greedy

最优化问题是计算机领域的一个很重要的问题，很多现实的问题本质上都是最优化问题，或者说都可以转化为最优化的问题。比如说怎么规划旅游线路最省钱，在指定的时间里做更多的事情等等，这些都是最优化问题。为了解决最优化问题，各种大神提出了各种算法，有穷举（这个是凑数的），贪婪，动态规划，分治算法，回溯算法等等。本文主要归纳整理贪婪算法。

贪婪算法是指，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，他所做出的是在某种意义上的局部最优解。

贪婪算法不能保证得到全局最优解（应该说大部分情况下都不是全局最优），最重要的是要选择一个优的贪婪策略，如果贪婪策略选的不好，结果就会比较差。然而就像Ivan说过的“我认为贪婪是健康的，你可以在贪婪的同时自我感觉很良好”（这货后来被监禁了，这货后来被监禁了，这货后来被监禁了，重要的话说三遍）。贪婪算法依旧是一个很不错的算法，这是一个简单同时还能得到比较不错的结果的算法（非常切合中庸之道啊）。

**贪婪算法可解决的问题通常大部分都有如下的特性：**

⑴随着算法的进行，将积累起其它两个集合：一个包含已经被考虑过并被选出的候选对象，另一个包含已经被考虑过但被丢弃的候选对象。

⑵有一个函数来检查一个候选对象的集合是否提供了问题的解答。该函数不考虑此时的解决方法是否最优。

⑶还有一个函数检查是否一个候选对象的集合是可行的，也即是否可能往该集合上添加更多的候选对象以获得一个解。和上一个函数一样，此时不考虑解决方法的最优性。

⑷选择函数可以指出哪一个剩余的候选对象最有希望构成问题的解。

⑸最后，目标函数给出解的值。

⑹为了解决问题，需要寻找一个构成解的候选对象集合，它可以优化目标函数，贪婪算法一步一步的进行。起初，算法选出的候选对象的集合为空。接下来的每一步中，根据选择函数，算法从剩余候选对象中选出最有希望构成解的对象。如果集合中加上该对象后不可行，那么该对象就被丢弃并不再考虑；否则就加到集合里。每一次都扩充集合，并检查该集合是否构成解。如果贪婪算法正确工作，那么找到的第一个解通常是最优的。

上面的话总结一下就是自己决定一个策略，从一个集合里拿向另一个空集合里装，什么时候拿满了就搞定。

一个典型的最优化问题就是0/1背包问题，下面我们通过这个问题来实践一下贪婪算法。

[0-1背包问题]有一个背包，背包容量是M=150kg。有7个物品，物品不可以分割成任意大小。（这句很重要）

要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

物品A B C D E F G

重量35kg 30kg 6kg 50kg 40kg 10kg 25kg

价值1040

3050 3540

30$

要想得到最优的结果，最容易想到的就是穷举法，那么首先用穷举法来处理这个问题。

穷举法的时间复杂度有些恐怖，那么我们利用贪婪算法来解决0/1背包问题

类的定义与穷举法一样：

# item have three attribute: name,weight,value

class Item(object):

def \_\_init\_\_(self,n,v,w):

self.name=n

self.weight=float(w)

self.value=float(v)

def getName(self):

return self.name

def getValue(self):

return self.value

def getWeight(self):

return

self.weight def \_\_str\_\_(self): result = ' < '+self.name+' , '+str(self.value)+' , '+str(self.weight) + '>'

return result

定义三种贪婪策略：分别为按价值，按最小重量，以及性价比（价值/重量）

def value(item):

return item.getValue()

def weightInverse(item):

return 1.0/item.getValue()

def density(item):

return item.getValue()/item.getWeight()

利用贪婪来获取解空间：  
这里keyFunciton是具体的贪婪策略  
函数实现了按照具体的贪婪策略来逐步选择物品，直到满足限制条件为止。  
sorted函数是按照具体的贪婪策略进行排序。

def greedy(items,maxWeight,keyFunction):

itemsCopy = sorted(items,key=keyFunction,reverse = True)

result = []

totalValue = 0.0

totalWeight = 0.0

for i in range(len(itemsCopy)):

if(totalWeight + itemsCopy[i].getWeight()) <= maxWeight:

result.append(itemsCopy[i])

totalValue +=itemsCopy[i].getValue()

totalWeight +=itemsCopy[i].getWeight()

return (result,totalValue)

测试代码：

def buildItem():

names=['A','B','C','D','E','F','G']

vals=[35,30,6,50,40,10,25]

weights=[10,40,30,50,35,40,30]

Items=[]for i in range(len(names)):

Items.append(Item(names[i],vals[i],weights[i]))

return Items

def testGreedy(items,constraint,keyFunction):

taken,val=greedy(items,constraint,keyFunction)

print'Total value of items taken=',val

for item in taken:

print'',item

def testGreedys(maxWeight=150):

items=buildItem()

print'Use greedy by value to fill knapsack of size',maxWeight

testGreedy(items,maxWeight,value)

print'\n Use greedy by weight to fill knapsack of size',maxWeight

testGreedy(items,maxWeight,weightInverse)

print'\n Use greedy by density to fill knapsack of size',maxWeight

testGreedy(items,maxWeight,density)

具体测试结果如下：

Use greedy by value to fill knapsack of size 150

Total value of items taken = 155.0

< D , 50.0 , 50.0>

< E , 40.0 , 35.0>

< A , 35.0 , 10.0>

< B , 30.0 , 40.0>

Use greedy by weight to fill knapsack of size 150

Total value of items taken = 106.0

< C , 6.0 , 30.0>

< F , 10.0 , 40.0>

< G , 25.0 , 30.0>

< B , 30.0 , 40.0>

< A , 35.0 , 10.0>

Use greedy by density to fill knapsack of size 150

Total value of items taken = 150.0

< A , 35.0 , 10.0>

< E , 40.0 , 35.0>

< D , 50.0 , 50.0>

< G , 25.0 , 30.0>

runing time is 0.000358

从结果中可以得出两个事实：

1）贪婪不一定能得到全局最优解，贪婪得到的是局部最优解，最终结果取决于贪婪策略

2）贪婪的时间消耗比穷举法低好多

总结：贪婪算法或许不是一个很好的算法，但是在解决一些问题时，如果选择好的贪婪策略，结果也是可以很优秀的。

彩蛋：贪婪算法没有办法得到全局最优解，那么0/1背包问题怎么办，岂不是很孤独？不用担心，大神们是很有爱的，他们早已搞定了这个问题，就是利用动态规划的方法，下面的链接将讲解动态规划