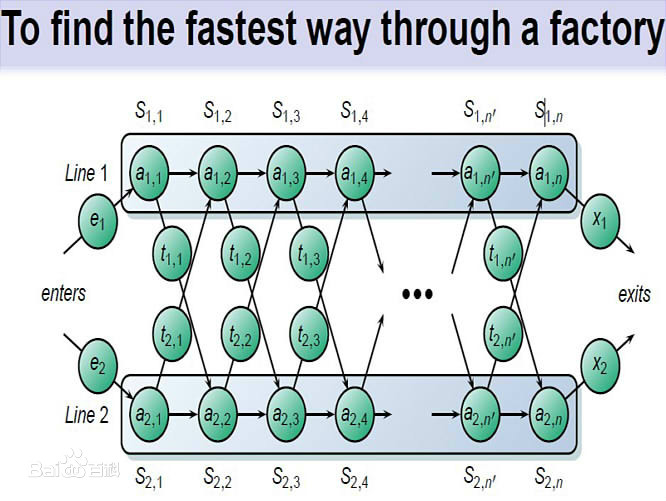
# 动态规划法

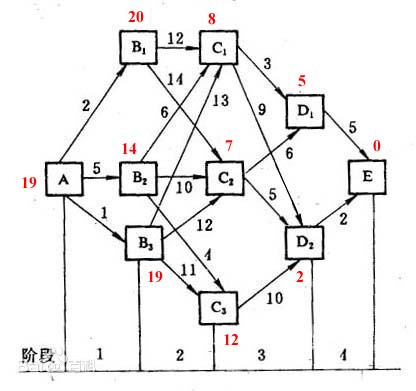
多阶段决策问题中，各个阶段采取的[决策](https://baike.baidu.com/item/%E5%86%B3%E7%AD%96)，一般来说是与时间有关的，决策依赖于当前状态，又随即引起状态的转移，一个决策序列就是在变化的状态中产生出来的，故有“动态”的含义，称这种解决多阶段决策最优化问题的方法为动态规划方法。



动态规划程序设计是对解最优化问题的一种途径、一种方法，而不是一种特殊算法。不像搜索或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法。动态规划程序设计往往是针对一种最优化问题，由于各种问题的性质不同，确定最优解的条件也互不相同，因而动态规划的设计方法对不同的问题，有各具特色的解题方法，而不存在一种万能的动态规划算法，可以解决各类最优化问题。

## 概念引入

在现实生活中，有一类活动的过程，由于它的特殊性，可将过程分成若干个互相联系的阶段，在它的每一阶段都需要作出决策，从而使整个过程达到最好的活动效果。当然，各个阶段决策的选取不是任意确定的，它依赖于当前面临的状态，又影响以后的发展，当各个阶段决策确定后，就组成一个决策序列，因而也就确定了整个过程的一条活动路线，如图所示：这种把一个问题看作是一个前后关联具有链状结构的多阶段过程就称为多阶段决策过程，这种问题就称为多阶段决策问题。



## 基本思想

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有[最优值](https://baike.baidu.com/item/%E6%9C%80%E4%BC%98%E5%80%BC)的解。动态规划算法与[分治法](https://baike.baidu.com/item/%E5%88%86%E6%B2%BB%E6%B3%95)类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从[这些子](https://baike.baidu.com/item/%E8%BF%99%E4%BA%9B%E5%AD%90)问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。

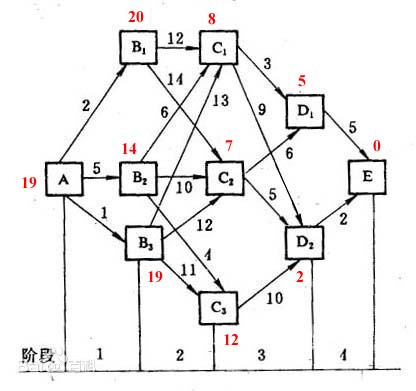
## 基本概念

### 多阶段决策问题

如果一类活动过程可以分为若干个互相联系的阶段，在每一个阶段都需作出决策(采取措施)，一个阶段的决策确定以后，常常影响到下一个阶段的决策，从而就完全确定了一个过程的活动路线，则称它为多阶段决策问题。

各个阶段的决策构成一个决策序列，称为一个策略。每一个阶段都有若干个决策可供选择，因而就有许多策略供我们选取，对应于一个策略可以确定活动的效果，这个效果可以用数量来确定。策略不同，效果也不同，多阶段决策问题，就是要在可以选择的那些策略中间，选取一个最优策略，使在预定的标准下达到最好的效果.

### 阶段

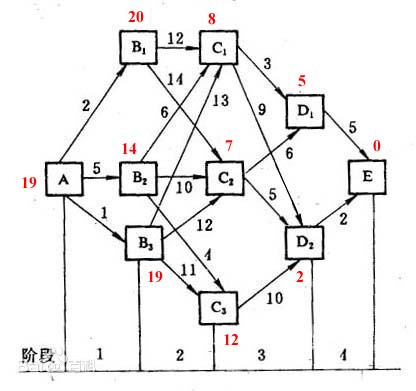
把所给求解问题的过程恰当地分成若干个相互联系的阶段，以便于求解，过程不同，阶段数就可能不同。描述阶段的变量称为阶段变量。在多数情况下，阶段变量是离散的，用k表示。此外，也有阶段变量是连续的情形。如果过程可以在任何时刻作出决策，且在任意两个不同的时刻之间允许有无穷多个决策时，阶段变量就是连续的。

如图所示：第一个阶段就是点A，而第二个阶段就是点A到点B，第三个阶段是点B到点C，而第四个阶段是点C到点D。

### 状态

状态表示每个阶段开始面临的自然状况或客观条件，它不以人们的主观意志为转移，也称为不可控因素。在上面的例子中状态就是某阶段的出发位置，它既是该阶段某路的起点，同时又是前一阶段某支路的终点。在前面的例子中，第一个阶段有一个状态即A，而第二个阶段有两个状态B1和B2，第三个阶段是三个状态C1，C2和C3，而第四个阶段又是一个状态D。

### 无后效性

我们要求状态具有下面的性质：如果给定某一阶段的状态，则在这一阶段以后过程的发展不受这阶段以前各段状态的影响，所有各阶段都确定时，整个过程也就确定了。换句话说，过程的每一次实现可以用一个状态序列表示，右图的例子中每阶段的状态是该线路的某个结点，确定了这些点的序列，整个线路也就完全确定。从某一阶段以后的线路开始，当阶段的始点给定时，不受以前线路（所有通过的点）的影响。状态的这个性质意味着过程的历史只能通过当前的状态去影响它未来的发展，这个性质称为无后效性。

### 决策

一个阶段的状态给定以后，从该状态演变到下一阶段某个状态的一种选择（行动）称为决策。在许多问题中，决策可以自然而然地表示为一个数或一组数。不同的决策对应着不同的数值。描述决策的变量称[决策变量](https://baike.baidu.com/item/%E5%86%B3%E7%AD%96%E5%8F%98%E9%87%8F)，因状态满足无后效性，故在每个阶段选择决策时只需考虑当前的状态而无须考虑过程的历史。决策变量的范围称为允许决策集合。

### 策略

由每个阶段的决策组成的序列称为策略。对于每一个实际的多阶段决策过程，可供选取的策略有一定的范围限制，这个范围称为允许策略集合。允许策略集合中达到最优效果的策略称为最优策略。

## 适用条件

任何思想方法都有一定的局限性，超出了特定条件，它就失去了作用。同样，动态规划也并不是万能的。适用动态规划的问题必须满足最优化原理和无后效性。

1.[最优化原理](https://baike.baidu.com/item/%E6%9C%80%E4%BC%98%E5%8C%96%E5%8E%9F%E7%90%86)（最优子结构性质） 最优化原理可这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。

2.[无后效性](https://baike.baidu.com/item/%E6%97%A0%E5%90%8E%E6%95%88%E6%80%A7)将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

3.子问题的重叠性 动态规划将原来具有指数级时间复杂度的[搜索算法](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E7%AE%97%E6%B3%95)改进成了具有多项式时间复杂度的算法。其中的关键在于解决冗余，这是动态规划算法的根本目的。动态规划实质上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其它的算法。

## 实现问题

[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)实现是比较好考虑的。但有时也会遇到一些问题，而使算法难以实现。动态规划思想设计的算法从整体上来看基本都是按照得出的[递推](https://baike.baidu.com/item/%E9%80%92%E6%8E%A8)关系式进行递推，这种递推相对于计算机来说，只要设计得当，效率往往是比较高的，这样在时间上溢出的可能性不大，而相反地，动态规划需要很大的[空间](https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4/55280)以存储中间产生的结果，这样可以使包含同一个子问题的所有问题共用一个子问题解，从而体现动态规划的优越性，但这是以牺牲空间为代价的，为了有效地访问已有结果，数据也不易压缩存储，因而空间矛盾是比较突出的。另一方面，动态规划的高时效性往往要通过大的测试数据体现出来（以与[搜索](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2/2791632)作比较），因而，对于大规模的问题如何在基本不影响运行速度的条件下，解决空间溢出的问题，是动态规划解决问题时一个普遍会遇到的问题。

一个思考方向是尽可能少占用[空间](https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4/55280)。如从结点的[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/1450)上考虑，仅仅[存储](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8)必不可少的内容，以及数据存储范围上精打细算(按位存储、压缩存储等)。当然这要因问题而异，进行分析。另外，在实现动态规划时，一个我们经常采用的方法是用一个与结点数一样多的数组来存储每一步的决策，这对于倒推求得一种实现最优解的方法是十分方便的，而且处理速度也有一些提高。但是在内存空间紧张的情况下，我们就应该抓住问题的主要矛盾。省去这个存储决策的数组，而改成在从最优解逐级倒推时，再计算一次，选择某个可能达到这个值的上一阶段的状态，直到推出结果为止。这样做，在程序编写上比上一种做法稍微多花一点时间，运行的时效也可能会有一些(但往往很小)的下降，但却换来了很多的空间。因而这种思想在处理某些问题时，是很有意义的。

但有时，即使采用这样的方法也会发现空间溢出的问题。这时就要分析，这些保留下来的数据是否有必要同时存在于内存之中。因为有很多问题，动态规划递推在处理后面的内容时，前面比较远处的内容实际上是用不着的。对于这类问题，在已经确信不会再被使用的数据上[覆盖](https://baike.baidu.com/item/%E8%A6%86%E7%9B%96/5946016)数据，从而使空间得以重复利用，如果能有效地使用这一手段，对于相当大规模的问题，空间也不至于溢出（为了求出最优方案，保留每一步的决策仍是必要的，这同样需要空间）。

一般地说，这种方法可以通过两种思路来实现：一种是递推结果仅使用Data1和Data2这样两个数组，每次将Data1作为上一阶段，推得Data2数组，然后，将Data2通过复制覆盖到Data1之上，如此反复，即可推得最终结果。这种做法有一个局限性，就是对于递推与前面若干阶段相关的问题，这种做法就比较麻烦；而且，每递推一级，就需要复制很多的内容，与前面多个阶段相关的问题影响更大。另外一种实现方法是，对于一个可能与前N个阶段相关的问题，建立数组Data[0..N]，其中各项为前面N个阶段的保存数据。这样不采用这种内存节约方式时对于阶段k的[访问](https://baike.baidu.com/item/%E8%AE%BF%E9%97%AE)只要对应成对数组Data中下标为k mod (N+1)的单元的访问就可以了。这种处理方法对于程序修改的代码很少，速度几乎不受影响，而且需要保留不同的[阶段](https://baike.baidu.com/item/%E9%98%B6%E6%AE%B5)数也都能很容易实现。

当采用以上方法仍无法解决内存问题时，也可以采用对内存的动态申请来使绝大多数情况能有效出解。而且，使用动态内存还有一点好处，就是在重复使用内存而进行交换时，可以只对指针进行交换，而不复制数据，这在实践中也是十分有效的。