天津大学

《算法设计与分析》实验报告

0/1 背包问题不同求解算法的性能分析与比较

0/1 背包问题不同求解算法的性能分析与比较

摘要:本次实验针对 0/1 背包问题，基于 Knapsack 数据集，利用不同求解算法进行求解，并对各种算法的时间复杂度进行了分析、验证和比较。主要做了以下三方面的工作，一是对动态规划算法、回溯法、分支限界法求解 0/1 背包问题的时间复杂度进行了实验验证；二是分析了数据相关性（不相关、弱相关、强相关）对三种算法性能的影响；三是在不同相关性的数据下，比较三种算法性能随着问题规模增大的优劣。实验结果表明，不同数据规模下和不同相关性的数据下，最优的算法选择并不一致，三种算法都有在其特定范围和条件下比其他两种更优。

关键词: 0/1 背包、动态规划、回溯法、分支限界、时间复杂度、数据相关性

1 实验目的

（1） 设计动态规划算法、回溯法、分支限界算法的程序求解 0/1 背包问题。

（2） 验证动态规划算法、回溯法、分支限界算法解决 0/1 背包问题的时间复杂度。

（3） 分析数据的相关性对三种求解算法的性能影响。

（4） 比较三种算法在不同数据规模和条件下的优劣。

2 实验说明

（1）实验数据集

实验数据集来源：Datasets (unicauca.edu.co)

该数据集包含了小规模实验数据和大规模实验数据。大规模数据的规模，物品数量可从 100 到 10000，背包容量规模可 995 到 49877。大规模数据，其有无相关性数据、弱相关性数据、强相关性数据三种类型。本文使用该数据集的小规模数据进行算法的正确性验证，使用大规模数据进行算法的时间复杂度分析等。

(2) 实验仓库

本实验所有的实验代码、实验数据、结果数据、图表文件所在的仓库地址：GGbondpro/ZeroOneKnapsack (github.com)

(3) 符号说明

文中 n 代表可选物品的个数，c 代表背包容量，w_i 代表物品重量，v_i 代表物品价值。t 代表算法求解该问题的时间，$z=\log_2(t)$ 是对算法求解时间取以 2 为底的对数。

3 实验设计与结果分析

3.1 验证动态规划算法的时间复杂性

(1) 实验原理

理论分析表明，动态规划算法需要遍历每一个物品和每一个背包容量，即需要 O(n*c)的时间复杂度，其中 n 为可选物品个数，c 为背包容量。本文使用实验方法对该复杂度进行验证。

实验时，保证背包容量 c 不变，改变物品数量 n，统计求解问题的时间 t 随着可选物品个数 n 的变化，验证 t 与 n 是否成正比关系。同理，保证物品数量 n 不变，改变背包容量 c，统计求解时间 t 随着背包容量的变化，验证 t 与 c 是否成正比关系。如果以上两个正比关系都成立，便可验证该算法的时间复杂度为 O(n*c)。

(2) 实验设计

设计动态规划算法。假设有 n 个物品和一个背包，其容量为 c。设 w_i 和 v_i 分别

为第 i 个物品的重量和价值。令 dp[i][j] 表示前 i 个物品中选择总重量不超过 j 的物品时的最大价值。那么，动态规划方程为：

$$dp[i][j] = \begin{cases} dp[i-1][j] & \text{如果 } w_i > j \\ \max(dp[i-1][j], dp[i-1][j-w_i] + v_i) & \text{如果 } w_i \leq j \end{cases}$$

根据算法原理，设计代码的核心部分如下：

```
// Function to solve the 0/1 Knapsack problem using dynamic programming
void dynamic(vector<vector<int>> &dp,int n, int c, vector<int> &values, vector<int> &weights)
{
    // Fill the table in a bottom-up manner
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= c; j++)
        {
            if (weights[i - 1] <= j)
            {
                dp[i][j] = max(values[i - 1] + dp[i - 1][j - weights[i - 1]], dp[i - 1][j]);
            }
            else
            {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
}
```

经过测试验证，该代码可以求解出正确的结果，下面我们利用该代码进行算法的时间复杂度验证。

1、使用数据集“large_scale/knapPI_1_1000_1000_1”进行测试，该数据集的包含了 1000 个可选物品的价值和重量，以及给定的背包大小 5002。

背包容量 5002 保持不变，将可选物品数量从 2 变化到 1000，步长为 2，得到 500 个数据点，统计每个数据点利用动态规划算法求解问题的时间。每个数据点重复求解 10 次，取时间的平均值。作出求解时间随物品个数的变化的图表 t-n，如图 1。

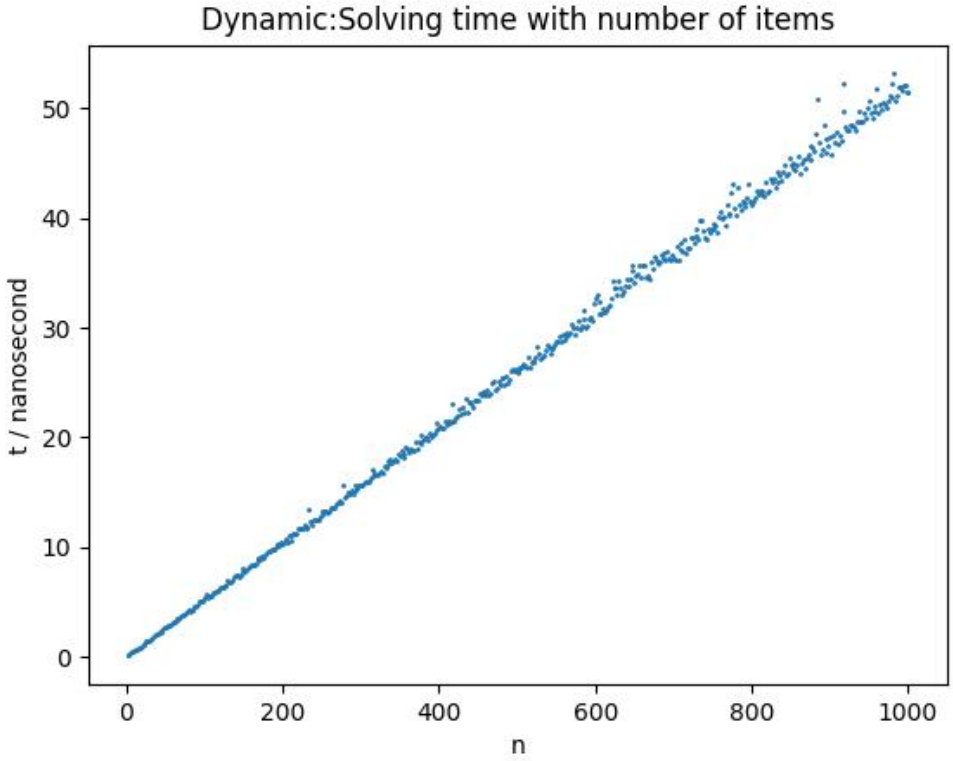

图 1

2、保持物品数量为 500 不变，将背包容量从 10 变化到 5000，步长为 10，得到 500 个数据点，统计每个数据点利用动态规划算法求解问题的时间。每个数据点重复求解 10 次，取时间的平均值。作出时间随背包容量的变化的图表 t-c，如图 2。

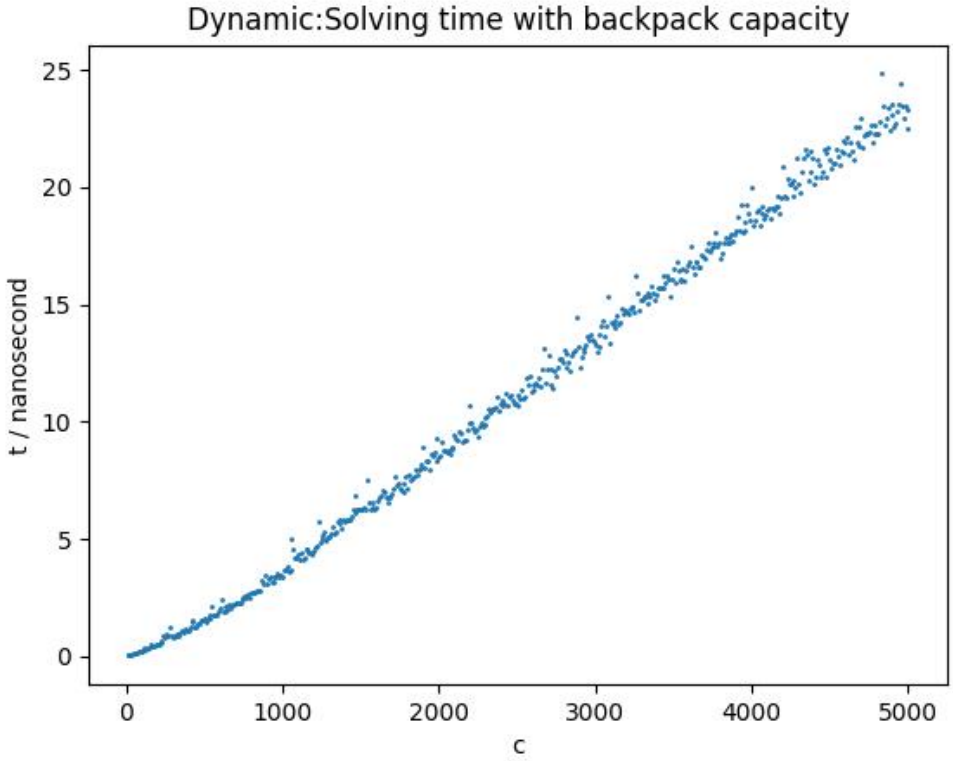

图 2

3、使用三个具有不同相关性的数据集“large_scale/knapPI_1_1000_1000_1”、“large_scale/knapPI_2_1000_1000_1”、“large_scale/knapPI_3_1000_1000_1”进行测试。三个数据集都包含了 1000 个可选物品的价值和重量，不同的是，物品的重量与物品的价值是否存在相关性，三个数据集依次是：不相关、弱相关、强相关。统计三种相关性的数据下，时间随物品个数的变化的图表，如图 3。

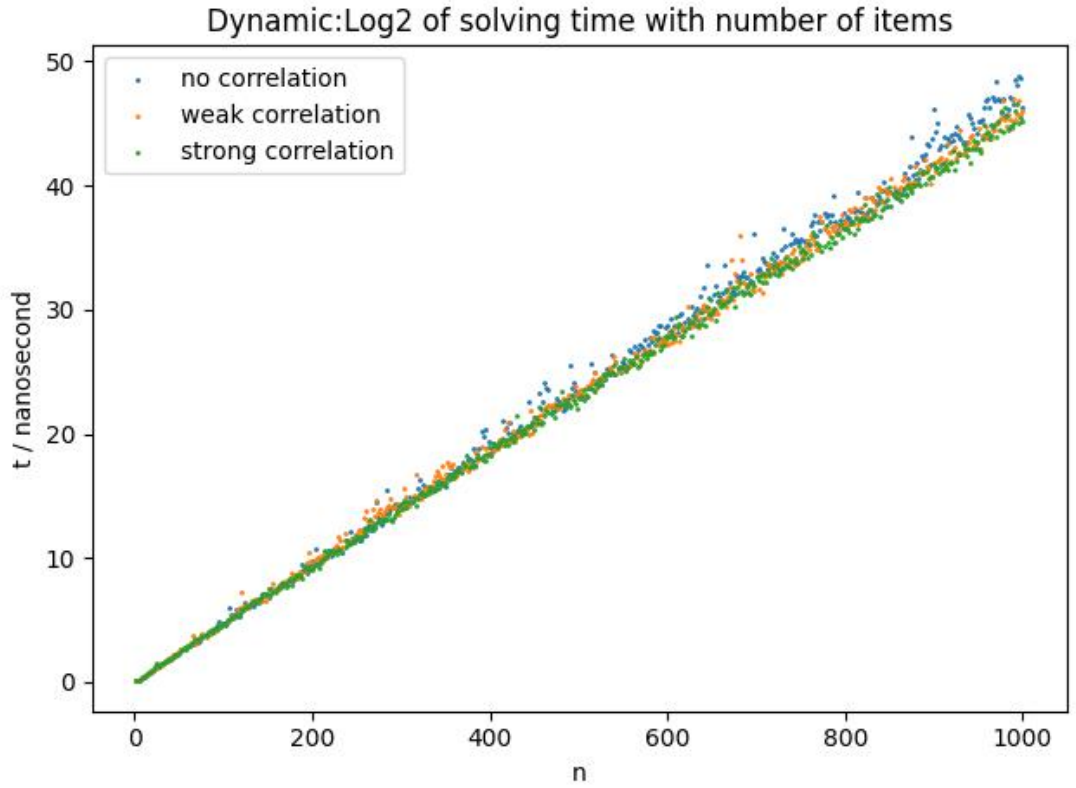

图 3

（3）结果分析

如图 1 所示，在背包容量一定的情况下，动态规划算法的求解时间与可选物品个数成正比。如图 2 所示，在可选物品个数一定的情况下，动态规划算法的求解时间与背包容量成正比。因此可以得到动态规划算法的求解时间 t 与可选物品个数和背包容量的乘积 n*c 成正比，即动态规划算法的时间复杂度为 O(n*c)。

从图 3 中可以发现，问题的数据是否具有相关性对动态规划算法的性能并没有明显的影响。

3.2 回溯法的时间复杂度验证

（1）实验原理

对于 0/1 背包问题，使用回溯法，每次可以选择将一个物品放入背包或不放入背包。如果有 n 个物品，那么总共会有 2^n 种可能的组合。因此，理论上回溯法求解 0/1 背包问题的时间复杂度是 $O(2^n)$。本文使用实验方法对其进行验证。

实验时，保证背包容量 c 不变，改变物品数量 n，统计求解时间 t 随着可选物品个数的变化观察 t 与 n 是否具有指数级的增长关系。进一步地，令 $z = \log_2(t)$, 如果有 $z \sim t$, 则可验证 $t \sim 2^n$,即验证回溯法的时间复杂度为 $O(2^n)$。

(2) 实验设计

设计回溯算法的核心代码如下：

```
// Backtracking pruning search on the solution space
void backtrack_2(vector<int> &values, vector<int> &weights, vector<int> &solution,
                 int capacity, int current_value, int current_weight, int index)
{
    if (index == values.size())
    {
        if (current_value > max_value)
        {
            max_value = current_value;
            best_solution = solution;
        }
        return;
    }

    if (current_weight + weights[index] <= capacity)
    {
        solution[index] = 1;
        backtrack_2(values, weights, solution, capacity,
                    current_value + values[index], current_weight + weights[index], index + 1);
    }

    solution[index] = 0;
    backtrack_2(values, weights, solution, capacity, current_value,
                current_weight, index + 1);
}
```

1、使用数据集“large_scale\\knapPI_1_200_1000_1”进行测试，该数据集的包含了 200 个可选物品的价值和重量，以及给定的背包大小 1008。物品数量从 2 变化到 130，步长为 2，背包容量 1008 不变。每个数据点统计 5 次求解时间，取平均值。作出求解时间随 t 随物品数量 n 变化的图表 t-n。令 $z = \log_2(t)$，作出 z - t，如图 4。

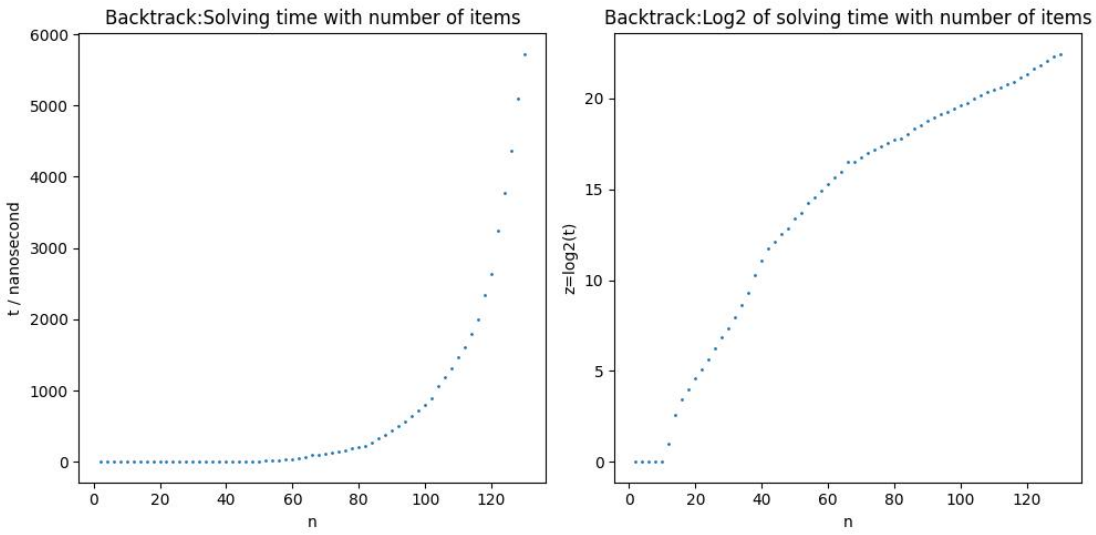

图 4

2、使用三个不同相关性的数据集“large_scale/knapPI_1_200_1000_1”、“large_scale/knapPI_2_200_1000_1”、“large_scale/knapPI_3_200_1000_1”进行测试，统计求解问题的时间，每个数据点重复求解 5 次取时间的平均值。作出三种相关性数据下，时间的对数 z 随物品个数 n 的变化的图表 z-n，如图 5。

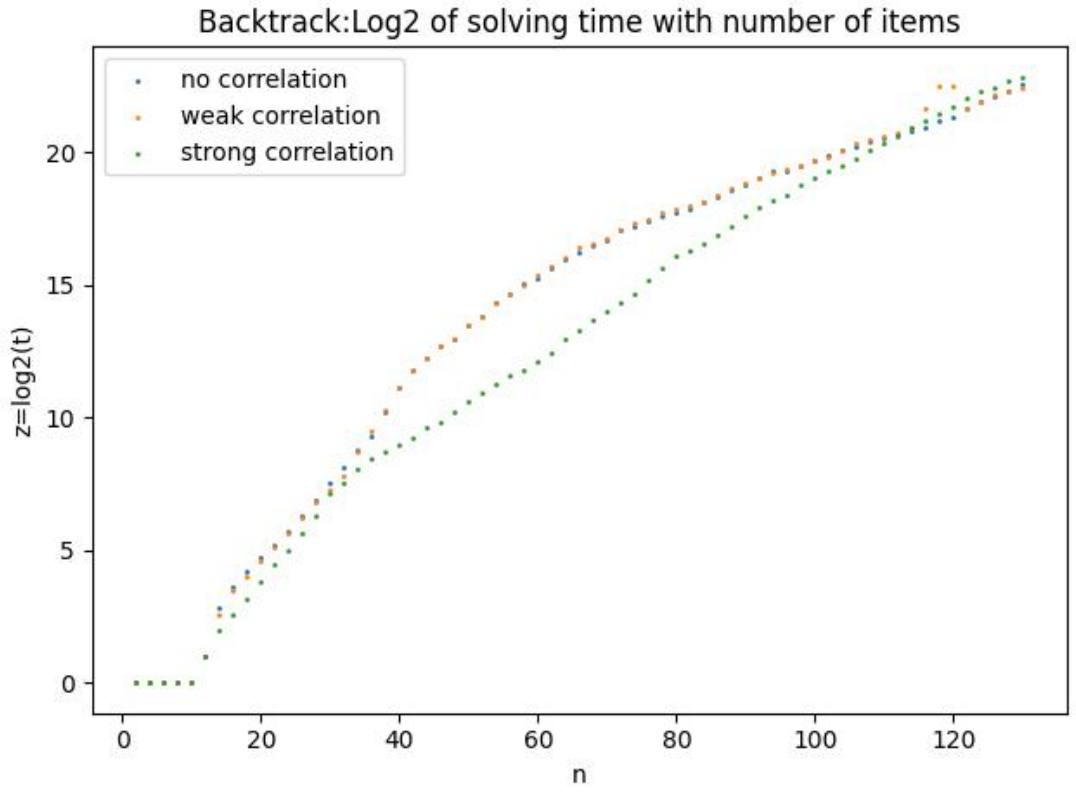

图 5

(3) 结果分析

由图 4 可以发现，回溯法求解 0/1 背包问题的时间曲线 t-n 呈指数型增长。对求解时间取对数，z-t 曲线近似直线并向下弯曲。这是由于回溯法在求解 0/1 背包问题时，并没有对解空间进行完全的搜索，而是基于背包容量限制进行了剪枝，这一剪枝过程一定程度上使时间复杂度低于 $O(2^n)$（z-t 曲线并向下弯曲），但仍趋近于 $O(2^n)$（z-t 曲线近似直线）。因此，验证了回溯法的时间复杂度仍然是 $O(2^n)$。

从图 5 中可以发现，问题的数据是否具有相关性对动态规划算法的性能并没有明显的影响。

3.3 验证分支限界算法的时间复杂度

(1) 实验原理

分支限界算法，每次选择当前叶节点中具有最优值的节点进行拓展，这里涉及一个堆排序的过程，时间复杂度为 O(n)。在最坏的情况下，分支限界法仍可能会对整个解空间进行遍历，解空间的大小为 $O(2^n)$。因此，分支限界算法在最坏的情况下，时间复杂度为 $O(n*2^n)$。但是进行剪支之后，算法的实际搜索范围降到非常低，平均求解时间大大减少。本文通过实验，分析限界剪支对降低算法时间复杂度的效果。

分支限界法具有左支限界、右支限界两个限界条件，左支限界是利用背包容量的进行限界，右支限界是利用当前的最优值进行下限限界。

(2) 实验设计

设计分支限界算法,关键部分代码如下：

```
// Include the item at the current level, left branch limit
if (weight + items[level].weight <= capacity)
{
    solution[level] = 1;
    pq.push({level, value + items[level].value,
    weight + items[level].weight, solution});
}

// Exclude the item at the current level, right branch limit
// calculate the remaining maximum value,
//if it is greater than the current maximum value, then push it to the
int remaining_capacity = capacity - weight;
int remaining_max_value = 0;
for (int i = level + 1; i < n; i++)
    if (remaining_capacity >= items[i].weight)
    {
        remaining_capacity -= items[i].weight;
        remaining_max_value += items[i].value;
    }
    else
    {
        remaining_max_value += items[i].density * remaining_capacity;
        break;
    }
if (value + remaining_max_value >= max_value)
{
    solution[level] = 0;
    pq.push({level, value, weight, solution});
}
```

1、使用数据集“large_scale/knapPI_1_200_1000_1”进行测试，该数据集的包含了 200 个可选物品的价值和重量。物品数量从 1 变化到 30，背包容量 1008 保持不变，统计三种限界条件下的问题求解时间。每个数据点统计 5 次求平均值，令 $z=\log_2(t)$，作出 t-n 曲线，如图 6 和图 7。

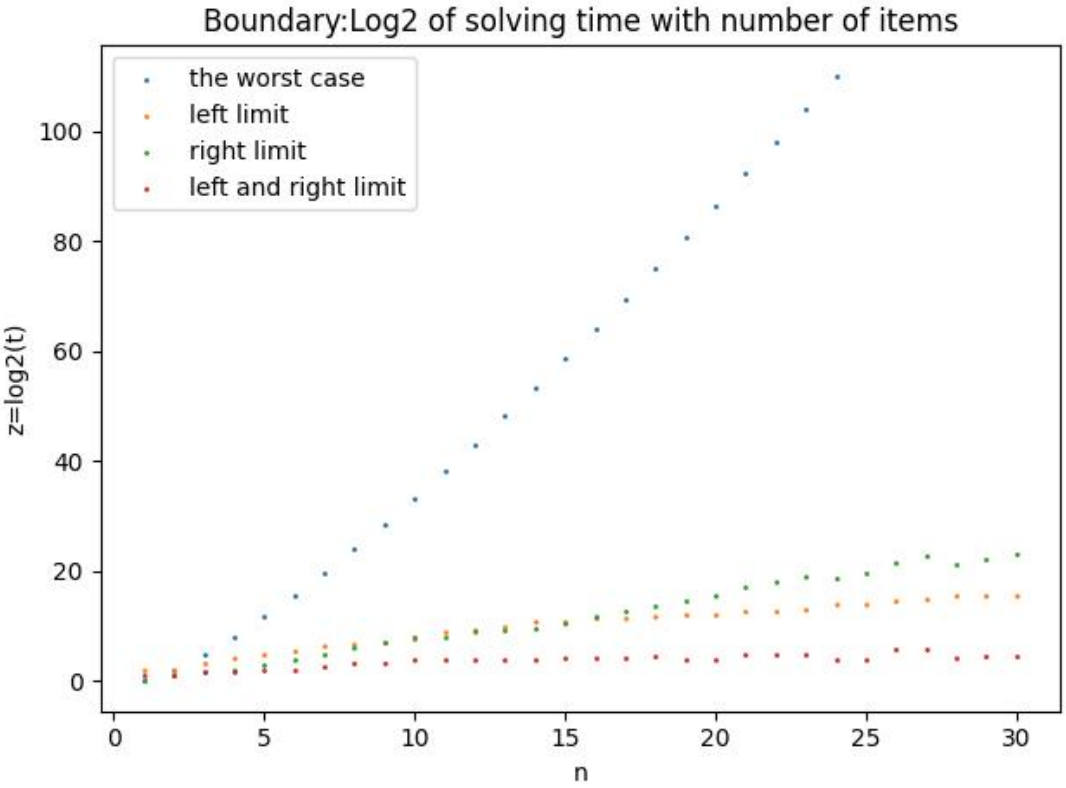
Boundary:Log2 of solving time with number of items
the worst case
left limit
right limit
left and right limit
z=log2(t)
n
0
20
40
60
80
100
0
5
10
15
20
25
30

图 6

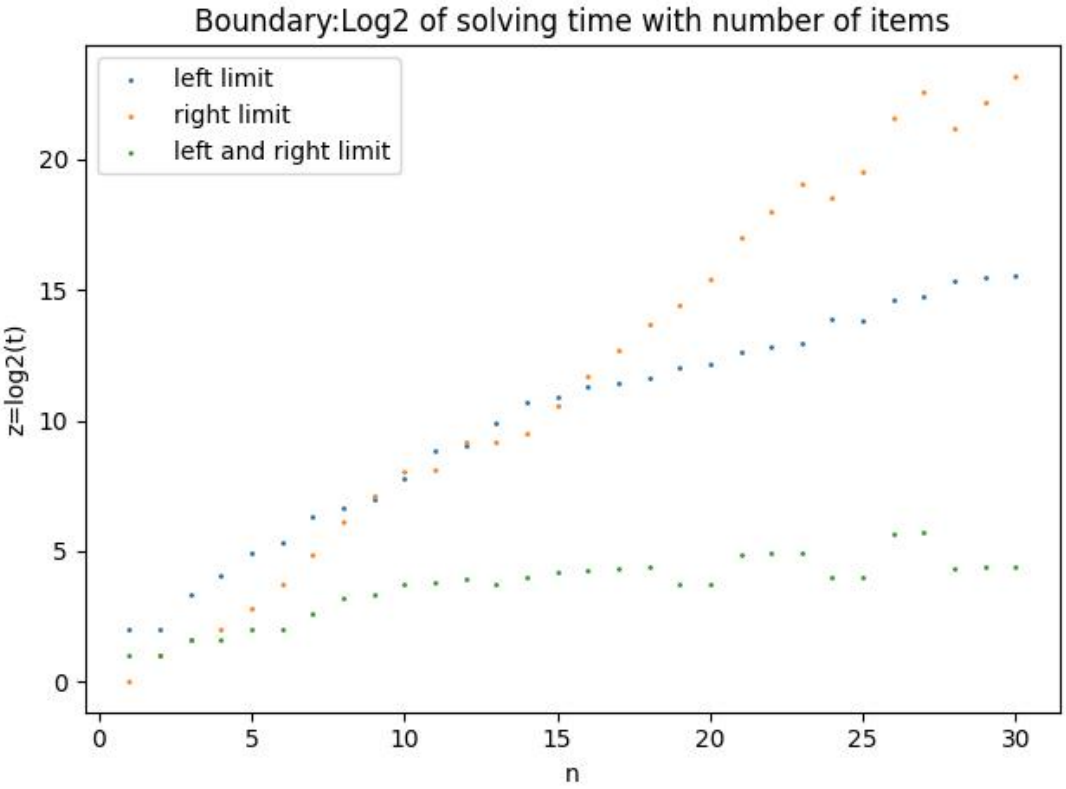
Boundary:Log2 of solving time with number of items
left limit
right limit
left and right limit
z=log2(t)
n
0
5
10
15
20
0
5
10
15
20
25
30

图 7

2、同样，使用三个不同相关性的数据集“large_scale/knapPI_1_200_1000_1”、“large_scale/knapPI_2_200_1000_1”、“large_scale/knapPI_3_200_1000_1”进对该算法行测试。作出三种相关性数据下，时间的对数 z 随物品个数 n 的变化的图表 z-n,如图 8。

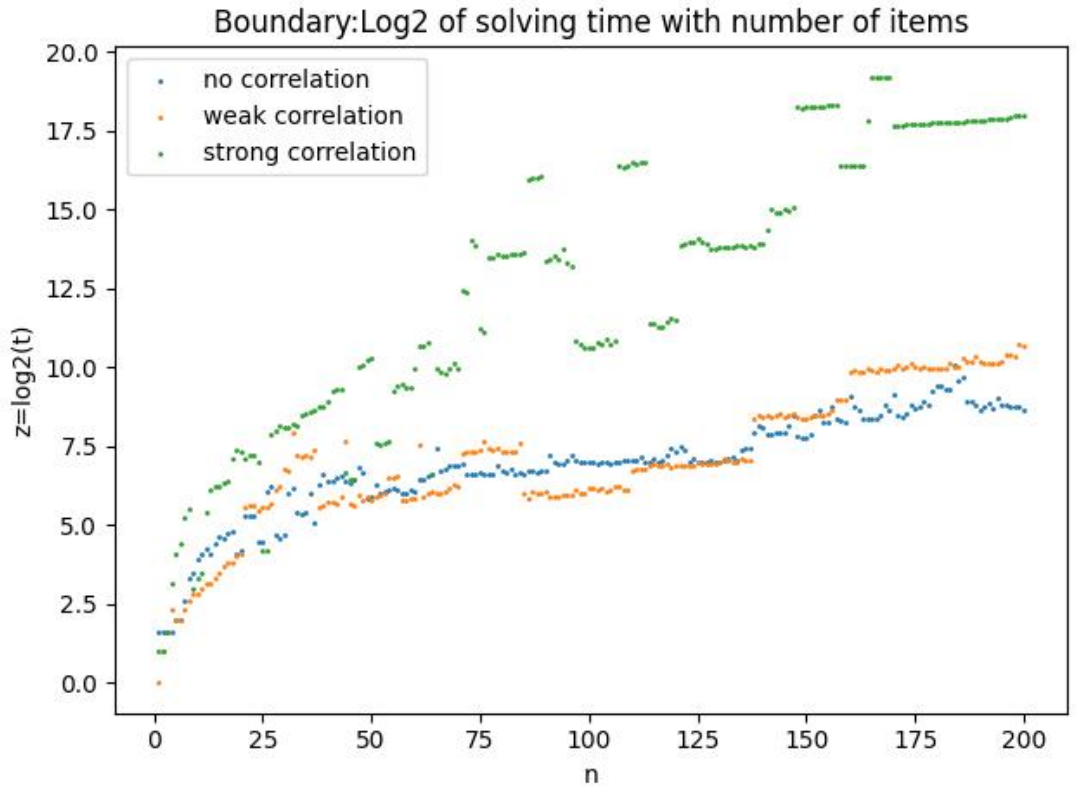

图 8

(3) 结果分析

从图 6 和图 7 可以看出，与不剪枝相比(the worst case)，限界剪枝方法大大地降低了算法的实际求解时间。其中同时进行左枝和右枝限界时，算法的时间性能达到最好，验证了限界剪枝对减低算法的求解时间具有很好的效果。

从图 8 可以看出，与动态规划和回溯法不同，分支限界算法对数据的相关性有较大的敏感性。使用分支限界法解决 0/1 背包问题时，强相关的数据比弱相关和不相关的数据，需要耗费更多的求解时间，并且时间表现出明显的抖动性。

3.4 三种算法的性能比较

(1) 无相关数据下的性能比较

在无相关数据下，使用数据集“large_scale/knapPI_1_200_1000_1”，物品数量从1 变化到 130，步长为 1，背包容量 1008 保持不变，统计三种算法解决问题的时间，每个数据点统计 5 次求平均值，如图 9 所示。

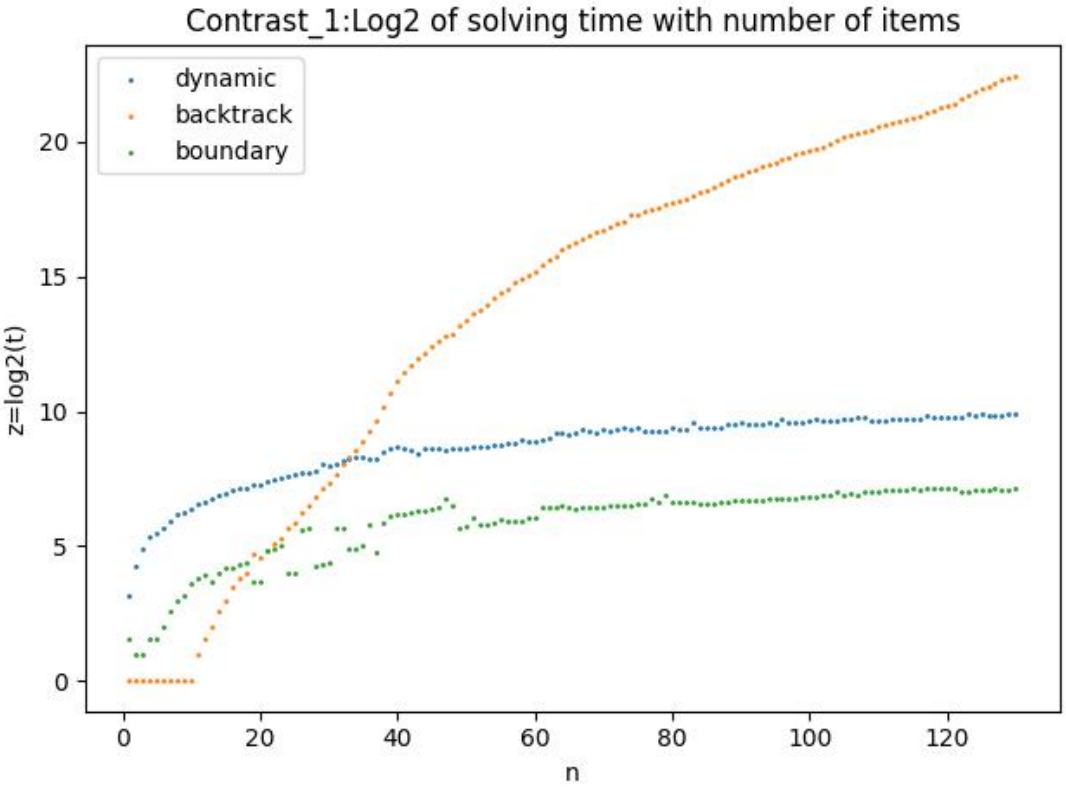

图 9

使用更大的数据集“large_scale/knapPI_1_10000_1000_1”，进一步比较动态规划和分支限界算法在更大规模数据下的性能表现。物品数量从 10 变化到 10000，步长为 10，背包容量 49877 保持不变，统计两种算法解决问题的时间，每个数据点统计 1 次，如图 10 所示。

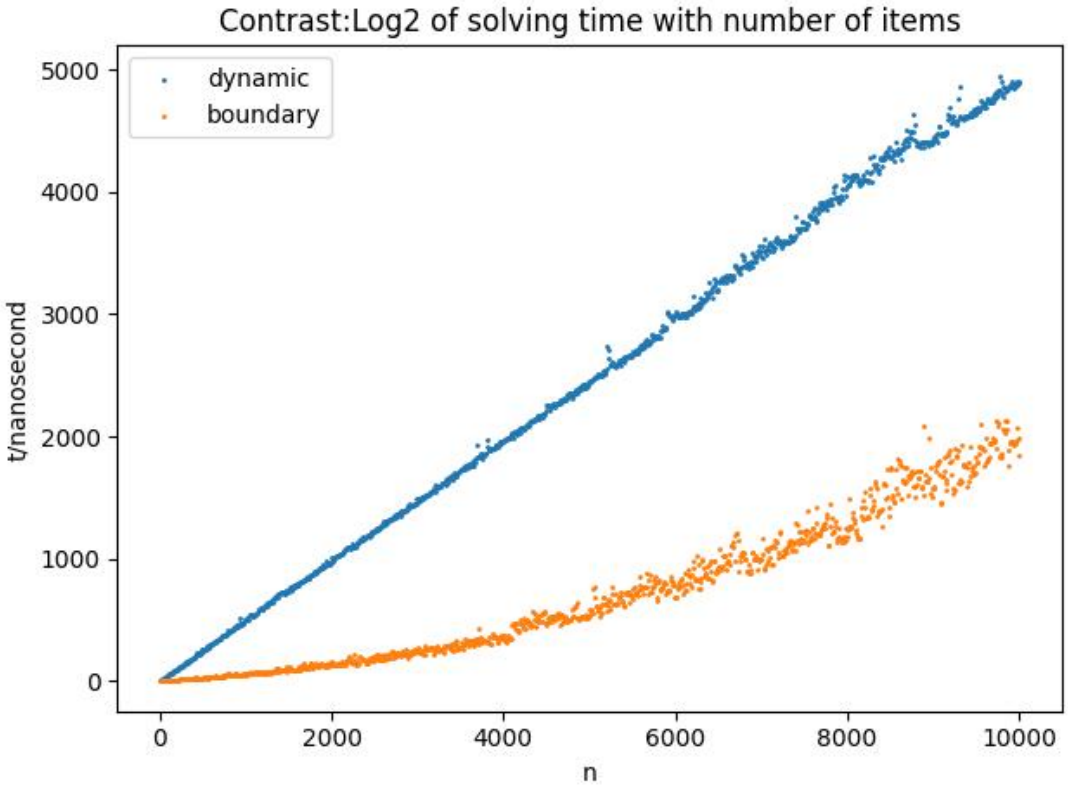

图 10

由图 9 可知，在本实验特定的环境和背包容量为 1008 条件下，对于无相关的数据，当数据规模较小时(约小于 20)，回溯算法表现出更好的时间性能。但随着数据规模的增大，回溯法的求解时间呈指数增长（图中的时间已经做了取对数处理），性能迅速下降。由图 9 和图 10 可知，当数据规模大于一定值以后，分支限界算法表现出更好的性能。动态规划算法求解时间则始终稍长于分支限界算法。

(2) 弱相关数据下的性能比较

在弱相关数据下，使用数据集“large_scale/knapPI_2_200_1000_1”，物品数量从 1 变化到 130，步长为 1，背包容量 1008 保持不变，统计三种算法解决问题的时间，每个数据点统计 5 次求平均值，如图 11 所示。

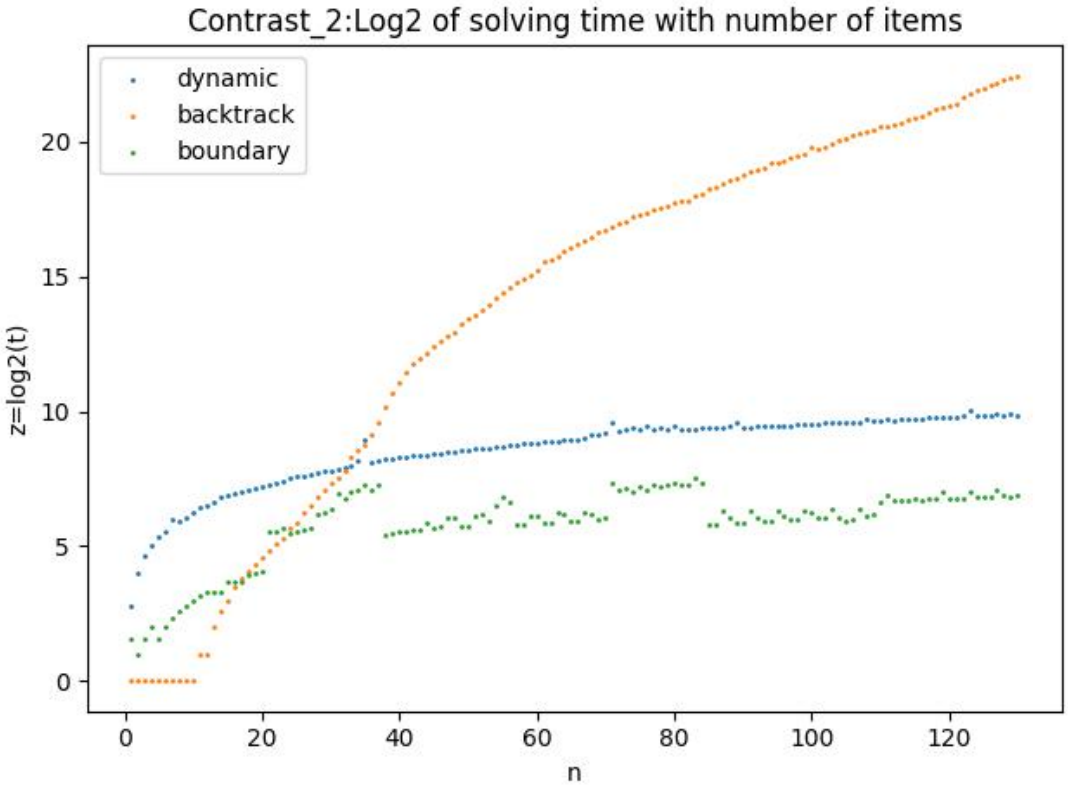

图 11

由图 11 并对比图 9 可知，弱相关数据下，三种算法的性能表现差异与不相关数据下的性能表现类似。不同的是，分支限界算法的曲线开始表现出抖动性。

(3) 强相关数据下的性能比较

在强相关数据下，使用数据集“large_scale/knapPI_3_200_1000_1”，物品数量从 1 变化到 130，步长为 1，背包容量 1008 保持不变，统计三种算法解决问题的时间，每个数据点统计 5 次求平均值，如图 12 所示。

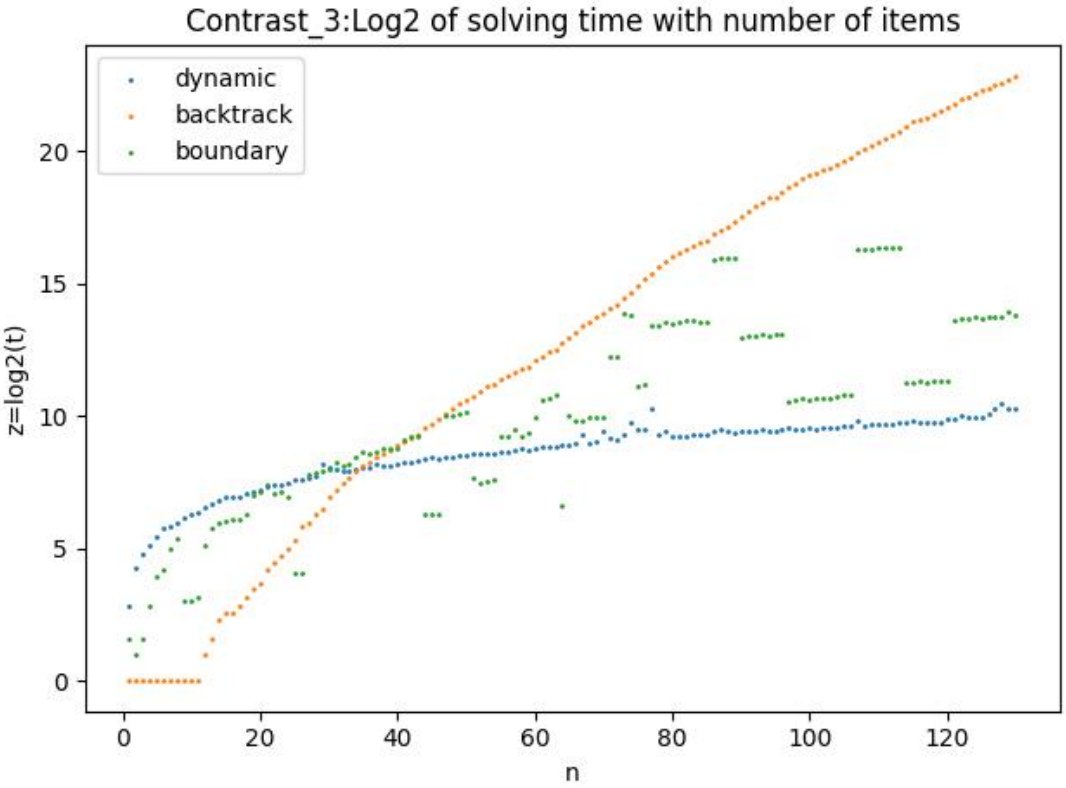

图 12

由图 12 可知，与弱相关数据相比，在强相关数据下,动态规划算法和回溯法表现并无太大差异，但分支限界算法表现出了更高的时间复杂性以及更大的抖动性。数据规模较小时，回溯法性能最优；数据规模较大时，动态规划算法性能已经比分支限界算法更优了。

4 实验结论

(1) 对动态规划算法而言，背包容量一定，算法的时间复杂性与可选物品数量成正比；可选物品个数一定，算法的时间复杂性与背包容量成正比。动态规划算法解决 0/1 背包问题的时间复杂度为 O(n*c),其中 n 为可选物品的个数，c 为背包容量大小。

(2) 对回溯法而言，最坏情况下，算法的时间复杂性随可选物品呈指数 2^n 增长。又由于背包容量的限制和剪枝，其实际时间往往低于 2^n，但依旧在 2^n 的数量级，时间复杂度为 $O(2^n)$。

(3) 对于分支限界算法而言，实验验证了限界剪支能够大大降低算法的求解时间，

甚至可以比时间复杂度为 O(n*c)的动态规划算法更优。

(4) 动态规划算法和回溯算法的时间性能对实验数据的相关性不敏感，分支限界算法则随着数据相关性的增强，时间复杂性增大，并且表现出无规律性。

(5) 不同数据规模和条件下最优算法的选择不同。对于小规模的数据，回溯法表现出更好的时间复杂性。对于大规模数据，当数据不具有或具有较弱的相关性时，分支限界法往往具有更好的时间性能；当数据具有较强相关性时，动态规划算法具有更好的时间性能。

5 进一步工作的展望

本文通过实验发现，分支限界法随着数据相关性的增强，其时间复杂度明显增大，性能下降，并且其时间曲线出现了较大抖动。对于出现这种情况的原因和原理，仍需进一步的理论分析和实验验证。