**调度算法研究**

**贪婪算法**

贪婪算法是一种短视算法，它会每次选择一个局部最优解，而不会考虑到该解对于今后的影响。一般来说，他的时间复杂度比较低，虽然贪婪算法并不从整体上最优加以考虑，导致它所作出的选择只是某种意义下的局部最优选择，这在工程实践中反而比较常用。

通常贪婪算法具有五部分构成：

一个待选元素集合，从该集合中选择元素，得到问题的解决方案。

一个选择函数，每次依据该函数进行选择，选择当前最优的情况，加入已选序列。

一个可行性函数，用来判断，该候选是否可以被用于作为解决方案的一部分。

一个目标函数，用来衡量最终解决方案。

一个解决方案函数，用来展示发现完全解决方案的时间。

对于一些数学问题，贪婪算法可以产生很好的解，但是对一些可能并没有好的解。对于可以产生优秀解的问题，一般有贪婪选择和最优子结构这两个特性。

我们在每一刻都选择当前时刻最优的解，然后解决以后出现的子问题。由贪婪算法做出的选择可能取决于迄今取得而不是对未来的选择或者所有的解决方案的子问题的选择。它反复让人贪婪的选择，将每个给定的问题变成一个较小的一个。换句话说，贪婪算法从未重新考虑其选择。这不同于动态规划，每一个阶段后，动态规划基于在前一阶段作出的所有决定进行决策，并可能重新考虑前一阶段的算法路径得到解决方案。如果这个问题的最佳解决方案包含最优解的子问题那么这个问题呈现最优子结构  
建立数学模型来描述问题。  
把求解的问题分成若干个子问题。对每一子问题求解，得到子问题的局部最优解。把子问题的解局部最优解合成原来解问题的一个解。

输入：重量序列，其中，容积b

输出：解的集合序列，价值

初始化解的序列，剩余容积，以及价值

for i = 1 to n(每一项) do

if (物体重量没有超过目前的容积) then

把物体i添加到解序列,

更新剩余容积 ，以及价值

else

Stop and return S, U

end if

end for

Return S, U

**动态规划算法**

一、基本概念  
动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。  
二、基本思想与策略  
基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。  
由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。  
与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

三、适用的情况  
能采用动态规划求解的问题的一般要具有3个性质：  
(1) 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。  
(2) 无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。  
（3）有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

四、求解的基本步骤  
动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。动态规划的设计都有着一定的模式，一般要经历以下几个步骤。  
初始状态→│决策１│→│决策２│→…→│决策ｎ│→结束状态  
(1)划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。  
(2)确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。  
(3)确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。  
(4)寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。  
一般，只要解决问题的阶段、状态和状态转移决策确定了，就可以写出状态转移方程（包括边界条件）。  
实际应用中可以按以下几个简化的步骤进行设计：  
（1）分析最优解的性质，并刻画其结构特征。  
（2）递归的定义最优解。  
（3）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值  
（4）根据计算最优值时得到的信息，构造问题的最优解

五、算法实现的说明  
动态规划的主要难点在于理论上的设计，也就是上面4个步骤的确定，一旦设计完成，实现部分就会非常简单。  
使用动态规划求解问题，最重要的就是确定动态规划三要素：  
（1）问题的阶段 （2）每个阶段的状态  
（3）从前一个阶段转化到后一个阶段之间的递推关系。  
递推关系必须是从次小的问题开始到较大的问题之间的转化，从这个角度来说，动态规划往往可以用递归程序来实现，不过因为递推可以充分利用前面保存的子问题的解来减少重复计算，所以对于大规模问题来说，有递归不可比拟的优势，这也是动态规划算法的核心之处。  
确定了动态规划的这三要素，整个求解过程就可以用一个最优决策表来描述，最优决策表是一个二维表，其中行表示决策的阶段，列表示问题状态，表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值（如最短路径，最长公共子序列，最大价值等），填表的过程就是根据递推关系，从1行1列开始，以行或者列优先的顺序，依次填写表格，最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。  
f(n,m)=max{f(n-1,m), f(n-1,m-w[n])+P(n,m)}  
动态规划算法是一个精确的算法。动态规划，简单说来就是知道一个状态，然后根据状态转移方程得到另外一个状态。根据这两个状态间的转换，递推获得最终解。

**rollout算法研究**

rollout算法对各种动态和离散优化问题具有十分优异的表现。作为近似的动态规划算法，一个rollout算法，在每个决策阶段会通过以称为基策略的贪心策略模拟未来事件的方法来使得选择近似理想的方案。虽然在很多情况下rollout算法，可以保证执行情况和它们的基策略一样，已经出现了一些理论成果显示采用rollout算法，性能会有额外的改善。根据现有文献的随机模型，本文分析被称之为consecutive rollout和exhaustive rollout的两种rollout方法，这两者采用简单的贪婪算法作为基策略。我们分析这两种算法在普遍问题下的应用场景及算法流程，然后结合我们的实际问题，对这两种算法进行分析。

**consecutive rollout**

* 算法思路  
  consecutive rollout算法的程序流程见下面，该算法输入为。。。，输出为。。。。，并调用贪婪算法作为其子程序。  
  算法流程。。77

输入：重量序列，其中，容积b

输出：解的集合序列，价值

初始化解的序列，剩余容积，以及价值

for i = 1 to n(每一项) do

估计加入物体i的价值，

估计未加入物体i的价值，

if (加入物体的估计值更大) then

把物体i添加到解序列,

更新剩余容积 ，以及价值

end if

end for

Return S, U

**exhaustive rollout**

* 算法思路  
  在每次迭代中，用t索引，该算法考虑在现有的顺序一它计算由移动每个项目序列的前部，施加致盲贪婪算法得到的值的所有项目。然后，该算法具有最高估计值添加项（如果它存在）到溶液中。我们暗含的假设一致的打破平局方法，如该项目优先考虑以最低的指数。下一次迭代，然后用物品的其余序列前进。  
  考虑  
  算法流程。

输入：重量序列，其中，容积b

输出：解的集合序列，价值

初始化解的序列，剩余容积，以及价值

for t = 1 to n do

for(剩余项序列中的每一项) do

用表示将i移动第一位的序列

估计价值序列，

end for

if then

决定具有最大估计值的项,

把项添加到解序列，

更新剩余容积 ，以及价值

end if

end for

Return S, U

我们可以发现，对于 consecutive rollout 算法，他会依次对当前的是否选择情况进行一个基策略，看是选择还是不选择的结果会好一些，对于 exhaustive rollout 算法，它需要对每次剩余的结果进行穷举，然后把其中的某一项移动最前方，对该序列再次进行遍历，然后从其中选择一个最优的。  
结合我们的实际情况，本文需要调度的是M个传感器的选择问题，每一步的选择情况都是固定的，是一个和M有关的表达式。可以很方便的使用 consecutive rollout 算法。而在传感器的选择方面，具有很强的时序相关性，时间序列无法随意改变，所以 exhaustive rollout 算法显然不适用于本文的情况。  
另外本文采用的rollout算法，均为单次迭代算法，在今后的工作中我们可以考虑rollout算法的第二次迭代。一个相关的议题是仍考虑rollout算法只是第一次迭代，但具有较大的超前长度（如试图对所有项目进行了 exhaustive rollout，而不仅仅是单独的每个项）。