算法分析与设计 第七讲

动态规划及实例分析

主要内容

- 最优二分搜索树
- 流水作业调度
- 备忘录方法

最优二分搜索树

- 什么是二分搜索树
  - 或者是一棵空树
  - 或者是具有下列性质的二叉树
    - 若左子树不空，则左子树上所有结点的值均小于它的根结点的值
    - 若右子树不空，则右子树上所有结点的值均大于它的根结点的值
    - 左、右子树分别为二叉搜索树

最优二分搜索树

- 常见操作：查询二分搜索树
- 指令MEMBER(x,S)：若x在S中则返回”yes”，否则返回”no”
- 设有n个实数 $a_1<a_2<\ldots<a_n$构成集合S，考察由MEMBER指令构成的序列
- 指令MEMBER(x,S)中的x可能是某个a_i，也可能不在S中，把不在S中的数按区间分为n+1类，以b_0，b_1，…，b_n作为每一类数的代表（虚节点）。

最优二分搜索树

- 定义
  - p_i为MEMBER (a_i,S)出现的频率（i=1,2,…,n）
  - q_j为MEMBER (b_j,S)出现的频率（j=0,1,2,…,n）
- 因此有$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$
- 定义一棵二分搜索树的总耗费：

$$\sum_{i=1}^{n} p_i (depth(a_i)+1) + \sum_{j=0}^{n} q_j (depth(b_j))$$

- 最优二分搜索树：耗费最小的二分搜索树

最优二分搜索树

- 分析
  - 假定T_0是最优二分搜索树，它的根是a_k（第k小的数）
  - 则a_k的左子树中必然包含了$\{a_1 \ldots a_{k-1}\}$，
  - a_k的右子树中必然包含了$\{a_{k+1} \ldots a_n\}$。

最优二分搜索树

- 考察一棵树接到另一个结点之下构成一棵新树时耗费的增加
- 设有一棵由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的树
- 按定义该树的耗费为

$$\sum_{l=i+1}^{j} p_l(depth(a_l)+1)+\sum_{l=i}^{j} q_l(depth(b_l))$$

最优二分搜索树

- 当这棵由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的树接到另一个结点之下构成一棵新树时
- 这棵子树中的每个结点的深度在新树中均增加了1
- 该子树在新树中的耗费增加了

$$\sum_{l=i+1}^{j} p_l + \sum_{l=i}^{j} q_l = q_i + (p_{i+1} + q_{i+1}) + \ldots + (p_j + q_j) = W_{ij}$$

最优二分搜索树

- 根据前面的约定以及二分搜索树的性质，任何一颗子树中结点的编号都是连续的
- 而且，最优树中的任何一棵子树，也必然是关于子树中结点的最优树
- 因此最优二分搜索树具有最优子结构性质

最优二分搜索树

- 若规模为m≤n-1的最优子树均已知
- 就可以通过逐一计算以a_1，a_2，…，a_n为根的树的耗费来确定（使耗费达到最小的）根a_k并找出最优二分搜索树
- 在上述计算中，规模较小（ m≤n-1 ）的最优子树在计算中要多次被用到，因此，该问题具有高度重复性

最优二分搜索树

- 在所有由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的树中，把耗费最小的树记为T_{ij}
- 若树以a_k作为根 $(i+1 \le k \le j)$
  - 则b_i，a_{i+1}，b_{i+1}，…，a_{k-1}，b_{k-1}必然在其左子树中
  - 则b_k，a_{k+1}，b_{k+1}，…，a_j，b_j必然在其右子树中
  - 这样的树中耗费最小的必然是以$T_{i,k-1}$为其左子树，以$T_{k,j}$为其右子树
- 记c_{ij}是最优子树T_{ij}的耗费，则$c_{i,k-1}$是最优子树$T_{i,k-1}$的耗费，$c_{k,j}$是最优子树$T_{k,j}$的耗费

最优二分搜索树

- 考察以a_k (i+1≤k≤j)为根、由结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j构成的、耗费最小的树的总耗费
- 由三部分组成
  - 左子树的耗费为：$c_{i,k-1}+ w_{i,k-1}$
  - 右子树的耗费为：$c_{k,j}+w_{k,j}$
  - 根的耗费为：p_k
- 总耗费为：$c_{i,k-1}+ w_{i,k-1}+c_{kj}+w_{k,j}+p_k$
- 总耗费为：$c_{i,k-1}+c_{kj}+w_{i,j}$

最优二分搜索树

- 对于以a_k为根、耗费最小的树的总耗费$c_{i,k-1}+c_{kj}+w_{ij}$
- p_i（i=1,2,…,n），q_j（j=0,1,2,…,n）已知
- 若$w_{i,j-1}$已知，则根据$w_{i,j}= w_{i,j-1}+p_j + q_j$可以计算出$w_{ij}$（由$w_{ij}$的定义）
- 故当$c_{i,k-1}$与c_{kj}已知时，以a_k为根的树的最小总耗费在O(1)时间就可以计算出来

最优二分搜索树

- 根据以上分析
- 分别计算以a_{i+1}，a_{i+2}，…，a_j为根、含有结点b_i，a_{i+1}，b_{i+1}，…，a_j，b_j的树的总耗费
- 从中选出耗费最小的树，此即最优子树T_{ij}
- 因此，最优子树T_{ij}的耗费为

$$c_{ij} = \min_{i<k\le j}\{c_{i,k-1} + c_{kj} + w_{ij}\}$$

最优二分搜索树

- 递推求c_{ij}及记录T_{ij}的根的算法

$w_{ii} \leftarrow q_i (i=1,2,\ldots,n)$; $c_{ii} \leftarrow 0$

for l←1 to n do

{ for i←0 to n-l do

{j←i+l;

$w_{i,j} \leftarrow w_{i,j-1} + p_j + q_j$;

$c_{ij} \leftarrow \min_{(i<k<=j)} \{c_{i,k-1} + c_{kj} + w_{ij}\}$;

$r_{ij} \leftarrow k'$;

}

}

最优二分搜索树

$w_{ii} \leftarrow q_i (i=1,2,\ldots,n)$; $c_{ii} \leftarrow 0$

```
for l←1 to n do
 { for i←0 to n-l do
     {j←i+l;
      w_{i,j}←w_{i,j-1}+p_j + q_j ;
      c_{ij}← min_{(i<k<=j)}{c_{i,k-1}+c_{kj}+w_{ij}};
      r_{ij}←k';
     }
 }
```

最优二分搜索树

- 动态规划方法：$\Theta(n^3)$
  - 三层循环，每个循环至多规模为n
- 穷举法
  - N个结点的二叉树共有$\Omega(4^n/n^{3/2})$个，使用穷举法需要检查指数个数个二分搜索树

最优二分搜索树

- 如何找出最优二分搜索树:根据r_{ij}去找
- 设T_{ij}的根为a_k (r_{ij}记录到的值是k)，则从根开始建结点

```
Build-tree(i,j,r,A)                          /*建立最优子树Tij*/
 {If i≥j return "nill";
  pointer←newnode(nodetype);
  k←rij;                                     /*必有i < k ≤ j*/
  pointer→value←A[k];      /*A[k]即ak*/
  pointer→leftson←Buildtree(i,k-1,r,A);  /*建立最优左子树Ti,k-1*/
  pointer→rightson←Buildertree(k,j,r,A); /*建立最优右子树Tk,j*/
  return pointer;}
```

最优二分搜索树

- 递推求c_{ij}及记录T_{ij}的根的算法可以改进，把算法时间复杂度从$\Theta(n^3)$降到$\Theta(n^2)$
- 可以证明：如果最小耗费树$T_{i,j-1}$和$T_{i+1,j}$的根分别为a_p和a_q，则必有⑴$p \le q$；⑵最小耗费树T_{ij}的根a_k满足$p \le k \le q$
- 因此，求$\min\{c_{i,k-1}+c_{kj}+w_{ij}\}$时，无需在$a_{i+1}$～$a_j$之间去一一尝试，而只要从$a_p$～$a_q$之间去找一个根即可

最优二分搜索树

$w_{ii} \leftarrow q_i (i=1,2,\ldots,n)$； $c_{ii} \leftarrow 0$

for l←1 to n do

{ for i←0 to n-l do

{j←i+l;

$w_{i,j} \leftarrow w_{i,j-1} + p_j + q_j$;

$c_{ij} \leftarrow \min_{(1<k<=j)} \{c_{i,k-1} + c_{kj} + w_{ij}\}$;

$r_{ij} \leftarrow k'$;

}

}

流水作业调度

- 设有n个作业，每一个作业i均被分解为m项任务: T_{i1}, T_{i2}, …, T_{im}(1≤i≤n，故共有n×m个任务)，要把这些任务安排到m台机器上进行加工
- n个作业
- m项任务
- m台机器

流水作业调度

- 如果任务的安排满足下列3个条件，则称该安排为流水作业调度：
  - 1. 每个作业i的第j项任务T_{ij} $(1\le i\le n, 1\le j\le m)$ 只能安排在机器P_j上进行加工
  - 2. 作业i的第j项任务$T_{ij}$$(1\le i\le n, 2\le j\le m)$的开始加工时间均安排在第j-1项任务$T_{i,j-1}$加工完毕之后
  - 3. 任何一台机器在任何一个时刻最多只能承担一项任务

流水作业调度

- 最优流水作业调度
- 设任务T_{ij}在机器P_j上进行加工需要的时间为t_{ij}，如果所有的t_{ij} $(1\le i\le n, 1\le j\le m)$均已给出，要找出一种安排任务的方法，使得完成这n个作业的加工时间为最少，这个安排称之为最优流水作业调度
- 完成n个作业的加工时间：从安排的第一个任务开始加工, 到最后一个任务加工完毕，其间所需要的时间

流水作业调度

- 注意点
- 优先调度
  - 允许优先级较低的任务在执行过程中被中断，转而去执行优先级较高的任务
- 非优先调度
  - 任何任务一旦开始加工，就不允许被中断，直到该任务被完成
- 流水作业调度一般均指的是非优先调度

流水作业调度

- 当机器数(或称工序数)m≥3时，流水作业调度问题是一个NP-hard问题
- 当m=2时，该问题可有多项式时间的算法
- 为讨论的方便
  - 记t_{i1}为a_i（作业i在P_1上加工所需时间）
  - 记t_{i2}为b_i（作业i在P_2上加工所需时间）

流水作业调度

- 当机器P_1为空闲时，则任何一个作业的第一个任务都可以立即在P_1上执行
- 必有一个最优调度使得在P_1上的加工是无间断的
- 一定有一个最优调度使得在P_2上的加工空闲时间（从0时刻起算）为最小，同时还满足在P_1上的加工是无间断的

流水作业调度

- 如果在P_2上的加工次序与在P_1上的加工次序不同，则只可能增加加工时间（在最好情况下，增加的时间为0）
- 请注意，这里机器数m的值
- 仅需要考虑在P_1和P_2上加工次序完全相同的调度
- 为简化起见，假定所有$a_i \neq 0$

流水作业调度

- 最优调度具有如下性质
  - 在所确定的最优调度的排列中去掉第一个执行作业后，剩下的作业排列仍然还是一个最优调度，即该问题具有最优子结构的性质
  - 在计算规模为n的作业集合的最优调度时，该作业集合的子集合的最优调度会被多次用到，即该问题亦具有高度重复性
- 可以用动态规划方法求解？

流水作业调度

- 设N={1，2，---，n}是全部作业的集合，作业集S是N的子集合即有$S \subseteq N$
- 设对机器P_2需等待t个时间单位以后才可以用于S中的作业加工（t也可以为0即无须等待）
- 记g(S,t)为在此情况下完成S中全部作业的最短时间，则g(S,t)可递归表示为
- $g(S,t)=\min_{i\in S}\{a_i+ g(S-\{i\},b_i+\max\{t-a_i,0\})\}$

流水作业调度

- 当S=N即全部作业开始加工时，t=0。
- $g(N,0)=\min_{1\le i\le n}\{a_i+ g(N-\{i\},b_i)\}$
- 根据上式可以实现计算g(N,0)
- 该算法的时间复杂度为指数量级，因为算法中对N的每一个非空子集都要进行一次计算，而N的非空子集共有2^n-1个
- 因此不能直接使用动态规划方法来求解该问题

流水作业调度

- $\min\{a_j, b_i\} \geq \min\{a_i, b_j\}$ （Johnson不等式）
- 即当$\min\{ a_i , a_j , b_i , b_j\}$为$a_i$或者$b_j$时，Johnson不等式成立，此时把i排在前j排在后的调度用时较少
- 反之，若$\min\{ a_i , a_j , b_i , b_j\}$为$a_j$或者$b_i$时，则j排在前i排在后的调度用时较少

流水作业调度

- 推广到一般情况
  - 当$\min\{ a_1, a_2,\text{---}, a_n , b_1, b_2,\text{---}, b_n \}=a_k$时，则对任何$i\neq k$，都有$\min\{a_i, b_k\} \geq \min\{a_k, b_i\}$成立，故此时应将作业k安排在最前面，作为最优调度的第一个执行的作业
  - 当$\min\{ a_1, a_2,\text{---}, a_n , b_1, b_2,\text{---}, b_n \}= b_k$时，则对任何$i\neq k$，也都有$\min\{a_k, b_i\}\geq\min\{a_i, b_k\}$成立，故此时应将作业k安排在最后面，作为最优调度的最后一个执行的作业

流水作业调度

- n个作业中首先开工（或最后开工）的作业确定之后，对剩下的n-1个作业采用相同方法可再确定其中的一个作业,应作为n-1个作业中最先或最后执行的作业；
- 反复使用这个方法直到最后只剩一个作业为止，即可确定最优调度
- 时间主要耗费在对任务集的排序，因此，其时间复杂度为O(nlgn)

流水作业调度

- 满足1）高度重复性 2）最优子结构性质时，一般采用动态规划法，但偶尔也可能得不到高效的算法
- 若问题本身不是NP-hard问题
  - 进一步分析后就有可能获得效率较高的算法
- 若问题本身就是NP-hard问题
  - 与其它的精确算法相比，动态规划法性能一般不算太坏， 但有时需要对动态规划法作进一步的加工

备忘录方法

- 当某个问题可以用动态规划法求解,但二维数组中有相当一部分元素在整个计算中都不会被用到
- 因此，不需要以递推方式逐个计算二维数组中元素，而采用备忘录方法：数组中的元素只是在需要计算时才去计算，计算采用递归方式，值计算出来之后将其保存起来以备它用

备忘录方法

- 若有大量的子问题无需求解时，用备忘录方法较省时
- 但当无需计算的子问题只有少部分或全部都要计算时，用递推方法比备忘录方法要好（如矩阵连乘，最优二分搜索树）

备忘录方法 LCS

- LCS问题，当$x_i=y_j$时，求C[i,j]只需知道C[i-1,j-1]，而无需用到C[i,0]～C[i,j-1]及C[i-1,j]～C[i-1,n]
- 当只需求出一个LCS时，可能有一些C[p,q]在整个求解过程中都不会用到
- 首先将C[i,0]与C[0,j] 初始化为0
- 其余m×n个C[i,j]全部初始化为-1

备忘录方法 LCS

- 计算C[i,j]的递归算法LCS_L2(X,Y, i,j,C)
- 若x[i]=y[j]，则去检查C[i-1,j-1]
  - 若C[i-1,j-1]> -1（已经计算出来），就直接把C[i-1,j-1]+1赋给C[i,j]，返回
  - 若C[i-1,j-1]=-1（尚未计算出来），就递归调用LCS_L2(X,Y, i-1,j-1,C) 计算出C[i-1,j-1]，然后再把C[i-1,j-1]+1赋给C[i,j] ，返回
- 若x[i]不等于y[j]，则检查C[i-1,j]和C[i,j-1]
  - 若两者均 > -1（已经计算出来），则把max{ C[i-1,j], C[i,j-1]} 赋给C[i,j]，返回
  - 若C[i-1,j], C[i,j-1] 两者中有一个等于-1（尚未计算出来），或两者均等于-1，就递归调用LCS_L2将其计算出来，然后再把max{ C[i-1,j], C[i,j-1]} 赋给C[i,j]