****

算法设计结课论文

01背包问题的求解性能比较

学生：高铭泽，袁树宽 | 指导老师：王征 | 日期：2020.12.25

实验目的

掌握利用算法进行问题求解的实验流程

强化对课堂所学算法原理的理解

提升利用算法原理进行编码实践的能力

了解算法研究的前沿动态，拓展算法改进的思路

实验课题

2.3算法求解类问题4.1对于01背包问题利用动态规划，回溯和分枝限界三种算法进行求解并分析不同算法解决同一问题的复杂度，归纳不同求解算法的优缺点。

研究问题描述

01背包是在M件物品取出若干件放在空间为W的背包里，每件物品的体积为W1，W2至Wn，与之相对应的价值为P1,P2至Pn。01背包是背包问题中最简单的问题。01背包的约束条件是给定几种物品，每种物品有且只有一个，并且有权值和体积两个属性。在01背包问题中，因为每种物品只有一个，对于每个物品只需要考虑选与不选两种情况。如果不选择将其放入背包中，则不需要处理。如果选择将其放入背包中，由于不清楚之前放入的物品占据了多大的空间，需要枚举将这个物品放入背包后可能占据背包空间的所有情况。

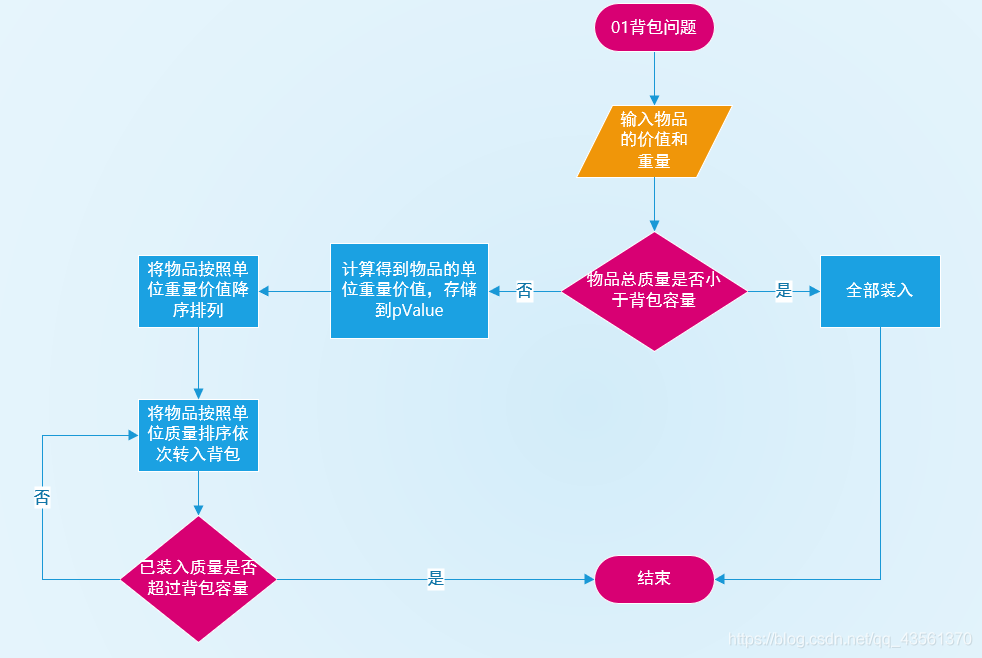
实验方法介绍

本实验针对0/1背包问题，利用动态规划，回溯，分支限界三种算法，对Florida State University: 8 个数据集，Unicauca University: 31 个数据集进行性能分析。实验表明​ 一般来说动态规划算法只要设计得当，效率往往是很高的，在时间上溢出的可能性不大，但是由于动态规划需要记录大量的中间过程，对内存空间的需求较大，设计不合理很有可能导致内存的溢出，回溯法采用深度优先方法搜索解空间，逐步生成解，在当前解不满足条件时会回退一步去求解，对解空间进行了限界对于结果是否符合条件进行预判，降低时间成本。分支限界法一般采用广度优先或以最小消耗优先的方式搜索解空间树，每一次活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有子结点；存储空间的要求不同，因此所需的存储空间比回溯法大得多。

动态规划代码设计流程

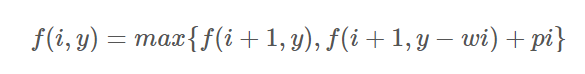
# 设计思路

动态规划，​是将原问题分解为一个个子问题，寻找子问题的最优解，构成员问题的最优解。这是一种自底向上求解算法。使用动态规划的前提是能够找到最优子结构。将最优的子结构存储起来，完成自下而上的递推求解。流程图如下：

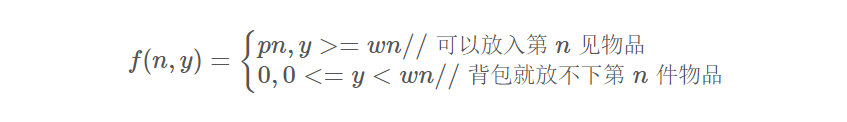


不选择贪心所用的密度值递减顺序放入物品，甚至在用k-优化优化算法提高可信度的方法，而是选择直接用递归枚举求解物品放还是不放的方法。即无论优化解是否放物品1，相对剩余背包容量,优化解对物品2,…,n的放法, 也是优化解。

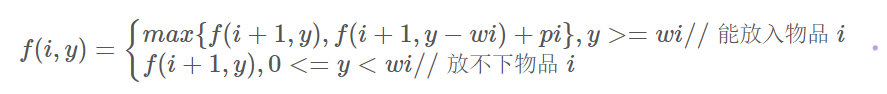
例如n=5,c=10,w=[2,2,6,5,4],p=[6,3,5,4,6]。其优化解为(1,1,0,0,1),即优化的物品装入背包的方法为物品1,2,5放入。物品1占背包容量2,剩下容量为8.优化解中包含的子问题转化为n=4,c’=c-2(物品1的重量),物品为2,3,4,5(1,0,0,1),即放物品2和5,是上述子问题的优化解，背包问题满足的优化原理。即每次我们都枚举物品是否放入来列递归式，设f(i,y)表示背包现有容量为y,放入物品i,..,n得到的优化效益值，则有：



当然，上述式子成立的条件是物品i假设可以放入，然后在讨论是否放入的问题，如果无法放入则，就直接抛弃，式子改写为f(i+1,y)，所以可以列出0/1背包dp思路模板如下：



上面的是前提已知条件，下面是递归模板

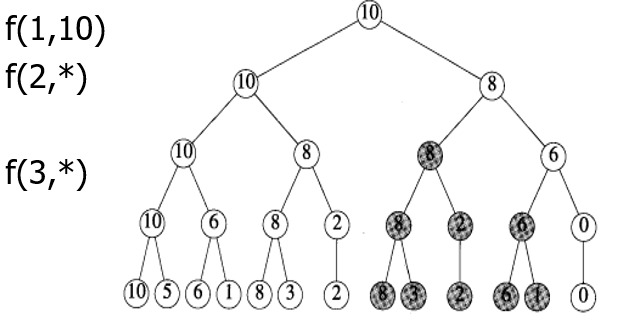


我有两种实现的思路不过总体原理不尽相同，都是以上述表达式为基础进一步实现的。一种是课堂上讲的递归算法，实现图如下：

# 二、伪代码

1. **int** F(**int** i,**int** y){
2. //返回f(i,y)
3. **if**(i==n){
4. **return** (y<w[n])?0:p[n];
5. }
6. **if**(y<w[i]){
7. **return** F(i+1,y);
8. }
9. **return** max(F(i+1,y),F(i+1,y-w[i]+p[i]));
10. }

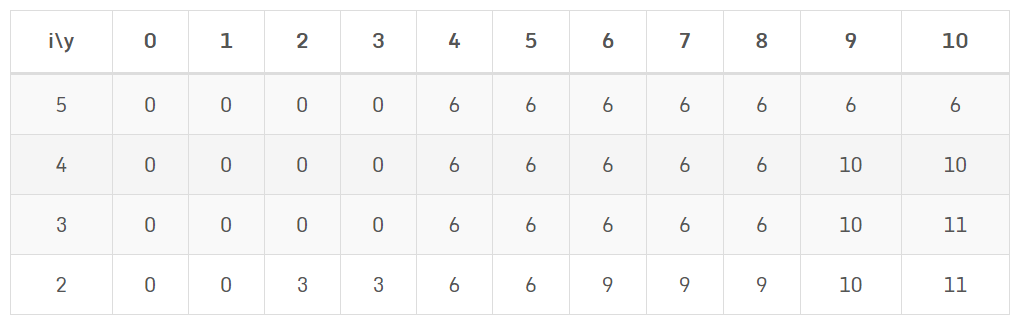
例如对于n=5,c=10,w=[2,2,6,5,4],p=[6,3,5,4,6]，首先我们先不想那么多，直接按照上面的递归表达式思路走一发。我们为了求解,调用F(1,10)求解f(1,10)。这里用递归树展开计算经过，其中每个节点用y值来标记，第j层的节点对应F(j,\*)，因此根节点表示F(1,10),而它有左、右子节点,分别对应F(2,10)和F(2,8)。



上图中节点标出的是背包剩余容量，为标出的则为后面无法在放物品了，我们会发现本次运行执行了28次递归调用，并且灰色部分是重复进行了计算，这部分就和dfs递归调用等递归求解问题共性缺点，重复计算消耗大量时间，为了减少没有必要的计算，我们可以暂存以前的计算结果，这样就只需要计算19次了，这也是dp算法总是要借助表格的原因。因此我们接下来给出另一种利用一维数组（二维数组也行）存储对于每件物品所选与否的当前最优价值，不断更新数组直到比较完所有物品，（与递归想法差不多）：

1. **template**<**class** T>
2. //这里的yMax为wi
3. **void** Knapsack(T p[],**int** c,**int** n,T\*\* f){
4. //对于所有i和y计算f[i][y]
5. //初始化f[n][]
6. **for**(**int** y=0;y<=yMax;y++){
7. f[n][y]=0;
8. }
9. **for**(**int** y=w[n];y<=c;y++){
10. //初始化第一行
11. f[n][y]=p[n]
12. }
13. //计算剩下的f
14. **for**(**int** i=n-1;i>1;i--){
15. **for**(**int** y=0;y<=yMax;y++){
16. //都默认为不装入这个物品时为最大值
17. f[i][y]=f[i+1][y]
18. }
19. **for**(**int** y=w[i],y<=c;y+=){
20. //比较装入和不装入哪个效益值更大取大值
21. f[i][y]=max(f[i+1][y],f[i+1][y-w[i]]+p[i]);
22. }
23. }
24. f[i][c]=f[2][c];
25. **if**(c>=w[i]){
26. f[i][c]=max(f[1][c],f[2][c-w[1]]+p[1]);
27. }
28. }
29. //对比分析，实际上元组法就是这个思路，只不过是用线性表优化了空间复杂度而已

那么对于上面的 使用二维数组打表最终得到的表的数值如下：

三、代码展示

（1）python语言实现：

1. n, c = map(int, input().split())
2. weight = [int(n) **for** n **in** input().split()]
3. value = [int(n) **for** n **in** input().split()]
5. #动态规划(非递归)
6. **def** backpack(n,c):
7. dp = [0 **for** i **in** range(c+1)]
8. **for** i **in** range(n):
9. **for** j **in** range(c,-1,-1): # 从后往前
10. **if** j >= weight[i]:
11. dp[j] = max(dp[j], dp[j-weight[i]] + value[i])
13. **print**(dp[-1])
14. #递归
15. **def** f(i,c):
16. **if**(i==(n-1)):
17. **if**(weight[i]>c):
18. **return** 0
19. **else**:
20. **return** value[i]
21. **if** weight[i]>c:
22. **return** f(i+1,c)
23. **return** max(f(i+1,c),f(i+1,c-weight[i])+value[i])

26. **if** \_\_name\_\_ == "\_\_main\_\_":
27. backpack(n,c)

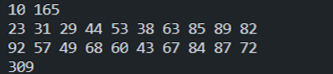
（2）C++语言实现

1. #include<iostream>
2. #include <time.h>
3. #include<windows.h>
4. **using** **namespace** std;
5. #define max(x,y) ((x)>(y)?(x):(y))
6. #define MAXN 25
7. #define MAXW 800
8. **int** n = 15;
9. **int** W = 750;
10. **int** w[ ]= {0,70,73,77,80,82,87,90,94,98,106,110,113,115,118,120};
11. **int** v[ ] = {0,135,139,149,150,156,163,173,184,192,201,210,214,221,229,240};
13. **int** dp[MAXN][MAXW];
14. **int** x[MAXN];
15. **int** maxv;
17. **void** Knap();
18. **void** Buildx();
20. **int** main()
21. {
22. LARGE\_INTEGER fre={0};//储存本机CPU时钟频率
23. LARGE\_INTEGER startCount={0};
24. LARGE\_INTEGER endCount = {0};
25. QueryPerformanceFrequency(&fre);
27. Knap();
28. Buildx();
29. printf("动态规划求解结果\n");
30. printf("选取的物品：");
31. **for**(**int** i =1;i<=n;i++)
32. {
33. **if**(x[i] == 1)
34. printf("%d ",i);
35. }
36. printf("\n");
37. printf("总共价值 = %d\n",maxv);
39. QueryPerformanceCounter(&startCount);
40. Sleep(1000);
41. //结束计时
42. QueryPerformanceCounter(&endCount);
43. //计算时间差
44. **double** dTimeTake = ((**double**)endCount.QuadPart - (**double**)startCount.QuadPart) / (**double**)fre.QuadPart;
45. printf("用时%f\n", dTimeTake);
46. **return** 0;
47. }
49. **void** Knap()
50. {
51. **int** i,r;
52. **for**( i =0;i<=n;i++)
53. dp[i][0] = 0;
54. **for**(r = 0;r<= W;r++)
55. dp[0][r] = 0;
56. **for**(i = 1;i<=n; i++)
57. {
58. **for**(r =1;r<=W;r++)
59. **if**(r < w[i])
60. dp[i][r] =dp[i-1][r];
61. **else**
62. dp[i][r]  = max(dp[i-1][r],dp[i-1][r-w[i]]+v[i]);
63. }
64. }
66. **void** Buildx()
67. {
68. **int**  i =n;
69. **int** r = W;
70. maxv = 0;
71. **while**(i>=0)
72. {
73. **if**(dp[i][r]!= dp[i-1][r])
74. {
75. x[i] = 1;
76. maxv += v[i];
77. r = r - w[i];
78. }
79. **else** x[i] = 0;
80. i--;
81. }
82. }

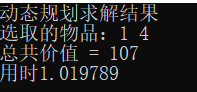
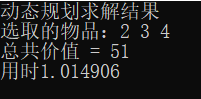
# 四、测试结果

所选数据集为Florida State University: 8 个数据集，Unicauca University: 31 个数据集进行性能分析（具体的输入样例请查看文件夹内01背包.txt），这里仅展示最终输出结果

1. python测试结果部分展示



1. c++测试结果部分展示



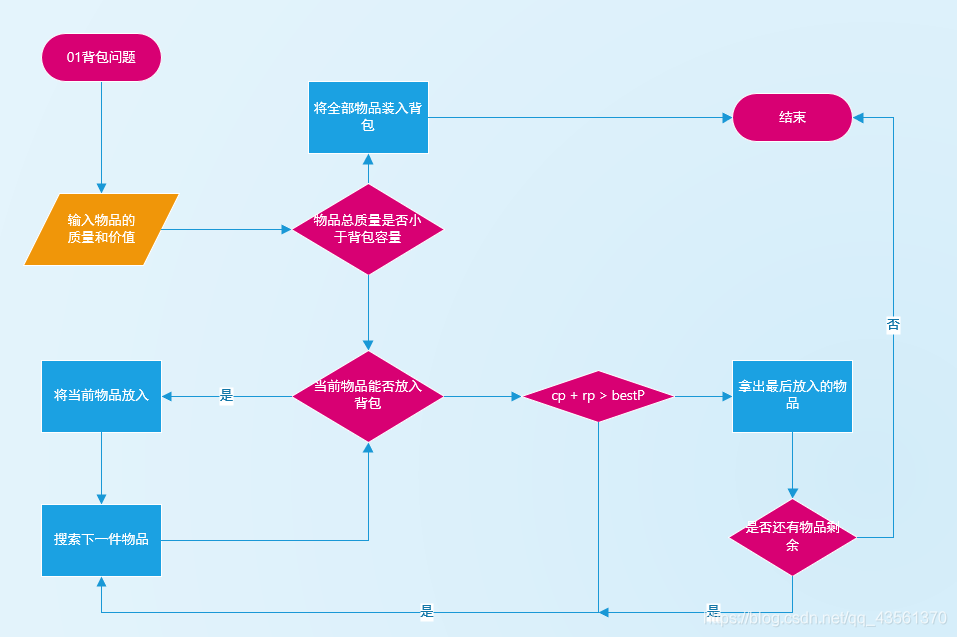
# 五、算法分析

在加入递归式求解并且建立表来记录中途计算过得数值以减少重复计算可以将时间复杂度控制在O(n^2)内，并且在使用二维数组后空间复杂度为O(n\*Wmax)。相较于暴力枚举搜索效率大幅提升，同时借用打表记录中途数值也可以在保证结果精度准确的同时加快搜索速度。

回溯法代码设计流程

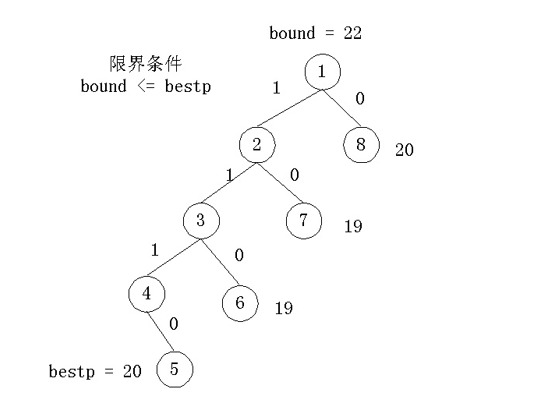
# 一、设计思路

回溯法，其基本思想就是对于每一个物品i，对于该物品只有选与不选2个决策，总共有n个物品，可以顺序依次考虑每个物品，这样就形成了一棵解空间树，遍历这棵树，以枚举所有情况。对于在该子集树中每个节点处进行选与不选的决策，如果剩余空间能容下当前物品则将其装入，并搜索左子树；如果不装入当前物品，且满足限界条件（剩余物品价值与当前价值的总和大于最优解）就搜索其右子树。此时我们也需要找到一个限界条件，此时我们先对背包的物品和效益值计算效益密度，按照效益密度重新排列，然后注意，此时我们这里要做的不是逐一尝试，而是将其看成连续背包问题来计算最大效益值，即即使装不下我们也要拿效益密度最大的进行填充来计算出最理想效益密度值bestp，而限界条件就是拿每次的最大贪心解bound来和bestp相比较，如果bound<=bestp,就不展开，一句话就是每次我都尽量多拿，如果大了我们在回退考虑不拿某个物体的情况，所以回溯虽然可以省略某些不必要的情况，但是我们还是计算出来了所有可能取得答案解的情况。流程图如下：

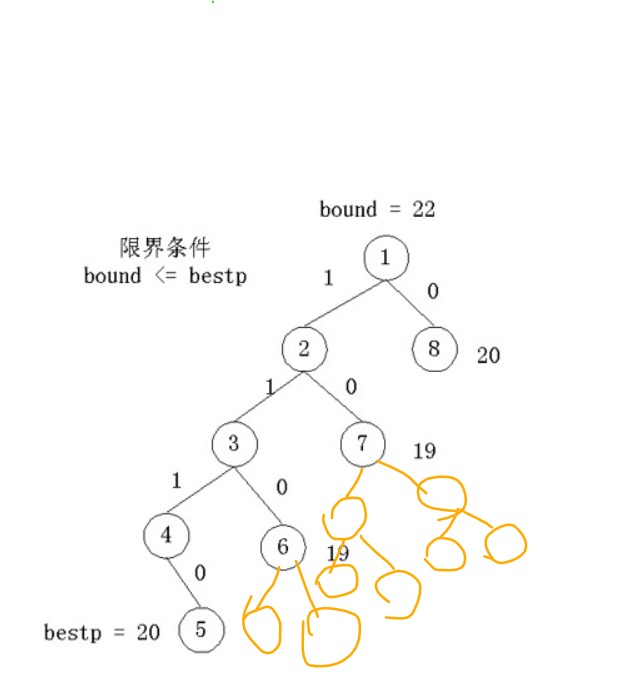


它在传统递归的基础上加了限界条件，采用深度优先（DFS）的模式去寻找最优的解。我们对于回溯法的数据先进行按价值密度排序的预处理（使得之后的选择会更加合理，在限界过程中不用多次排序）。

下面我们举一道例题来进行对比：考察一个背包例子，n=4,c=7,p=[9,10,7,4],w=[3,5,2,1]，按密度排列为(4,3,1,2)，w=[1,2,3,5],p=[4,7,9,10]，最终按照回溯法展开状态空间树如下：



思考我们具体加快搜索怎样实现的？我们对比一下dfs的步骤，你就会发现在节点6,7处我们都没有再次展开，这里可是省略了许多步骤呢😲！而我们付出的代价仅仅是每次计算一下最大贪心解执行一下限界判断而已，为此我们可是少进行了以下8个节点（即8中情况）的讨论呢！这就是加快搜索的原理。



# 伪代码

1. #按密度对于物品重排序
2. **def** sort\_by\_pw(weight,value,n):
3. **for** i **in** range(n):
4. **for** j **in** range(i+1,n,1):
5. **if** (value[i]/weight[i])<(value[j]/weight[j]):
6. value[i],value[j] = value[j],value[i]
7. weight[i],weight[j] = weight[j],weight[i]

对于右子树展开的限界条件，我写了两种不同的限界，一种是比较（当前总价值与剩余物品总价值之和）与当前最优价值比较，第二种是在第一种基础上改进，对于剩下未选择的物品通过连续背包处理相比前一种限界得到更合理的上界值。

1. #限界条件1（剩余物品价值与当前价值的总和大于最优解）
2. **def** bound(k,cv):
3. rv = 0
4. **while**(k<n):
5. rv += value[k]
6. k+=1
7. **return** cv + rv

10. #限界条件2（剩余物品连续背包处理的结果作为上界）
11. **def** bound\_profit(k,cv):
12. rv = 0
13. rw = 0
14. **for** i **in** range(k,n):
15. **if** weight[i]<=c - (cw + rw):
16. rw += weight[i]
17. rv += value[i]
18. **else**:
19. **if** c-(cw + rw)!=0:
20. rv += (weight[i]/value[i])\*(c - (cw + rw))
21. rw == c -cw
22. **else**:
23. **break**
24. **return** cv + rv

对于左子树的限界，需要（当前物品重量与当前的总重量之和）小于 （背包容量），实现的回溯算法如下：

1. **def** backpack(k):
2. **global** cv, cw, bestV
3. **if** k > n-1:
4. **if** cv > bestV:
5. **for** i **in** range(0, n):
6. bestX[i] = flag[i]
7. bestV = cv
8. **return**
10. **if** cw + weight[k] <= c:    #判断左子树是否符合条件
11. flag[k] = True
12. cw += weight[k]
13. cv += value[k]
14. backpack(k+1)
15. cw -= weight[k]
16. cv -= value[k]
17. **if** (bound(k+1, cv) > bestV):   #右子树
18. flag[k] = False
19. backpack(k+1)
20. 代码展示

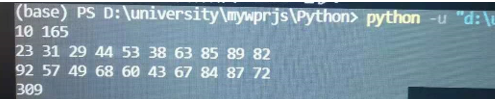
（1）python语言实现

1. **import** numpy as np
3. n, c = map(int, input().split())
4. weight = [int(n) **for** n **in** input().split()]
5. value = [int(n) **for** n **in** input().split()]
6. flag = np.full(len(value), False)
7. bestV = 0
8. bestX = flag
9. cv = 0
10. cw = 0
12. **def** backpack(k):
13. **global** cv, cw, bestV
14. **if** k > n-1:
15. **if** cv > bestV:
16. **for** i **in** range(0, n):
17. bestX[i] = flag[i]
18. bestV = cv
19. **return**
21. **if** cw + weight[k] <= c:    #判断左子树是否符合条件
22. flag[k] = True
23. cw += weight[k]
24. cv += value[k]
25. backpack(k+1)
26. cw -= weight[k]
27. cv -= value[k]
28. **if** (bound(k+1, cv) > bestV):   #右子树
29. flag[k] = False
30. backpack(k+1)
32. #限界条件（剩余物品价值与当前价值的总和大于最优解）
33. **def** bound(k,cv):
34. rv = 0
35. **while**(k<n):
36. rv += value[k]
37. k+=1
38. **return** cv + rv
39. #限界条件（剩余物品连续背包处理的结果作为上界）
40. **def** bound\_profit(k,cv):
41. rv = 0
42. rw = 0
43. **for** i **in** range(k,n):
44. **if** weight[i]<=c - (cw + rw):
45. rw += weight[i]
46. rv += value[i]
47. **else**:
48. **if** c-(cw + rw)!=0:
49. rv += (weight[i]/value[i])\*(c - (cw + rw))
50. rw == c -cw
51. **else**:
52. **break**
53. **return** cv + rv
55. #按密度对于物品重排序
56. **def** sort\_by\_pw(weight,value,n):
57. **for** i **in** range(n):
58. **for** j **in** range(i+1,n,1):
59. **if** (value[i]/weight[i])<(value[j]/weight[j]):
60. value[i],value[j] = value[j],value[i]
61. weight[i],weight[j] = weight[j],weight[i]
63. **if** \_\_name\_\_ == "\_\_main\_\_":
64. sort\_by\_pw(weight,value,n)
65. backpack(0)
66. **print**(bestV)
67. c++语言实现
68. #include<iostream>
69. #include<time.h>
70. #include<windows.h>
71. #define MAXN 25
72. using namespace std;
73. int n = 7;
74. int W = 170;
75. int w[ ]= {0,41,50,49,59,55,57,60};
76. int v[ ] = {0,442,525,511,593,546,564,617};
77. int x[MAXN];
78. int maxv;
80. void dfs(int i, int tw,int tv,int op[]);
81. void display();
83. int main()
84. {
85. LARGE\_INTEGER fre={0};//储存本机CPU时钟频率
86. LARGE\_INTEGER startCount={0};
87. LARGE\_INTEGER endCount = {0};
88. QueryPerformanceFrequency(&fre);
90. int op[MAXN];
91. dfs(1,0,0,op);
92. display();
94. QueryPerformanceCounter(&startCount);
95. Sleep(1000);
96. //结束计时
97. QueryPerformanceCounter(&endCount);
98. //计算时间差
99. double dTimeTake = ((double)endCount.QuadPart - (double)startCount.QuadPart) / (double)fre.QuadPart;
100. printf("用时%f\n", dTimeTake);
101. **return** 0;
102. }
104. void dfs(int i, int tw,int tv,int op[])
105. {
106. **if**(i>n)
107. {
108. **if**(tw <=W && tv >maxv)
109. {
110. maxv  = tv;
111. **for**(int j=1;j<=n;j++)
112. {
113. x[j] = op[j];
114. }
115. }
116. }
117. **else**
118. {
119. **if**(tw+w[i] <= W)
120. {
121. op[i] = 1;
122. dfs(i+1,tw+w[i],tv+v[i],op);
123. }
124. op[i]=0;
125. dfs(i+1,tw,tv,op);
126. }
127. }
129. void display()
130. {
131. int i;
132. printf("回溯求解结果\n");
133. printf("选取的物品：");
134. **for**(i =1;i<=n;i++)
135. **if**(x[i] == 1)
136. cout<<i<<" ";
137. cout<<endl;
138. printf("总重量=%d,总价值=%d\n",W,maxv);
139. }

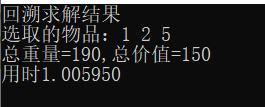
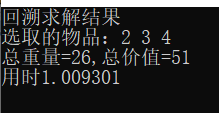
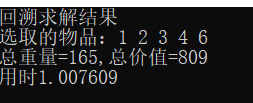
# 测试结果

所选数据集为Florida State University: 8 个数据集，Unicauca University: 31 个数据集进行性能分析（具体的输入样例请查看文件夹内01背包.txt），这里仅展示最终输出结果

（1）python测试结果部分展示



（2）c++测试结果部分展示



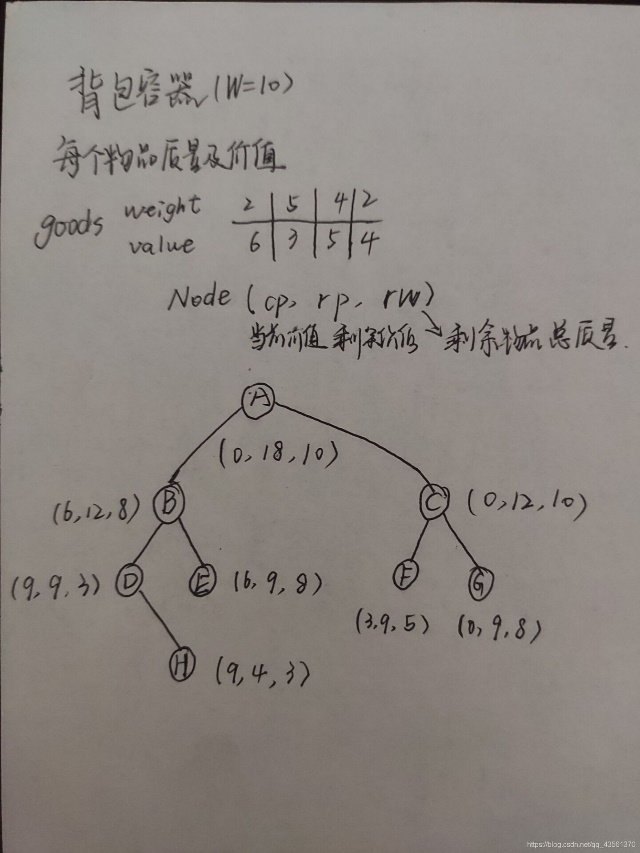
# 算法分析

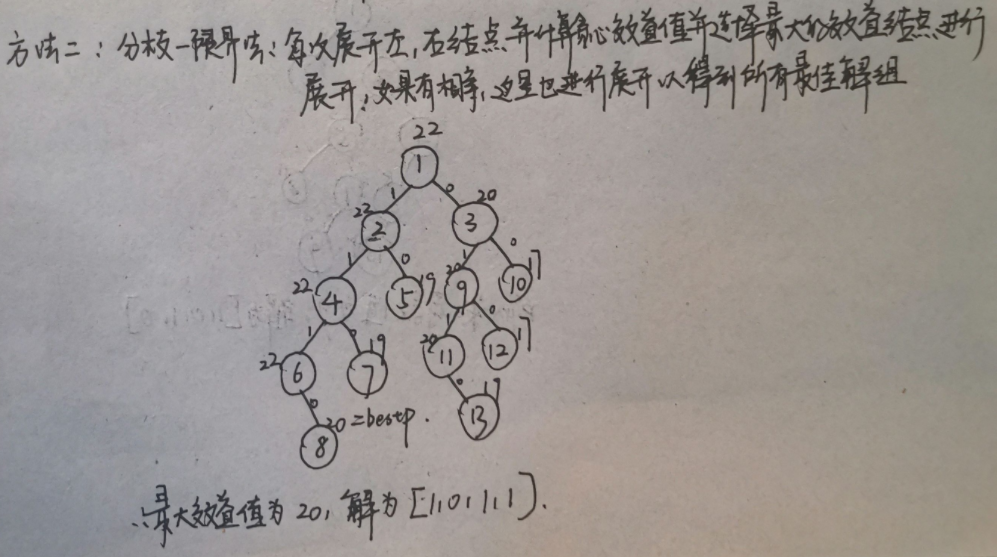
时间复杂度为O(n^2)，利用限界条件加以剪枝展开状态空间树，避免了许多已知不可能得到更优解的情况，加快了搜索速度。

分枝限界法设计流程

# 设计思路

采用FIFO（先进先出），物品放入背包中或物品不放入背包中，根据约束条件舍去无效情况。当前物品k放入背包中时，获取当前最大总价值，下一个物品k+1也存在放入背包和不放入背包中两个节点。当物品k不放入背包时，上个节点的最总价值为当前节点的最总价值，下一个物品k+1仍然存在放入背包和不放入背包两个节点。对于物品k+1以此类推，最终根据不同的放入情况选择一个最优解作为可以放入背包物品的最大总价值。我简单模拟一组数据，并列出状态空间树，如下图：





# 代码展示

（1）python语言实现

1. **import** numpy as np
2. **import** queue
3. **import** math
4. n, c = map(int, input().split())
5. w = [int(n) **for** n **in** input().split()]
6. v = [int(n) **for** n **in** input().split()]
7. **def** test(capacity):
8. vec\_len = 2\*\*(len(v)+1) - 1#tree `s size
9. vec\_v = np.zeros(vec\_len)#当前总价值
10. vec\_w = np.zeros(vec\_len)#当前剩余容量
11. vec\_w[0]=capacity
12. que = queue.Queue();
13. que.put(0)
14. best = 0
15. **while**(**not** que.empty()):
17. current = que.get()
19. level = int(math.log(current+1,2))#当前级数
21. **if**(vec\_v[current] > vec\_v[best]):
22. best = current
24. left = 2\*current+1#左子树根节点序号
26. right = 2\*current+2#右子树根节点序号
28. #左子树限界
29. **if**(left < vec\_len **and** vec\_w[current]-w[level] >= 0):
30. vec\_v[left] = int(vec\_v[current]+v[level])
31. vec\_w[left] = vec\_w[current]-w[level]
32. que.put(left)
34. #右子树限界
35. **if**(right < vec\_len **and** sum(v[level+1:-1])+vec\_v[current] > vec\_v[best]):
36. vec\_v[right] = vec\_v[current]
37. vec\_w[right] = vec\_w[current]
38. que.put(right)
40. **print** (vec\_v[best])

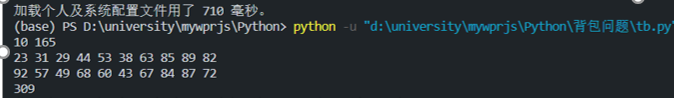
43. **if**  \_\_name\_\_ == '\_\_main\_\_':
44. test(c)

（2）c++语言实现

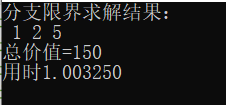
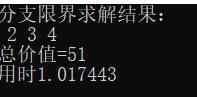
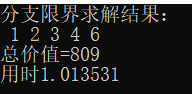
1. #include<iostream>
2. #include<queue>
3. #include<time.h>
4. #include<windows.h>
5. **using** **namespace** std;
6. #define MAXN 25
7. **int** n = 7;
8. **int** W = 170;
9. **int** w[ ]= {0,41,50,49,59,55,57,60};
10. **int** v[ ] = {0,442,525,511,593,546,564,617};
12. **int** maxv = -9999;
13. **int** bestx [MAXN];
14. **int** total=1;
16. **struct** NodeType
17. {
18. **int** no;
19. **int** i;
20. **int** w;
21. **int** v;
22. **int** x[MAXN];
23. **double** ub;
24. };
25. **void** bound(NodeType &e );
26. **void** EnQueue(NodeType e,queue<NodeType> &qu);
27. **void** bfs();
28. **int** main()
29. {
30. LARGE\_INTEGER fre={0};//储存本机CPU时钟频率
31. LARGE\_INTEGER startCount={0};
32. LARGE\_INTEGER endCount = {0};
33. QueryPerformanceFrequency(&fre);
35. bfs();
36. printf("分支限界求解结果：\n ");
37. **for**(**int** i = 1;i<=n;i++)
38. **if**(bestx[i] == 1)
39. cout<<i<<" ";
40. cout<<endl;
41. printf("总价值=%d\n",maxv);
43. QueryPerformanceCounter(&startCount);
44. Sleep(1000);
45. //结束计时
46. QueryPerformanceCounter(&endCount);
47. //计算时间差
48. **double** dTimeTake = ((**double**)endCount.QuadPart - (**double**)startCount.QuadPart) / (**double**)fre.QuadPart;
49. printf("用时%f\n", dTimeTake);
50. **return** 0;
51. }
53. **void** bound(NodeType &e )
54. {
55. **int** i =e.i+1;
56. **int** sumw = e.w;
57. **double** sumv = e.v;
58. **while**(i<= n && (sumw + w[i] <= W))
59. {
60. sumw += w[i];
61. sumv+= v[i];
62. i++;
63. }
64. **if**(i<=n)
65. e.ub = sumv+(W-sumw)\*v[i]/w[i];
66. **else**
67. e.ub = sumv;
68. }
70. **void** EnQueue(NodeType e,queue<NodeType> &qu)
71. {
72. **if**(e.i==n)
73. {
74. **if**(e.v>maxv)
75. {
76. maxv = e.v;
77. **for**(**int** j =1;j<=n;j++)
78. bestx[j]=e.x[j];
79. }
80. }
81. **else** qu.push(e);
82. }
84. **void** bfs()
85. {
86. **int** j;
87. NodeType e,e1,e2;
88. queue<NodeType> qu;
89. e.i = 0;
90. e.w = 0;
91. e.v = 0;
92. e.no = total++;
93. **for**(j = 1;j<=n;j++)
94. e.x[j]=0;
95. bound(e);
96. qu.push(e);
97. **while**(!qu.empty())
98. {
99. e = qu.front();
100. qu.pop();
101. **if**(e.w+w[e.i+1]<=W)
102. {
103. e1.no =total++;
104. e1.i = e.i+1;
105. e1.w = e.w+w[e1.i];
106. e1.v = e.v+v[e1.i];
107. **for**(j =1;j<=n;j++)
108. e1.x[j]=e.x[j];
109. e1.x[e1.i] =1;
110. bound(e1);
111. EnQueue(e1,qu);
112. }
113. e2.no = total++;
114. e2.i=e.i+1;
115. e2.w=e.w;
116. e2.v=e.v;
117. **for**(j =1;j<=n;j++)
118. e2.x[j]=e.x[j];
119. e2.x[e2.i]=0;
120. bound(e2);
121. **if**(e2.ub>maxv)
122. EnQueue(e2,qu);
123. }
124. }

# 测试结果

（1）python语言测试部分结果展示



（2）c++语言测试部分结果展示



# 算法分析

分枝-限界方法我们了解到了他总是伴随bfs出现，并且每次都要计算c(x)来记录随时切换到最理想的情况进行求解，最终即可同时得到最优值和最优解，所以其根本难点在于计算c(x)，对于罚款额，背包等一般问题我们按照正常思维（每次寻找最少罚款额，每次选取效益值大的情况）即可写出c(x）计算式，时间复杂度可以控制在O(n^2)以内加以限界条件：所展开节点的效益值上界（即最理想情况）要大于已知最有效益值时才可以加入到FIFO活结点表中展开。相较于bfs广搜剪枝去掉了已知不可能在产生更优解的情况，在保证结果正确的情况下加快了搜索效率。

三种算法的对比分析

动态规划：寻找递归式求解并借用二维数组打表或者一维线性索引表（元组法）记录中途的结果以减少计算次数来加快计算速度，无剪枝策略，讨论所有情况。

回溯法：用限界条件（所展开E-节点的效益值上界要好于已知最优解）时可以展开左子树节点来实现剪枝同时回溯讨论右子树情况，加快搜索效率。

分枝限界法：对于任何一个E-节点都需要满足限界条件（同上）才可以加入到FIFO队列中等待展开，每次展开都是从当前节点和FIFO活结点表中选取可能产生最好情况的节点，最终到达叶子节点所组成的一组元组解就是最优解。

三种方法各有优势，没有绝对的优劣之分，对于不同的输入样例三种方法的搜索速率大致相同，时间复杂度都可以控制在O(n^2)之内相较于暴力枚举都大幅提升了搜索效率，减少了计算开销。

文献参考

1. <https://blog.csdn.net/qq_37437983/article/details/103583451>
2. <https://blog.csdn.net/gl620321/article/details/108698065>
3. <https://www.cnblogs.com/ChristalR/p/Dynamic_programming.html>
4. 《算法导论》动态规划，回溯法与分支限界法章节以及0/1背包求解问题案例分析
5. 《c++语言描述数据结构，算法与应用》机械工业出版社