MATLAB: 背包问题

郭龙昕，江诗毅，胡进

2019 年 4 月 21 日

任务一

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,:)。

具体实现

下面分别给出伪代码和具体的代码

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。

    % 找到所有重量比小并且前面没有用到的物品的索引x
     for  j = 重量比小的物品索引i
          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;
```

程序:*knapsack.m*

```
function  [plan,opt] = knapsack(v,w,x)
%knapsack  浣跨敤鍔ㄦ€佽鍒掓眰瑙ｂ B 鑳屽寘闂     0-1
%
%杈撳叆鍙傛暟%    v(vector) : 姣忎釜鐗╁搧鐨勪环鍊%    w(vector) : 姣忎釜鐗╁搧鐨勯噸閲%    x(vector) : 鑳屽寘鐨
    勬   閲%
%杈撳嚭鍙傛暟%    plan(vector) : 闀垮害绛変簬闀垮害鐗╁搧鏁伴噺鐨勫悜閲〃绀烘槸鍚︽斁鍏   鐗╁搧10
%    opt(number) : 鏈€澶х殑浠峰€
leng = length(v);
resultArr = zeros(1,x+1);   % g(x)鐨勬粨鏋s = zeros(x+1,leng);   % 璺   嚎

for  i = 1:x
     % 璧嬪湪涓   垎鍊
     resultArr(i+1) = resultArr(i);
```

```
    s(i+1,:) = s(i,:);

    % 涓釜瀛楁數绗  笉澶х殑鑻扮粍
    %tem = zeros(1,length(v));
    for j = find(w<=i)   % 鎵惧埌鎵滄閲嶉噺蹇呴』灏忎簬瀹归噺鐨勭墿鍝佸唴閮ㄥ惊鐜?
        if ismember(j,find(s(i-w(j)+1,:) == 1) )
            continue;
        end
        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) = 1;
        end
    end
end


plan = s(i+1,:);
opt = resultArr(x+1);

end
```

正确性测试

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。

我们发现是任务一的代码存在问题：

改进方法

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 纠正中的背包问题解法，改为物品个数无限。    10-1
%
%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) : 表示第个物品选了几个i
%    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)  % 找到所有重量比小的索引x
            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
```

改进后的代码正确性测试

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。

算法效率分析

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。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$(x+1)(n+1)$

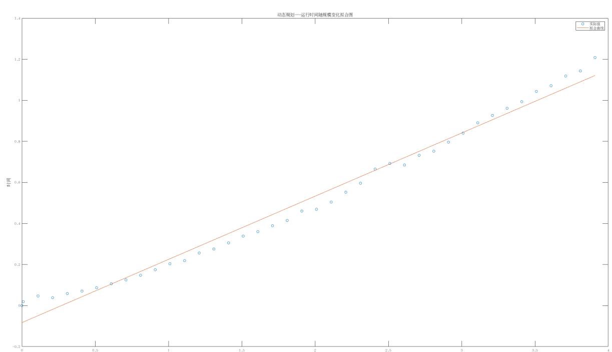

图 1: 稀疏点拟合

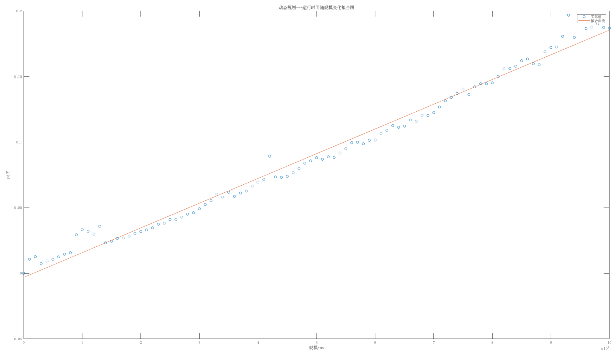

图 2: 密集点拟合

任务二

贪婪算法

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。

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。

具体程序

```
function [plan,opt] = greedy(v,w,x)
%greedy 涜跨敤璐 ┆绠楁硶姹傝В鑳屽寘闂    鐨勮繎浼艰В鍒0-1%
%杈撳叆锛%    v(vector) : 姣忎釜鐗╁搧鐨勫环鍊%    w(vector) : 姣忎釜鐗╁搧鐨勯噸閲%    x(vector) : 鑳屽寘鐨
    勫  閲%
%杈撳嚭锛%    plan(vector) : 闀旳總鎴栧闀旳總鍚戦噺锛岃〃绀烘槸鍚︽斁鍏ヨ  鐗╁搧10
%    opt(number) : 鏈€浼樿В鐨勪环鍊unit = v./w;
plan = zeros(1,length(w));
opt = 0;

i = find(unit==max(unit));
i = i(1);

while x >= w(i)
    plan(i) = 1;
    opt = opt + v(i);
    x = x - w(i);
    unit(i) = 0;
    i = find(unit==max(unit));
    i = i(1);
end
```

```
end
```

算法效率分析

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：

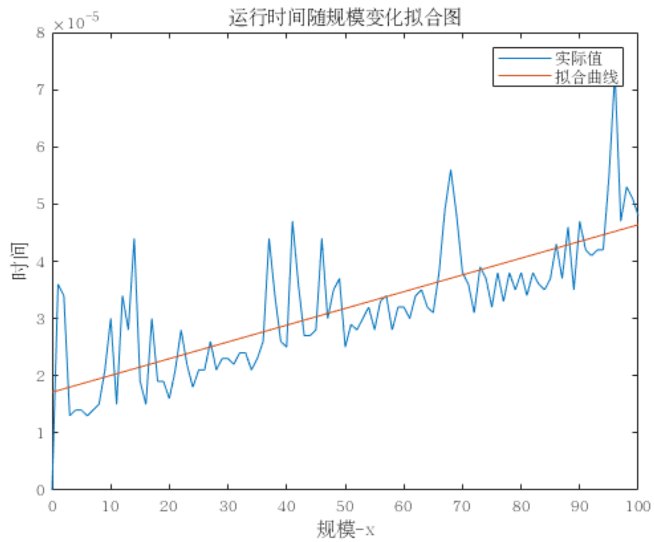

图 3: 对比

任务三

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。

具体程序

程序：*knapsack.m*

```
function [plan,opt] = knapsack(v,w,x)
%knapsack 浣跨敤鍔ㄦ€佽鍒掓硶姹傝В 0-1
%
%权搬叆锛%    v(vector) : 姣忎釜鐗╁搧鐨勪环鍊%    w(vector) : 姣忎釜鐗╁搧鐨勯噸閲%    x(vector) : 鑳屽寘鐨
    勫 閲%
%权搬嚭锛%    plan(vector) : 闃舵緫鎯呴樁鏂囨嚭锛岃〃绀烘槸鍚︽斁鍏ヨ鐗╁搧10
%    opt(number) : 鏈€鐖剳墿鍝佷环鍊
leng = length(v);
resultArr = zeros(1,x+1);  % g(x)鐨勭粨鏋s = zeros(x+1,leng);  % 璺 嚎

for i = 1:x
    % 璧嬪鍒濆 垵鍊
    resultArr(i+1) = resultArr(i);
    s(i+1,:) = s(i,:);

    % 涓釜瀛樻斁绗  簩澶x 殑鏁扮粍
    %tem = zeros(1,length(v));
    for j = find(w<=i)  % 鎵惧埌鎵€鏈夊彲浠ユ斁杩涘幓鐨勭墿鍝侊紝鍒嗗埆鍋氬嚭鍒ゆ柇
        if ismember(j,find(s(i-w(j)+1,:) == 1) )
            continue;
        end
        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) = 1;
        end
    end
end


plan = s(i+1,:);
opt = resultArr(x+1);

end
```

正确性测试

在任务一中已经证明，这里不再赘述

算法效率分析

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。

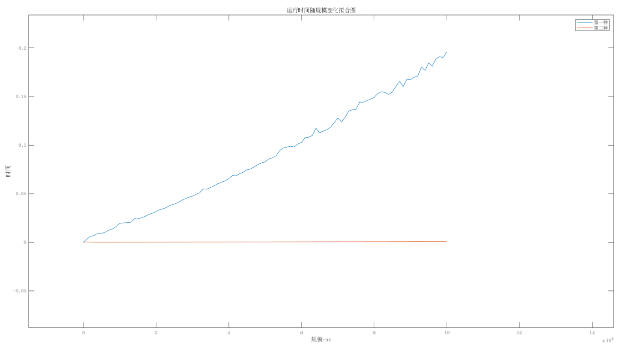
运行时间随规模变化拟合图
第一种
第二种
0.2
0.15
0.1
0.05
0
-0.05
时间
0
2
4
6
8
10
12
14
×10^4
规模-nx

图 4: 对比