动态规划求解 0-1 背包问题

U201715825　管实-江诗毅

2019 年 4 月 17 日

目录

摘要

首先根据 0-1 背包问题的描述给出了该问题的两种动态规划方程式 (修正了老师给出的表达式)，然后使用 matlab 实现了上述两种算法，分别验证了其时间复杂度，并且对比了两种方案的差异之处。

然后，根据一种启发式贪心算法，描述了解决问题的思路，使用 matlab 实现，并分析了其优劣。

最后，总结了实验收获。

关键词： 动态规划

1 动态规划一

1.1 方程式

其他参数不变，原方程忽略了不加入物品的情况，故将动态规划方程式变为：

$$g(x) = \max_{i:w_i \leq x} \{v_i + g(x - w_i), g(x-1)\}$$

1.2 给出用法及验证时间复杂度

确定参数及用法

```
%knapsack 使用动态规划求解0-1背包问题
%
%输入：
%    v(vector) : 每个物品的价值
%    w(vector) : 每个物品的重量
%    x(vector) : 背包的容量
%
%输出：
%    plan(vector) : 逻辑1或者逻辑0向量， 表示是否选择该物品
%    opt(number) : 最优的物品价值
```

实现思路会在下一小节谈到，这里先验证使用该动态规划方程得出的算法的时间复杂度为 $O(nx)$, 其中 n 为物品的个数，x 为背包的容量大小。

绘制时间随规模–nx 衡量的散点图，并用一次函数拟合，结果如下图 (图 1, 图 2).

由图可知，该算法的时间复杂度能较好的使用 $O(nx)$ 来描述，下一小节会解释原因，以及其内存的耗费 (空间复杂度).

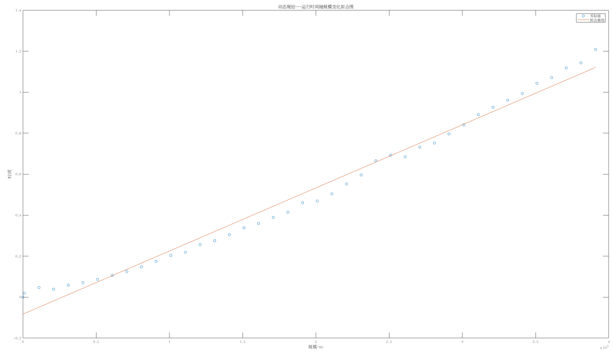

图 1: 稀疏点拟合

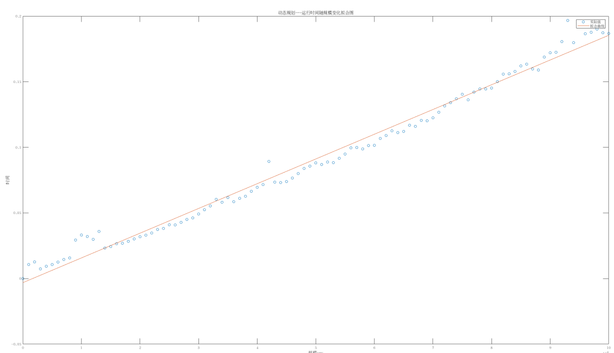

图 2: 密集点拟合

1.3 实现思路

实现思路：将求 g(x) 的最大值转换成求一个个小问题，背包容量从 0 到 x 逐步求出当前容量下的最优值 (并且将子问题的最优选品方案存储起来)，这样逐一迭代就能给出 g(x) 的最优值及其选品方案。

步骤：

- 预分配内存: 一个 $1\times(x+1)$ 一维数组 resultArr 用来存储子问题的最优值, 一个 $(x+1)\times n$ 二维数组 s 用来存储每个子问题的选品方案。
- 从 0 迭代到 x，逐一选择比当前背包容量小的物品放入 (或者不选任何物品)。将该物品的价值加上当前容量减去该物品重量这个子问题的

最优值。

- 去掉重选，如果选择了一物品，并且该物品在子问题进行了选择，那么就不考虑该种方案。
- 从上述方案中选择价值最大的方案，这就是当前背包容量下的最优选品方案。
- 这样，最优值 resultArr(x), 最优的决策方案就是 s(x,:)。

伪代码

```
预分配内存resultArr,s
for i=1 to x
    resultArr(i+1) = resultArr(i);  % 这就是不选物品的最优值。
    s(i+1,:) = s(i,:);  % 更改最优方案。

    % 找到所有重量比x小并且前面没有用到的物品的索引
    for j = 重量比i小的物品索引
        if 物品在前面用过
        continue;
    end

    % 比较选出价值最大的方案。
    if 当前的方案比之前的优
        % 更新方案
        resultArr(i+1) = v(j) + resultArr(i-w(j)+1);
        s(i+1,:) = s(i-w(j)+1,:);
        s(i+1,j) = 1;
```

时间空间复杂度分析

用伪代码的迭代语句中能够看出，首先是对 1 到 x 的背包容量进行了迭代，然后在每一种背包容量下对每一个物品进行了查找。共有 n 个物品，故算法的时间复杂度为 $O(n)$。

内存开销，用到了一个 $1{\times}(x+1)$ 一维数组，一个 $(x+1){\times}n$ 二维数组，故空间复杂度为 $(x+1)(n+1)$

1.4 测试用例

为了较好的验证算法的正确性，我使用了 DP1 与 DP3(任务 3 中的动态规划) 做对比，(见 test.m)，发现它们给出的解答在少数情况下是不相同的。

例如，当

重量向量为 [24 41 22 32 33 43 21 18 5 3 16 28 24 4 41 1 28 41 22 34]

价值向量为 [4 29 19 27 26 7 32 17 14 1 26 15 11 37 32 17 20 6 27 27]

DP1 得到的解为 305，DP3 得到的最优解为 306, 也就是说其中一个 DP 存在问题。下一节阐述 DP1 的问题所在。

1.5 由测试用例发现的问题

前面说到，DP1 得到的最优解有问题，于是重新 check 代码，发现这个 DP 方程对 0-1 背包问题来说有一个问题，那就是在后面选择物品的时候，可能该物品已经在子问题中选择过了，而我采取的办法是直接将这种方案从整个可行解集中去掉。这样就带来错误。正确的做法是将这种重复选择的物品的重量从容量中减去，并且在子问题中也不考虑这种物品。这样就不符合 DP 方程的定义了。

于是更改 0-1 背包问题为 0-n 背包问题，也就是说每个物品可以选择多个，而不是一个，同样最优方案用一维数组表示，第 i 个位置的值表示选择了该物品几次。

更改后的代码

```
function [plan,opt] = correctKnapsack(v,w,x)
%correctKnapsack    纠正1中的0-1背包问题解法，改为物品个数无限。
%
%输入：
%   v(vector) : 每个物品的价值
%   w(vector) : 每个物品的重量
%   x(vector) : 背包的容量
%
%输出：
%   plan(vector) : 表示第i个物品选了几个
%   opt(number) : 最优的物品价值
```

```
    leng = length(v);
    resultArr = zeros(1,x+1);  % g(x)的结果
    s = zeros(x+1,leng);  % 路线

    for i = 1:x
        % 赋一个初值
        resultArr(i+1) = resultArr(i);
        s(i+1,:) = s(i,:);
        for j = find(w<=i)  % 找到所有重量比x小的索引
            if v(j) + resultArr(i-w(j)+1) > resultArr(i+1)
                resultArr(i+1) = v(j) + resultArr(i-w(j)+1);
                s(i+1,:) = s(i-w(j)+1,:);
                s(i+1,j) = s(i+1,j) + 1;
            end
        end
    end
    plan = s(i+1,:);
    opt = resultArr(x+1);
end
```

同样针对上面的例子：

重量向量为 [24 41 22 32 33 43 21 18 5 3 16 28 24 4 41 1 28 41 22 34]

价值向量为 [4 29 19 27 26 7 32 17 14 1 26 15 11 37 32 17 20 6 27 27]

对于容量 25，得出的结果为：[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 0 0 0 0]，很容易验证这是正确的。

2 动态规划二

2.1 方程式

针对另一种动态规划解法，相当于从两个维度进行分割成子问题。一个维度是物品的种类，从只有一个物品到包含所有物品。另一个维度是从背包的容量，从 0 容量到 x 容量。

$$G(m,x)=\max\{G(m-1,x),G(m-1,x-w_m)+v_m\}$$

该动态规划方程的含义简单的理解，就是对于新增加的第 m 个物品，我可以选或者不选。

2.2 给出用法及验证时间复杂度

用法以及验证过程同上，不再赘述，只给出图示。

由图 3，图 4 大致能看出这种动态规划算法时间复杂度也符合 $O(nx)$，并且耗时少于第一种动态规划。原因后再后面进行论述。

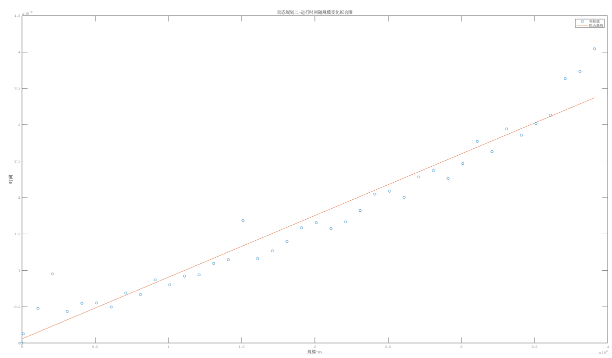

图 3: 稀疏点拟合

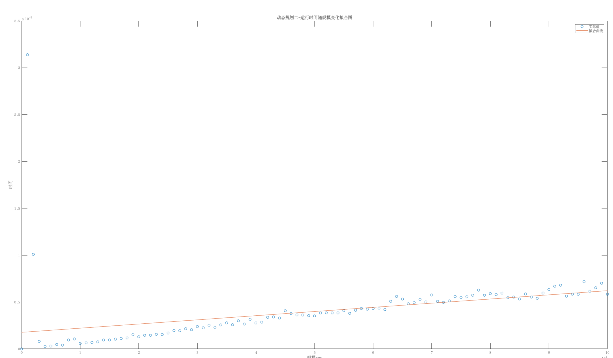

图 4: 密集点拟合

2.3 实现思路

步骤:

- 预分配内存:使用一个 n×(x+1) 二维数组 temp 来存储所有子问题的最优值,以及一个 n×(x+1) 的 0-1 二维数组 s 来存储当前最优方案下是否选择了新增的物品。
- 从只有一个物品开始迭代,同时背包容量也从 0 开始迭代直到 x 为止。
- 在当前物品集合下,当前背包容量下做出是否选择当前物品的决策。并且记录下来
- 最后的 temp(n,x+1) 就是最优值,决策方案根据 s 给出,具体步骤如下。

根据 s 给出决策方案步骤:

- 从 s(n,x+1) 开始,如果为 1,证明选择了第 n 个物品。如果为 0,证明没选。
- 如果上一步为 1,那么跳到 s(n-1,x+1-v(n)) ,看它为 1 还是 0。如果上一步为 0,跳到 s(n-1,x+1) 进行判断。
- 重复上述步骤,直到全部判断完毕,这样就得出了最优方案。

代码见 knapsack.m 附件。

2.4 测试用例

思路:寻找已经得到证实的背包问题解,验证自己的算法。
将 test.m 附件。

3 两种动态规划的对比

从实现思路上来说,第一种只对背包的容量进行了“动态规划”,而第二种在对背包容量进行动态规划的同时,也对物品的种类进行了划分。所以针对第一种,我使用了一个一维数组存储最优值,对于第二种,使用了二维的数组。在求解决策方案上,对于第一种,直接将每个子问题的选品方案存

在一个二维数组里面，对于第二种，我是从物品的角度来看，如果选择了当前迭代的物品，那么存 1, 否则存 0.

空间复杂度：第一种使用一个一维数组，一个二维数组。第二种动态规划使用两个二维数组。故易知，第二种方案在空间上开销优于第一种。

时间复杂度：两种方案都近似于 $O(nx)$。但在循环内部，第一种每一次都要查找所有物品，而第二种只用决策是否选择当前物品。故总的来说，第二种动态规划在时间上消耗比第一种少很多。由前面给出了图也看出，第一种的数量级为 1，第二种的数量级为 10^{-4}

下面给出图示验证。

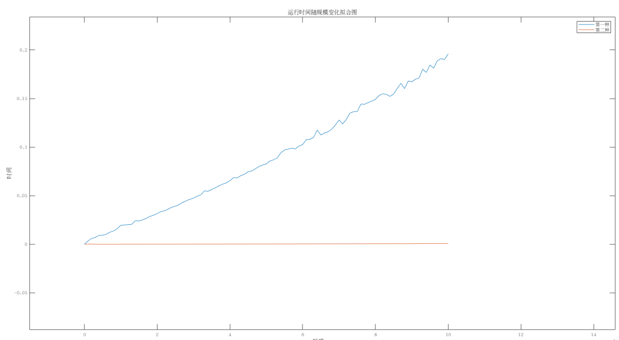

图 5: 对比

4 启发式: 贪婪算法

4.1 问题描述

针对背包问题，一种很自然的想法就是先选择性价比最高的物品，也就是 v/w 最高的。但是这只是在当前看来是最优的选着，但是对于长远来说，可能存在浪费背包容量的问题，所以这只是一种近似的解法。

4.2 实现思路

步骤：

- 将物品按找 v/w 从大到小排序。

- 选择当前背包剩余容量能装下的最大性价比的物品。
- 重复第二步，直到背包再也装不下任何物品。

4.3 算例分析

按照问题描述，由于所有的物品都只扫描了一次，所以算法的时间复杂度为 $O(n)$。

启发的式的贪婪算法虽然得不到问题的最优解，但是其在问题规模较大的时候，能得到比较理想的结果，并且其时间复杂度相比与动态规划大幅降低了。

如下图：

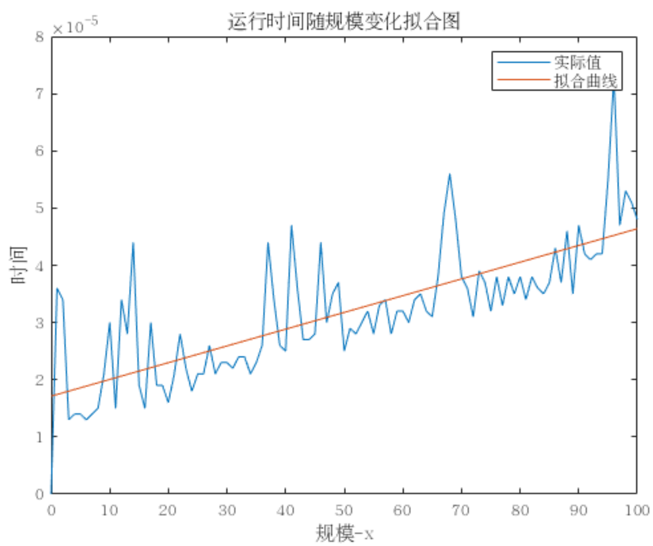

图 6: 对比

5 总结

二维的动态规划形式不一定劣于一维的动态规划形式，反而因为它在迭代过程中将大量子问题存储起来，这样在后续计算中能减少很多次的比较，故效率高于一维的动态规划形式。

启发的式的贪婪算法虽然得不到问题的最优解，但是其在问题规模较大的时候，能得到比较理想的结果，并且其时间复杂度相比与动态规划大幅降低了。