW += g[t.ne].w;

                        t.items.push\_back(g[t.ne].id);

                        t.ne ++;

                        q.push(t);

                    }

                }

                t2.upV = bound(t2.ne + 1, t2.totW, t2.totV);

                if(t2.upV > ans ) {

                    t2.ne ++;

                    q.push(t2);

                }

            }

        }

        cout << "分支限界解决01背包问题\n\n";

        cout << "物品个数：" << n << ' ' << "背包容量：" << m << "\n\n";

        for(int i = 0; i < n; i ++) {

            cout << "物品" << i + 1 << "的体积价值：" << tmp[i].w << ' ' << tmp[i].v << '\n';

        }

        cout << '\n';

        vector<bool> selected(n);

        for(int id : bestNode.items) {

            selected[id] = true;

        }

        cout << "最大价值为：" << ans << '\n';

        cout << "解向量为：";

        for(auto c : selected) cout << c << ' ';

        cout << '\n';

    };

    bfs();

    return 0;

}

1. **动态规划**

#include <bits/stdc++.h>

using namespace std;

int main() {

    // ios::sync\_with\_stdio(false);

    // cin.tie(nullptr);

    // cout << "物品个数和背包体积：";

    int n, m;

    cin >> n >> m;

    // cout << "\n";

    // cout << "各个物品的体积和价值：\n";

    vector<int> w(n), v(n), dp(m + 1); //w代表体积, v代表价值，dp[i]表式容量为i的背包能够装入的最大价值

    // cout << "重量：\n";

    for(int i = 0; i < n; i++ ) {

        // cout << "物品" << i + 1 << ": ";

        cin >> w[i];

    }

    // cout << "\n";

    // cout << "体积：\n";

    for(int i = 0; i < n; i ++) {

        // cout << "物品" << i + 1 << ": ";

        cin >> v[i];

    }

    // cout << "\n";

    //状态转移

    for(int i = 0; i < n; i ++) {

        for(int j = m; j >= w[i]; j --) {

            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);

        }

    }

    //求解向量

    vector<bool> is\_selected(n);

    int V = m;

    for(int i = n - 1; i >= 0; i --) {

        if(dp[V] == dp[V - w[i]] + v[i]) {

            is\_selected[i] = true;

            V -= w[i];

        } else if(dp[V] == dp[V - w[i]]) {

            is\_selected[i] = false;

        }

    }

    cout << "动态规划解决01背包问题：\n\n";

    cout << "物品个数：" << n << "  背包体积：" << m << "\n\n";

    for(int i = 0; i < n; i ++) {

        cout << "物品" << i + 1 << "体积价值: ";

        cout << w[i] << " " << v[i];

        cout << "\n";

    }

    cout << '\n';

    cout << "最大价值为：" << dp[m] << '\n';

    cout << "解向量为：";

    for(auto c : is\_selected) {

        cout << c << ' ';

    }

    cout << '\n';

    return 0;

}

1. **实验结果**
2. **回溯法**

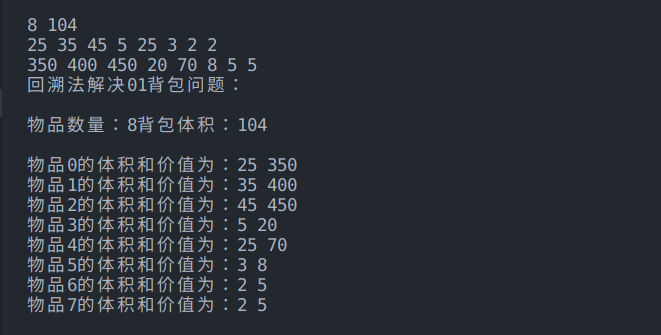


图1回溯法代码测试（数据来自FSU-05）

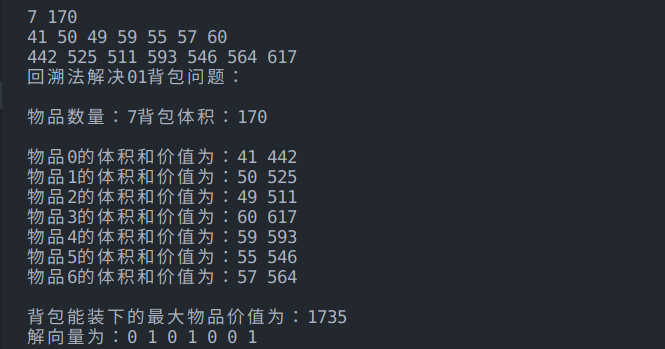


图2 回溯法代码测试（数据来自FSU-06）

1. **分支限界法**

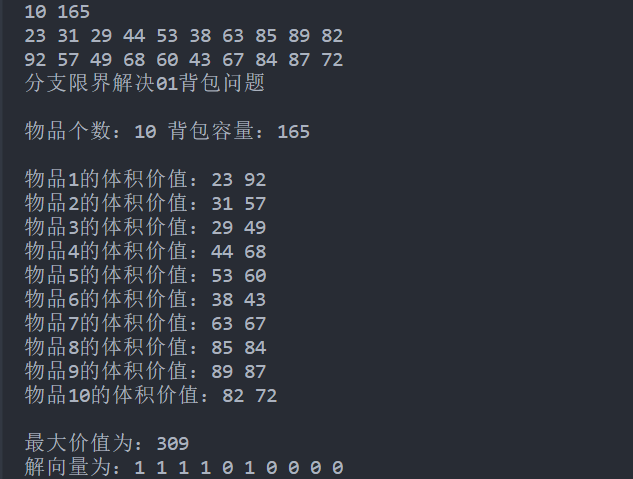


图3分支限界代码测试（数据来自FSU-01）

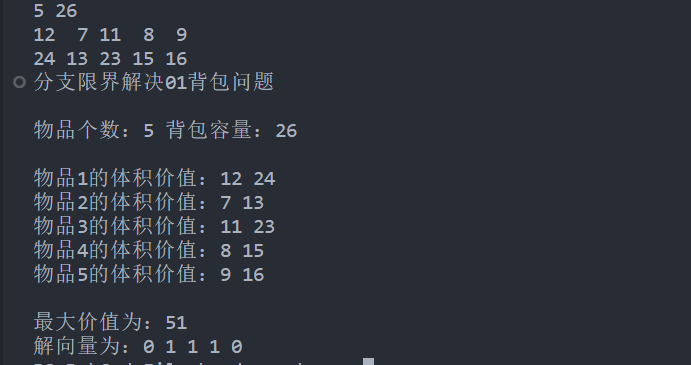


图4 动态规划代码测试（数据来自FSU-02）

1. **动态规划**



图5 动态规划代码测试（数据来自FSU-03）

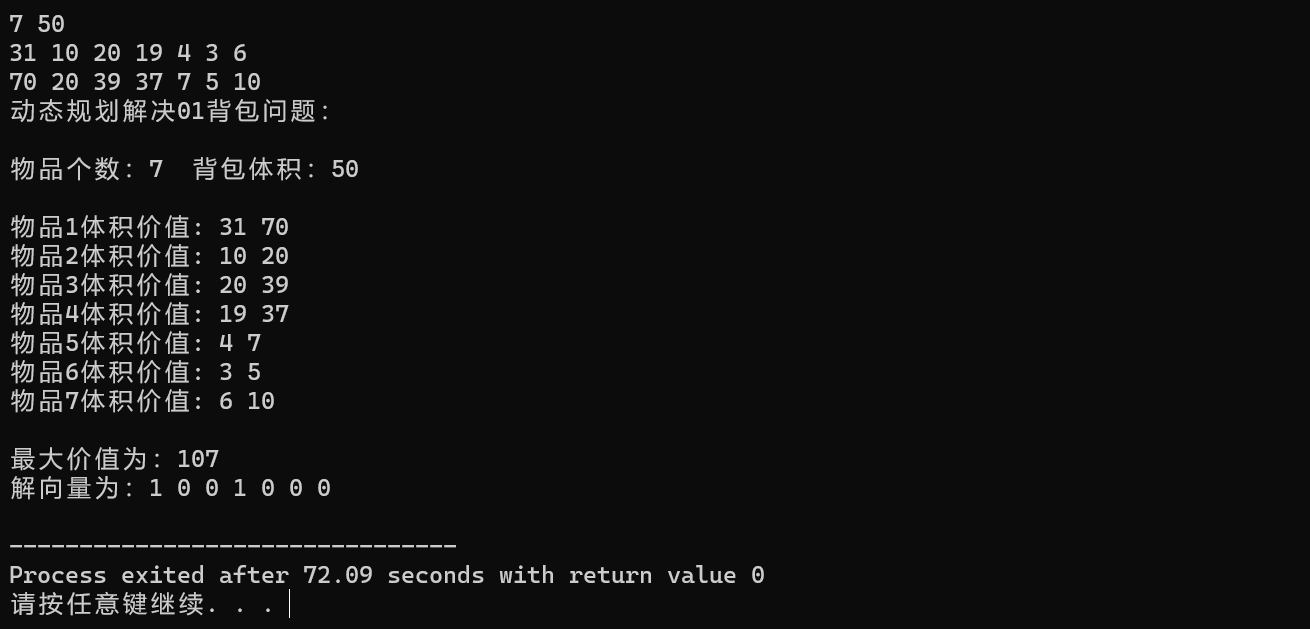


图6 动态规划代码测试（数据来自FSU-04）

1. **复杂性分析**
2. **回溯法**

首先子集树结点为2^(N+1)-1个，即O(2^N)个。在每个结点处都需要求解限界函数，时间复杂度显然为O(N)。所以回溯的时间复杂度为O(N\*2^N)。而排序需要花费O(NlogN)时间，在渐进意义下，总时间复杂度为O(N\*2^N)。

搜索深度最多为N+1，即O(N)，而排序的栈空间开销为O(logN)，渐进意义下的空间复杂度为O(N)

1. **分支限界法**

分支限界法确实是比回溯法更简单，因为我们广度优先遍历，而且每次找优先级最高的，当达到叶结点时一定是最优的。算法的时间复杂度主要是在计算最优解的过程，这取决于我们达到叶结点的时候经过了多少节点，有可能我们没有遍历完一层，下一层的优先级更高，我们就会更进一层，所以取决于我们中间经历过的结点。树的结点数目为2^N。

但是由于最后一层我们只会找到一个结点，而最后一层的结点数目应该是2^(N-1)。

所以我们的最坏时间复杂度应为2^(N-1)(2^N-2^(N-1))，但基本不可能实现。因为这是一个非常松的上界，并且我们不难发现回溯法和分支限界法都比号称O(N^2)时间复杂度的动态规划快，这证明剪枝时剪去了非常多的枝条限界函数行之有效。

空间复杂度O(2^N)，这是个非常松的上界，因为我们剪去了很多树枝，事实上从运行空间大小来看，与回溯法相差无几，我们有理由相信经过剪枝后的分支限界法的运行空间在大多数情况下是接近O(N)的。

1. **动态规划法**

时间复杂度O(NC)，两层循环，外层循环为N，内层循环为C，总时间复杂度为O(NC)。若利用滚动数组转化为一维则时间复杂度为O(N)。若利用课上的递归则最差时间复杂度为O(2^N)。

空间复杂度O(N)。

1. **总结**
2. **适用性**

动态规划可以解决的题目相对较少，需要题目满足最优子结构和重叠子问题两个条件。

回溯法和分支限界法都是通用求解方法，适用范围更广，可以用来求解NP问题。但是一般分支限界法会倾向于去求解最优解问题，当然也可以用来求解所有可行解，但是此时我们会更倾向于使用回溯法。

1. **使用难度**

动态规划涉及到最优子结构和重叠子问题的证明，状态的定义，状态转移方程的构造等。想要设计一个正确的动态规划算法难度较大。

回溯法和分支限界法的编写难度较低，只需对解空间树进行搜索即可。但是约束函数和限界函数的编写的好坏对算法的效率影响非常大，如果不进行优化，这两个算法的时间和空间花费是不可接受的。剪枝是一个技巧性很强的操作，想要写出一个高效的回溯法或者分支限界法难度也较高。回溯法和分支限界法应该属于易学难精的算法，但是如果真的写出了优秀的回溯法或者分支限界法，它们的效率会非常高。

1. **算法效率**

动态规划的时间复杂度和空间复杂度一般都是多项式级别的，相对于一般的暴力解法会高效很多。

回溯法和分支限界法的时间复杂度上界虽然很大，但是经过剪枝可以达到非常高效。但是到底效率能达到多高与剪枝的方法息息相关，一个好的剪枝方案可以带来优越的算法，而一个差的剪枝方案会带来一个没有实用性的算法。也就是说虽然回溯法和分支限界法的效率的上限很高，但是下限也很低。

1. **算法实现**

我们利用三种算法都正确地实现了0/1背包问题的求解，并进行了复杂度分析。