天津大学

算法设计与分析期末大作业

题目：对 **0-1** 背包问题的多种求解算法分析

学　院：智能与计算学部

专　业：计算机科学与技术

年　级：**2020** 级

姓　名：刘锦帆

学　号：**3020202184**

2021 年 **12** 月 **25** 日

摘　　要

本实验针对 0/1 背包问题，通过使用动态规划、回溯法和分支限界法，对 Florida State University 的 5 个数据集进行问题求解与性能的分析。本实验分析并对比了三种算法的时间和空间开销，并通过插入测量变量，进行实际测量，加以验证。同时，分析比对了是否使用优先队列进行剪枝优化的分支限界方法的实际表现与回溯法进行对比。结果表明，剪枝的回溯法在时间复杂度方面显著优于朴素的回溯法、朴素的分支限界法以及动态规划方法；回溯法在特殊数据集内明显优于分支限界法；使用优先队列的搜索算法在有限的数据集内，相较于朴素的搜索算法，没有特别突出的表现，但在较大数量级上，有更优的效率。

关键词： 0/1 背包 动态规划 回溯法 分支限界法 算法比较

目　录

第一章　　实验目的

0-1 背包问题 (0-1 Knapsack Problem, 0-1KP) 的主要内容是：给定一个背包最大承受重量为 W，以及 n 个物品，第 i 个物品的重量为 w_i，价值为 v_i，并限定每种物品只能选择 0 个或 1 个，求在满足总重量小于 W 的前提下，总价值能达到的最大值。

0-1 背包问题是一种组合优化的 NP 问题，对其的深入研究可以惠及商业、组合数学、计算复杂性理论、密码学和应用数学中的研究。

通过对该问题的研究，可以加深我们对 NP 问题的理解，以及对动态规划、回溯法、分支限界法以及其优化的理解；通过对该问题的实际编程实现与实践，我们能更加熟练的掌握与体会、巩固对知识的掌握与运用能力。

第二章　　实验设计流程

2.1　动态规划算法设计

我们利用第一章中使用的记号，进行接下来的假设与计算。

首先，我们再次重述问题的定义：在不超过给定总重量 W 的情况下，选取物品使得最终的价值总和最大。该问题的限制条件为总重量；求解的目标为总价值的最大，是一个最优化问题。于是我们得出一个启发式的动态规划的转移状态：$f[w_{tot}]$，在总重量小于等于 w_{tot} 时的最大总价值。该状态需要转移，考虑单个转移方程：在增加一个物品 i 时，总重量增加了 w_i 且总价值增加了 v_i，即应该有：

$$f[w_{tot} + w_i] = f[w_{tot}] + v_i \tag{2-1}$$

但这样的方程还不够清晰，于是我们对这个一维的状态数组进行二维的扩展：$f[n][w_{tot}]$ 表示在选择了 n 个物品之后，组合总重量小于等于 w_{tot} 时的最大总价值。这样一来，我们的转移方程变为：$f[n][w_{tot} + w_i] = f[n-1][w_{tot}] + v_i$，同理，这个方程还可以写为：

$$f[n][w_{tot}] = f[n-1][w_{tot} - w_i] + v_i \tag{2-2}$$

乍一看这个方程已经非常完美了，但分析这个方程的物理意义还是可以发现，如果我们利用这个方程进行状态转移，我们在对每个物品进行评判时，隐式的认为应当选择它，即在分析加入第 i 个物品后，前 i 个物品的最优解一定包含第 i 个物品，这显然时不正确的。所以需要对其进行改进，即，可能存在不加入第 i 个物品，才是最优解，具体改变则为：

$$f[n][w_{tot}] = max\{f[n-1][w_{tot} - w_i] + v_i, f[n-1][w_{tot}]\} \tag{2-3}$$

在具体状态转移时，我们采取从第 1 个物品到最后一个物品的顺序进行。类似填写一张 n 行 W 列的表格。每一个格子对应一个 $f[n][w_{tot}]$，每一行对应选择前 n 个物品时，所有重量可能性下的价值最大值。

2.2　回溯法算法设计

回溯法本质是一种搜索算法，即搜索所有的解空间，在 0-1 背包的问题中，解空间可看作一个 n 维的 0, 1 向量：

$$X = (x_1, x_2, ..., x_n), x_i \in \{0, 1\} \tag{2-4}$$

所以回溯法的本质，是利用深度搜索的搜索顺序对解空间进行遍历。按顺序对 n 个物品的解进行遍历、搜索。

增加一些限制条件，以减少搜索解空间的大小。例如增加一个目前能得到的

最优解：$best_v$，这样如果当前的解加上不考虑重量限制时的剩余解都无法达到这个 $best_v$ 时，可以进行减枝。

2.3 分支限界法算法设计

分支限界法本质上也是一种遍历解空间的搜索策略。在0-1背包问题中，即通过广度优先搜索策略，对解空间进行遍历。（解空间同回溯法）

增加一些限制条件，可以起到剪枝的作用，提升算法效率。和回溯法一致，可以增加一个贪心最优解，在每次决定是否扩展当前分支时进行判断，决定是否进行扩展。

此外，还可以先用一个贪心策略，在O(n)时间复杂度内算出一个最优解的下界，并设置一个目标函数，在本例中，我们设置为：

$$u[i] = v[i] + (W - w) \times (d_{i+1}) \tag{2-5}$$

式中，u为目标函数，v为当前最优解，W为限制的总重量，w为当前选择后的总重量，d_{i+1} 为下一个物品的密度。

需要注意的是，在该策略下，我们要求先对所有物品根据其密度进行从大到小的排序，这样我们的目标函数才满足贪心选择性。

第三章　代码实现

3.1　基础框架

所有程序都由输入数据、处理数据、输出数据三部分组成，不同之处仅在于处理数据中，于是我们可以规范地写出一个通用的 main 函数，如下：

```
int main(){
    readDataIn();
    t1 = clock();
    solutionBfs();
    t2 = clock();
    printDataOut();
    return 0;
}
```

其中，函数 readDataIn() 负责进行数据的读入，solutionBfs() 代表是使用 bfs 即广度优先搜索策略进行的分支限界法进行求解，printDataOut() 则负责输出数据。

3.1.1　数据读入与存储

数据读入和存储，为了显明起见，采用如下结构体存储：

```
struct ITEM{
    int w, v, id;
    double d;
}item[SIZE_N];
bool cmp_d(const ITEM &a, const ITEM &b){ return a.d < b.d; }
```

其中，w 为物品重量，v 为物品价值，id 为物品编号，因为在分支限界法中需要预先根据其密度进行排序，故需要存储其原本的编号，d 为其密度，配套的函数 cmp_d() 负责重载运算符，使 sort() 函数能依照其密度进行从大到小的排序。

于是读入的代码就非常简洁明了了:

```
void readDataIn(){
    cin>>n>>W;
    for(int i=1;i<=n;i++) scanf("%d" ,&item[i].w);
    for(int i=1;i<=n;i++) scanf("%d" ,&item[i].v);
```

```
}
```

3.1.2 时间函数

由于各个解决方案中不同之处主要在与其 solution() 函数，同时为了排除不同情况下的输出流影响，我们将时间测量的点设置在了 solution() 函数的前后。

3.1.3 输出函数

输出的格式都是统一的，于是也可以使用统一的代码进行封装，便于编写。

```
void printDataOut(){
    printf("Solution Found: \nTotal Used Capacity: %d\nTotal Value: %
        d\nChoice of Item:\n",tot_w,f[n][c]);
    for(int i=ans.size()-1;i>=0;i--) printf("%d ",ans[i]);
    printf("\n");
    printf("Time Usage: %f mini-seconds in tot\n" ,1000*(double) (t2
        - t1) / CLOCKS_PER_SEC);
}
```

3.2 Makefile

使用 Makefile 管理多个解决方案的文件，使用统一的调用方式与测试样例，可以提升测试效率。Makefile 见附录。

3.3 动态规划代码实现

3.3.1 朴素的动态规划

```
void solutionDp(){
    for(int i=1;i<=n;i++){
        for(int j=item[i].w;j<=W;j++){
            f[i][j] = max(f[i-1][j-item[i].w]+item[i].v, f[i-1][j]);
        }
    }
    int cc = W;
    for(int i=n;i>=1;i--){
        if(f[i][cc]==f[i-1][cc]) continue;
        ans.push_back(i);
        tot_w += item[i].w;
        cc -= item[i].w;
```

```
    }
}
```

朴素动态规划的 solution() 函数主要由两部分组成，一部分是动态规划的转移方程，另一部分是通过转移方程留下的结果来反推选择的物件。为了方便转移方程的代码，我们使第一个物品存储在 item[1] 中而不是 item[0] 中。

3.3.2 特殊情况下的动态规划

下面讨论在不需要求出具体选择物品情况下的动态规划解法，在该情况下，我们可以对空间复杂度进行大幅度地降低，以期获得更有的算法效率。

注意到，我们动态转移方程的顺序，是从总质量等于 item[i].w 开始逐渐增大到 W。如果我们将这个顺序反向，其结果应当不变。那么，我们会发现，在更新顺序从 W 逐渐减少到 item[i].w 的过程中，更新第 j 个总质量时，只会用到小于等于 j 的信息，而不会用到大于等于 j 的信息，即我们可以将这个二维数组 f[n][W] 压缩为一维数组，减少大量的空间开销。代码如下：

```
for(int i=1;i<=n;i++){
    for(int j=c;j>=w[i];j--){
        f[j] = max(f[j-w[i]]+p[i], f[j]);
    }
}
```

3.4 回溯法代码实现

3.4.1 朴素的回溯法

这是一个朴素的没有减枝的回溯法代码，对于每个结点都有选择当前物品和不选择当前物品两种分支。

```
void dfs(int now, STATUS temp){
    /* Reach to the end */
    if(now == n+1){
        if(ans.tot_value < temp.tot_value || (ans.tot_value == temp.
            tot_value && ans.tot_weight > temp.tot_value)){
            ans = temp;
        }
        return ;
    }
    /* Choose now_th item */
```

```
10      if(temp.tot_weight + item[now].w <= W){
11          temp.PutIn(now);
12          dfs(now + 1, temp);
13          temp.PutOut(now);
14      }
15      /* Don't choose now_th item */
16      dfs(now+1, temp);
17  }
```

其中 STATUS 为一个结构体:

```
1  struct STATUS{
2      bitset<40>s;
3      int tot_value, tot_weight, now;
4  };
```

变量解释如下表所示：

表 3-1 变量解释

变量	类型	含义
s	bitset<>	当前选择的物品 (解向量)
tot_value	int	当前选择中总价值
tot_weight	int	当前选择造成的总重量
now	int	当前状态更新到了第 now 个物品

从理论上来讲，tot_value, 和 tot_weight 是冗余的量，因为这两个量都可以通过 s 分别和物品价值、物品重量的一位矩阵做乘法得到，为了方便后续的描述以及方便在 O(1) 时间内访问，故设置于此。

3.4.2 剪枝策略的回溯法

核心思想为，当前的解在贪心范围内大于等于目前能找到的最优解时，对其进行更新，否则不与理睬。

具体策略和之后的分支限界法一致，在此不过多介绍，其中回溯法的 bound 初始值为 0，每次更新答案 ans 时对其进行更新。

3.5 分支限界法

3.5.1 朴素的分支限界法

分支限界法核心代码如上，该部分主要内容在与使用队列 Q 进行状态的存储，STATUS 与回溯法中一致。

```
queue<STATUS>Q;
void bfs(){
    bitset<40>s0;
    s0.reset();
    Q.push(STATUS(s0, 0, 0, 0));
    while(!Q.empty()){
        STATUS temp = Q.front();
        Q.pop();
        cnt++;
        temp.now++;
        // printf("%d %d %d \n" ,temp.now, temp.tot_value,temp.
           tot_weight);
        if(temp.now==n+1){
            if(temp.tot_value >= ans.tot_value){
                ans = temp;
            }
            continue;
        }
        Q.push(temp);
        if(temp.tot_weight + item[temp.now].w>W) continue;
        STATUS have_temp = temp;
        have_temp.PutIn(have_temp.now);
        Q.push(have_temp);
    }
}
```

分支为每种状态是否需要再加上该物品，限制条件为加上该物品之后总重量是否超过限制重量。

3.5.2 减枝策略地分支限界法

增加限制条件，前面由提到过一种目标函数，在此重新予以描述与证明。

在搜索开始前通过贪心得到一个优化解的下界 bound：

```
void makeBound(){
    sort(item+1, item+n+1, cmp_d);
    int cnt=0;
    for(int i=1;i<=n;i++){
        cnt += item[i].w;
        if(cnt>W) break;
        bound += item[i].v;
    }
    printf("bound: %d\n" ,bound);
    return ;
}
```

通过一个排序，得到按照密度大小排序的物品集合，然后按照密度由大到小进行选择，得到贪心解，即优化解的下界。

然后在每次更新状态时增加一个判断函数：

```
bool Test(int value){
    int nextMax = now==n ? 0:item[now+1].d * (double)(W-tot_weight);
    return nextMax + tot_value >= bound;
}
```

封装在 struct STATUS 结构体内。目标函数为公式 2-5

下面说明为何这样的减枝不会减去最优解：

假设这样的减枝减去了最优解，即设最优解的状态为 S_0（类型为 STATUS），其某个前序状态 S_1 时被限界函数去除。则有：

$$S_1.v + d[now+1] \times (W-w) < bound \tag{3-1}$$

其中 d[now+1] 为下一个被搜索的物品的密度，w 为当前已用的重量。

而由于 *bound* 是贪心解，故必有：

$$bound \le S_0.v \tag{3-2}$$

又因为从 S_1 到 S_0 还需要增加的物品之密度在逐渐减少，即:

$$S_0.v = S_1.v+(d[now+1]\times w[now+1])+...+(d[now+t]\times w[now+t]) \le S_1.v+d[now+1]\times(w[now+1]+...+w[no \tag{3-3}$$

注：此处不要求 now+1 连续加到 now+t。

又因为是按照密度从小到大排序且物品可能无法刚好填满总重量 W，于是

有：

$$\begin{aligned} S_0.v &= S_1.v + (d[now+1] \times w[now+1]) + ... + (d[now+t] \times w[now+t]) \\ &\leq S_1.v + d[now+1] \times (w[now+1] + ... + w[now+t]) \\ &\leq S_1.v + d[now+1] \times (W - w) \\ &< bound \end{aligned} \tag{3-4}$$

即 $S_0.v < bound$ 与条件3-1不符，推出矛盾，则假设不成立。

即这样的减枝策略不会减去最优解，是安全的减枝。

减枝策略的具体性能将在第 6 章实验分析。

第四章　　复杂度分析

4.1　动态规划复杂度分析

4.1.1　时间复杂度

动态规划的执行过程就是在填写一张 n * W 的表格，且填写每个格子的时间开销为 O(1)，故其时间复杂度为 O(nW)。

4.1.2　空间复杂度

有两种空间复杂度，如果是完全的动态规划，则需要展开所有的格子，即总共 n*W 个，故总共开销为 $O(nW + n * 2) = O(nW)$。

如果是简约的动态规划，即压缩成一维数组时，则为 $O(W + 2 * n) = O(W)$ 或 O(n)。

4.2　回溯法复杂度分析

4.2.1　时间复杂度

最坏情况下，回溯法会将解空间中所有的可能性都进行依次遍历，于是解空间的时间开销为 $O(2^n)$。

4.2.2　空间复杂度

空间复杂度主要体现在每一次递归的最深层次为 n 层，故空间复杂度的开销为 $O(n)$。

4.3　分支限界法复杂度分析

4.3.1　时间复杂度

和回溯法一样，分支限界法的搜索策略不同，但其遍历的解空间和回溯法完全一致，故时间开销仍然为 $O(2^n)$。

4.3.2　空间复杂度

分支限界法是按照每一层的顺序进行遍历的，遍历第 i 层之后，队列 Q 中的元素为第 i + 1 层中的所有解，可以推测最大的队列应当为第 n 层，共有 2^n 个解，即空间开销为 $O(2^n)$，但实际上通过优先队列和减枝策略，空间复杂度会大大降低。

第五章 实验平台及测试样例

5.1 实验平台

实验采用的电脑为普通个人电脑。处理器为 Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz。

在第二系统（Linux）上运行，内存为 16 GB，操作系统为 Ubuntu 20.04.3 LTS，编译统一使用 g++ -O1 优化。

5.2 测试样例

测试样例来自 Florida State University 的 P02, P03, P04, P07, P08。

其中 P02, P03, P04 为小数据，P07 为中等大小数据，P08 为大数量级数据。

第六章　实验结果及数据分析

根据前述的时间函数，我们以毫秒为单位，对每个方案进行计时，整体数据如表6-1所示：

表 6-1　动态规划实验结果

	p02	p03	p04	p07	p08
dpmin	0.01	0.016	0.014	0.08	746.737
dp	0.011	0.015	0.018	0.091	817.387
dfs	0.002	0.002	0.004	0.664	292.114
dfsb	0.002	0.001	0.004	0.006	0.038
bfs	0.004	0.007	0.011	2.557	999.561
bfsq	0.017	0.023	0.041	4.202	38.532

6.1　动态规划实验结果分析

动态规划算法的实验结果如表6-2所示：

表 6-2　动态规划实验结果

	p02	p03	p04	p07	p08
dpmin	0.01	0.016	0.014	0.08	746.737
dp	0.011	0.015	0.018	0.091	817.387

结果表明，动态规划的简化算法，会改进其空间效率，同时也能对时间复杂度有细微的提升，推测应该是在运行时寻址时效率更高。

下图能更加清晰地看出二者的差距：

从图中我们可以看出，优化过的 dpmin 的确在所有数据集都优于普通的 dp 方法。由于 p08 的数量级过于庞大，故图中没有 p08。

6.2　回溯法实验结果分析

回溯法实验结果如表6-3所示：

实验结果表明，优化过的 dfs 在几乎所有数量级上，都优于朴素的 dfs 方法，即便其在小数量级上多了剪枝的操作。

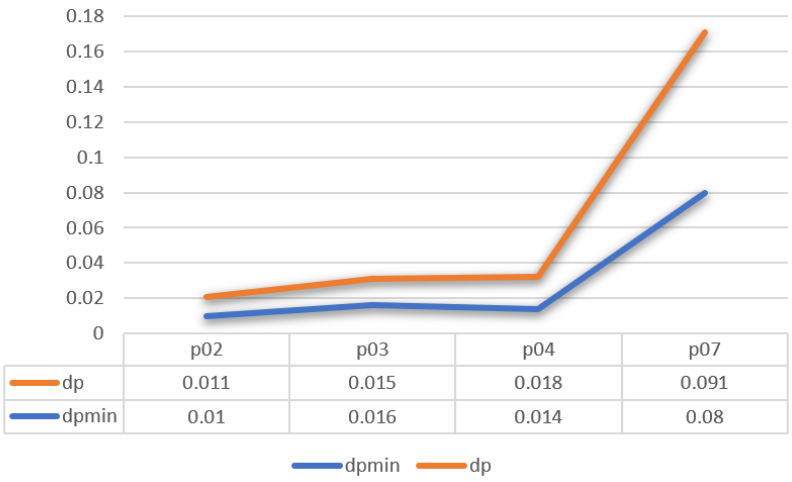


图 6-1　　dp 方案优化对比

表 6-3　　回溯法实验结果

	p02	p03	p04	p07	p08
dfsb	0.002	0.001	0.004	0.006	0.038
dfs	0.002	0.002	0.004	0.664	292.114

在大数量级上，可以预见的是，其剪枝操作应当减去了超过 90 的分支，具体实验结果将在下一小节一起讨论。

6-2能更加清晰地看出二者的差距：

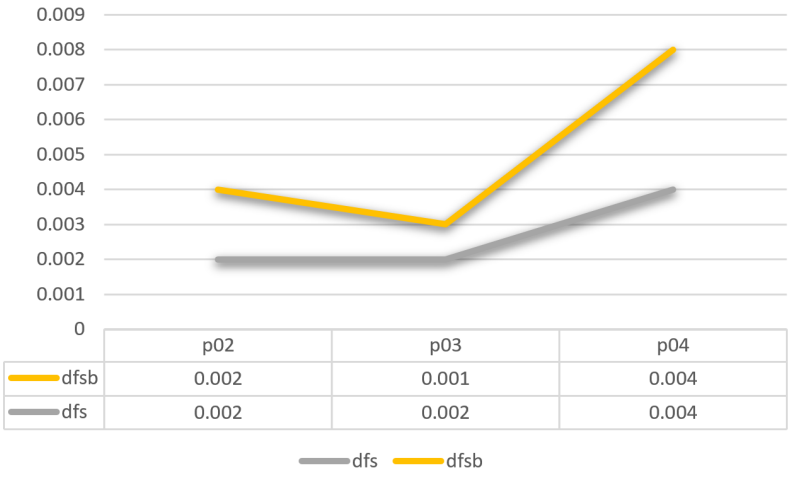


图 6-2　　dfs 方案优化对比

6.3　分支限界法实验结果分析

分支限界法的实验结果如表6-4所示：

表 6-4 分支限界法实验结果

	p02	p03	p04	p07	p08
bfsq	0.017	0.023	0.041	4.202	38.532
bfs	0.004	0.007	0.011	2.557	999.561

实验结果标明，优化过的 bfs 在小数量级上并没有太大优势。这是因为 1. 小数量级上剪枝并没有太大作用，实际效果并不明显。2. bfs 的剪枝策略决定了其必须要先使用 O(NlogN) 的时间复杂度完成一次贪心，这样的时间消耗，也是朴素 bfs 不需要的。

然而，在较大数量级上，朴素 bfs 显得力不从心，而优化 bfs 则可以轻松应对。

6-3能更加清晰地看出二者的差距：

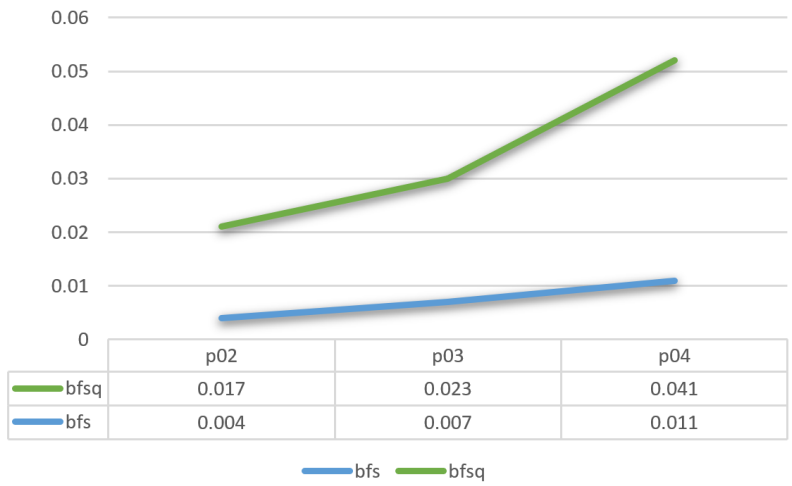


图 6-3 bfs 方案优化对比

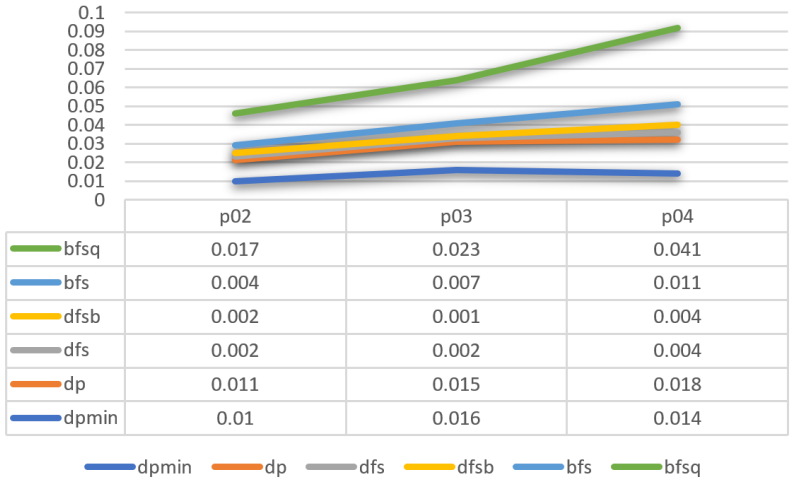
6.4 整体对比

6.4.1 小数量级

如图6-4所示，我们得知，小数量级上朴素的回溯法是效率最高的算法。

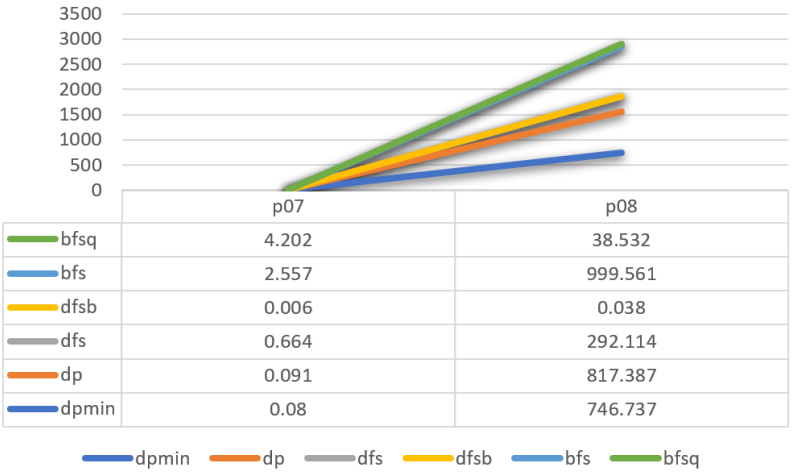
6.4.2 大数量级

如图6-5 和 6-6所示，我们得知，大数量级上优化后的分支限界法和回溯法效率远优于剩余算法。



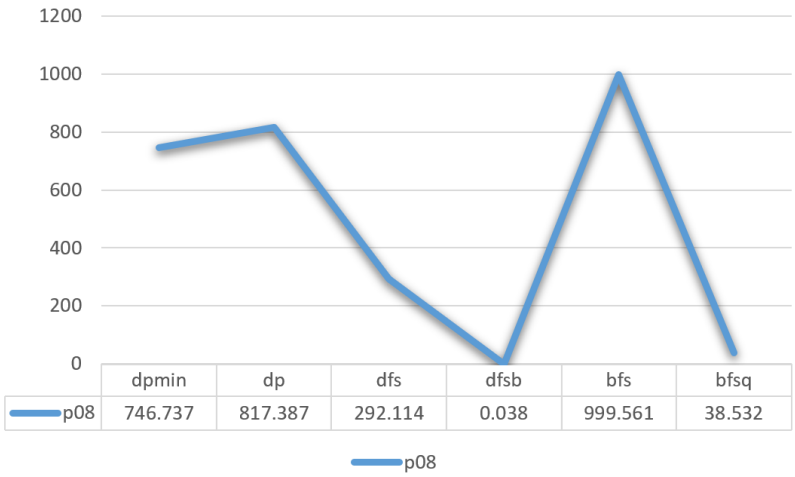
	p02	p03	p04
bfsq	0.017	0.023	0.041
bfs	0.004	0.007	0.011
dfsb	0.002	0.001	0.004
dfs	0.002	0.002	0.004
dp	0.011	0.015	0.018
dpmin	0.01	0.016	0.014

图 6-4　　小数量级下比较



	p07	p08
bfsq	4.202	38.532
bfs	2.557	999.561
dfsb	0.006	0.038
dfs	0.664	292.114
dp	0.091	817.387
dpmin	0.08	746.737

图 6-5　　大数量级下比较



	dpmin	dp	dfs	dfsb	bfs	bfsq
p08	746.737	817.387	292.114	0.038	999.561	38.532

图 6-6　　P08 执行效果

表 6-5 分支限界法实验结果

	p02	p03	p04	p07	p08
dfs	44	78	157	4448	17770698
dfsb	11	17	72	141	1192
bfs	45	85	145	44018	17692176
bfsq	26	57	128	28855	110749
all	63	127	255	65535	33554431

6.5 讨论分支限界和回溯法的剪枝策略

二者的解空间总大小相同，不同点在于其策略导致的搜索空间，6-5展示了不同的搜索策略的空间大小，其中 all 为解空间大小。

将其归一化并做成图6-7图像，可清晰地看出：

无论在哪个数量级，优化的回溯法，总能大幅度减少搜索空间。

优化的分支限界法也能减少部分搜索空间，但在较大数量级上的表现不如回溯法。

朴素的分支限界法、回溯法在大数量级上的搜索空间大小约等于总共解空间的一半，大概可以推测，其平均在最后一层搜索时才会被迫停止，这样的效率是及其低下。

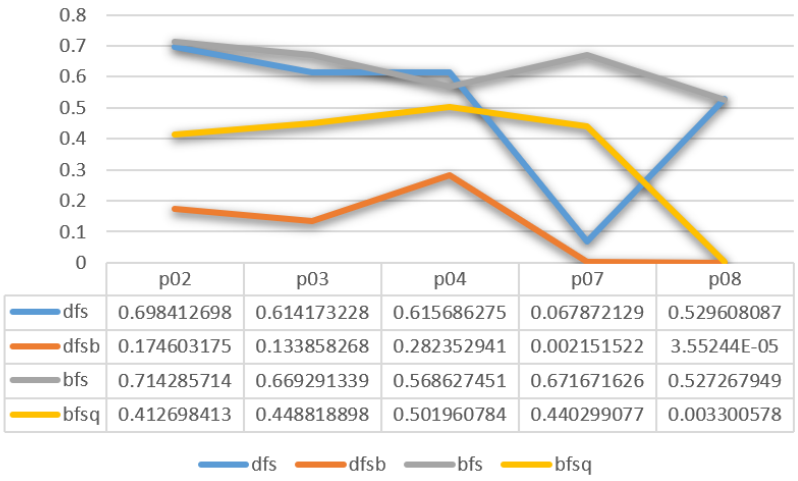


图 6-7 P08 执行效果

结 论

本实验分别实现了动态规划、回溯法、分支限界法求解0-1背包问题的5个样例，并记录了各自的时间，分析了三种算法的时间、空间复杂度，并通过实际测试，对比了各自算法与其剪枝算法优化的效果。

总体而言，优化的回溯法 > 优化的分支限界法 > 动态规划法 > 朴素的回溯法和分支限界法。

由于0-1背包是NPC问题，其本身是无法在多项式时间内求解的，所以无论是动态规划、回溯法、分支限界法，都无法有效的解决大规模的0-1背包问题。

但是，在知道数据集特点时，选择合适的算法，能起到提升效率的效果。例如在总重量较大而物品数量较少时，采用剪枝后的分支限界法的效率优于动态规划法。

附 录

Makefile

```
dir=./data
file=6.in
files=5.in 6.in 7.in 15.in 24.in
runningfile=${dir}/${file}
allout=dpmin.out dp.out dfs.out dfsb.out bfs.out bfsq.out
name=$(allout:%.out=%)

compileall: dp.out dfs.out bfs.out bfsq.out dpmin.out

run: ${allout}
    @echo "[[[[[[[[[ BASIC MODE ]]]]]]]]]"
    @for va in ${name};do\
        echo ">>>>>>>>>>>>>>> $$va <<<<<<<<<<<<"; \
        ./$$va.out < ${runningfile}; \
        echo "";\
    done

forrun: ${allout}
    @echo "[[[[ FOR MODE ]]]]"
    @for va in ${name};do\
        echo ">>>>>>>>>>>>>>> $$va <<<<<<<<<<";\
        for f in ${files};do\
            echo "file=$$f";\
            ./$$va.out < ${dir}/$$f;\
            echo "";\
        done;\
        echo "";\
    done

dfs: dfsb.out dfs.out
```

```
31          ./dfsb.out < ${runningfile}
32          ./dfs.out < ${runningfile}
33
34      bfs: bfsq.out bfs.out
35          ./bfsq.out < ${runningfile}
36          ./bfs.out < ${runningfile}
37
38      dp.out:
39          @g++ -o dp.out dp.cpp
40
41      dfs.out:
42          @g++ -o dfs.out dfs.cpp
43
44      bfs.out:
45          @g++ -o bfs.out bfs.cpp
46
47      bfsq.out:
48          @g++ -o bfsq.out bfsq.cpp
49
50      dpmin.out:
51          @g++ -o dpmin.out dpmin.cpp
52
53      dfsb.out:
54          @g++ -o dfsb.out dfsb.cpp
55
56      clean:
57          rm -f *.out *.o
```