湖南大學
HUNAN UNIVERSITY

# 第十五章 回溯与分支限界

湖南大学信息科学与工程学院

湖南大学 HUNAN UNIVERSITY

# 目录

# 引言

- 理论上
  - 寻找问题的解的一种可靠的方法是首先列出所有候选解，然后依次检查每一个，在检查完所有或部分候选解后，即可找到所需要的解。
- 但是
  - 当候选解数量有限并且通过检查所有或部分候选解能够得到所需解时，上述方法是可行的。
  - 若候选解的数量非常大（指数级，大数阶乘），即便采用最快的计算机也只能解决规模很小的问题。

# 引言

- 于是
  - 回溯和分支限界法是比较常用的对候选解进行系统检查两种方法。
  - 按照这两种方法对候选解进行系统检查通常会使问题的求解时间大大减少。
  - 可以避免对很大的候选解集合进行检查，同时能够保证算法运行结束时可以找到所需要的解。
  - 通常能够用来求解规模很大的问题。

# 解向量

## 问题的解向量

回溯法希望一个问题的解能够表示成一个n元式$(x_1,x_2,\ldots,x_n)$的形式。

## 显约束

对分量$x_i$的取值限定。

## 隐约束

为满足问题的解而对不同分量之间施加的约束

# 解空间

- 对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

注意：同一问题可有多种表示，有些表示更简单，所需状态空间更小（存储量少，搜索方法简单）。

# 解空间实例

例如

对于有n种可选物品$(a_1, a_2, ..., a_n)$的0-1背包问题

✓解空间由n!个物品排列组成，用来表示物品被选中放入背包

✓n=3时，解空间为$\{(a_1),(a_2),(a_3),(a_1,a_2),(a_1,a_3),(a_2,a_1),(a_2,a_3),(a_3,a_1),(a_3,a_2),(a_1,a_2,a_3),(a_1,a_3,a_2),(a_2,a_1,a_3),(a_2,a_3,a_1),(a_3,a_1,a_2),(a_3,a_2,a_1)\}$

用**n叉树**表示的**解空间**

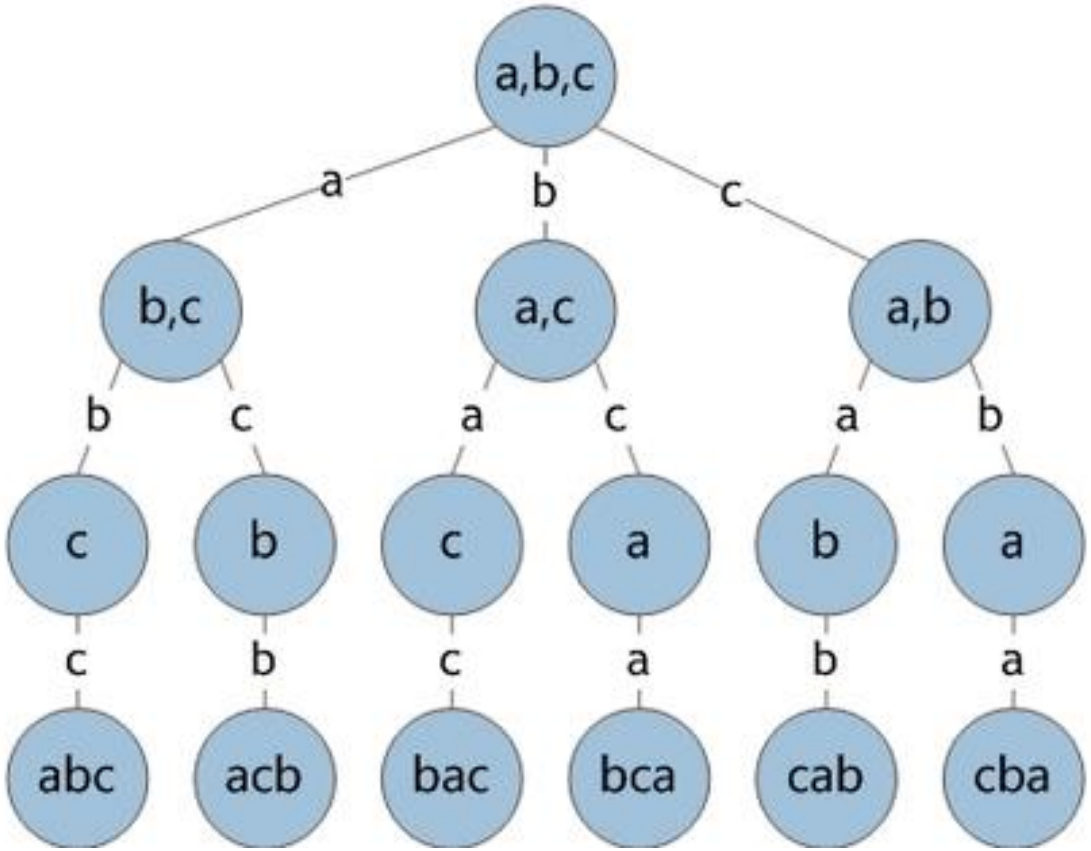
✓边上的数字给出了向量x中第i个分量的值$a_i$

✓树上每一个节点定义了解问题的一个解

# 典型的解空间树——排列树

排列树（Permutation Trees）：当所给问题是确定$n$个元素满足某种性质的排列时，相应的解空间树称为排列树。在排列树中，通常情况下，$|S_0|=n$，$|S_1|=n-1$，…，$|S_{n-1}|=1$，所以，排列树中共有$n!$个叶子结点，因此，遍历排列树需要$O(n!)$时间。

# 排列树实例



# 针对最优子集问题的排列树

●当所给问题是从$n$个元素的集合中找出满足某种性质的子集时，因为计算机求解过程都是依次将元素放入待求解子集，所以也可以对应成元素排列进入最优子集的排列树

# 回溯法

- 回溯法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。
- 回溯法以深度优先的方式系统地搜索问题的解，按选优条件向前搜索，以达到目标。

# 分支限界法的基本思想

- 分支限界法通常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。
- 在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。其中导致不可行解或非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。
- 从活结点表中取下一结点成为当前扩展结点并重复上述结点扩展过程，直到找到所求的解或活结点表为空时为止。

湖南大学 HUNAN UNIVERSITY

# 目录

# 基于回溯法的批处理作业调度

## 问题描述：

- 给定n个作业的集合$\{J_1, J_2, \cdots, J_n\}$。
- 每一个$J_i$作业都有两项任务分别在两台机器上完成。每个作业必须先由机器1处理，然后由机器2处理。
- 作业$J_i$需要机器j的处理时间为$t_{ji}$，其中i=1, 2, . . . n， j=1, 2。
- 对于一个确定的作业调度，设$F_{ji}$是作业i在机器j上完成处理的时间。
- 所有作业在机器2上完成处理的时间和 $f = \sum_{i=1}^{n} F_{2i}$ 称为该作业调度的完成时间和。

# 基于回溯法的批处理作业调度

**批处理作业调度问题要求**对于给定的n个作业，制定最佳作业调度方案，**使其完成时间和达到最小**

# 批处理作业调度实例

问题解析：

| $t_{ji}$ | 机器1 | 机器2 |
|---|---|---|
| 作业1 | 2 | 3 |
| 作业2 | 5 | 2 |
| 作业3 | 4 | 1 |

- 这个3个作业的6种可能的调度方案是：

（1, 2, 3） （1, 3, 2） （2, 1, 3）

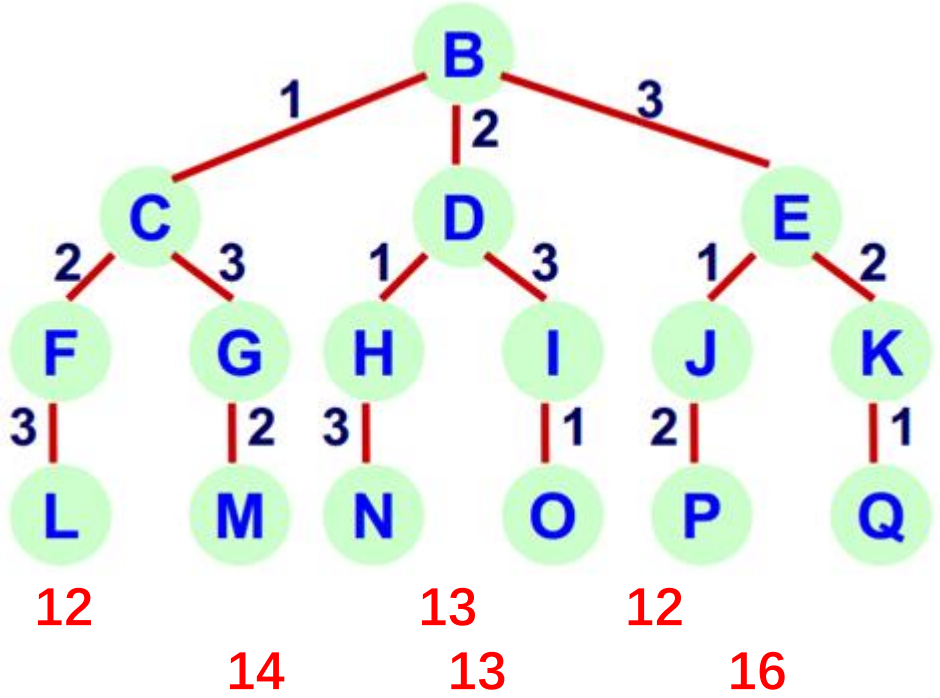
（2, 3, 1） （3, 1, 2） （3, 2, 1）

- 对应的完成时间和分别是：12， 13， 12， 14， 13， 16。
- 最佳调度方案是1, 2, 3和2, 1, 2， 其完成时间和为12。

# 批处理作业调度

## 算法设计：

批处理作业调度问题要从n个作业的所有排列中找出有最小完成时间和的作业调度，所以批处理作业调度问题的解空间是一颗**排列树**。



# 基于分支限界的批处理作业调度问题

问题描述：给定$n$个作业的集合$J=\{J_1, J_2, \ldots, J_n\}$，每个作业都有3项任务分别在3台机器上完成，作业$J_i$需要机器$j$的处理时间为$t_{ij}(1\leq i\leq n, 1\leq j\leq 3)$，每个作业必须先由机器1处理，再由机器2处理，最后由机器3处理。

- 批处理作业调度问题要求确定这$n$个作业的最优处理顺序，使得从第1个作业在机器1上处理开始，到最后一个作业在机器3上处理结束所需的时间最少。

# 批处理作业调度问题实例

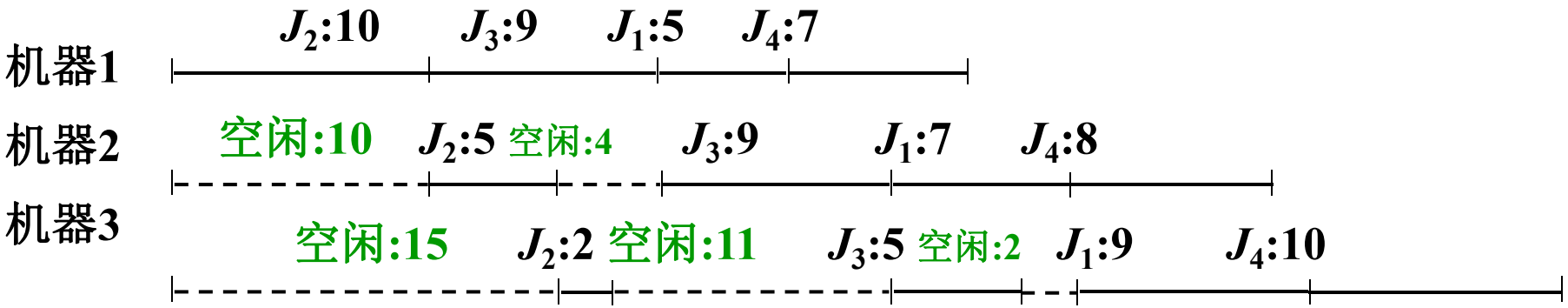
实例：设$J=\{J_1, J_2, J_3, J_4\}$是4个待处理的作业，需要的处理时间如下所示

$$T= \begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array} \begin{array}{c} \begin{array}{ccc} \text{机器1} & \text{机器2} & \text{机器3} \end{array} \\ \begin{pmatrix} 5 & 7 & 9 \\ 10 & 5 & 2 \\ 9 & 9 & 5 \\ 7 & 8 & 10 \end{pmatrix} \end{array}$$

若处理顺序为$(J_2, J_3, J_1, J_4)$，则从作业2在机器1处理开始到作业4在机器3处理完成的调度方案如下：

# 批处理作业调度问题实例

$$T=\begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \text{机器1} \quad \text{机器2} \quad \text{机器3} \\ \begin{pmatrix} 5 & 7 & 9 \\ 10 & 5 & 2 \\ 9 & 9 & 5 \\ 7 & 8 & 10 \end{pmatrix} \end{array}$$




(----表示空闲，最后完成处理时间为54)

**等待时间+处理时间**

# 批处理作业调度问题实例

分析：

- 解向量：$X=(x_1, x_2, ..., x_n)$——排列树
- 约束条件：显式：$x_i=J_1, J_2, ..., J_n$
- 目标函数:$sum3[n]= max\{sum2[n], sum3[n-1]\}+t_{n3}$极小化

  其中，$sum2[n]=max\{sum1[n], sum2[n-1]\} + t_{n2}$；

  $sum1[n]=sum1[n-1]+ t_{n1}$

# 批处理作业调度问题实例

下限界函数：设M是已安排好的作业集合，$M \subset \{1, 2, ..., n\}$，$|M|=k$。现在要处理作业$k+1$，有：

$$db = \max\{\text{sum1}[k+1], \text{sum2}[k]\} + \sum_{i \notin M} t_{i2} + \min\{\ t_{j3}\ \}_{j \neq k+1, j \notin M}$$

$$\boldsymbol{sum1[k+1]=sum1[k]+ t_{k+1,1}}$$

$$\boldsymbol{sum2[k+1]=max\{sum1[k+1], sum2[k]\} + t_{k+1,2}}$$

# 批处理作业调度问题实例

目标函数的下界：$sum3_{db}= \quad t_{i1}+\sum_{j=1}^{n} t_{j2}+\min_{k\neq i}\{t_{k3}\}$

说明：最理想的情况下，机器1和机器2均无空闲，最后处理的作业恰好是在机器3上处理时间最短的作业。

如上实例，$sum3_{db}= \quad t_{11}+\sum_{j=1}^{4} t_{j2}+t_{23}=36$

# 批处理作业调度问题实例

遍历并估算解空间树上各结点的目标函数的下界值：

根结点：$sum1=0, sum2=0, M=\{\}$

$k=0$

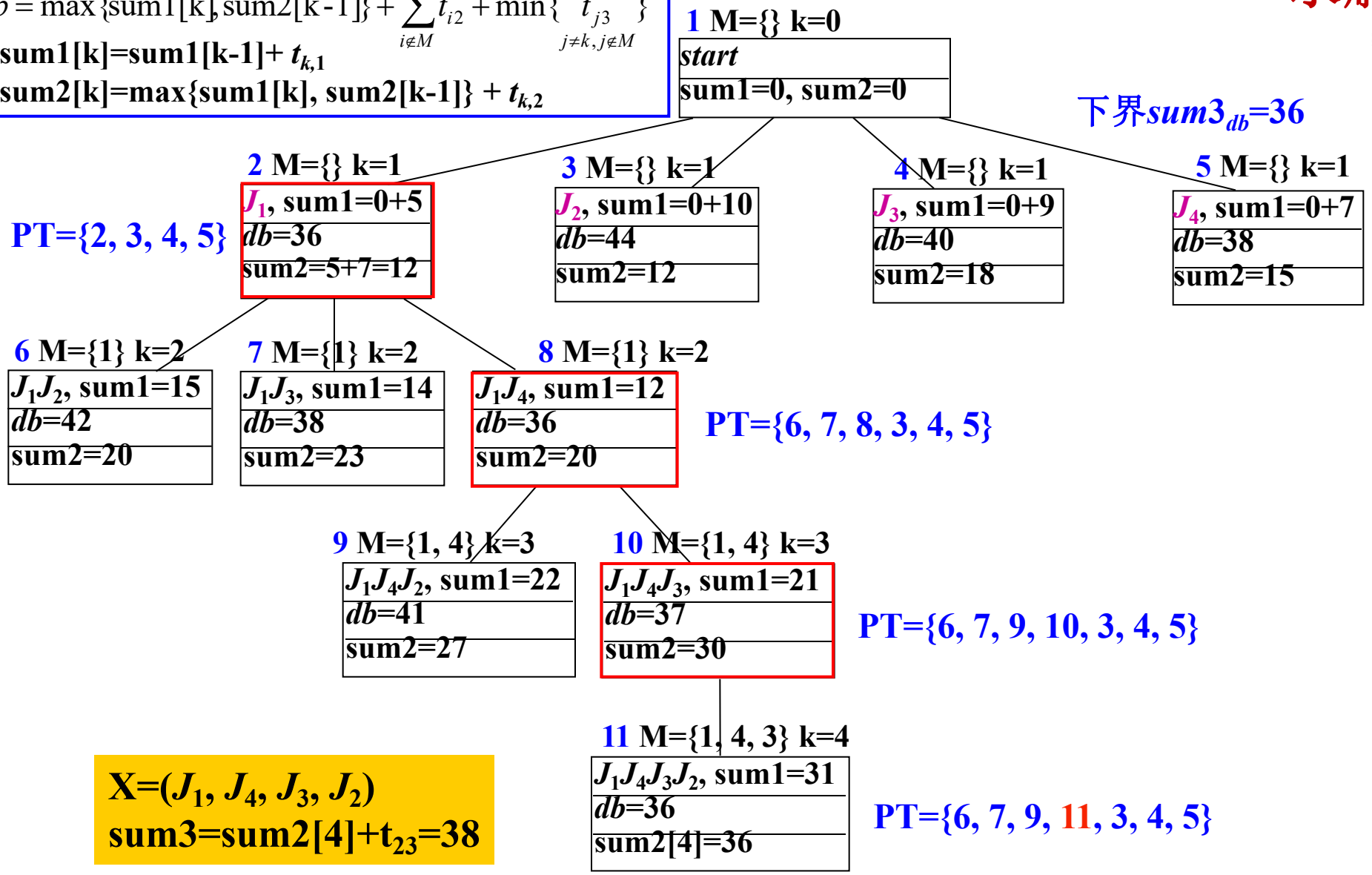
$$T=\begin{array}{c} \\ J_1 \\ J_2 \\ J_3 \\ J_4 \end{array}\begin{array}{c} \begin{array}{ccc} \text{机器1} & \text{机器2} & \text{机器3} \end{array} \\ \begin{pmatrix} 5 & 7 & 9 \\ 10 & 5 & 2 \\ 9 & 9 & 5 \\ 7 & 8 & 10 \end{pmatrix} \end{array}$$

# 求解过程

$$db=\max\{\text{sum1}[k],\text{sum2}[k-1]\}+\sum_{i\notin M}t_{i2}+\min\{\ t_{j3}\ \}_{j\neq k,j\notin M}$$

sum1[k]=sum1[k-1]+ $t_{k,1}$

sum2[k]=max{sum1[k], sum2[k-1]} + $t_{k,2}$




**X=($J_1$, $J_4$, $J_3$, $J_2$)**
**sum3=sum2[4]+$t_{23}$=38**

# 批处理作业调度问题实例总结

从上例可知：分支限界法中，

- 扩展结点表PT取得极值的叶子结点就对应的是问题最优解；
- 扩展结点的过程，一开始实际类似“深度优先”

湖南大学 HUNAN UNIVERSITY

# 目录

# 0-1背包问题

在0-1背包问题中，需对容量为c的背包进行装载。从n个物品中选取装入背包的物品，每件物品i的重量为$w_i$, 价值为$v_i$。对于可行的背包装载，背包中的物品的总重量不能超过背包的容量，最佳装载是指所装入的物品价值最高

$$\max \sum_{1\le i\le n} v_i x_i$$

约束条件为：$\sum_{1\le i\le n} w_i x_i \le C, x_i \in \{0,1\}$

在这个表达式中，需求出$x_i$的值。$x_i$=1表示物品i装入背包中，$x_i$=0表示物品i不装入背包。其中C、$w_i$以及$v_i$不限定为一个整数

# 0-1背包问题

- 令cw($i$)表示目前搜索到第$i$层已经装入背包的物品总重量，即部分解($x_1$，$x_2$，…，$x_i$)的重量：

$$cw(i)=\sum_{j=1}^{i}x_j w_j$$

- 对于左子树，$x_i$=1，其约束函数为：

$$constraint(i)=cw(i-1)+w_i$$

- 若$constraint(i)$>C，则停止搜索左子树，否则继续搜索。

# 0-1背包问题

- 对于右子树，为了提高搜索效率，采用上界函数Bound(*i*)剪枝。
- 令cv(*i*)表示目前到第*i*层结点已经装入背包的物品价值：

$$cv(i)=\sum_{j=1}^{i}x_j v_j$$

- 令r(*i*)表示剩余物品的总价值：

$$r(i)=\sum_{j=i+1}^{n}v_j$$

- 则限界函数Bound(*i*)为：

$$Bound(i)=cv(i)+r(i)$$

# 0-1背包问题

- 假设当前最优值为bestv，若Bound(*i*)＜bestv，则停止搜索第*i*层结点及其子树，否则继续搜索。显然r(*i*)越小， Bound(*i*)越小，剪掉的分支就越多。
- 为了构造更小的r(*i*)，将物品以单位重量价值比$d_i=v_i/w_i$递减的顺序进行排列：$d_1 \geq d_2 \geq \ldots \geq d_n$
- 对于第*i*层，背包的剩余容量为C-cw(*i*)，采用贪心算法把剩余的物品放进背包，根据贪心算法理论，此时剩余物品的价值已经是最优的，因为对剩余物品不存在比上述贪心装载方案更优的方案。

# 0-1背包问题

0-1背包问题的回溯算法

```
Backtrack(int i)
{ if (i > n)
     { bestv=cv; return;}
  if (cw+w[i]<=c) {//x[i]=1
       cw+=w[i];  cv+=v[i];
       Backtrack(i+1);
       cw-=w[i];  cv-=v[i]; }
if(Bound(i+1)>bestv)     //x[i]=0
   Backtrack(i+1) ;
}
```

```
值递减序装入
   物品
                    cleft)
   }
//装满背包
if（i<=n） b+=v[i]/w[i]*cleft;
return b;
}
```

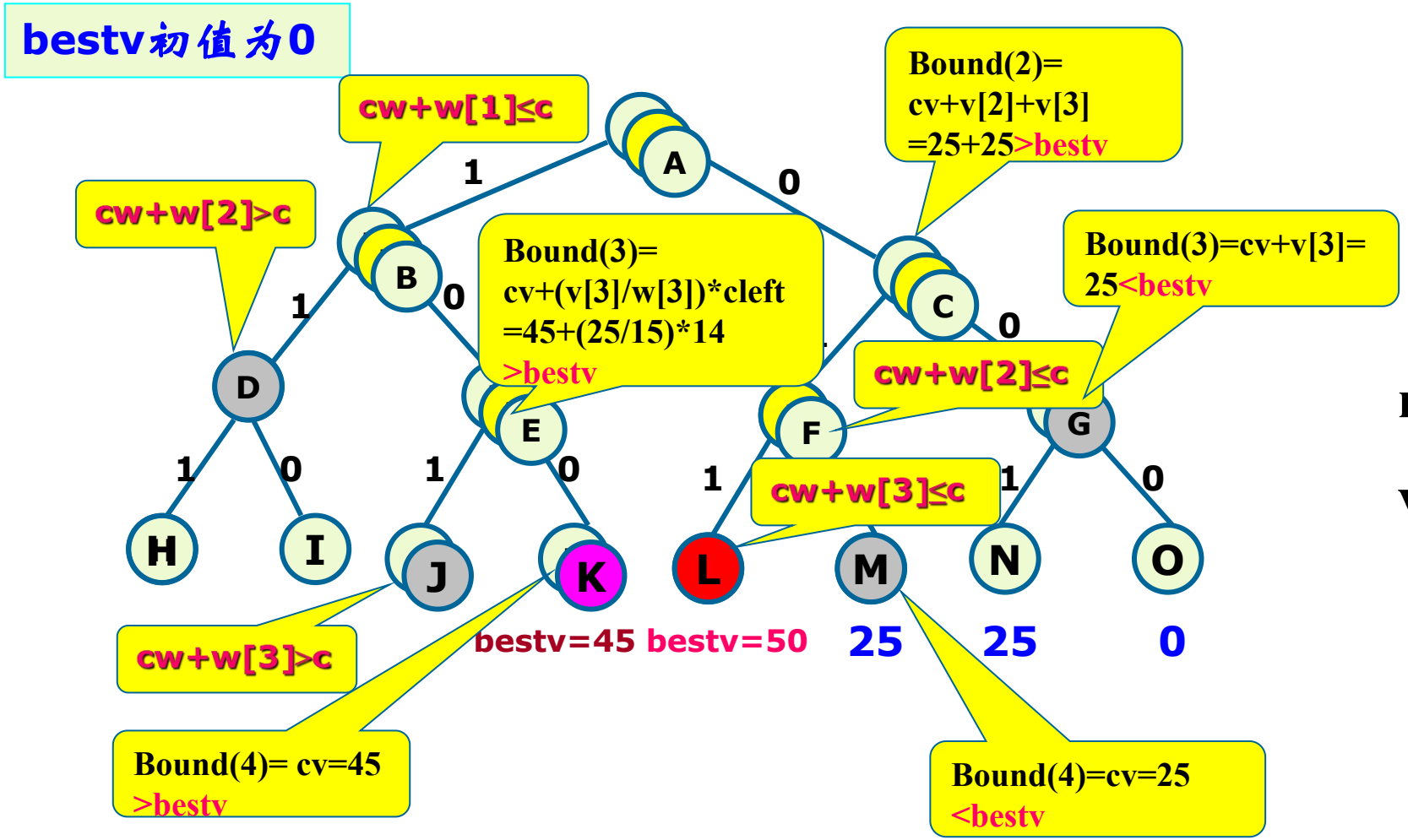
bestv表示当前最优解的装包价值，初值为0。
能走到这儿，意味着得到更优解，因此才改写。

cw和cv是全局变量，表示当前路径已装包重量及装包价值，初值均为0 。

约束条件

限界条件

# 0-1背包问题




n=3，W={16，15，15}，

v={45，25，25}，C=30。

# 0-1背包问题

- 算法效率

计算上界函数Bound 需要O(n)时间，在最坏情况下有$O(2^n)$个右儿子结点需要计算上界函数，所以解0-1背包问题的回溯算法Backtrack的计算时间为：

$T(n)=O(n2^n)$

# 0-1背包问题-队列式分支限界法

用一个队列存储活结点表，初始为空

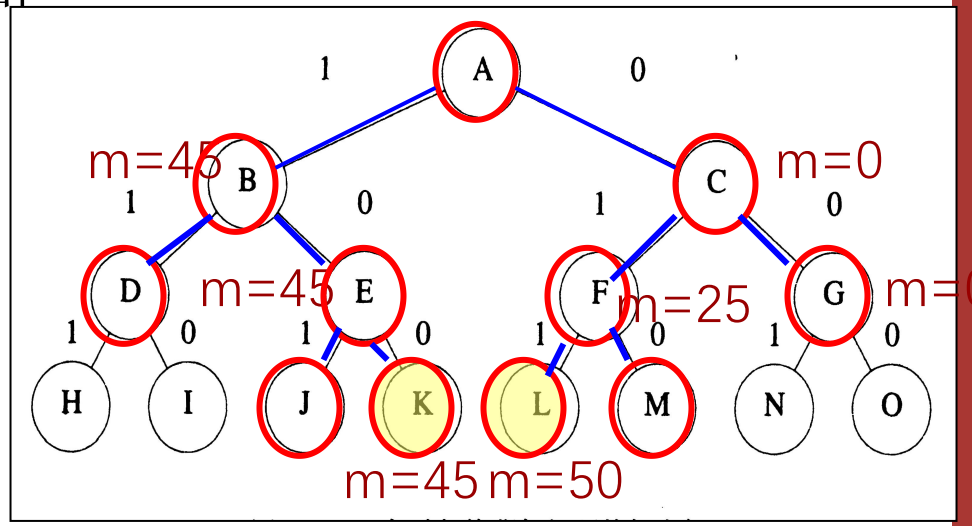
A为当前扩展结点，其儿子结点B和C均为可行结点，

将其按从左到右顺序加入活结点队列，并舍弃A。

按FIFO原则，下一扩展结点为B，其儿子结点D不可行，舍弃；E可行，加入。舍弃B

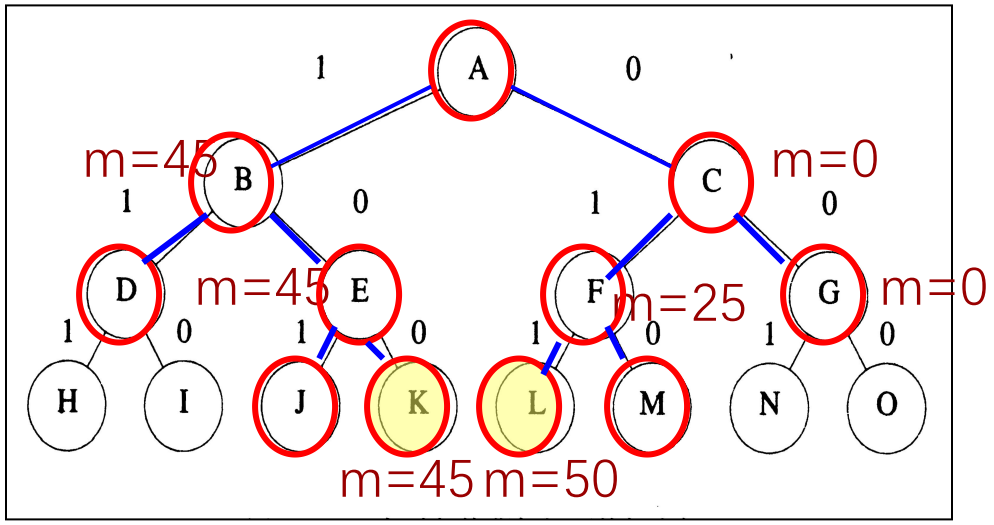
C为当前扩展结点，儿子结点F、G均为可行结点，加入活结点表，舍弃C

扩展结点E的儿子结点J不可行而舍弃；K为可行的叶结点，是问题的一个可行解，价值为45



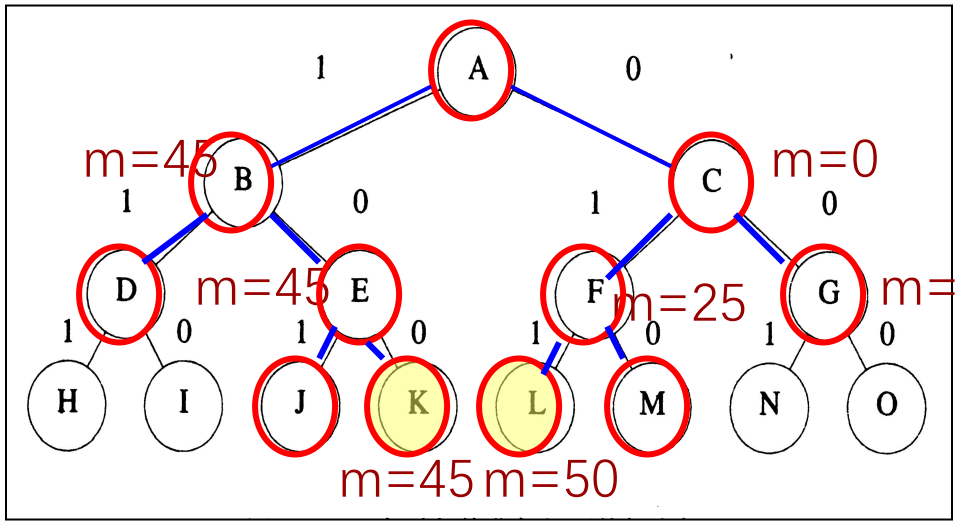
# 0-1背包问题-队列式分支限界法

- 当前活结点队列的队首为F, 儿子结点L、M为可行叶结点，价值为50、25
- G为最后一个扩展结点，儿子结点N、O均为可行叶结点，其价值为25和0
- 活结点队列为空，算法结束，其最优值为50



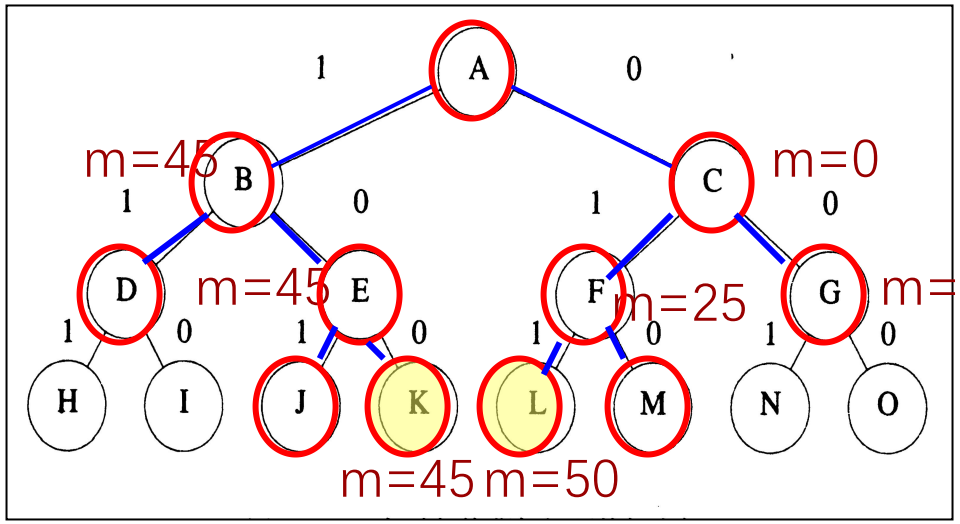
# 0-1背包问题-队列式分支限界法

- 用一个极大堆表示活结点表的优先队列，其优先级定义为活结点所获得的价值。初始为空。
- 由A开始搜索解空间树，其儿子结点B、C为可行结点，加入堆中，舍弃A。
- B获得价值45，C为0. B为堆中价值最大元素，并成为下一扩展结点。



# 0-1背包问题-队列式分支限界法

- B的儿子结点D是不可行结点，舍弃。E是可行结点，加入到堆中。舍弃B。
- E的价值为40，是堆中最大元素，为当前扩展结点。
- E的儿子J是不可行叶结点，舍弃。K是可行叶结点，为问题的一个可行解价值为45.
- 继续扩展堆中唯一活结点C，直至存储活结点的堆为空，算法结束。
- 算法搜索得到最优值为50，最优解为从根结点A到叶结点L的路径（0，1，1）。



湖南大学 HUNAN UNIVERSITY

# 目录

# 数据挖掘经典故事：啤酒与尿布

“啤酒与尿布”的故事产生于20世纪90年代的美国沃尔玛超市中，沃尔玛的超市管理人员分析销售数据时发现了一个令人难于理解的现象：在某些特定的情况下，“啤酒”与“尿布”会经常出现在同一个购物篮中

# 频繁项集挖掘的提出

“啤酒与尿布”的故事必须具有技术方面的支持。1993年美国学者Agrawal提出通过分析购物篮中的商品集合，从而找出商品之间关联关系的关联算法，并根据商品之间的关系，找出客户的购买行为。Agrawal提出的这个问题及其设计的相应算法也被认为是开启了 数据挖掘 领域。

# 频繁项集定义——支持度

一个项集的支持度被定义为数据集中包含该项集的记录所占的比例

支持度是针对项集来说的，因此可以定义一个最小支持度，而只保留满足最小支持度的项集

# 频繁项集的单调性

频繁项集的单调性是说如果某个项集是频繁的，那么它的所有子集也是频繁的。反之，如果一个项集是非频繁项集，那么它的所有超集也是非频繁的。

# 基于分支限界法与回溯法的方法

基于分支限界的在频繁项集搜索空间中进行宽度优先搜索——Apriori算法；

基于回溯的在频繁项集搜索空间中进行深度优先搜索——FP-growth算法

# Apriori算法实例

【例3】一个Apriori的具体例子，该例基于右图某商店的事务DB。DB中有9个事务，Apriori假定事务中的项按字典次序存放。

| *TID* | 项目 |
|---|---|
| *1* | *1，2，5* |
| *2* | *2，4* |
| *3* | *2，3* |
| *4* | *1，2，4* |
| *5* | *1，3* |
| *6* | *2，3* |
| *7* | *1，3* |
| *8* | *1，2，3，5* |
| *9* | *1，2，3* |

# Apriori算法实例

（1）在算法的第一次迭代，每个项都是候选*1*-项集的集合$C_1$的成员。算法简单地扫描所有的事务，对每个项的出现次数计数。

扫描D,对每个候选计数

$C_1$

| 项集 | 支持度计数 |
| --- | --- |
| {1} | 6 |
| {2} | 7 |
| {3} | 6 |
| {4} | 2 |
| {5} | 2 |

# Apriori算法实例

（2）设最小支持计数为2，可以确定1-频集的集合$L_1$。它由具有最小支持度的候选1-项集组成。

比较候选支持度计数与最小支持度计数

$L_1$

| 项集 | 支持度计数 |
|---|---|
| {1} | 6 |
| {2} | 7 |
| {3} | 6 |
| {4} | 2 |
| {5} | 2 |

# Apriori算法实例

（3）为发现繁-频集的集合$L_2$，算法使用$L_1$ $\bowtie$ $L_1$产生候选2-项集集合$C_2$。

由$L_1$产生候选$C_2$

$C_2$

| 项集 |
| --- |
| {1，2} |
| {1，3} |
| {1，4} |
| {1，5} |
| {2，3} |
| {2，4} |
| {2，5} |
| {3，4} |
| {3，5} |
| {4，5} |

# Apriori算法实例

（4）扫描D中事务，计算$C_2$中每个候选项集的支持计数。

扫描D, 对每 个候选计数

$C_2$

| 项集 | 支持度计数 |
|---|---|
| {1，2} | 4 |
| {1，3} | 4 |
| {1，4} | 1 |
| {1，5} | 2 |
| {2，3} | 4 |
| {2，4} | 2 |
| {2，5} | 2 |
| {3，4} | 0 |
| {3，5} | 1 |
| {4，5} | 0 |

# Apriori算法实例

（5）确定2-频集的集合$L_2$，它由具有最小支持度的$C_2$中的候选2-项集组成。

比较候选支持度计数与最小支持度计数

$L_2$

| 项集 | 支持度计数 |
| --- | --- |
| {1，2} | 4 |
| {1，3} | 4 |
| {1，5} | 2 |
| {2，3} | 4 |
| {2，4} | 2 |
| {2，5} | 2 |

# Apriori算法实例

（6）候选3-项集的集合$C_3$的产生如下： 连接：$C_3=L_2 \bowtie L_2$

$L_2$

| 项集 |
| --- |
| 1,2 |
| 1,3 |
| 1,5 |
| 2,3 |
| 2,4 |
| 2,5 |

$C_3$

| 项集 | |
| --- | --- |
| 1,2,3 | √ |
| 1,2,5 | √ |
| 1,3,5 | × |
| 2,3,4 | × |
| 2,3,5 | × |
| 2,4,5 | × |

# 利用Apriori性质剪枝

频繁项集的所有子集必须是频繁的。存在候选项集，判断其子集是否频繁。

- *{1,2,3}*的*2*-项子集是*{1,2}*，*{1,3}*和*{2,3}*，它们都是$L_2$的元素。因此保留*{1,2,3}*在$C_3$中。
- *{1,2,5}*的*2*-项子集是*{1,2}*，*{1,5}*和*{2,5}*，它们都是$L_2$的元素。因此保留*{1,2,5}*在$C_3$中。
- *{1,3,5}*的*2*-项子集是*{1,3}*，*{1,5}*是$L_2$的元素，*{3,5}*不是$L_2$的元素，因而不是频繁的，由$C_3$中 删除*{1,3,5}*。

# 利用Apriori性质剪枝

- *{2,3,4}*的*2*-项子集是*{2,3}*，*{2,4}*和*{3,4}*,其中*{3,4}*不是$L_2$的元素，因而不是频繁的， 由$C_3$中删除*{2,3,4}*。
- *{2,3,5}*的*2*-项子集是*{2,3}*，*{2,5}*和*{3,5}*, 其中*{3,5}*不是$L_2$的元素，因而不是频繁的， 由$C_3$中删除 *{2,3,5}*。
- *{2,4,5}*的*2*-项子集是*{2,4}*，*{2,5}*和*{4,5}*, 其中*{4,5}*不是$L_2$的元素，因而不是频繁的， 由$C_3$中删除*{2,4,5}* 。
- 这样，剪枝后$C_3$=*{{1,2,3}，{1,2,5}}*。

# Apriori算法实例

（7）扫描D中事务，以确定$L_3$，它由$C_3$中具有最小支持度的的候选3-项集组成。

由$L_2$产生候选$C_3$

*C3*

| 项集 |
| --- |
| {1，2，3} |
| {1，2，5} |

扫描D, 对每 个候选计数

*C3*

| 项集 | 支持度计数 |
| --- | --- |
| {1，2，3} | 2 |
| {1，2，5} | 2 |

# Apriori算法实例

（8）算法使用$L_3$产生候选4-项集的集合$C_4$。尽管连接产生结果{{1,2,3,5}}, 这个项集被剪去，因为它的子集{2,3,5} 不是频繁的。则$C_4$为空,因此算法终止， 找出了所有的频繁项集。

比较候选支持度计数
与最小支持度计数

$L_3$

| 项集 | 支持度计数 |
| --- | --- |
| {1，2，3} | 2 |
| {1，2，5} | 2 |

# Apriori算法的缺点

1. 在每一步产生侯选项目集时循环产生的组合过多，没有排除不应该参与组合的元素；
2. 每次计算项集的支持度时，都对数据库D中的全部记录进行了一遍扫描比较

# FP-growth算法详解

FP-growth算法是UIUC的Jian Pei在2000年提出的频繁项集挖掘算法

FP-growth算法只需要对数据库进行两次扫描，FP-growth算法发现频繁项集的基本过程如下：

**1. 构建FP树**

**2. 从FP树中挖掘频繁项集**

# 构建FP树

| 事务ID | 事务中的元素项 |
| --- | --- |
| 001 | r, z, h, j, p |
| 002 | z, y, x, w, v, u, t, s |
| 003 | z |
| 004 | r, x, n, o, s |
| 005 | y, r, x, z, q, t, p |
| 006 | y, z, x, e, q, s, t, m |

为构建FP树，需要对原始数据集扫描两遍。

第一遍找出满足最小支持度的元素项。L= {'s': 3, 'r': 3, 't': 3, 'y': 3, 'x': 4, 'z': 5}

# 构建FP树

FP-growth算法还需要一个称为头指针表来指定给定元素项的第一个树节点并记录各个元素项的出现次数。这样每个元素项都构成一条单链表，可以快速访问FP树中一个给定元素项的所有元素。

# 构建FP树

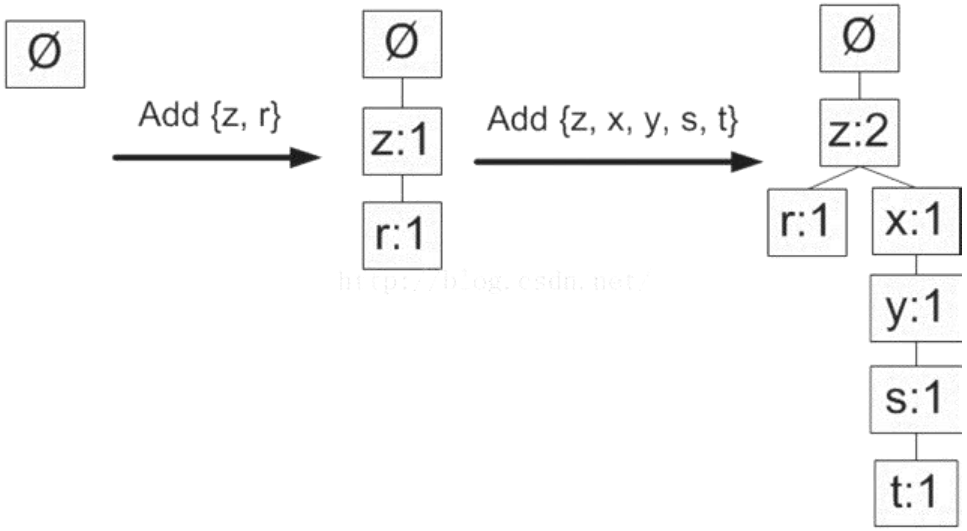
第二遍扫描数据集用来构建FP树。在构建时，读入每个项集并将其添加到一条已经存在的路径中。如果该路径不存在，则创建一条新路径。在将每条事务加到树之前，需要对每个集合进行排序。排序基于元素项的绝对出现频率由高到低来进行，并过滤出不在L中的非频繁项。过滤及重排后的事务如下：

# 构建FP树

| 事务ID | 事务中的元素项 | 过及重排序后的事务 |
| --- | --- | --- |
| 001 | r, z, h, j, p | z,r |
| 002 | z, y, x, w, v, u, t, s | z,x,y,s,t |
| 003 | z | z, |
| 004 | r, x, n, o, s | x,s,r |
| 005 | y, r, x, z, q, t, p | z,x,y,r,t |
| 006 | y, z, x, e, q, s, t, m | z,x,y,s,t |

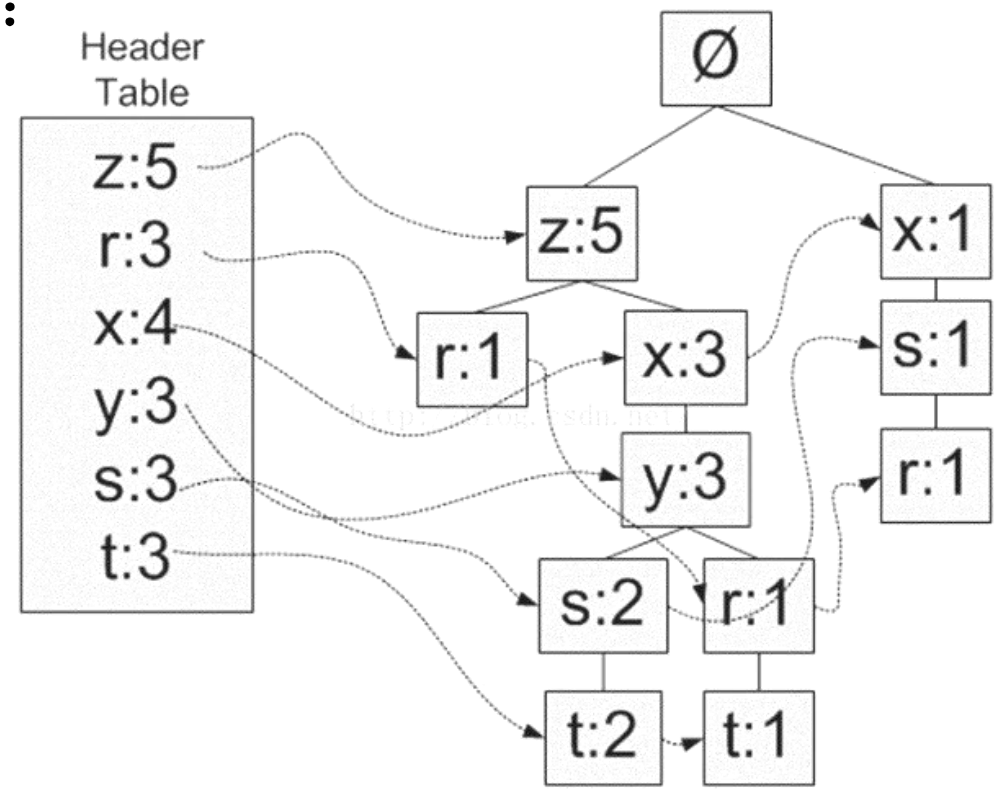
# 构建FP树

在对事务记录过滤和排序之后，就可以构建FP树了。从空集开始，将过滤和重排序后的频繁项集一次添加到树中。如果树中已存在现有元素，则增加现有元素的值；如果现有元素不存在，则向树添加一个分支。



# 构建FP树

依次将每个事务都加到FP树，最后构成的带头指针表(HeaderTable)的FP树如下图所示：



# 从FP树中挖掘频繁项集

从FP树中抽取频繁项集的三个步骤如下：

1. 从FP树中获得条件模式基（conditional pattern base）；
2. 利用条件模式基，构建一个条件FP树；
3. 迭代重复步骤1步骤2，直到树包含一个元素项为止。

# 从FP树中获得条件模式基

首先从头指针表中的每个频繁元素项开始，对每个元素项，获得其对应的条件模式基。条件模式基是以所查找元素项为结尾的路径集合。每一条路径其实都是一条前缀路径（prefix path）。简而言之，一条前缀路径是介于所查找元素项与树根节点之间的所有内容。

每一个频繁项的所有前缀路径（频繁模式基）为：

# 从FP树中挖掘频繁项集

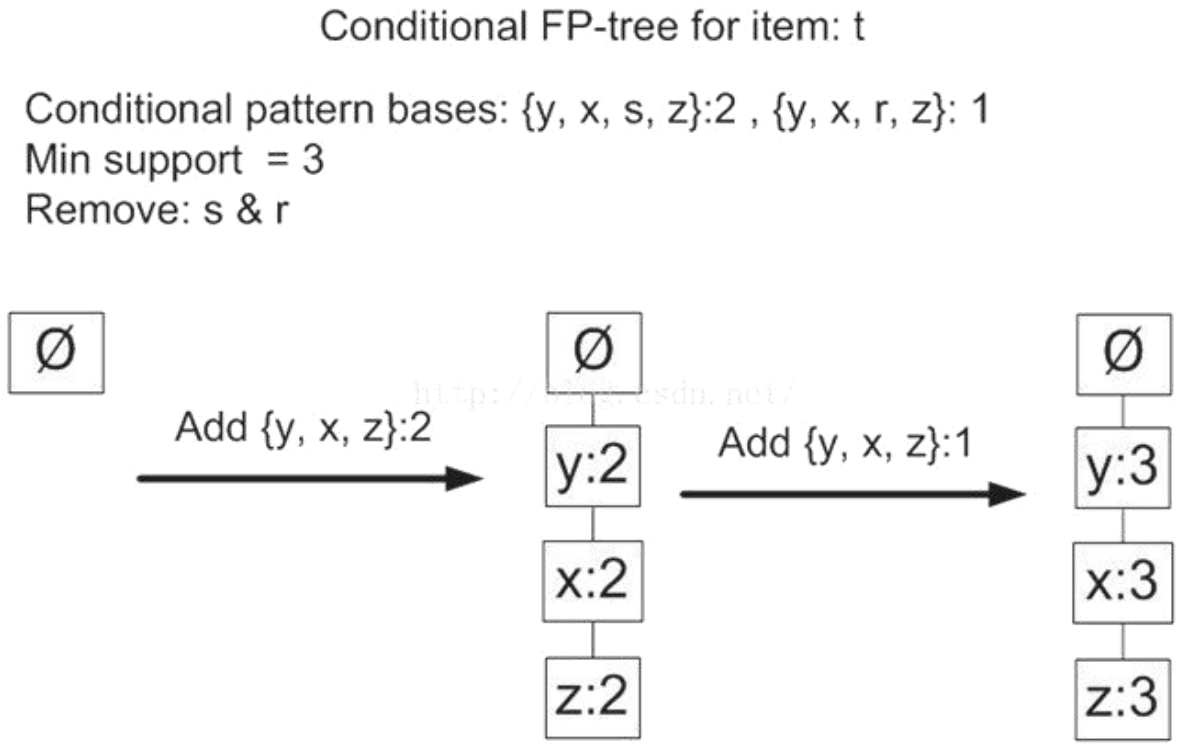
| 频繁项 | 前缀路径 |
| --- | --- |
| z | {}: 5 |
| r | {x, s}: 1, {z, x, y}: 1, {z}: 1 |
| x | {z}: 3, {}: 1 |
| y | {z, x}: 3 |
| s | {z, x, y}: 2, {x}: 1 |
| t | {z, x, y, s}: 2, {z, x, y, r}: 1 |

# 根据条件模式基构建条件FP树

对于每一个频繁项，都要创建一棵条件FP树。可以使用刚才发现的条件模式基作为输入数据，并通过相同的建树代码来构建这些树。例如对于为频繁项t构建一个条件模式树，然后对{t,y},{t,x}…重复该过程

# 从FP树中获得条件模式基

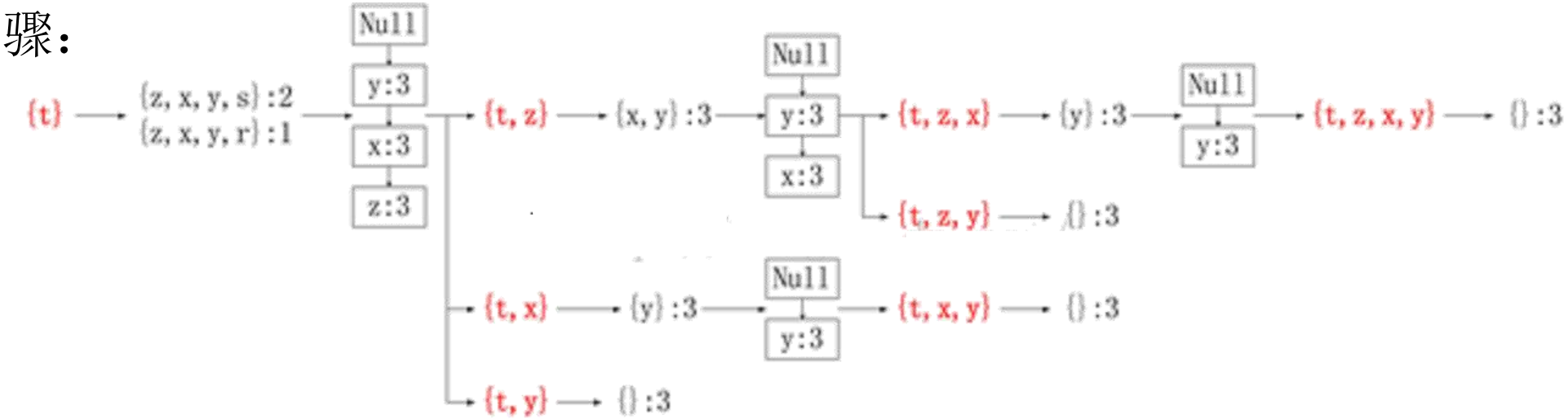
每一个频繁项的所有前缀路径（频繁模式基）为元素t的条件模式树构建过程如下图所示：




注意到元素项s以及r是条件模式基的一部分，但是它们并不属于条件FP树。

# 递归查找频繁项集

有了FP树和条件FP树，就可以在前两步的基础上递归得查找频繁项集。仍然以元素项t为例，下图为查找元素项t的频繁项集的步骤：




图中红色的部分即频繁项集

# 总结

FP-growth算法是一种用于发现数据集中频繁模式的有效方法。由于只对数据集扫描两次，因此FP-growth算法执行更快。在FP-growth算法中，数据集存储在一个称为FP树的结构中。

湖南大學
HUNAN UNIVERSITY

# 谢谢观赏

实事求是　敢为人先