实验十一 从贪心到模拟退火解决0-1背包问题

摘 要

本文从0-1背包问题入手，分别使用了贪心算法、深度优先搜索、动态规划、模拟退火等算法，从时间复杂度以及空间复杂度的角度比较了不同算法的优劣。

关键词：深度优先搜索搜索 动态规划 模拟退火

# 问题重述

给定n种物品和一背包。物品 i 的重量似乎 wi，其价值为 vi，背包的容量为 c。问应该如何选择装入背包中的物品，使得装入背包中物品的总价值最大？

题目中给定物体个数为5，背包容量为10，重量价值对应表如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 重量 | 2 | 3 | 5 | 1 | 4 |
| 价值 | 2 | 5 | 8 | 3 | 6 |

# 模型的建立与求解

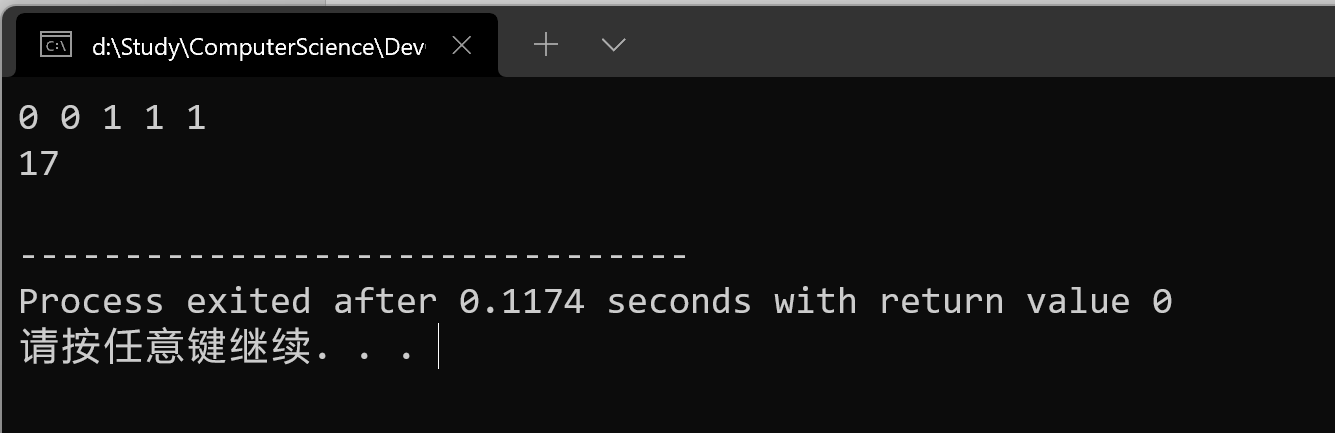
## 贪心算法

贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解。对于最优化问题最容易想到的算法就是也就是贪心算法，在局部寻求最优解后分步寻找全局最优解。

下面用贪心算法给出0-1背包问题的求解，初始化参数之后按如下算法解决问题，首先设置解向量为全0，一个变量存当前价值，一个变量存当前重量。循环如下操作：找到第i个物体，看看目前的背包有没有满，如果没有满的话那就把东西拿上，自然而然这时最优解。如果目前背包已经放满了东西，那么就要从背包里看看，是不是有东西的价值比这个新的东西低且交换物品的空间足够呢，如果满足条件，那就交换物体。如此循环自然得到最优解。实际上我这个算法是对普通的贪心算法做出了改进，实际贪心算法的时间复杂度应该为O(n)，而改进之后的算法时间复杂度为O(n^2),具体实现代码如下所示：

|  |
| --- |
| 代码 |
| 介绍：C语言 贪心算法求解0-1背包问题 |
| #include<iostream>  using namespace std;  int main(){  int c=10,n=5;  int solution[5]={0,0,0,0,0};  int nowvalue=0,nowweight=0;  int w[5]={2,3,5,1,4};  int v[5]={2,5,8,3,6};  for(int i=0;i<5;i++){  if(c-nowweight>=w[i]){  solution[i]=1;  nowvalue+=v[i];  nowweight+=w[i];  }  else{  for(int j=0;j<5;j++){  if(solution[j]==1){  if(c-(nowweight-w[j])>=w[i])  if(v[i]>v[j]){  solution[i]=1;  solution[j]=0;  nowvalue-=v[j];  nowweight-=w[j];  nowvalue+=v[i];  nowweight+=w[i];  }  }  }  }  }  for(int i=0;i<5;i++)cout<<solution[i]<<" ";  cout<<endl<<nowvalue<<endl;  } |

运行代码之后得到如下结果：



解决方案为[0,0,1,1,1]

众所周知贪心算法不能解决0-1背包问题，而题目地结果却是以外的正确，可能是题目设置的较为简单，使得一般的贪心算法也能完成问题的求解，实际上贪心算法并不能解决该问题。

## 搜索

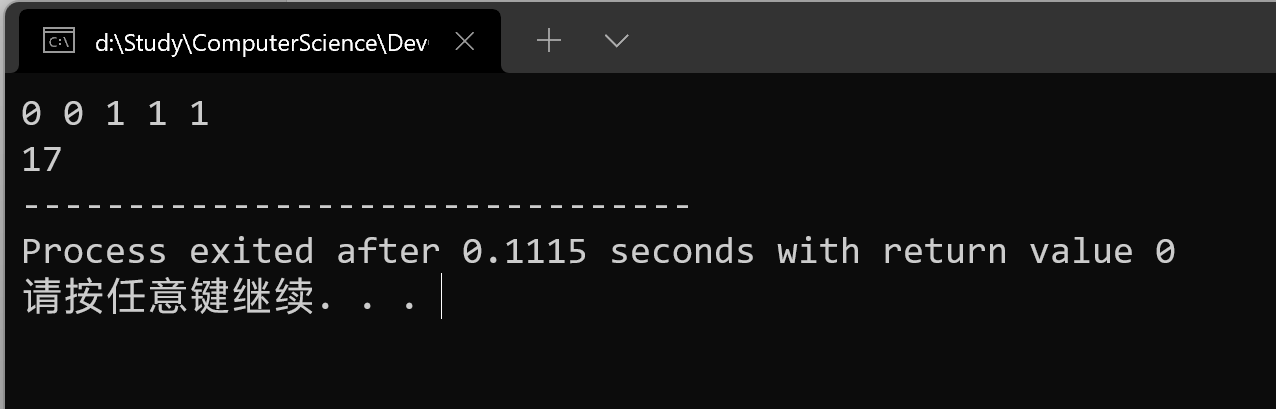
因为一般的贪心算法不能解决0-1背包问题，自然就会想到一种时间复杂度更高的算法，就是搜索算法。

搜索算法是利用计算机的高性能来有目的的穷举一个问题解空间的部分或所有的可能情况，从而求出问题的解的一种方法。现阶段一般有枚举算法、深度优先搜索、广度优先搜索、A\*算法、回溯算法、蒙特卡洛树搜索、散列函数等算法。在大规模实验环境中，通常通过在搜索前，根据条件降低搜索规模；根据问题的约束条件进行剪枝；利用搜索过程中的中间解，避免重复计算这几种方法进行优化。

以下使用最普通的深度优先搜索求解问题，实际搜索就是对情况地一种遍历，对每一种可能性进行计算代码如下：

|  |
| --- |
| 代码 |
| 介绍：C语言 深度优先搜索求解0-1背包问题 |
| #include<iostream>  using namespace std;  int main(){  int c=10,n=5;  int solution[5]={0,0,0,0,0};  int nowvalue=0,nowweight=0;  int w[5]={2,3,5,1,4};  int v[5]={2,5,8,3,6};  for(int i=0;i<5;i++){  if(c-nowweight>=w[i]){  solution[i]=1;  nowvalue+=v[i];  nowweight+=w[i];  }  else{  for(int j=0;j<5;j++){  if(solution[j]==1){  if(c-(nowweight-w[j])>=w[i])  if(v[i]>v[j]){  solution[i]=1;  solution[j]=0;  nowvalue-=v[j];  nowweight-=w[j];  nowvalue+=v[i];  nowweight+=w[i];  }  }  }  }  }  for(int i=0;i<5;i++)cout<<solution[i]<<" ";  cout<<endl<<nowvalue<<endl;  }#include <iostream>  #include <cmath>  using namespace std;  int c = 10, n = 5;  int solution[5] = {0, 0, 0, 0, 0};  int ss[5]={};  int nowvalue = 0, nowweight = 0;  int w[5] = {2, 3, 5, 1, 4};  int v[5] = {2, 5, 8, 3, 6};  int count\_value(int s[]){  int value=0;  for (int i=0; i<5;i++){  value+=s[i]\*v[i];  }  return value;  }  int count\_weight(int s[]){  int weight=0;  for (int i=0; i<5;i++){  weight+=s[i]\*w[i];  }  return weight;  }  void f(int n,int s[])  {  if (n == 5)  if(count\_weight(s)<=c)  { if(nowvalue<count\_value(s)){  for(int i=0;i<5;i++)ss[i]=solution[i];  nowvalue=count\_value(s);  }  return;}  else return;  s[n]=0;  f(n+1,s);  s[n]=1;  f(n+1,s);  }  int main()  {  f(0,solution);  for(int i=0; i<5;i++){  cout<<ss[i]<<" ";  }  cout<<endl<<nowvalue;  } |

得到结果：



不出所料得到了最优结果，但这种算法地复杂度也非常高，实际上这个问题的规模不大还并不能看出该搜索地巨大复杂度O(2^n),当随着问题规模的变大，这种时间消耗将变得难以忍受。

## 动态规划

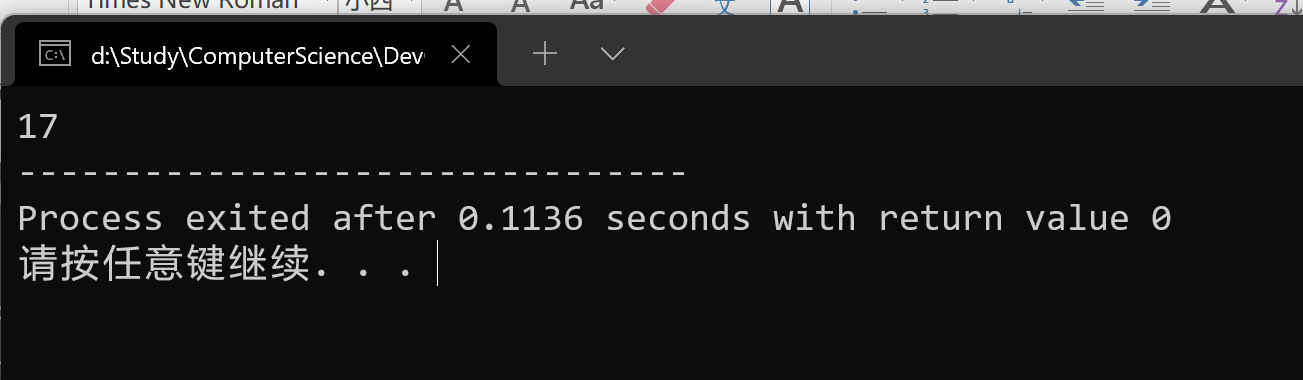
动态规划方法是解决背包问题效率最高的算法之一，具体构建的方法不再展开赘述，对于0-1背包问题，可以构间如下的转移方程：

dp[i][j] = max(dp[i - 1][m], dp[i - 1][m - w[i]] + v[i]);

代码部分如下：

|  |
| --- |
| 代码2 |
| 介绍：C语言 动态规划求解0-1背包问题 |
| #include <iostream>  #include <cmath>  using namespace std;  int c=10;  int w[5] = {2, 3, 5, 1, 4};  int v[5] = {2, 5, 8, 3, 6};  int m[6][11];  int main(){  for(int i=1;i<=5;i++){  for(int j=10;j>=0;j--){  if(j-w[i-1]>=0)  m[i][j]=max(m[i-1][j],m[i-1][j-w[i-1]]+v[i-1]);  else m[i][j]=m[i-1][j];  }  }  cout<<m[5][10];  } |

运行得到结果如下：



## 模拟退火

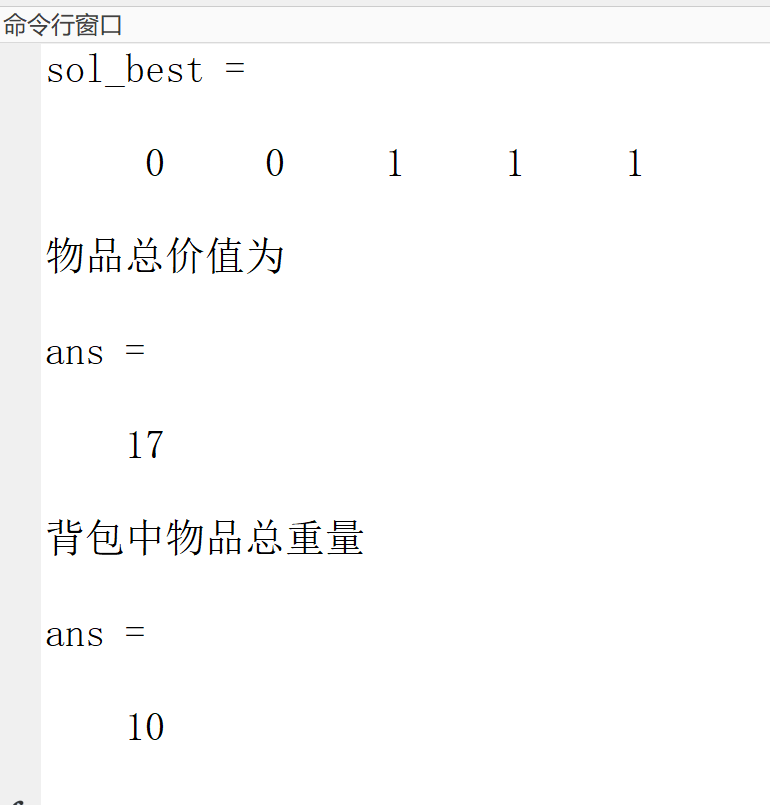
模拟退火算法是通过赋予搜索过程一种时变且最终趋于零的概率突跳性，从而可有效避免陷入局部极小并最终趋于全局最优的串行结构的优化算法。

首先设定好参数，包括物品价值、背包重量等参数，设定好初始温度后进入循环，产生随机扰动，根据公式判断是否接受新解，随着温度的降低最终得到答案。

代码部分如下：

|  |
| --- |
| 代码 |
| 介绍：MATLAB 模拟退火求解0-1背包问题 |
| clc  clear all  close all  a=0.95;%温度衰减速度  mar\_length=1000;%马氏链长度  d=[2 3 5 1 4];%物品价值  k=[2 5 8 3 6];%物品重量  restriction=10;%背包能够承受的最大重量  num=length(k);%物品数量  sol\_new=round(rand(1,num));%随机生成初始解  E\_current=inf;E\_best=inf;%E\_current是当前解对应的目标函数，E\_best是最优解，E\_new是新解的目标函数值  t0=97;tf=3;t=t0;  while t>tf  for i=1:mar\_length  %产生随机扰动  temp1=ceil(rand\*num);  sol\_new(1,temp1)=~sol\_new(1,temp1);  %检查是否满足约束  while (1)  s=(sol\_new\*d'>restriction);  if s  %如果不满足约束随机放弃一个物品  temp2=find(sol\_new==1);  temp3=ceil(rand\*length(temp2));  sol\_new(temp2(temp3))=~sol\_new(temp2(temp3));  else  break  end  end  %计算背包中物品的价值模拟退火算法只能求最小值，所以价值取负  E\_new=sol\_new\*(-k');  if E\_new<E\_current  E\_current=E\_new;  sol\_current=sol\_new;  if E\_new<E\_best  E\_best=E\_new;  sol\_best=sol\_new;  end  else  if (rand<exp(-(E\_new-E\_current)./t))  E\_current=E\_new;  sol\_current=sol\_new;  else  sol\_new=sol\_current;  end  end  end  t=t\*a;  end  disp('最优解为')  sol\_best  disp('物品总价值为')  -E\_best  disp('背包中物品总重量')  sol\_best\*d' |

运行结果如下：



# 模型的分析与检验

|  |  |
| --- | --- |
| **算法** | **时间复杂度** |
| 搜索 | O(2^n) |
| 动态规划 | O(n\*m) |
| 模拟退火 | O(n\*m\*a) |

从时间复杂度来看动态规划是解决0-1背包问题的最优算法，而模拟退火算法相对来说复杂度要高不少。但是不同算法都有不同应用场景，模拟退火问题在求解某些无法直接求解的问题上可以很好的求解出答案。