Last Section

减 治

- 插入排序：n^2, n, $n^2/4$
- 快速排序+插入排序
- 拓扑排序：减一
- 生成排列+ Johnson-Trotter
- 生成子集+比特串方法

- 假币问题
- 俄式乘法
- 约瑟夫斯问题

- 欧几里德算法
- 插值查找
- 二叉查找树

变治_实例化简

- 预排序
  - 检验数组中元素的惟一性： $n(n-1)/2$, $n\log n+n$
  - 模式计算： $n(n-1)/2+n-1$, $n\log n+\Theta(n)$

- 高斯消去法
  - Partial pivoting
  - LU 分解
  - 矩阵的逆

- AVL树: $1.39\log n$, $1.01\log n$

变治_改变表现

变换为同样实例的不同表现—**改变表现**
(Representation Change)

- 2-3 树
- 堆和堆排序
- 霍纳法则
- 二进制幂

变治

变换为另一个问题的实例， 这种问题的算法是已知的**—问题化简(Problem reduction)**。

- Lcm
- 图中的路径数量
- 函数极值
- 综合除法
- 凸包，解析几何
- 线性规划
- 简化为图
- MST vs. Element Uniqueness

MST & Element Uniqueness

- MST 与 EU 哪一个可以规约为另一个问题？
- 规约的过程是怎样的?
- 哪一个更“容易”？

动态规划

- 最优化原理:
  - 无论过去的状态和决策如何，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

- 构成动态规划模型的条件
  - 正确选择状态变量x_k, 使它既能描述过程的状态，又要满足无后效性(如果某段状态给定，则在这段以后过程的发展不受前面各阶段状态的影响)
  - 确定决策变量u_k及每段的允许决策集合
  - 写出状态转移方程
  - 列出指标函数$V_{k,n}$关系，并要满足递推性

问题举例

例2 机器负荷分配问题

某种机器，可以在高低两种不同的负荷下进行生产。

在高负荷下进行生产时，产品的年产量s_1和投入生产的机器数量u_1的关系为

$$s_1 = g(u_1)$$

这时，机器的年折损率为a，即如果年初完好机器的数量为u，到年终时完好的机器就为au, $0<a<1$.

在低负荷下生产时，产品的年产量s_2和投入生产的机器数量u_2的关系为

$$s_2 = h(u_2)$$

相应的机器的年折损率为b, $0<b<1$.

假定开始生产时完好的机器数量为x_1。要求制定一个五年计划，在每年开始时，决定如何重新分配完好的机器在两种不同的负荷下生产的数量，使在五年内产品的总产量达到最高。

机器负荷分配问题

例2 机器负荷分配问题

设机器在高负荷下生产的产量函数为$S_1=8u_1$,年折损率为$a=0.7$;

在低负荷下生产的产量函数为$S_2=5u_2$,年折损率为$b=0.9$。

开始生产时完好机器的数量$x_1=1000$台。按题意要安排好五年的生产计划，使产品的总产量最高。

机器负荷分配问题

问题的动态规划模型：

- 设阶段序数K表示年度。
- 状态变量x_k为第K 年度初拥有的完好机器数量，亦为第K-1年度末时的完好机器数量。
- 决策变量u_k为第K年度中分配高负荷下生产的机器数量。则x_k-u_k为该年度中分配在低负荷下生产的机器数量。
- 这里x_k和u_k均取连续变量。则u_k=0.3， 表示一台机器在该年度只有3/10的时间在高负荷下工作。

机器负荷分配问题

状态转移方程为

$$x_{k+1}= au_k+b(x_k-u_k)$$

$$=0.7u_k+0.9(x_k-u_k) \qquad k=1,2,\ldots,5$$

k段允许决策集合为$D_k(x_k)=\{u_k|0\leq u_k\leq x_k\}$

设$v_k(x_k, u_k)$为第K年度的产量，则

$$v_k = 8u_k+5（x_k-u_k）$$

故指标函数为

$$V_{1,5}= \sum_{k=1}^{5} v_k(x_k,u_k)$$

机器负荷分配问题

- 令$f_k(x_k)$表示由x_k出发采用最优分配方案到第5年度结束这段期间的产品产量。
- 根据最优化原理，则有递推关系式：

$$\begin{cases} f_6(x_6)=0 \\ f_k(x_k)=\max\limits_{u_k\in D_k(x_k)}\{8u_k+5(x_k-u_k)+f_{k+1}[0.7u_k+0.9(x_k-u_k)]\} \quad k=1,2,3,4,5 \end{cases}$$

- 逆序计算，得$u_k=\{0,0,x_3,x_4,x_5\}$,最高产量23700

动态规划应用举例

- 资源分配问题
- 生产与存储问题
- 复合系统工作可靠性问题
- 排序问题
- 设备更新问题

$$\begin{cases} \max[\, g_1(x_1)+g_2(x_2)+.....+g_n(x_n)] \\ x_1+x_2+.....+x_n=a \\ x_i \geq 0 \qquad i=1,2.....,n \end{cases}$$

$$\max P(z_1,z_2,.....,z_n)=\prod_{i=1}^{N} p_i(z_i) \qquad R=\left\{ z \mid \sum_{i=1}^{N} c_i z_i \leq c, \sum_{i=1}^{N} w_i z_i \leq w, z_i \geq 0 \right\}$$

排序问题

- 在加工工件所需经过的工序、各种工件在各道工序加工需要的时间为给定的条件下，如何确定加工工件的顺序，使得总的加工时间最短，这也可看作是一个多阶段决策问题。

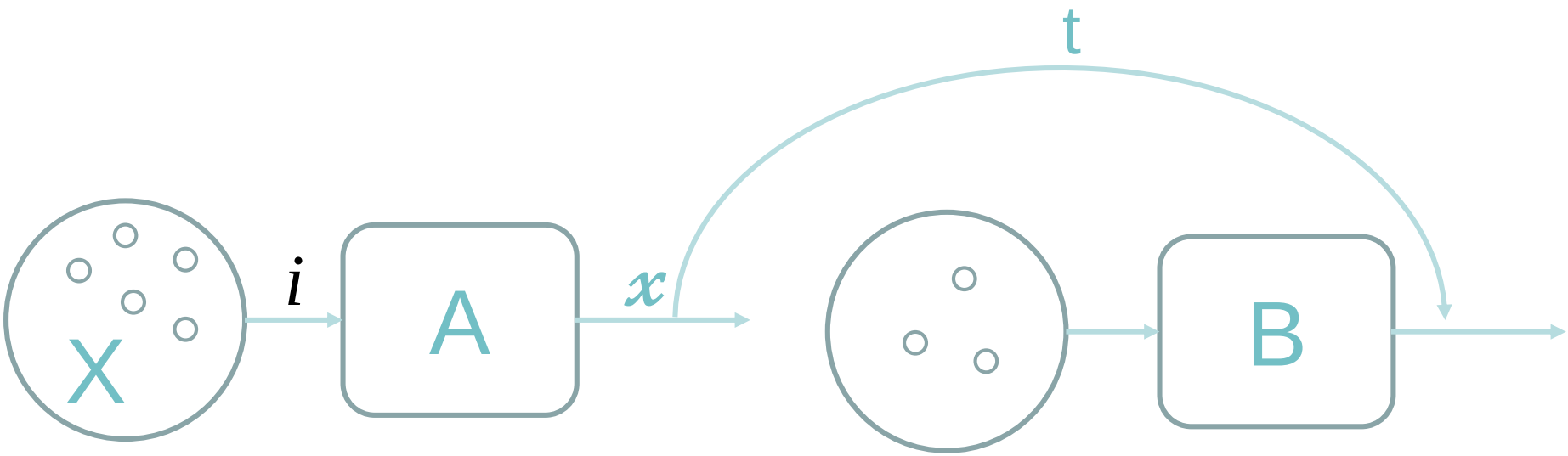
- 设有n个工件需要在机床A、B 上加工，每个工件都必须经过先A而后B的两道加工工序。以a_i、b_i分别表示工件 i $(1\leq i\leq n)$在A、B上的加工时间。问：

  应如何在两机床上安排各工件加工的顺序，使在机床A上加工第一个工件开始到在机床B上将最后一个工件加工完为止，所用的加工总时间最少？

排序问题

- 以在机床A上更换工件的时刻作为时段起止。以X表示在机床A上等待加工的按取定顺序排列的工件集合。以*x*表示不属于X的在A上最后加工完的工件。以*t*表示在A上加工完*x*的时刻算起到在B上加工完*x*所需的时间。这样，在A上加工完一个工件之后，就有（X，*t*）与之对应。

- 选取（X，*t*）作为描述机床A、B在加工过程中的状态变量，则当X包含有*s*个工件时，过程尚有*s*段，其时段数已隐含在状态变量之中。


t
X
i
A
x
B

排序问题

- 令f（X，t）为由状态（X，t）出发，对未加工的工件采取最优加工顺序后，将X中所有工件加工完所需时间。

  f（X，t, i）为由状态（X，t）出发，在A上加工工件i，然后再对以后的加工工件采取最优顺序后，把X中工件全部加工完所需要的时间。

f（X，t, i, j）为由状态（X，t）出发，在A上相继加工i与j后，对以后的加工工件采取最优顺序后，将X中工件全部加工完所需要的时间。

则

$$f(\mathrm{X},\ t, i) = a_i + f(X/i,\ t - a_i + b_i) \qquad \text{当} t \geq a_i \text{时}$$

$$a_i + f(X/i,\ b_i) \qquad \text{当} t \leq a_i \text{时}$$

- 记

$$z_i(t) = \max(t - a_i, 0) + b_i$$

上式就可合并写成

$$f(\mathrm{X},\ t, i) = a_i + f[X/i,\ z_i(t)]$$

其中X/i表示在集合X中去掉工件i后剩下的工件集合。

排序问题

- 由定义，可得

  f（X，t, i, j）$= a_i + a_j + f\,[X/\{\,i, j\}$，$z_{ij}\,(t)]$

  其中$z_{ij}\,(t)$是在机床A上从X出发相继加工工件i、j，并从它将j加工完的时刻算起，至在B上相继加工工件i、j并将工件加工完所需时间。
- 故（$X/\{\,i, j\}$，$z_{ij}\,(t)$）是在A加工i、j后所形成的新状态。即在机床A上加工i、j后由状态（X，t）转移到状态（$X/\{\,i, j\}$，$z_{ij}\,(t)$）。
- 仿照$z_i\,(t)$的定义， $z_i\,(t)$代替t，有

$$z_{ij}(t) = \max[\,z_i(t) - a_j, 0] + b_j$$

排序问题

$$z_{ij}(t) = \max[\max(t - a_i, 0) + b_i - a_j, 0] + b_j$$

$$= \max[\max(t - a_i - a_j + b_i, b_i - a_j), 0] + b_j$$

$$= \max[t - a_i - a_j + b_i + b_j, b_i + b_j - a_j, b_j]$$

- 将i、j对调，可得

  f（X，t, j, i）$= a_i + a_j + f[X/\{i, j\}, z_{ji}(t)]$

$$z_{ji}(t) = \max[t - a_i - a_j + b_i + b_j, b_i + b_j - a_i, b_i]$$

- 由于f（X，t）为t的单调上升函数，故当$z_{ij}(t) \le z_{ji}(t)$时，
  f（X，t, i, j）$\le f$（X，t, j，i）。

- 由此，对任意t，当$z_{ij}(t) \le z_{ji}(t)$ 时， 工件i放在工件j之前加工可以使总的结果时间短些。

排序问题

- 由$z_{ij}(t)$和$z_{ji}(t)$的表示可知，若

$$\max(b_i + b_j - a_j, b_j) \le \max(b_i + b_j - a_i, b_i)$$

成立时，就可推得

$$z_{ij}(t) \le z_{ji}(t)$$

- 将上式两边同减去b_i与b_j，得

$$\max(-a_j, -b_i) \le \max(-a_i, -b_j)$$

- 则有

$$\min(a_i, b_j) \le \min(a_j, b_i)$$

排序问题

以上条件就是工件i应该排在工件j之前的条件。因此，我们得到下面的最优排序规则：

1. 找出$a_1,a_2,\ldots\ldots,a_n,\ b_1,b_2,\ldots\ldots,b_n$中的最小数；
2. 若最小者为a_i，则将工件i排在第一位，并从工件集合中去掉这个工件；
3. 若最小者为b_i，则将工件i排在最后一位，并从工件集合中去掉这个工件；
4. 对剩下的工件重复上述手续，直至工件集合为空集时停止。

设备更新问题

- 在工业和交通运输企业中，经常碰到设备陈旧或损坏需要更新的问题：

  一种设备应该用多少年后进行更新为最恰当，即更新的合理年限是多少？

- 一辆卡车使用的情况：

  一辆新卡车第一年使用时，显然出车时间长，故障少，因而一年的运输收入额高，维修费支出较少。

  但使用几年后，必然出车时间较少，维修费用增加。

  到一定时候，需要更换；卖掉旧车，购买新车。但旧车愈旧，价值就越低，新旧相抵后更新所花的净费用支出就愈多。

  另外，设备的更新不能只看当年的收入和支出情况，而应看若干年总的收益效果来决定。

设备更新问题

- 令t为卡车已经使用过的年数。
- 设$r(t)$为已使用t年的卡车每年出车所得的运输收入额。它随t而减少，因车愈旧收入愈少。
- $u(t)$为已使用t年的卡车每年所需的维修费。它随t而增加，因车愈旧愈要维修。
- $c(t)$为更新一辆使用了t年的旧车，卖掉旧车，买进新卡车所需的净费用。则$c(t)$随t而上升。

- 如果某年初把使用了t年的旧车，再继续使用一年，则在这一年内所得回收额为：

$$g^{(K)}(t) = r(t) - u(t)$$

- 如果把使用了t年的车更换后买进一辆新车，则这一年所得的回收额为：

$$g^{(P)}(t) = r(0) - u(0) - c(t)$$

设备更新问题

- 但是，设备的更新不能只比较一年$g^K(t)$和$g^{(P)}(t)$的大小，而要看若干年总的收益效果，从N年内的总回收额中来决定那年更换旧车为好。为此，引入指标函数$f(t)$。
- $f(t)$：为一个以使用t 年的旧车，从某年开始在以后继续用到规定的N_0年止这几年内的总回收额。如果N=5， 就表示到N_0年止还有5年，$f(t)$就是在这5年内的总回收额。因此，我们有
- $f(t+1)$：为该车已使用了$t + 1$ 年后，继续用到N_0年止者几年内的总回收额。
- $f(1)$：为已使用了1年的车，继续用到N_0年这段期间中的总回收额。
- $f(0)$：为一辆新车，用到N_0年这段期间中的总回收额。
- 于是，指标函数$f(t)$的基本关系式为：
- $$f(t)=\max\begin{Bmatrix} P: r(0)-u(0)-c(t)+f(1) \\ K: r(t)-u(t)+f(t+1) \end{Bmatrix}$$

  其中，P表示更新，K表示保留使用。

设备更新问题

- r(t)、u(t)、c(t)一般为非线性函数，解析法求f(t)最优解很难；随车辆数目增多问题愈发复杂。
- 用动态规划方法，把问题变为多阶段决策过程。先定义指标函数$f_n(t)$为已经使用了t年的卡车，在第n年又继续使用，直到N_0年止这几年内所得的总回收额。
- 在第n年更新的车辆，满足递推关系为：

$$f_n^{(P)}(t) = r_n(0) - u_n(0) - c_n(t) + f_{n+1}(1)$$

- 若第n年不更新，继续使用，则有递推关系：

$$f_n^{(K)}(t) = r_n(t) - u_n(t) + f_{n+1}(t+1)$$

- 比较$f_n^{(P)}(t)$和$f_n^{(K)}(t)$的大小可得第n年的决策和指标函数：

$$f_n(t) = \max[f_n^{(P)}(t), f_n^{(K)}(t)] \qquad \text{当} n \geq N_0 + 1 \text{时，} f_n(t) = 0$$

设备更新问题

- 以上递推关系说明了第n年使用的车辆，在以后若干年的总回收额的大小，它决定于第$n+1$年的指标函数。

- 从而，为求出$f_n(t)$，必须从继续使用的最后一年（N_0年）向回计算，即为逆序算法。

- $f_1(t)$即为使用了t年的旧车，按最优策略逐年使用或更新，从第1年到N_0年的总回收额。

可基于动态规划思想求解的

问题与算法

计算二项式系数

- 二项式系数， 记作C（n, k），等于 一个n元素集合的k元素组合（子集）的数量（0≤k≤n）。
- “二项式系数”这个名字来源于这些数字出现在二项式公式之中：

$$(a+b)^n = C(n,0)a^n + \ldots + C(n,i)a^{n-i}b^i + \ldots + C(n,n)b^n$$

且有如下特性：当n>k>0时

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

以及 $C(n,0) = C(n,n) = 1$

计算二项式系数

- 可以用动态规划技术来对它求解。把二项式系数记录在一张n+1行k+1列的表中，行从0到n 计数，列从0到k计数。
- 为了计算C（n,k），逐行填充表格，从行0开始，到行n结束。
- 每一行（0≤i≤n）从左向右填充，第一个数字填1，因为C（n,0）=1。
- 行0直到行k也是以1结束在表的主对角线上：当0≤i≤k时，C(i,i)=1。
- 用上述公式计算其他单元格的值，即把前一行前一列那个单元格的值和前一行当前列那个单元格的值相加。（帕斯卡三角形）

计算二项式系数

- 算法*Binomial(n,k)*
- 　　//用动态规划算法计算C(n,k)
- 　　//输入：一对非负整数n≥k≥0
- 　　//输出：C(n,k)的值

- 　　**for** i ←0 **to** n **do**
- 　　　　**for** j←0 **to** min(i,k) **do**
- 　　　　　　**if** j= 0 **o**r j=i
- 　　　　　　　　C[i,j] ←1
- 　　　　　　**else** C[i,j] ←C[i-1,j-1]+C[i-1,j]
- 　　**return** C[n,k]

计算二项式系数

- 该算法的基本操作是加法，所以把A（n,k）记作该算法在计算C(n,k)时所作的加法总次数。
- 因为表格的前k+1行构成了一个三角形，而余下的n-k行构成了一个矩形，将求和表达式A（n,k）分成两个部分,有：

$$A(n,k)=\sum_{i=2}^{k}\sum_{j=1}^{i-1}1+\sum_{i=k+1}^{n}\sum_{j=1}^{k}1=\sum_{i=2}^{k}(i-1)+\sum_{i=k+1}^{n}k$$

$$=\frac{(k-1)k}{2}+k(n-k)\in\Theta(nk)$$

最长公共子序列

- 一个给定序列的子序列是在该序列中删去若干元素后得到的序列。
- 存在一个严格递增下标序列。

  例如，序列Z= {B，C，D，B}是序列X= {A，B，C，B，D，A，B}的子序列， 相应的递增下标序列为{2，3，5，7}。
- 给定两个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。
- 例如， 若X= {A，B，C，B，D，A，B}，Y= {B，D，C，A，B，A}则序列{B，C，A}是X和Y的一个公共子序列。
- 最长公共子序列问题：给定两个序列X= {x_1, x_2, …, x_m}和Y= {y_1, y_2, …, y_n}， 找出X和Y的一个最长公共子序列。

最长公共子序列

- 蛮力搜索的方法：

  令$A= a_1a_2\ldots a_n$和$B= b_1b_2\ldots b_m$。例举A所有的2^n个子序列， 对于每一个子序列在$\Theta(m)$ 时间内来确定它是否也是B的子序列。此方法的时间复杂性是$\Theta(m2^n)$。

- 动态规划：

  寻找一求最长公共子序列长度的递推公式：

  令$L[i, j]$表示$a_1a_2\ldots a_i$ 和$b_1b_2\ldots b_j$的最长公共子序列的长度。（i 和j 可能是0，即，$a_1a_2\ldots a_i$和$b_1b_2\ldots b_j$中的一个或同时可能为空字符串。）-----如果i=0或j=0, 则$L[i, j]$=0。

最长公共子序列

观察结论：如果i 和j 都大于0， 那么

- 若$a_i = b_j$，$L[i, j]= L[i\text{-}1, j\text{-}1] + 1$；
- 若$a_i \neq b_j$，$L[i, j] = \max\{ L[i, j\text{-}1], L[i\text{-}1, j]\}$。
- 则计算 A和B的最长公共子序列长度的递推式为：

$$L[i, j] = \begin{cases} 0 & \text{若i=0或j=0} \\ L[i\text{-}1, j\text{-}1] + 1 & \text{若i>0, j>0 和} a_i = b_j \\ \max\{ L[i, j\text{-}1], L[i\text{-}1, j]\} & \text{若i>0, j>0 和} a_i \neq b_j \end{cases}$$

- 用动态规划技术求解最长公共子序列的问题。对于每一对i 和j 的值，$0 \le i \le n$和$0 \le j \le m$, 我们用一个（n+1）x (m+1) 表来计算$L[i, j]$的值，只需要用上面的公式逐行地填表$L[0\ldots n, 0\ldots m]$。

最长公共子序列

算法LCS
输入：字母表上的两个字符串A和B，长度分别为n 和m。
输出：A和B最长公共子序列的长度。

```
1.  for i←0 to n
2.          L[i, 0] ←0
3.  for j←0 to m
4.          L[0, j] ←0
5.  for i←1 to n
6.          for j←1 to m
7.                  if a_i = b_j then L[i, j] ←L[i-1, j-1] + 1
8.                  else L[i, j] ←max{ L[i, j-1], L[i-1, j]}
9.                  end if
10.         end for
11. end for
12. return L[n, m]
```

算法复杂性正好是表的大小$\Theta(nm)$

矩阵链相乘

- 假设要用标准的矩阵乘法来计算M_1，M_2，M_3 三个矩阵的乘积$M_1M_2M_3$，这三个矩阵的维数分别是 2 x 10, 10 x 2 和 2 x 10。

- 若先把M_1和M_2相乘，然后把结果和M_3相乘，那么要进行2 x 10 x 2 +2 x 2 x 10 = 80次乘法；
- 代之以用M_2和M_3先相乘，则乘法的次数就变成了 10 x 2 x 10 + 2 x 10 x 10 = 400；
- 执行乘法M_1（M_2M_3）耗费的时间是执行乘法（M_1M_2）M_3的5倍。

矩阵链相乘

- 一般化地，顺序数等于乘这n个矩阵时用每一种可能的途径放置括弧的方法数。
- 设$f(n)$是求n个矩阵乘积的所有放置括弧的方法数，假定要进行以下的乘法

$$（M_1 M_{2\ldots} M_{\mathrm{k}}）\mathrm{x}\ (M_{\mathrm{k+1}} M_{\mathrm{k+2}\ \ldots} M_{\mathrm{n}})$$

- 则，对于前k个矩阵有$f(k)$种方法放置括弧。对于$f(k)$中的每一种方法，可对余下的$f(n\text{-}k)$个矩阵放置括弧，总共有$f(k)\ f(n\text{-}k)$种方法。由于可以假设k是1到n-1中的任意值，对于n个矩阵放置括弧的所有方法数由下面的和式给出

$$f(n)=\sum_{k=1}^{n-1} f(k)\,f(n-k)$$

矩阵链相乘

- 两个矩阵相乘只有一种方法，三个矩阵相乘有两种方法。因此f（2）=1，f（3）=2。为了使递推式有意义，令$f(1) = 1$。可以证明

$$f(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

- 由Stirling公式：

$$n! \approx \sqrt{2\pi n}(n/e)^n$$

其中e=2.718 28…

有

$$f(n) = \frac{1}{n}\binom{2n-2}{n-1} = \frac{(2n-2)!}{n((n-1)!)^2} \approx \frac{4^n}{4\sqrt{\pi}n^{1.5}}$$

- 因此 $f(n) = \Omega\left(\frac{4^n}{n^{1.5}}\right)$

矩阵链相乘

- 应用动态规划技术可以找出一个有效算法来计算以上递推式

- 由于对于每个i， $1 \le i < n$，矩阵M_i的列数一定等于矩阵M_{i+1}的行数，因此指定每个矩阵的行数和最右面矩阵M_n的列数就足够表示矩阵链规模。我们假设r_i和r_{i+1}分别是矩阵M_i中的行数和列数，$1 \le i \le n$。

- 用$M_{i,j}$表示$M_i M_{i+1} \ldots M_j$的乘积，并假设链$M_{i,j}$的乘法的耗费用数量乘法的次数来测度， 记为$C[i, j]$。

矩阵链相乘

- 对于给定的一对索引i和j，　$1 \le i < j \leqslant n$，　$M_{i,\ j}$可用如下方法计算：
- 设k是$i+1$和j之间的一个索引，计算两个矩阵

  $M_{i,\ k-1}=M_i M_{i+1} \ldots M_{k-1}$和$M_{k,\ j}=M_k M_{k+1} \ldots M_j$,

  则$M_{i,\ j}=M_{i,\ k-1} M_{k,\ j}$。
- 显然，用这种方法计算$M_{i,\ j}$的总耗费, 是

  计算$M_{i,\ k-1}$的耗费+计算$M_{k,\ j}$的耗费+$M_{i,\ k-1}$乘上$M_{k,\ j}$的耗费（$r_i r_k r_{j+1}$）。
- 从而有以下找出k值的公式，其中的k使执行矩阵乘法$M_i M_{i+1} \ldots M_j$所需要的数量乘法次数最小。

矩阵链相乘

$$C[i,j]=\min_{i<k\le j}\{C[i,k-1]+C[k,j]+r_i r_k r_{j+1}\}$$

由此，为了找出执行矩阵乘法$M_1 M_{2\ldots} M_n$所需要数量乘法的最小次数， 只需要解递推式

$$C[1,n]=\min_{1<k\le n}\{C[1,k-1]+C[k,n]+r_1 r_k r_{n+1}\}$$

矩阵链相乘

算法MATCHAIN

输入：n个矩阵的链的维数对应于正整数数组$r[1\ldots n+1]$, 其中$r[1\ldots n]$是n个矩阵的行数，$r[n+1]$是M_n的列数。

输出：n个矩阵相乘的数量乘法的最小次数。

- 1. for i←1 to n　　{填充对角线d_0}
- 2.　　$C[i,i] \leftarrow 0$
- 3. end for
- 4. for $d \leftarrow 1$ to n-1　　{填充对角线d_1到d_{n-1}}
- 5.　　for $i \leftarrow 1$to n-d　　{填充对角线d_i的项目}
- 6.　　$j \leftarrow i+d$　　{对角线上第i行元素的列数}
- 7.　　$C[i,j] \leftarrow \infty$
- 8.　　for $k \leftarrow i+1$ to j　　{i，j之间的索引}
- 9.　　$C[i,j] \leftarrow \min\{ C[i,j], C[i,k-1] + C[k,j] + r[i]r[k]r[j+1]\}$
- 10.　　end for
- 11.　　end for
- 12. end for
- 13. return $C[1,n]$

$\Theta(n^3)$

所有点对的最短路径问题

- 设G=（V，E）是一个有向图，其中的每条边（i, j）有一个非负的长度$l[i, j]$, 如果从顶点i 到顶点j没有边，则$l[i, j]=\infty$。

- 问题：找出从每个顶点到其他所有顶点的距离（从顶点x到顶点y的距离是指从x到y 的最短路径的长度）。

- 我们假设V ={1，2，…,n}，设i和j 是V中两个不同的顶点，定义$d^k_{i,j}$是从i到j， 并且不经过{k+1, k+2,…,n}中任何顶点的最短路径的长度。

所有点对的最短路径问题

- 例如$d^0_{i,j}=l[i,j]$，$d^1_{i,j}$是从i到j，除了可能经过顶点1以外，不经过任何其他顶点的最短路径，$d^2_{i,j}$是从i到j，除了可能经过顶点1、顶点2或同时经过它们以外，不经过任何其他顶点的最短路径，等等。
- 则$d^n_{i,j}$是从i到j的最短路径长度，也就是从i到j的距离。给出这个定义，可以递归地计算$d^k_{i,j}$如下

$$d^k_{i,j}=l[i,j] \qquad \text{若} k=0$$

$$=\min\{d^{k-1}_{i,j}, d^{k-1}_{i,k}+ d^{k-1}_{k,j}\} \qquad \text{若} 1\leqslant k\leqslant n$$

所有点对的最短路径问题

- Floyd算法，用自底向上地解以上递推式的方法来处理。它用n+1 个n x n维矩阵D_0，D_1，…,D_n来计算最短约束路径的长度。
- 开始时，如果$i \neq j$ 并且（i ,j）是G中的边，则置D_0 [i,i] = 0, D_0 [i,j] = l[i,j]; 否则置D_0 [i,j] =∞。
- 然后做n次迭代，使在第k次迭代后，D_k [i,j]含有从顶点i到顶点j，且不经过编号大于k的任何顶点的最短路径的长度。这样在第k次迭代中，可以用公式

$$D_k[i, j] = \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]$$

计算D_k [i,j]。

**在第K次迭代中，第K行和第K列均不变，且其它元素只用K行列，故D不需做副本*

所有点对的最短路径问题

算法FLOYD

输入：n x n维矩阵$l[1\ldots n,1\ldots n]$, 对应于有向图$G=$（$\{1,2,\ldots,n\},E$）中的边（i,j）的长度为$l[i,j]$。

输出：矩阵D, 使得$D[i,j]$等于i到j的距离。

```
1.  D←l                    {将输入矩阵l复制到D}
2.  for k←1 to n
3.          for i←1 to n
4.                  for j←1 to n
5.                          D[i,j]=min{D[i,j],D[i,k] +D[k,j]}
6.                  end for
7.          end for
8.  end for
```

显然，算法的运行时间是 $\Theta(n^3)$. 图例？

0/1 背包问题

- 0/1 背包问题可以定义如下：

  设$U=\{u_1,u_2,\ldots,u_n\}$是一个准备放入容量为C的背包中的n项物品的集合。对于$1\le j\le n$,令s_j和v_j分别为第j项物品的体积和价值，C, s_j, v_j 都是正整数。

  要解决的问题是用U中的一些物品来装背包，这些物品的总体积不超过C，然而要使它们的总价值最大。假设每项物品的体积不大于C，给出有n项物品的U，我们要找出一个子集合$S\subseteq U$，使得

  $$\sum_{u_i\in S} v_i$$

  在约束条件 $\sum_{u_i\in S} s_i \le C$ 下最大。

0/1 背包问题

- 为导出递归公式，

  设$V[i,j]$ 表示从前i项$\{u_1, u_2, \ldots, u_i\}$中取出来的装入体积为$j$的背包的物品的最大价值。这里，$i$的范围是从0到$n$, j的范围是从0到C。
- 则，要寻求的是值$V[n,C]$。
- 有$V[0,j]$对于所有j的值是0，当背包中没有物品。

  $V[i,0]$对于所有i的值为0，当没有物品可放到容积为0的背包里。

0/1 背包问题

当*i*和*j*都大于0时，有以下的结论：

$V[i,j]$是下面两个量的最大值：

- $V[i-1,j]$:仅用最优的方法取自$\{u_1, u_2, \ldots, u_{i-1}\}$的物品去装入体积为$j$的背包所得到的价值最大值。
- $V[i-1,j-s_i]+v_i$:用最优的方法取自$\{u_1, u_2, \ldots, u_{i-1}\}$的物品去装入体积为$j-s_i$的背包所得到的价值最大值加上物品$u_i$的价值。这仅应用于如果$j \geq s_i$以及它等于把物品$u_i$加到背包上的情况。

- 观察结论对于找出最优装背包时的值，有下面的递推式

$$
\begin{aligned}
V[i,j] &= 0 && \text{若 } i=0 \text{ 或 } j=0 \\
&= V[i-1,j] && \text{若 } j<s_i \\
&= \max\{V[i-1,j], V[i-1,j-s_i]+v_i\} && \text{若 } j \geq s_i
\end{aligned}
$$

0/1 背包问题

- 现在可以很简单地用动态规划来求解这个整数规划问题。

- 用一个$(n+1)$ x $(C+1)$的表来计算$V[i, j]$的值，只需利用上面的公式逐行地填表$V[0\ldots n, 0\ldots C]$即可。

0/1 背包问题

算法**KNAPSACK**
输入：物品集合$U=\{u_1, u_2, \ldots, u_n\}$，体积分别为$v_1, v_2, \ldots, v_n$，容量为$C$的背包。
输出：$\sum_{u_i \in S} v_i$ 的最大总价值，且满足 $\sum_{u_i \in S} s_i \le C$ ，其中 $S \subseteq U$。

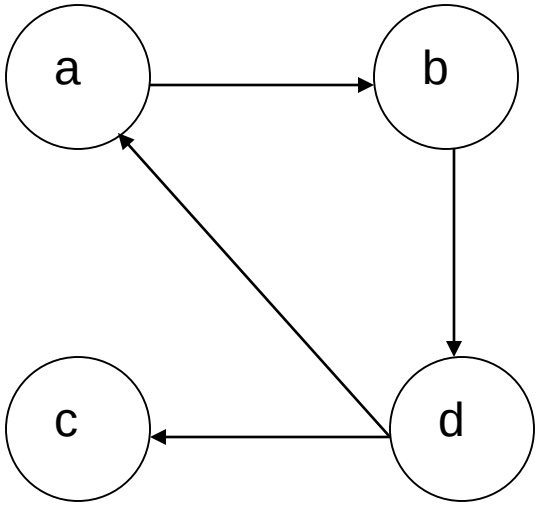
```
1.  for i←0 to n
2.          V[i, 0] ←0
3.  end for
4.  for j←0 to C
5.          V[0, j] ←0
6.  end for
7.  for i←1 to n
8.          for j←1 to C
9.                  V[i, j] ←V[i-1, j]
10.                 if s_i≤j then V[i, j] ←Max{V[i, j],V[i-1, j-s_i]+v_i}
11.         end for
12. end for
13. return V[n, C]
```

$\Theta(nc)$

计算有向图的传递闭包

- 定义
  一个n顶点有向图的传递闭包可以定义为一个n阶布尔矩阵T= $\{t_{ij}\}$;
  如果从第i个顶点到第j个顶点之间存在一条有效的有向路径，矩阵第i行（1≤i≤n）第j列（1≤j≤n）的元素为1;
  否则，t_{ij}为0。

- 可以通过深度优先查找和广度优先查找生成有向图的传递闭包;
  但对同一个有向图遍历了多次（需以每个顶点为起点做一次遍历）。



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Warshall 算法

- Warshall算法通过一系列n阶布尔矩阵来构造一个给定的n个顶点有向图的传递闭包：

  $R^{(0)},\ldots,R^{(k-1)},R^{(k)},\ldots,R^{(n)}$ （1）
- 当且仅当，从第i个顶点到第j个顶点之间存在一条有向路径（长度大于0）并且路径的每一个中间顶点的编号不大于k时，矩阵$R^{(k)}$的第i行第j列的元素$r_{ij}^{(k)}$的值等于1。
- 这一系列矩阵从$R^{(0)}$ 开始，这个矩阵不允许它的路径中包含任何中间顶点；所以$R^{(0)}$ 就是有向图的邻接矩阵。
- $R^{(1)}$ 包含允许使用第一个顶点作为中间顶点的路径信息；每一个后继矩阵相对它的前趋来说，都允许增加一个顶点作为其路径上的顶点。
- 序列中的最后一个矩阵，反映了能够以有向图的所有n个顶点作为中间顶点的路径，因此它就是有向图的传递闭包。

Warshall 算法的主要思想

- 任何R（k）中的所有元素都可以通过它在序列（1）中的直接前趋R（k-1）计算得到。把矩阵R（k）中第i行第j列的元素$r_{ij}^{(k)}$置为1意味着存在一条从第i个顶点到第j个顶点的路径，路径中每一个中间顶点的编号都不大于k：

  v_i，每个顶点编号都不大于k的中间顶点列表，v_j　(2)

- 对于路径（2），有两种可能：

  1、路径的中间顶点列表中不包含第k个顶点，那么这条从v_i到v_j的路径中顶点的编号不会大于k-1，所以$r_{ij}^{(k-1)}$也等于1。

  2、路径的中间顶点$r_{ij}^{(k-1)}$ 的确包含第k个顶点v_k。

Warshall 算法

- 假设v_k在列表中只出现一次，路径（2）可以改写成以下形式：

  （若不只一次，只要简单地把路径中第一个v_k和最后一个v_k之间的顶点全部消去，就可以创建一条从v_i到v_j的新路径）

  v_i，编号≤k-1的顶点，v_k，编号≤k-1的顶点，v_j

- 这表明：

  存在一条从v_i到v_k的路径，路径中每个中间顶点的编号都不大于k-1（因此$r_{ik}^{(k-1)}=1$）；

  存在一条从v_k到v_j的路径，路径中每个中间顶点的编号也都不大于k-1（因此$r_{kj}^{(k-1)}=1$）。

Warshall 算法

- 上述说明：
  如果$r_{ij}^{(k)}=1$， 则：
  $\boldsymbol{r_{ij}^{(k-1)}=1}$**，或** $\boldsymbol{r_{ik}^{(k-1)}=1}$**且** $\boldsymbol{r_{kj}^{(k-1)}=1}$。

- 其逆命题也成立。因此，对于如何从矩阵$R^{(k-1)}$的元素中生成矩阵$R^{(k)}$的元素，我们有下面的公式：

$$r_{ij}^{(k)}= r_{ij}^{(k-1)} \ \textbf{or} \ \ r_{ik}^{(k-1)} \ \ \textbf{and} \ \ r_{kj}^{(k-1)}$$

- 该公式意味着以下规则：

  如果一个元素r_{ij}在$R^{(k-1)}$中是1，它在$R^{(k)}$中仍然是1。
  如果一个元素r_{ij}在$R^{(k-1)}$中是0，当且仅当矩阵中第i行第k列的元素和第k行第j列的元素都是1，该元素在$R^{(k)}$中才能变成1。（如图）

  * 例

Warshall 算法

算法*Warshall*(*A*[1..n,1..n])
//实现计算传递闭包的Warshall算法
//输入：包括n个顶点有向图的邻接矩阵A
//输出：该有向图的传递闭包

$R^{(0)} \leftarrow A$

```
for k←1 to n do
    for i←1 to n do
        for j←1 to n do
            R(k)[i, j] ←Rk-1[i, j]or Rk-1[i,k]and Rk-1[k, j]
return R(n)
```

$\Theta(n^3)$

最优二叉查找树

- 最优二叉查找树：在查找中的平均键值比较次数是最低的。

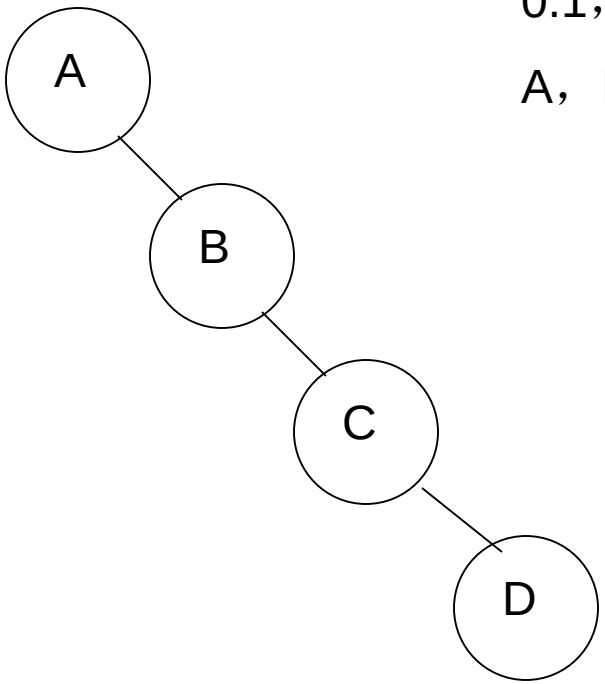
  （集合中元素的查找概率是已知的）

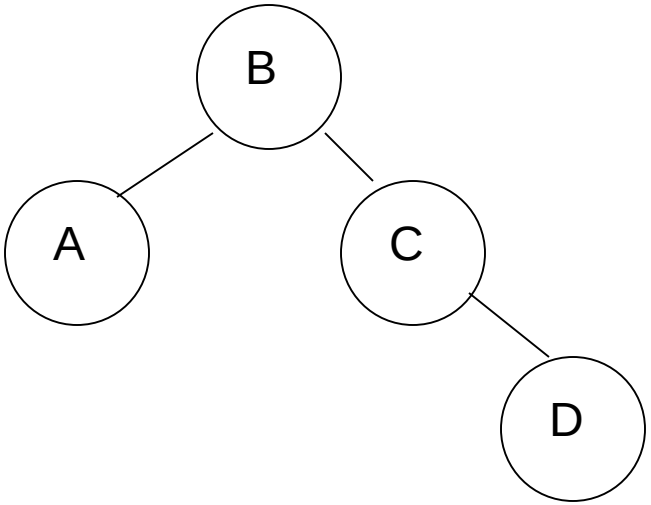
- 例：分别以概率0.1，0.2，0.4，0.3来查找4个键A，B，C，D。
- 在成功查找时，第一棵树的平均键值比较次数为2.9， 而第二棵树是2.1。

最优二叉查找树

0.1，0.2，0.4，0.3

A，B，C，D





0.1·1+0.2·2+0.4·3+0.3·4=2.9

0.1·2+0.2·1+0.4·2+0.3·3=2.1

最优二叉查找树

- 最优二叉查找树：在查找中的平均键值比较次数是最低的。（集合中元素的查找概率是已知的）

- 例：分别以概率0.1，0.2，0.4，0.3来查找4个键A，B，C，D。
- 在成功查找时，第一棵树的平均键值比较次数为2.9， 而第二棵树是2.1。

- 构造*的穷举方法是不现实的：包含n个键的二叉查找树的总数量等于第n个Catalan数，它以$4^n/n^{1.5}$的速度逼近无穷大。(1,1,2,5,14,42,132,429,1430,4862,16796…)

最优二叉查找树

- 设$a_1,\ldots a_n$ 是从小到大排列的互不相等的键，$p_1,\ldots\ldots,p_n$是它们的查找概率。T_i^j是由键$a_i,\ldots,a_j$构成的二叉树，C[i,j]是在这棵树中成功查找的最小的平均查找次数，其中，i,j 是一些整数下标， $1\le i\le j\le n$.
- 为了推导出动态规划算法中隐含的递推关系，需要考虑从键$a_i,\ldots,a_j$中选择一个根a_k的所有可能的方法。
- 以a_k 为根的左子树T_i^{k-1}和右子树T_{k+1}^{j}中的键$a_i,\ldots,a_{k-1}$ 和$a_{k+1},\ldots,a_j$也应分别是最优排列的。

最优二叉查找树

- 若从1开始对树的层数进行计数，有下列递推关系式(顶点层数为0)：

（设α_s为a_s在T_i^{k-1}中的层数，β_s为a_s在T^j_{k+1}中的层数）

$$C[i,j]=\min_{i\le k\le j}\{p_k\cdot 1+\sum_{s=i}^{k-1}p_s\cdot(\alpha_s+1)+\sum_{s=k+1}^{j}p_s\cdot(\beta_s+1)\}$$

$$=\min_{i\le k\le j}\{p_k+\sum_{s=i}^{k-1}p_s\cdot\alpha_s+\sum_{s=i}^{k-1}p_s+\sum_{s=k+1}^{j}p_s\cdot\beta_s+\sum_{s=k+1}^{j}p_s\}$$

$$=\min_{i\le k\le j}\{\sum_{s=i}^{k-1}p_s\cdot\alpha_s+\sum_{s=k+1}^{j}p_s\cdot\beta_s+\sum_{s=i}^{j}p_s\}$$

$$=\min_{i\le k\le j}\{C[i,k-1]+C[k+1,j]\}+\sum_{s=i}^{j}p_s$$

最优二叉查找树

- 因此有下列递推关系式：

当1≤i≤j≤n 时

$$C[i,j]=\min_{i\le k\le j}\{C[i,k-1]+C[k+1,j]\}+\sum_{s=i}^{j}p_s$$

当1≤i≤n+1 时，　　c[*i,i-1*]=0; empty tree

当1≤i≤n 时，　c[*i,i*]=p_i ； single node

最优二叉查找树

算法*OptimalBST(P*[1..n])
//用动态规划求最优二叉查找树
//输入：一个n个键的有序列表的查找概率数组P[1..n]
//输出：在最优BST中成功查找的平均比较次数，以及最优BST中子树的根表R

for *i*← 1 **to** *n* **do**
    C[*i*,*i*-1] ←0
    C[*i*,*i*] ←*P*[*i*]
    R[*i*,*i*] ←*i*
C[*n*+1,*n*] ←0
for *d*←1 **to** *n*-1 **do** //对角线计数
    for *i*←1**to** *n-d* **do**
        j←*i*+*d*
        minval←∞
        f**or** *k*←*i* to *j* **do**
            if *C*[*i*,*k*-1]+*C*[*k*+1,*j*]<*minval* ***then*** *minval*←*C*[*i*,*k*-1]+C[k+1,j];*kmin*←*k*
        R[*i*,*j*] ←*kmin*
        s*um*←*P*[*i*];
        for *s*←*i*+1 **to** *j* **do** *sum*←*sum*+*P*[*s*]
        C[*i*,*j*] ←*minval*+*sum*
return *C*[1,*n*], *R*
Figure？ ; O(n^3*)*

函数迭代与策略迭代

- 阶段数为n 的最短路线问题是一个定期的多阶段决策过程。

- 阶段不固定的最短路线问题？

- **问题**：设有N个点：1，2，……，N。任两点i、j之间有一弧连接，其长度为c_{ij}，$0 \leq c_{ij} \leq \infty$。设$N$为固定点，求任一点$i$至固定点$N$的最短路线。

函数迭代法

- 函数迭代法的基本思想：

  以段数（步数）作为参变数，先求在各个不同段数下的最优策略，然后从这些最优解中再选出最优者，从而同时确定了最优段数。

- 步骤：

  （1）先选定一初始函数$f_1(i)$:

  $f_1(i)=c_{iN}$ $\quad i=1,2,\ldots\ldots,N-1$

  $f_1(N)=0$ $\quad i=N$

  (2) 然后用下列递推关系求出$\{f_k(i)\}$。定义

  $f_k(i)=\min_j[c_{ij}+f_{k-1}(j)]$ $\quad i=1,2,\ldots,N-1$

  $f_k(N)=0$ $\quad k>1$

  这里$f_k(i)$表示由i点出发朝固定点走k步后的最短路线（不一定到达点N）。K增大时，$f_k(i)$逼近最优函数$f(i)$。

策略迭代法

- 策略迭代法的基本思想是：

  先给出初始策略$u_0(i)$

  (即$\{u_0(i)\}$, i=1,2,….,N-1);

  然后按某种方式求得新策略$u_1(i)$, $u_2(i)$,……,直至最终求出最优策略。

  若对某一k，$u_k(i) = u_{k-1}(i)$ 对所有i成立，则称策略收敛，此时$\{u_k(i)\}$就是最优策略。

策略迭代法的步骤

（1）先选出一无回路的初始策略$\{u_0(i)\}$ i=1,2,……,N-1。$u_0(i)$表示在此策略下由i点到达的下一个点。

（2）由策略$u_k(i)$求出指标值函数$f_k(i)$。

即由方程组：

$$f_k(i) = c_{i,u_k(i)} + f_k[u_k(i)] \qquad i=1,2,\ldots,N-1$$

$$f_k(N)=0 \qquad k=0,1\ldots$$

解出$f_k(i)$。其中 $c_{i,u_k(i)}$ 为已知。

（3）由指标值函数$f_k(i)$求策略$u_{k+1}(i)$，其$u_{k+1}(i)$是$\min_u\{c_{i,u}+f_k(u)\}$的解。

（4）按（2）、（3）步反复迭代，可逐次求得$\{u_k（i）\}$和$\{f_k(i)\}$。一直找到有某一k, 使 $u_k（i）=u_{k-1}（i）$对所有i成立。则$\{u_k（i）\}$就是最优策略，其相应$\{f_k(i)\}$为最优值。

回溯

回溯法

- 任何难问题，均可通过穷尽搜索数量巨大但有限多个可能性而获得一个解。
- 大多数难问题都不存在用穷尽搜索之外的方法来解决问题的算法。
- 产生了开发系统化的搜索技术的需要，并且希望能够将搜索空间减少到尽可能的小。
- 组织搜索的一般技术之一是回溯法。这种算法设计技术可以被描述为**有组织的穷尽搜索**，它常常可以避免搜索所有的可能性。

回溯法的基本思想

- 针对所要做的选择构造一棵所谓的状态空间树，树的每一层节点代表了对解的每一个分量所做的选择。
- 用深度优先法搜索状态空间树。
- 在状态空间树中的任一节点，满足一定条件的情况下，搜索回溯。

3 着色问题

- 给出一个无向图$G=（V，E）$，需要用三种颜色之一为V中的每个顶点着色，三种颜色分别为1，2和3，使得没有两个邻接的顶点有同样的颜色。我们把这样的着色称为合法的；否则，如果两个邻接的顶点有同一种颜色就是非法的。

- 一种着色可以用n元组$\{c_1,c_2,\ldots, c_n\}$来表示，使$c_i \in \{1,2,3\}$, $1 \le i \le n$。
  例如，（1，2，2，3，1）表示一个有5个顶点的图的着色。

3 着色问题

- 一个n个顶点的图共有3^n种可能的着色，所有可能的着色的集合可以用一棵完全的三叉树来表示，称为搜索树。
- 在这棵树中，从根到叶子节点的每一条路径代表一种着色指派。

3 着色问题

- 回溯法依次生成搜索树的一个节点。
- 如果从根到当前节点的路径对应于一个合法着色，且图中所有点均已着色，过程终止（除非期望找到不止一种的着色）。
- 如果这条路径的长度小于n，并且相应的着色是合法的（部分解），那么就生成现节点的一个子节点，并将它标记为现节点。
- 如果对应的路径是非法的，那么现节点标记为死节点并生成对应于另一种颜色的新节点。
- 如果所有三种颜色都已经试过且没有成功，搜索就回溯到父节点，它的颜色被改变，依次类推。

3 着色问题

- 例

3 着色问题

算法 3-COLORREC
输入：无向图*G*=（*V*，*E*）。
输出：*G*的顶点的3着色*c*[1…*n*], 其中每个*c*[*j*]为1，2或3。

1. **for** *k*←1 **to** *n*
2. *c*[*k*] ←0
3. **end for**
4. *flag*←**false**
5. *graphcolor*(1)
6. **if** *flag* **then output** *c*
7. **else output** “no solution”

Procedure *graphcolor*(*k*)
1. **for** *color* = 1 **to** 3
2. *c*[*k*] ←*color*
3. **if** *c* 为原问题的合法解 **then** *flag*←**true** and **exit**
4. **else if** *c* 是部分解 **then** *graphcolor*(*k* + 1)
5. **end for**

3 着色问题

算法 3-COLORITER
输入：无向图$G=（V，E）$。
输出：G的顶点的3着色$c[1…n]$, 其中每个$c[j]$为1，2或3。

```
1.  for k←1 to n
2.          c[k] ←0
3.  end for
4.  flag←false
5.  k←1
6.  while k≥1          //点、层、分量
7.          while c[k] ≤2          //色、分量取值
8.                  c[k] ←c[k]+1
9.                  if c 为原问题的合法解 then flag←true ；从两个while 循环退出
10.                 else if c 是部分解 then k←k + 1          {前进}
11.         end while
12.          c[k] ← 0
13.          k←k-1    {回溯}
14. end while
15. if flag then oupput c
16. else output “no solution”
```

3 着色问题

- $T_{worst} = O(n3^n)$

- 3^n节点
- n合法

8皇后问题

- 经典的8皇后问题：如何在8 x 8的国际象棋棋盘上安排8个皇后，使得没有两个皇后能互相攻击？

  (如果两个皇后处在同一行、同一列或同一条对角线上，则她们能互相攻击。）

- N皇后：由于没有两个皇后能放在同一行或列上，则解向量必须是数1，2，…,*n*的一个排列。这样，蛮力方法可以由 n^n 改进为测试 $n!$ 种布局 。

4皇后问题之回溯

- 回溯法求解4皇后问题：
- 生成并以深度优先方式搜索一棵完全四叉树， 树的根对应于没有放置皇后的情况。
- 第一层的节点对应于皇后在第一行的可能放置情况，第二层的节点对应于皇后在第二行的可能放置情况， 依次类推。
- 每种可能的布局可用一有4个分量的向量 $x=\{x_1, x_2, x_3, x_4\}$来描述。 x_i的 值为棋盘上第i行的皇后所处的列。
- 放在位置x_i和x_j的两个皇后当且仅当$x_i=x_j$时处在同一列上，不难看出两个皇后处在同一条对角线上当且仅当

$$x_i - x_j = i - j \text{ 或 } x_i - x_j = j - i$$

4皇后问题

算法 4-QUEENS
输入：空。
输出：对应于4皇后问题的解的向量$x[1\ldots4]$。

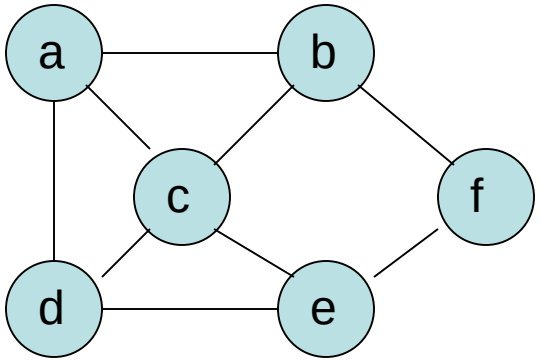
```
1   for k←1 to 4
2       x[k] ←0    {棋盘上无皇后}
3   end for
4   flag←false
5   k←1
6   while k≥1
7       while x [k] ≤3
8           x [k] ←x [k]+1
9           if x 为原问题的合法解 then flag←true 且从两个while 循环退出
10          else if x 是部分解 then k←k + 1 {前进}
11      end while
12      x [k] ←0
13      k←k-1      {回溯}
14  end while
15  if flag then oupput x
16  else output “no solution”
```

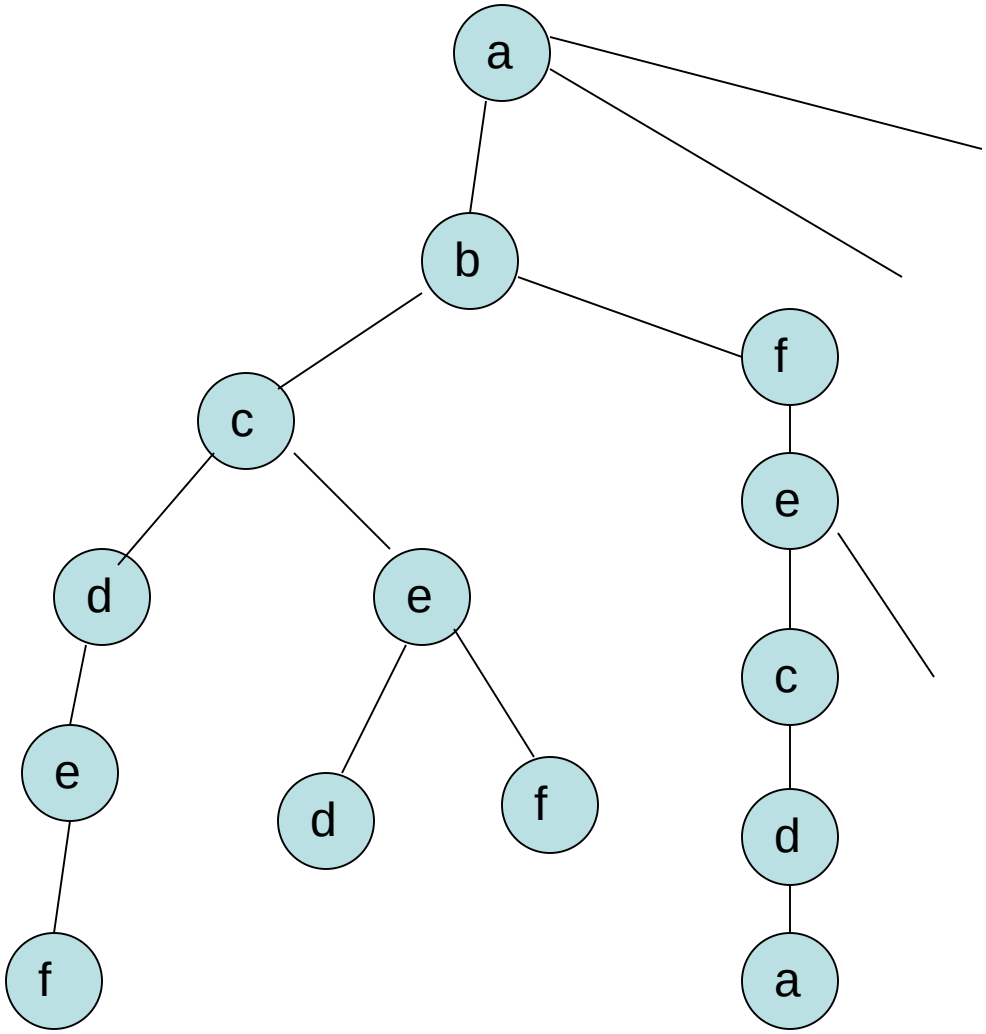
Graph?

n 皇后问题

- 蛮力方法可以改进为测试$n!$种布局
- 回溯法n^n种。
- 考虑对应于前两个皇后放在第一、二列的那些布局(（n-2）！)，蛮力方法要盲目地测试所有这些向量，而在回溯法中用O（1）次测试即可避免这些浪费。
- 尽管回溯法在最坏情况下要用O（n^n）时间来求解n皇后问题，根据经验它在有效性上远远超过蛮力方法的O（$n!$）时间，作为它的可期望运行时间通常要快得多。

哈密尔顿回路





一般回溯算法

- 一般回溯算法可以作为一种系统的搜索方法应用到一类搜索问题当中，这类问题的解由满足事先定义好的某个约束的向量（$x_1, x_2, \ldots, x_i$）组成。
- 如下的PARTITION问题中的一个变形。给定一个n个整数的集合$X = \{x_1, x_2, \ldots, x_n\}$和整数$y$，找出和等于$y$的$X$的子集$Y$。比如说，如果

  X ={10,20,30,40,50,60} 和y=60

  则有三种不同长度的解，它们包括

  {10，20，30}，{20，40} 和{60}
- 用另一种方法明确表达，使得解是一种明显的长度为n的布尔向量，以上三个解可用布尔向量表示为

  {1，1，1，0，0，0}，{0，1，0，1，0，0}

  和{0，0，0，0，0，1}

一般回溯算法

- 假定算法已经找到部分解为（$x_1, x_2, \ldots, x_j$），然后再考虑向量v=（$x_1, x_2, \ldots, x_j, x_{j+1}$），有下面的情况：
  （解向量中每个x_i都属于一个有限的线序集X_i）

1. 如果v表示问题的最后解，算法记录下它作为一个解，在仅希望获得一个解时终止，或者继续去找出其他解。
2. （向前步骤）。如果v表示一个部分解，算法通过选择集合X_{j+2}中的第一个元素向前。
3. 如果v既不是最终的解，也不是部分解，则有两种子情况。
   a. 如果从集合X_{j+1}中还有其他的元素可选择，算法将x_{j+1}置为X_{j+1}中的下一个元素。
   b. (回溯步骤)。如果从集合X_{j+1}中没有更多的元素可选择，算法通过将x_j置为X_j中的下一个元素回溯；如果从集合X_j中仍然没有其他的元素可以选择，算法通过将x_{j-1}置为X_{j-1}中的下一个元素回溯，依次类推。

一般回溯算法

算法 BACKTRACKREC
输入：集合X_1，X_2，…, X_n 的清楚的或隐含的描述。
输出：解向量v=（$x_1, x_2, \ldots, x_i$）, $i \le n$。

1. $v \leftarrow \Phi$
2. *flag*←**false**
3. *backrec*(1)
4. **if** *flag* **then** output *v*
5. **else output** “no solution”

procedure *backrec* (*k*)
1. **for** 每个$x \in X_k$
2. $x_k \leftarrow x$; 将x_k加入v
3. **if** *v* 为最终解**then set** flag←**true** and **exit**
4. **else** if *v* 是部分解**then** *backrec* (*k*+1)
5. **end for**

一般回溯算法

算法BACKTRACKITER
输入：集合X_1，X_2，…, X_n 的清楚的或隐含的描述。
输出：解向量v=（x_1, x_2, …, x_i）, $i \le n$。

1. $v \leftarrow \Phi$
2. *flag*←**false**
3. $k \leftarrow 1$
4. **while** $k \ge 1$
5. **while** X_k 没有被穷举
6. $x_k \leftarrow X_k$ 中的下一个元素；将x_k加入v
7. **if** v 为最终解**then** set *flag*←**true**, 且从两个while循环退出
8. **else if** v 是部分解**then** $k \leftarrow k+1$ {前进}
9. **end while**
10. 重置X_k， 使得排在第一位的元素为下一个元素
11. $k \leftarrow k-1$ {回溯}
12. **end while**
13. **if** *flag* **then** output v
14. **else** output “no solution”

分支定界

(Branch and Bound)