算法导论课程报告

——01 背包问题的求解与分析

赵伯远

211440128

2023 年 12 月 4 日

目录

1 问题描述 **2**

2 算法分析 **2**

2.1 蛮力法 2

2.1.1 朴素方法 2

2.1.2 位运算优化 3

2.2 动态规划 4

2.2.1 朴素方法 4

2.2.2 滚动数组优化 4

2.2.3 一维数组优化 5

2.3 遗传算法 6

2.3.1 遗传算法简介 6

2.3.2 锦标赛选择 6

2.4 算法效率分析 8

2.5 蛮力法 8

2.6 动态规划 8

2.7 遗传算法 8

3 实验结果 **8**

1 问题描述

01 背包问题是组合优化问题中的一个经典问题，其背景可以追溯到物品的装载和资源的分配。具体来说，假设有一个背包，其承载的最大重量为 W。同时，有 n 个物品，每个物品有各自的重量 w_i 和价值 v_i。01 背包问题的目标是选择一些物品装入背包中，使得这些物品的总重量不超过背包的最大承载重量，而它们的总价值尽可能大。

形式化地，可以将问题描述如下：

目标：

$$\max \sum_{i=1}^{n} v_i \cdot x_i \tag{1}$$

受到约束：

$$\sum_{i=1}^{n} w_i \cdot x_i \leq W \tag{2}$$

其中：

- x_i 是一个二元变量，如果物品 i 被选中，则 $x_i = 1$；如果没有被选中，则 $x_i = 0$。
- v_i 是物品 i 的价值。
- w_i 是物品 i 的重量。
- W 是背包的最大承重。

2 算法分析

2.1 蛮力法

2.1.1 朴素方法

朴素的蛮力法的核心思想是检查所有可能的物品组合。对于每种组合，我们计算其总重量和总价值，并检查总重量是否超过背包的最大容量。如果没有超过，我们将其与当前找到的最大价值进行比较，以更新最大价值。

Algorithm 1: BruteForceKnapsack

Input: $weights$, $values$, $capacity$

Output: Maximum value achievable

Function `BruteForceKnapsack`($weights, values, capacity$)**:**

- $maxValue \leftarrow 0$
- **foreach** $combination$ **do**
  - **if** $Weight(combination) \leq capacity$ **then**
    - $maxValue \leftarrow \max(maxValue, Value(combination))$
  - **end**
- **end**
- **return** $maxValue$

2.1.2 位运算优化

位运算优化方法是朴素方法的一种改进。在这种方法中，我们使用整数的二进制表示来表示物品的选择状态。每个位代表一个物品，其中 1 表示选择该物品，0 表示不选择。

通过遍历从 0 到 $2^n - 1$ 的所有整数，我们可以有效地遍历所有物品的组合。对于每个整数，我们使用位运算来确定哪些物品被选中，并计算这些物品的总重量和总价值。这种方法在理论上与朴素方法相同，但在实践中通常更高效，因为它直接利用了计算机的二进制运算能力（常数优化）。

Algorithm 2: BitwiseOptimizedKnapsack

Input: $weights$, $values$, $capacity$

Output: Maximum value achievable

Function `BitwiseOptimizedKnapsack`($weights, values, capacity$)**:**

- $maxValue \leftarrow 0$
- **for** $i \leftarrow 0$ **to** $2^n - 1$ **do**
  - $weight \leftarrow 0, value \leftarrow 0$
  - **for** $j \leftarrow 0$ **to** $n - 1$ **do**
    - **if** $i \mathbin{\&} (1 \ll j)$ **then**
      - $weight \leftarrow weight + weights[j]$
      - $value \leftarrow value + values[j]$
    - **end**
  - **end**
  - **if** $weight \leq capacity$ **then**
    - $maxValue \leftarrow \max(maxValue, value)$
  - **end**
- **end**
- **return** $maxValue$

2.2 动态规划

2.2.1 朴素方法

在二维数组方法中，我们定义 $dp[i][v]$ 为用剩余容量为 v 的背包来装前 i 件物品时可以达到的最大价值。因此，状态转移方程如下：

$$dp[i][0] = 0$$

对于每个物品 i 和每个可能的容量 v，有以下两种情况：

- 如果物品 i 的体积大于 v，即 $w[i] > v$，则当前物品无法装入背包，所以 $dp[i][v] = dp[i-1][v]$。
- 如果物品 i 的体积不大于 v，即 $w[i] \leq v$，则有两种选择：装入物品 i 或者不装入。因此，状态转移方程为：

$$dp[i][v] = \max(dp[i-1][v], dp[i-1][v-w[i]] + p[i])$$

在这里，$w[i]$ 和 $p[i]$ 分别代表物品 i 的重量和价值。

Algorithm 3: TwoDimensionalKnapsack

Input: $weights$, $values$, $capacity$

Output: Maximum value achievable

```
Function TwoDimensionalKnapsack(weights, values, capacity):
    dp ← array[0..length(weights)][0..capacity] initialized to 0
    for i ← 1 to length(weights) do
        for w ← 0 to capacity do
            if weights[i] ≤ w then
                dp[i][w] ← max(dp[i − 1][w], dp[i − 1][w − weights[i]] + values[i])
            else
                dp[i][w] ← dp[i − 1][w]
            end
        end
    end
    return dp[length(weights)][capacity]
```

2.2.2 滚动数组优化

在标准的二维数组方法中，我们使用一个二维数组 $dp[i][w]$ 来存储所有可能的状态。然而，实际上，在计算 $dp[i][w]$ 时，我们只需要前一行 $i-1$ 的数据。这意味着，我们并不需要存储整个二维数组来获得最终结果，而只需保留足够的信息来推导出当前行的值。

滚动数组的优化基于这样一个观察：在计算第 i 行的状态时，只需要第 $i-1$ 行的状态。因此，我们可以使用两个一维数组交替表示当前行和上一行，而不是使用完整的二维数组。这种

方法大大减少了空间复杂度，从 $O(nW)$ 减少到 $O(2W)$，其中 n 是物品的数量，W 是背包的容量。我们通过交替更新两个数组，以保持动态规划的正确性，同时节约空间。

状态转移方程改写如下：

$$dp[current][w] = \max(dp[previous][w], dp[previous][w - weights[i]] + values[i]) \tag{3}$$

这里，$current$ 和 $previous$ 分别代表当前和前一行的索引。这种方法通过交替使用两个数组（或两行），减少了空间的需求，同时保持了动态规划的核心逻辑不变。

Algorithm 4: RollingArrayKnapsack

Input: $weights, values, capacity$

Output: Maximum value achievable

```
Function RollingArrayKnapsack(weights, values, capacity):
    dp ← array[2][0..capacity] initialized to 0
    for i ← 1 to length(weights) do
        for w ← 0 to capacity do
            if weights[i − 1] ≤ w then
                dp[i%2][w] ←
                  max(dp[(i − 1)%2][w], dp[(i − 1)%2][w − weights[i − 1]] + values[i − 1])
            else
                dp[i%2][w] ← dp[(i − 1)%2][w]
            end
        end
    end
    return dp[length(weights)%2][capacity]
```

2.2.3 一维数组优化

在进一步优化，我们只使用一个一维数组 $dp[w]$。在这种情况下，更新 $dp[w]$ 时需要确保引用的 $dp[w - weights[i]]$ 是上一个物品状态的值，而非当前物品已经更新过的值。

为了确保这一点，我们采用逆序枚举的方式更新数组。这意味着对于每个物品 i，我们从背包的最大容量开始向下遍历到该物品的重量。逆序更新确保在更新 $dp[w]$ 时，$dp[w - weights[i]]$ 仍然代表不包含当前物品 i 的情况。如果我们采用正序枚举，那么在更新 $dp[w]$ 时，$dp[w - weights[i]]$ 可能已经被当前物品更新，从而导致错误的结果。

通过逆序枚举，我们可以保证每次更新 $dp[w]$ 时，所依赖的历史数据（较小的 w 值）是有效的，从而确保算法的正确性。这种方法的空间复杂度是 $O(W)$，在实际应用中非常有效，特别是对于背包容量大但可用内存有限的情况。

状态转移方程改写如下：

$$dp[w] = \max(dp[w], dp[w - weights[i]] + values[i]) \tag{4}$$

在这个方程中，只使用 $dp[w]$ 来存储每个容量下的最大价值。逆序更新是关键，它保证了在更新 $dp[w]$ 时，$dp[w - weights[i]]$ 仍然代表不包含当前物品 i 的情况。

Algorithm 5: OneDimensionalKnapsack

Input: $weights$, $values$, $capacity$

Output: Maximum value achievable

Function `OneDimensionalKnapsack`($weights, values, capacity$)**:**
    $dp \leftarrow$ array[0..$capacity$] initialized to 0
    for $i \leftarrow 0$ **to** *length(weights) - 1* **do**
        for $w \leftarrow capacity$ **to** $weights[i]$ ***step -1*** **do**
            $dp[w] \leftarrow \max(dp[w], dp[w - weights[i]] + values[i])$
        end
    end
    return $dp[capacity]$

2.3 遗传算法

2.3.1 遗传算法简介

遗传算法是一种模仿自然界生物进化机制的优化方法，通过迭代过程在解空间中搜索最优解。对于 01 背包问题，我们可以利用遗传算法高效地探索潜在的最优解。

首先，算法随机初始化一个种群，每个个体由一系列基因组成，这些基因代表背包中物品的装载状态，即 1 代表装入，0 代表不装入。种群的每个个体都是一个潜在的解决方案。

适应度函数是遗传算法的核心，用于评价个体的适应程度。对于 01 背包问题，适应度函数通常是个体中所选物品的总价值，但需要注意的是，若个体所选物品的总重量超过背包容量，则应对该个体进行惩罚，使其适应度为 0。这样可以确保选出的解是有效的。

在每一代中，算法首先根据适应度对种群进行排序，以确保更适应的个体有更高的存活概率。然后，算法通过交叉和变异操作产生新的个体。交叉操作模仿生物的繁殖过程，从两个父本个体中创造出新的后代，而变异操作则随机改变个体的某些基因，增加种群的多样性。

通过重复这个过程，种群逐渐进化，适应度较高的个体逐渐占据主导地位。最终，算法通过选择适应度最高的个体作为问题的最优解。这个过程不断迭代，直到满足特定的停止条件，如达到预设的代数。

2.3.2 锦标赛选择

由于基础的遗传算法表现不佳，因此我们可以考虑引入更加优秀的选择算子。锦标赛选择是一种常用的选择算子，用于从种群中选择个体。在锦标赛选择中，我们随机选择 k 个个体，然后从中选择适应度最高的个体作为父本。这个过程重复 n 次，直到选择出 n 个父本。

Algorithm 6: Knapsack Genetic Algorithm with Tournament Selection

Input: Items, W, PopulationSize, Generations, CrossoverRate, MutationRate, TournamentSize

Output: Maximum value achieved by the knapsack

```
Initialize random number generator
Initialize population with random individuals
foreach individual in population do
    foreach gene in individual do
        gene = random value (0 or 1)
    end
end
Define fitness function
for generation = 0 to Generations do
    Initialize newPopulation
    for i = 0 to PopulationSize do
        Select parent1 and parent2 using tournament selection
        Initialize child
        if random number < CrossoverRate then
            Perform crossover between parent1 and parent2
        else
            child = parent1
        end
        foreach gene in child do
            if random number < MutationRate then
                Mutate gene
            end
        end
        Add child to newPopulation
    end
    Replace population with newPopulation
    Sort population based on fitness
    Update answer with the fitness of the best individual
end
Sort population based on fitness
return Maximum fitness value
```

2.4 算法效率分析

在 01 背包问题中，不同的算法有各自的优势和限制。以下是一些主要算法的时间和空间复杂度的分析。

2.5 蛮力法

蛮力法通过枚举所有可能的物品组合来寻找最优解。对于每个物品，有两种可能的状态（选中或未选中），因此对于 n 个物品，总共有 2^n 种可能的组合。因此，蛮力法的时间复杂度为 $O(2^n)$。每个组合需要 $O(n)$ 的空间来存储，因此空间复杂度为 $O(n)$。

2.6 动态规划

动态规划通过构建一个二维数组来避免重复计算，数组的大小为物品数量 n 乘以背包容量 W。因此，时间和空间复杂度均为 $O(nW)$。通过优化方法如滚动数组和一维数组，可以将空间复杂度降低到 $O(2W)$ 和 $O(W)$，分别适用于空间受限和高度空间受限的情况。

2.7 遗传算法

遗传算法的时间复杂度取决于种群大小 p、代数 g 和每代中的操作数量 m，因此时间复杂度为 $O(g \cdot p \cdot m)$。空间复杂度主要受种群大小 p 影响，因此为 $O(p)$。遗传算法适用于大规模和复杂的问题，特别是在解空间较大时。

表 1: 蛮力法、动态规划和遗传算法在 01 背包问题中的性能对比

算法类型	时间复杂度	空间复杂度	适用场景
蛮力法	$O(2^n)$	$O(n)$	小规模问题
动态规划（朴素方法）	$O(nW)$	$O(nW)$	通用
动态规划（滚动数组优化）	$O(nW)$	$O(2W)$	空间受限
动态规划（一维数组优化）	$O(nW)$	$O(W)$	空间高度受限
遗传算法	$O(g \cdot p \cdot m)$	$O(p)$	大规模、复杂问题

3 实验结果