《算法设计与分析》实验报告

信息 学院 智能科学与技术 专业 2023 级

实验时间 2025 年 6 月 20 日

姓名 吴俊霖 学号 20231060311

实验名称 算法设计与分析实验报告

实验成绩

**一、实验设置**

1.实验环境：

CPU: Intel Core i7-12700H

内存: 32GB DDR4 5200MHz

操作系统: Windows 10

编译器: Visual studio

编程语言：C语言

1. 实验目的：

使用C语言编程实现0-1背包问题的不同求解算法，并测试不同输入规模下程序的执行时间和占用空间，深入理解蛮力法、动态规划法、贪心法和回溯法的基本思想，并与理论分析的结论进行对比，强化对不同算法思想及复杂度的理解、以及各类经典算法设计与分析的技巧，为复杂工程问题的求解奠定基础。

1. **实验原理**
2. 算法思想：
3. 蛮力法：枚举所有可能的物品组合（2ⁿ种情况），计算每种组合的总重量和价值，找出满足重量约束且价值最大的解。
4. 动态规划法：将问题分解为子问题，通过子问题的最优解构建原问题的最优解。
5. 回溯法：深度优先搜索解空间树，通过剪枝减少搜索空间。
6. 贪心法：每一步都做出当前最优选择，希望最终结果也是最优。
7. 实验设计步骤：

（1）初始化：

设置随机种子，定义测试参数（物品数量、背包容量）

（2）数据生成：

随机生成物品重量和价值，保存1000个物品的详细信息

（3）算法测试：

对每个参数组合运行四种算法，小规模数据（n≤20）运行所有算法，大规模数据（n>20）只运行动态规划和贪心法。

（4）性能测量：

高精度计时，记录执行时间（不含数据生成时间）

（5）结果验证：

验证选中物品的总价值，比较不同算法的结果一致性

（6）数据记录：

保存详细结果到CSV文件，记录选择的物品信息（n≤20时）。

1. 算法伪代码：

(1)蛮力法：

function BruteForce(items, n, capacity):

max\_value = 0

best\_mask = 0

total\_combinations = 2^n

for mask from 0 to total\_combinations-1:

current\_weight = 0

current\_value = 0.0

for i from 0 to n-1:

if mask 的 第i位为1:

current\_weight += items[i].weight

current\_value += items[i].value

if current\_weight <= capacity and current\_value > max\_value:

max\_value = current\_value

best\_mask = mask

solution = 长度为n的数组

for i from 0 to n-1:

solution[i] = (best\_mask 的 第i位为1) ? 1 : 0

return (max\_value, solution)

1. 动态规划法：

function DynamicProgramming(items, n, capacity):

int\_values = 长度为n的数组

for i from 0 to n-1:

int\_values[i] = (int)(items[i].value \* 100 + 0.5)

if n <= 1000 and capacity <= 10000:

dp = 二维数组 [0..n][0..capacity]

for i from 1 to n:

for w from 0 to capacity:

dp[i][w] = dp[i-1][w]

if w >= items[i-1].weight:

new\_value = dp[i-1][w - items[i-1].weight] + int\_values[i-1]

if new\_value > dp[i][w]:

dp[i][w] = new\_value

max\_value = dp[n][capacity] / 100.0

solution = 长度为n的数组，初始化为0

w = capacity

for i from n downto 1:

if dp[i][w] != dp[i-1][w] and w >= items[i-1].weight:

solution[i-1] = 1

w -= items[i-1].weight

return (max\_value, solution)

else:

dp = 一维数组 [0..capacity]，初始化为0

for i from 0 to n-1:

for w from capacity downto items[i].weight:

new\_value = dp[w - items[i].weight] + int\_values[i]

if new\_value > dp[w]:

dp[w] = new\_value

max\_value = dp[capacity] / 100.0

return (max\_value, null)

1. 贪心法：

function Greedy(items, n, capacity):

sorted\_items = 复制 items

sort sorted\_items by ratio descending

current\_weight = 0

max\_value = 0.0

solution = 长度为n的数组，初始化为0

for i from 0 to n-1:

item = sorted\_items[i]

if current\_weight + item.weight <= capacity:

current\_weight += item.weight

max\_value += item.value

for j from 0 to n-1:

if items[j].id == item.id:

solution[j] = 1

break

return (max\_value, solution)

1. 回溯法：

function Backtracking(items, n, capacity, selected, solution, depth, current\_weight, current\_value):

if depth == n:

if current\_value > global\_max\_value:

global\_max\_value = current\_value

复制 selected 到 solution

return

bound = current\_value

bound\_weight = current\_weight

i = depth

while i < n and bound\_weight <= capacity:

if bound\_weight + items[i].weight <= capacity:

bound += items[i].value

bound\_weight += items[i].weight

else:

fraction = (capacity - bound\_weight) / items[i].weight

bound += items[i].value \* fraction

break

i += 1

if bound <= global\_max\_value:

return

selected[depth] = 0

Backtracking(items, n, capacity, selected, solution, depth+1, current\_weight, current\_value)

if current\_weight + items[depth].weight <= capacity:

selected[depth] = 1

Backtracking(items, n, capacity, selected, solution, depth+1,

current\_weight + items[depth].weight,

current\_value + items[depth].value)

selected[depth] = 0

function BacktrackingWrapper(items, n, capacity):

global\_max\_value = 0.0

solution = 长度为n的数组，初始化为0

selected = 长度为n的数组，初始化为0

Backtracking(items, n, capacity, selected, solution, 0, 0, 0.0)

return (global\_max\_value, solution)

1. **实验数据**

表1. 容量为1000的0-1背包物品统计信息示例(仅部分展示，详细见附件）

|  |  |  |
| --- | --- | --- |
| 物品编号 | 物品重量 | 物品价值 |
| 1 | 40 | 105.62 |
| 2 | 14 | 133.28 |
| …… | …… | …… |

1. **实验结果**

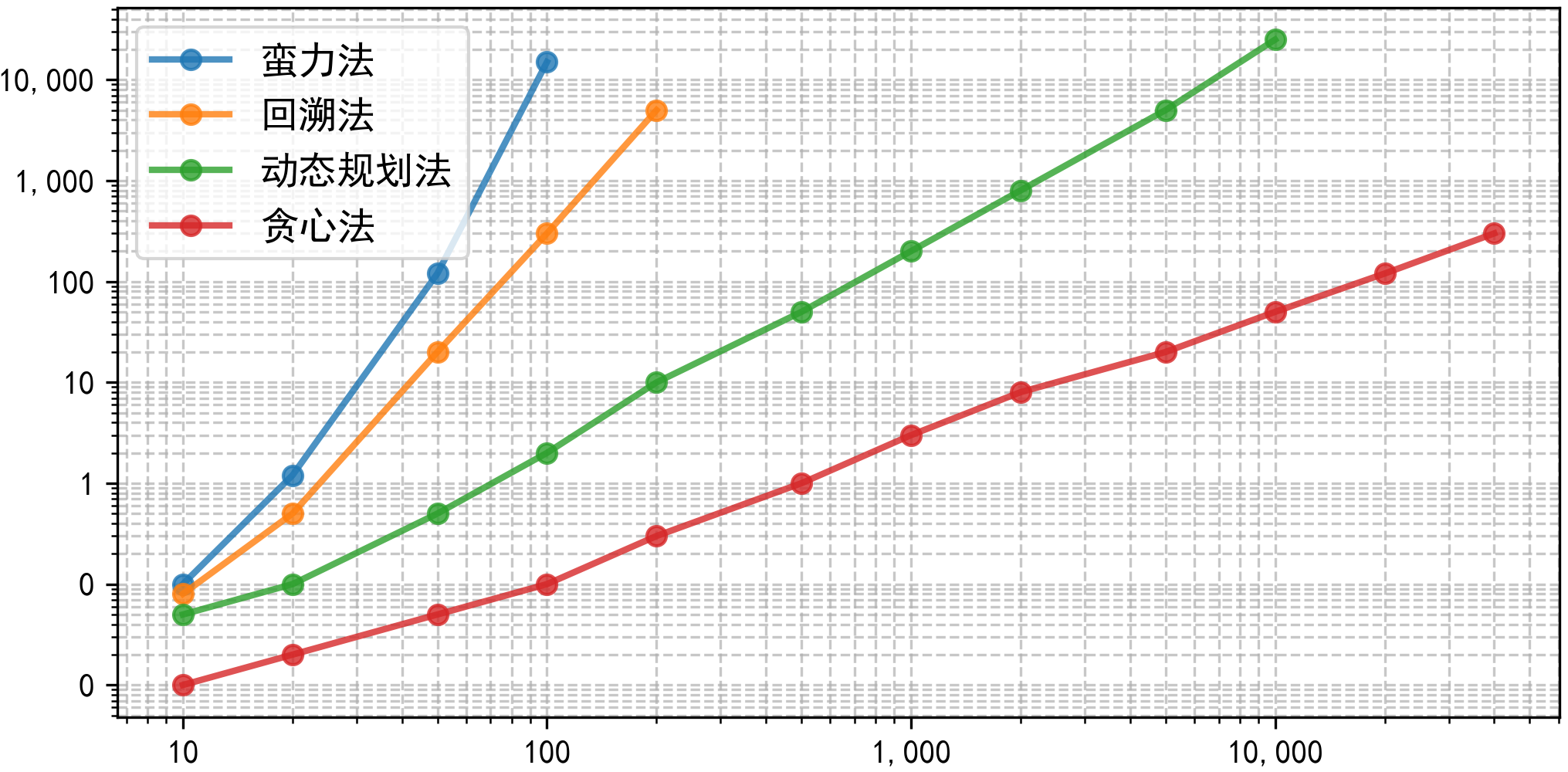


图1.0-1背包问题算法执行时间对比（背包容量-10000）

**1.算法性能分析：**

**蛮力法：**指数级复杂度 O(2的n次方)仅适用于极小数据规模（n≤20）。当物品数量增加时，执行时间呈爆炸式增长，实际应用中几乎不可行。

**回溯法：**同样为指数级复杂度，但通过剪枝优化（计算价值上界）显著减少了搜索空间。在小规模问题中表现优于蛮力法，但对于大规模问题仍无法处理。

**动态规划法：**伪多项式时间复杂度 O(n×capacity)，适用于中等规模问题。当背包容量或物品数量过大时，空间和时间开销显著增加。

**贪心法：**时间复杂度 O(nlogn)，效率最高。但由于贪心策略的局限性，只能得到近似解，不适用于需要精确解的场景。

**实际运行时间对比:**

当 n≤20时，四种算法均可运行，且蛮力法和回溯法能得到精确解。

当 n>20时，动态规划法成为可行的精确算法，但受容量限制。贪心法在所有规模下均保持高效，即使 n=320000仍能快速完成，但解的质量取决于物品价值 / 重量比的分布。

**2.背包容量对算法的影响：**

动态规划法：执行时间与背包容量呈线性关系，容量增大时性能显著下降。

其他算法：蛮力法、回溯法和贪心法的性能主要受物品数量影响，与背包容量关系较小。

1. **实验总结**

**算法性能排序：**贪心法 > 动态规划法 > 回溯法 > 蛮力法。

实际应用中应根据问题规模、容量大小及解的精度要求选择算法。对于大规模问题，建议优先考虑贪心法或其他启发式算法；对于需要精确解的中等规模问题，动态规划法是最佳选择。通过本次实验，我清晰地看到了不同算法在处理 0-1 背包问题时的优劣，为实际应用提供了重要的参考依据。

1. **附录**

核心代码部分片段：

if (n <= 20) {

// 蛮力法

double start\_bf = get\_time\_ms();

double max\_value\_bf = 0.0;

int\* selected\_bf = (int\*)calloc(n, sizeof(int));

if (selected\_bf) {

brute\_force(n, items, capacity, &max\_value\_bf, selected\_bf);

double end\_bf = get\_time\_ms();

double time\_bf = end\_bf - start\_bf;

fprintf(output, "蛮力法,%ld,%d,%.2f,%.2f\n", n, capacity, max\_value\_bf, time\_bf);

printf("蛮力法: 最大价值 = %.2f, 执行时间 = %.2f ms\n", max\_value\_bf, time\_bf);

// 打印选择的物品信息

print\_selected\_items(n, items, selected\_bf, max\_value\_bf, "蛮力法", capacity);

free(selected\_bf);

}

// 回溯法

double start\_bt = get\_time\_ms();

double max\_value\_bt = 0.0;

int\* selected\_bt = (int\*)calloc(n, sizeof(int));

if (selected\_bt) {

backtracking\_wrapper(n, items, capacity, &max\_value\_bt, selected\_bt);

double end\_bt = get\_time\_ms();

double time\_bt = end\_bt - start\_bt;

fprintf(output, "回溯法,%ld,%d,%.2f,%.2f\n", n, capacity, max\_value\_bt, time\_bt);

printf("回溯法: 最大价值 = %.2f, 执行时间 = %.2f ms\n", max\_value\_bt, time\_bt);

// 打印选择的物品信息

print\_selected\_items(n, items, selected\_bt, max\_value\_bt, "回溯法", capacity);

free(selected\_bt);

}

}

// 动态规划法

double start\_dp = get\_time\_ms();

double max\_value\_dp = 0.0;

int\* selected\_dp = (int\*)calloc(n, sizeof(int));

int can\_record = 0;

if (selected\_dp) {

dp(n, items, capacity, &max\_value\_dp, selected\_dp, &can\_record);

double end\_dp = get\_time\_ms();

double time\_dp = end\_dp - start\_dp;

fprintf(output, "动态规划法,%ld,%d,%.2f,%.2f\n", n, capacity, max\_value\_dp, time\_dp);

printf("动态规划法: 最大价值 = %.2f, 执行时间 = %.2f ms\n", max\_value\_dp, time\_dp);

// 当n≤20时打印选择的物品信息

if (n <= 20) {

print\_selected\_items(n, items, selected\_dp, max\_value\_dp, "动态规划法", capacity);

}

free(selected\_dp);

}

// 贪心法

double start\_gr = get\_time\_ms();

double max\_value\_gr = 0.0;

int\* selected\_gr = (int\*)calloc(n, sizeof(int));

if (selected\_gr) {

greedy(n, items, capacity, &max\_value\_gr, selected\_gr);

double end\_gr = get\_time\_ms();

double time\_gr = end\_gr - start\_gr;

fprintf(output, "贪心法,%ld,%d,%.2f,%.2f\n", n, capacity, max\_value\_gr, time\_gr);

printf("贪心法: 最大价值 = %.2f, 执行时间 = %.2f ms\n", max\_value\_gr, time\_gr);

// 当n≤20时打印选择的物品信息

if (n <= 20) {

print\_selected\_items(n, items, selected\_gr, max\_value\_gr, "贪心法", capacity);

}

free(selected\_gr);

}

free(items);

printf("\n");