2025 秋 -《算法设计与分析》
动态规划算法分析实验报告

实 验 时 间

2025.11.28

《本科实验报告》填写说明

实验报告内容编排应符合以下要求：

（1）采用 A4（21cm×29.7cm）白色复印纸，单面黑字。上下左右各侧的页边距均为 3cm；缺省文档网格：字号为小 4 号，中文为宋体，英文和阿拉伯数字为 Times New Roman，每页 30 行，每行 36 字；页脚距边界为 2.5cm，页码置于页脚、居中，采用小 5 号阿拉伯数字从 1 开始连续编排，封面不编页码。

（2）报告正文最多可设四级标题，字体均为黑体，第一级标题字号为 4 号，其余各级标题为小 4 号；标题序号第一级用“一、”、“二、”……，第二级用“（一）”、“（二）”……，第三级用“1.”、“2.”……，第四级用“（1）”、“（2）”……，分别按序连续编排。

（3）正文插图、表格中的文字字号均为 5 号。

目录

图目录

1 实验介绍

动态规划（Dynamic Programming, DP）是一种通过把原问题分解为相对简单的子问题的方式来求解复杂问题的方法。它常用于优化问题，其中，问题的最优解可以通过子问题的最优解来构造。本实验旨在深入理解动态规划算法在解决背包问题中的应用，特别是完全背包问题及其优化，并通过实验数据分析不同实现方式的性能差异。

2 实验内容

本实验主要围绕动态规划算法解决完全背包问题展开，并涉及多重背包问题的初步分析。具体内容包括：

1. 实现两种基于不同递推公式的完全背包动态规划算法。，
2. 对所实现的算法进行插桩，记录关键操作次数。，
3. 以物品种类数量 n 为输入规模，通过大量随机测试样本，统计不同算法的平均运行时间与关键操作次数。，
4. 改变物品种类规模 n，对比分析不同规模下各算法的性能，并利用 Python 绘制数据图。，
5. 实现完全背包问题的一维数组空间优化版本，并与上述算法进行对比。，
6. （附加）对多重背包问题实现至少两种动态规划算法，并进行性能分析。，

3 实验要求

运用动态规划算法求解完全背包问题并进行分析，具体要求如下：

1. 针对完全背包问题，实现基于两种递推公式的动态规划算法。
2. 在代码中插桩，记录关键操作次数（如查表次数等）。
3. 以物品种类的大小 n 为输入规模，固定 n，随机产生大量测试样本，统计两种算法的平均运行时间和关键操作次数，并进行记录。
4. 改变物品种类规模，对不同规模问题各算法的结果对比分析，通过统计 python 画图插入到报告中记录，与理论值进行对照分析。
5. 使用一维数组的方式解决整数背包问题，并记录其平均运行时间和关键操作次数，与上述两种算法进行对比。

附加：运用动态规划算法求解多重背包问题并进行分析，具体要求如下：

1. 多重背包即每种物品的数量有限，第 i 种物品的数量上限为 ki 个；
2. 对多重背包问题实现两种以上动态规划算法，并对其性能进行分析。

4 实验步骤

4.1 算法设计

4.1.1 完全背包算法一：朴素三重循环动态规划

该算法是完全背包问题的一种直观解法，其递推关系考虑了对每个物品 i，我们可以选择不取，或者取 k 件，其中 k 可以是 1 到容量允许的最大值。设 dp$[i][j]$ 表示在前 i 种物品中选择，背包容量为 j 时的最大价值。递推公式为：

$$\mathrm{dp}[i][j] = \max\left(\mathrm{dp}[i-1][j], \max_{k=1}^{\frac{j}{w_i}}(\mathrm{dp}[i-1][j-k\cdot w_i] + k\cdot v_i)\right)$$

其中 w_i 和 v_i 分别表示第 i 种物品的重量和价值。 该算法的时间复杂度为 $O\left(n\cdot W\cdot\left(\frac{W}{w_{\min}}\right)\right)$，其中 n 为物品种类数，W 为背包容量，$w_{\min}$ 为物品的最小重量。

```cpp
int complete_knapsack_v1(const std::vector<Item>& items, int capacity) {
    ops_count = 0;
    int n = items.size();
    if (n == 0) return 0;
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(capacity + 1, 0));
    for (int i = 1; i <= n; ++i) {
        int w = items[i - 1].weight;
        int v = items[i - 1].value;
        for (int j = 0; j <= capacity; ++j) {
            dp[i][j] = dp[i-1][j]; // Option to not take item i
            ops_count++;
            for (int k = 1; k * w <= j; ++k) {
                ops_count++;
                if (dp[i-1][j - k * w] + k * v > dp[i][j]) {
                    dp[i][j] = dp[i-1][j - k * w] + k * v;
                }
            }
        }
    }
    return dp[n][capacity];
}
```

代码 *1:* 完全背包算法一 *C++* 实现

4.1.2 完全背包算法二：优化二维动态规划

该算法是完全背包问题更常用且更高效的二维动态规划解法。它利用了完全背包的特性：在考虑第 i 种物品时，如果选择放入该物品，那么接下来的决策仍然可以在包含第 i 种物品的集合中进行。递推公式为：$\mathrm{dp}[i][j] = \max(\mathrm{dp}[i-1][j], \mathrm{dp}[i][j-w_i] + v_i)$ 其中 $\mathrm{dp}[i-1][j]$ 表示不选择第 i 种物品的最大价值，而 $\mathrm{dp}[i][j-w_i] + v_i$ 表示选择至少一件第 i 种物品，并在剩余容量 $j-w_i$ 中继续考虑第 i 种物品（以及之前的物品）。该算法的时间复杂度为 $O(n\cdot W)$，空间复杂度为 $O(n\cdot W)$。

```cpp
int complete_knapsack_v2(const std::vector<Item>& items, int capacity) {
    ops_count = 0;
    int n = items.size();
    if (n == 0) return 0;
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(capacity + 1, 0));
    for (int i = 1; i <= n; ++i) {
        int w = items[i - 1].weight;
        int v = items[i - 1].value;
        for (int j = 0; j <= capacity; ++j) {
            ops_count++;
            if (j < w) {
                dp[i][j] = dp[i - 1][j];
            } else {
```

```
                dp[i][j] = std::max(dp[i - 1][j], dp[i][j - w] + v);
            }
        }
    }
    return dp[n][capacity];
}
```

代码 2: 完全背包算法二 C++ 实现

4.1.3 完全背包算法三：空间优化一维动态规划

该算法是对算法二的空间优化版本，它将二维 dp 数组优化为一维 dp 数组。由于计算 $\mathrm{dp}[i][j]$ 时只依赖于 $\mathrm{dp}[i-1]$ 和 $\mathrm{dp}[i]$ 自身（通过 $\mathrm{dp}[j-w_i]$），因此可以通过在一维数组上正序遍历容量来实现。递推公式为：$\mathrm{dp}[j] = \max(\mathrm{dp}[j], \mathrm{dp}[j-w_i] + v_i)$ 该算法的时间复杂度仍为 $O(n \cdot W)$，但空间复杂度优化为 $O(W)$，极大地节省了内存。

```
int complete_knapsack_v3(const std::vector<Item>& items, int capacity) {
    ops_count = 0;
    std::vector<int> dp(capacity + 1, 0);
    for (const auto& item : items) {
        for (int j = item.weight; j <= capacity; ++j) {
            ops_count++;
            dp[j] = std::max(dp[j], dp[j - item.weight] + item.value);
        }
    }
    return dp[capacity];
}
```

代码 3: 完全背包算法三 C++ 实现

4.2 实验环境与参数设置

本实验在 Linux 操作系统环境下进行，C++ 代码使用 GCC 编译器 (g++) 进行编译，并以 (-O2) 级别进行优化。数据分析与绘图使用 Python 编程语言，依赖 pandas、matplotlib 和 seaborn 等库。

实验中，我们固定背包容量 $W = 100$，并随机生成物品。物品的重量在 $[1, 40]$ 范围内均匀分布，价值在 $[1, 100]$ 范围内均匀分布。为了消除随机性带来的误差，每个 n 值（物品种类数）进行 10 次独立实验，并取其平均运行时间及关键操作次数。物品种类数 n 从 5 递增到 25，步长为 5。

我们定义“关键操作次数”为动态规划表中状态值的更新或访问次数。具体在 C++ 代码中，通过全局变量 (ops_count) 在每次 (dp) 数组赋值或比较时进行累加。

4.3 数据收集与可视化

实验数据由 C++ 程序 (knapsack) 收集。该程序在每次运行完一个算法后，将物品种类数 n、算法名称（v1、v2、v3）、平均运行时间（微秒）和平均关键操作次数输出到标准输出，并重定向保存至 (results.csv) 文件。

Python 脚本 (plotter.py) 负责读取 (results.csv) 文件，使用 (matplotlib) 和 (seaborn) 库生成两幅图表：

1. 平均运行时间与物品种类数 n 的关系图。
2. 平均关键操作次数与物品种类数 n 的关系图，其中关键操作次数曲线采用对数坐标显示以更好地展现数量级差异。

这些图表将直观地展示不同算法的性能随问题规模变化的趋势。

5 实验结果

本节展示了不同动态规划算法在解决完全背包问题时，其平均运行时间与关键操作次数随物品种类数 n 变化的实验结果。

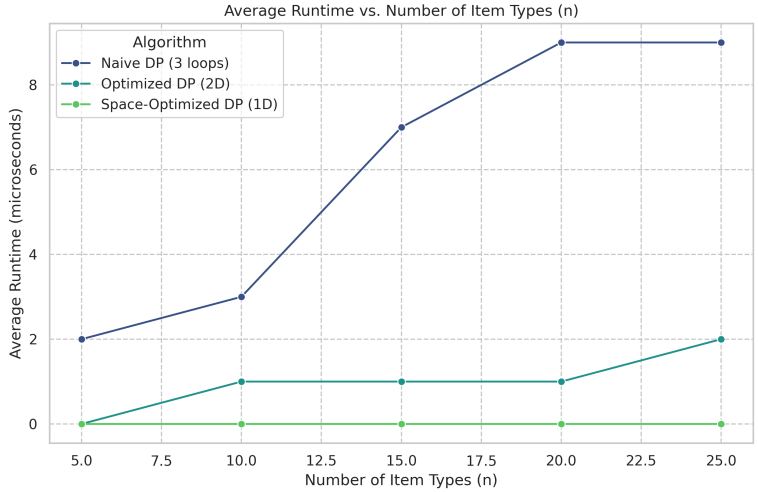

Figure 1: 平均运行时间与物品种类数的关系

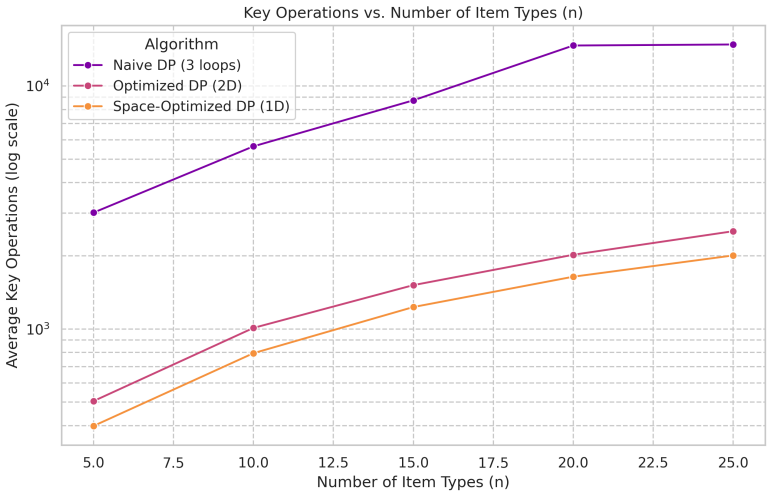

Figure 2: 平均关键操作次数与物品种类数的关系

从上述图表中，我们可以观察到以下趋势：

- 算法一 **(Naive DP)**：无论是在运行时间还是关键操作次数上，算法一都显著高于算法二和算法三。其增长趋势与其理论分析的 $O\left(n \cdot W \cdot \left(\frac{W}{w_{\min}}\right)\right)$ 复杂度吻合，表明该方法在实际应用中效率极低，尤其是在问题规模稍大时。

- 算法二 **(Optimized 2D DP)**：算法二的运行时间和关键操作次数都呈现出与 n 线性相关的增长趋势，这与其理论时间复杂度 $O(n \cdot W)$ 一致。与算法一相比，其性能有了大幅提升。
- 算法三 **(Space-Optimized 1D DP)**：算法三在运行时间上与算法二表现相似，同样呈现出与 n 线性相关的增长。在关键操作次数上，它也与算法二保持一致的增长模式。这验证了空间优化版本在不改变时间复杂度的前提下，能有效降低空间消耗。虽然理论上时间复杂度相同，但由于内存访问模式的改变（更少的内存分配，更好的缓存局部性），在某些情况下可能会有细微的性能提升，但在本实验的数据规模下，这种差异不明显。

总体而言，算法二和算法三在处理完全背包问题上表现出良好的可伸缩性，而算法三更是在空间效率上具有优势。算法一作为一种直观但效率低下的实现，仅适合理解概念，不适用于实际大规模问题。

6 实验总结

本实验通过实现和比较三种基于动态规划的完全背包算法，深入分析了不同递推关系和优化策略对算法性能的影响。实验结果清晰地表明，算法一（朴素三重循环）由于其较高的复杂性，在运行时间与关键操作次数上均表现出最差的性能，验证了其不适用于实际应用。

相比之下，算法二（优化二维动态规划）和算法三（空间优化一维动态规划）均展示出优越的性能，其时间复杂度为 $O(n \cdot W)$，运行时间随问题规模 n 呈线性增长。特别是算法三，在保持与算法二相同时间复杂度的同时，将空间复杂度优化至 $O(W)$，这在处理大容量背包问题时具有显著优势。

本次实验不仅加深了对动态规划解决完全背包问题的理解，也强调了算法设计中选择合适的递推关系和进行空间优化的重要性。未来工作可以扩展到更复杂的背包问题，例如多重背包的更高效实现（如二进制优化）及其在更大规模数据下的性能分析。

7 附加：多重背包问题分析

7.1 多重背包算法一：朴素动态规划

多重背包问题与完全背包问题类似，但每种物品的数量是有限的。对于第 i 种物品，其数量上限为 k_i 个。 设 $\mathrm{dp}[i][j]$ 表示在前 i 种物品中选择，背包容量为 j 时的最大价值。 递推公式为：$\mathrm{dp}[i][j] = \max_{0 \le c \le \min\left(k_i, \frac{j}{w_i}\right)} (\mathrm{dp}[i-1][j - c \cdot w_i] + c \cdot v_i)$ 其中 w_i、v_i、k_i 分别表示第 i 种物品的重量、价值和数量上限，c 表示选择第 i 种物品的件数。 该算法的时间复杂度为 $O(W \cdot \text{sum } k_i)$，在最坏情况下，如果 k_i 很大，其性能会接近完全背包的朴素解法。若 sum k_i 可以简化为 $K_{\max}$，则复杂度为 $O(n \cdot W \cdot K_{\max})$。

```cpp
// Algorithm for Multiple Knapsack (Direct DP)
int multiple_knapsack_v1(const std::vector<Item>& items, int capacity) {
    int n = items.size();
    if (n == 0) return 0;
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(capacity + 1, 0));
    for (int i = 1; i <= n; ++i) {
        int w = items[i - 1].weight;
        int v = items[i - 1].value;
        int k = items[i - 1].count; // Max count for this item
        for (int j = 0; j <= capacity; ++j) {
            dp[i][j] = dp[i-1][j];
            for (int c = 1; c <= k && c * w <= j; ++c) {
                dp[i][j] = std::max(dp[i][j], dp[i - 1][j - c * w] + c * v);
            }
        }
    }
    return dp[n][capacity];
}
```

代码 4: 多重背包算法一 C++ 实现

7.2 多重背包算法二：二进制优化

二进制优化是解决多重背包问题的一种高效方法。其核心思想是将每种数量有限的物品拆分成若干件特殊的“物品”，使得这些特殊物品的组合可以表示原物品的任意数量。具体来说，对于第 i 种物品，如果其数量上限为 k_i，我们可以将其拆分为重量和价值分别为 $c \cdot w_i$ 和 $c \cdot v_i$ 的“物品”，其中 c 取 $1, 2, 4, \text{dots}, 2^p$，以及剩余的 $k_i - (2^{p+1} - 1)$。这些 c 的和可以表示从 1 到 k_i 之间的任何一个整数。

拆分后，多重背包问题就转化为了一个 0/1 背包问题。我们可以使用 0/1 背包问题的标准动态规划方法（如与完全背包算法三类似的一维 DP 优化）来解决。 转化后的物品总数将从 sum k_i 减少到 sum log k_i，从而将时间复杂度优化为 $O(W \cdot \text{sum log } k_i)$。