第五章 动态规划方法

算法基本思想

多段图问题

0/1背包问题

流水线调度问题

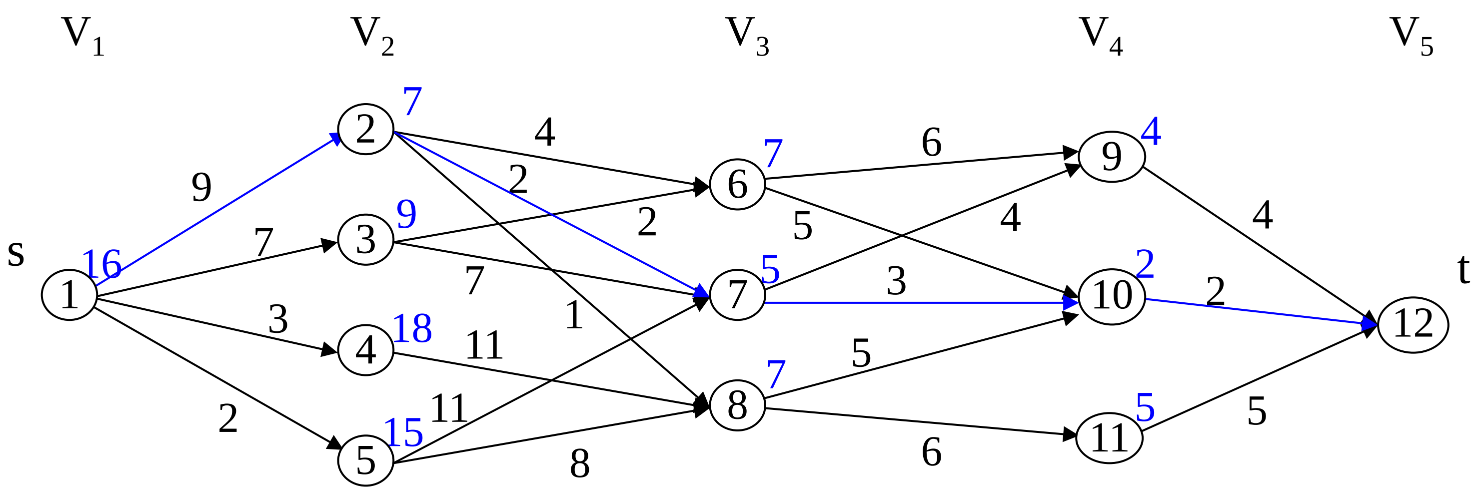
最优二叉搜索树问题

算法基本思想

- **多阶段决策过程**：由一系列决策解决问题。每部分决策都决定问题的一种状态，后面的决策所确定的状态只与当前状态有关，而与如何到达当前状态的过程无关。
- **最优决策序列**：为了达到最优解，当我们从某一状态出发确定后面的决策时，需要枚举后面决策的所有可能序列，从中选择达到最优解的决策序列。（计算量大）
- **最优化原理(Bellman)**：最优决策序列具有性质：不论起点选择何处，后面的决策序列必定关于起点状态构成最优决策序列
- **最优子结构性质**：求原问题计算模型的最优解需包含其（相干）子问题的一个最优解。
- **子问题重叠与备忘录**：同样的子问题多次出现，记录第一次计算信息，以备后面调用，节省计算时间。

多段图问题

- 赋权有向图 G=(V,E)，顶点集V被划分成 k(≥2)个不相交的子集V_i: 1≤i≤k，其中，V_1和V_k分别只有一个顶点s(称为源)和一个顶点t(称为汇)，所有的边(u,v)的始点和终点都在相邻的两个子集V_i和V_{i+1}中，而且u∈V_i，v∈ V_{i+1} 。
- **多阶段图问题**：求由s到t的最小成本路径（也叫最短路径）



- **具有最优子结构性质**：s→t： s, v_2, … , v_i,v_{i+1}, … , v_{k-1}, t一最优
  v_i → t: v_i,v_{i+1}, … , v_{k-1}, t 一最优子序列

多段图动态规划算法

- **最优值递推关系式：** COST(i,j)－第 i 段中节点 j 到 t 的最优路径成本

$$COST(i,j) = \min_{l \in V_{i+1},(j,l)\in E} \{c(j,l) + COST(i+1,l)\}$$

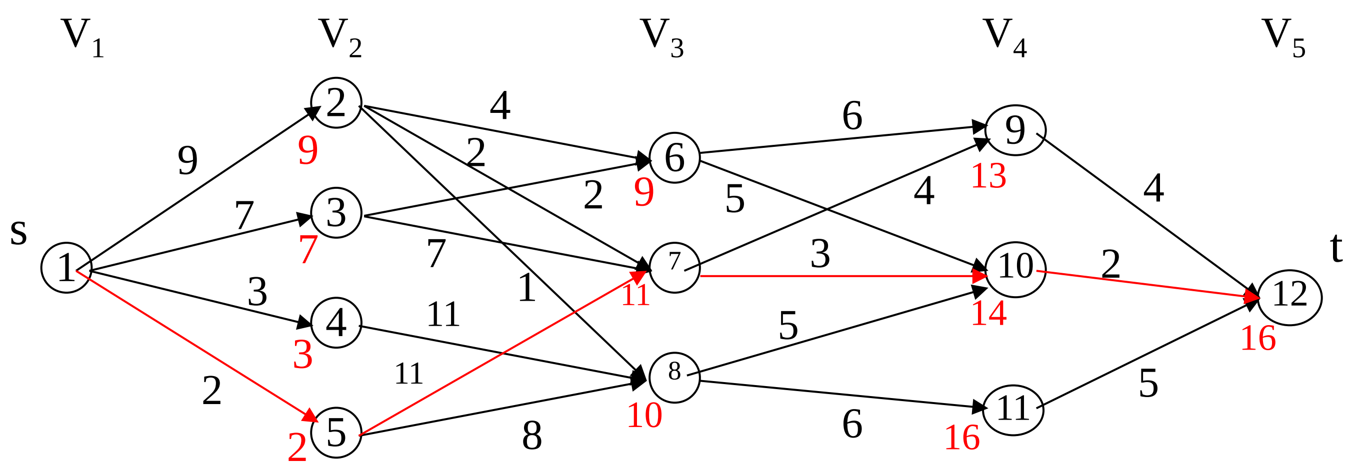
- **算法**

MultiGraph(E, k, n, P) //有n个顶点的k段图G（按段序统一编号），E是边集，
//c(i, j)是边(i, j)的成本，P[1..k]是最小成本路径。

```
float COST[1..n]; integer D[1..n-1]，P[1..k], r, j, n;
COST[n]:=0;
for j from n-1 by –1 to 1 do
  设 r 是这样一个顶点，(j, r)∈E 且使得 c(j, r)+COST[r] 取最小值
  COST[j]:= c(j, r)+COST[r];
  D[j]:=r;  //指出j的后继
end{for}
P[1]:=1; P[k]:=n; //最短路径的起点为s，终点为t
for i from 2 to k-1 do
  P[i]:=D[P[i-1]];  //最短路径上的第i个节点是第i-1节点的后继
end{for}
end{MultiGraph}
```

时间复杂度分析

- 算法中，D[j]将由j到汇顶点t的最短路径上j后面的顶点记录下来，P[i]则记录由源节点s到汇节点t的最短路径上处于第i 阶段中的顶点。语句10是根据数组D中的信息逐段寻找到由源顶点s到汇顶点t的最短路径上的各个顶点。
- 如果用邻接链表表示图G，则语句4中r的确定可以在与$d^{+}(j)$成正比的时间内完成。因此，语句3－7的for循环所需的时间是$\Theta(n+|E|)$ ，循环体9－11所需时间是$\Theta(k)\le n$ 。因而算法MultiGraph的时间复杂度是 $\Theta(n+|E|)$ 。
- **由前向后的处理方法(备忘录方法)**



矩阵连乘积问题

- 给定n个数字矩阵A_1，A_2，…，A_n，其中A_i与A_{i+1}是可乘的，设A_i是$p_{i-1}\times p_i$矩阵, i=1,2,…,n。
- 求矩阵连乘$A_1A_2\cdots A_n$的加括号方法，使得所用的数乘次数最少
- 三个矩阵连乘：$(A_1A_2)A_3$和$A_1(A_2A_3)$，乘法次数分别为

  $p_0p_1p_2+p_0p_2p_3$和$p_0p_1p_3+p_1p_2p_3$
- 例子：$p_0=10$, $p_1=100$, $p_2=5$, $p_3=50$, 两种方法：7500 和 75000
- 最优子结构性质：$(A_1\ldots A_k)(A_{k+1}\ldots A_n)$

$$m[1][n] = \min_{1\le k\le n}\{ m[1][k] + m[k+1][n] + p[0]*p[k]*p[n]\}$$

- 目标值递推关系式

$$m[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i\le k\le j}\{ m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]\} & i < j \end{cases}$$

矩阵连乘的递归算法

```
int RecurMatrixChain(int i, int j)
{
  if (i= =j) return 0;
  int u=RecurMatrixChain(i, i)
        +RecurMatrixChain(i+1,j)
        +p[i-1]·p[i]·p[j];
  s[i][j]=i;
  for(int k=i+1; k<j; k++){
  int t=RecurMatrixChain(i,k)
       +RecurMatrixChain(k+1,j)
       +p[i-1]·p[k]·p[j];
  if (t<u) {
    u=t;
    s[i][j]=k;}
  }
  return u;
}
```

加括号数

$$C(n)=\frac{1}{n+1}\binom{2n}{n}=\Omega(4^n/n^{3/2})$$

- T(n)表示该算法的时间复杂度，则

$$T(n)\ge\begin{cases}O(1) & n=1\\ 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1) & n>1\end{cases}$$

- 当 $n>1$ 时，

$$T(n)\ge 1+(n-1)+\sum_{k=1}^{n-1}T(k)+\sum_{k=1}^{n-1}T(n-k)$$

$$=n+2\sum_{k=1}^{n-1}T(k)$$

- 用数学归纳法直接证明

$$T(n)\ge 2^{n-1}=\Omega(2^n)$$

- 子问题的重复出现：A_iA_{i+1}大约会出现$4^{i-1}+4^{n-i-1}$次。加括号数P(n)

$$P(n)=\begin{cases}1 & n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & n>1\end{cases}$$

- P(n)=C(n-1)—Catalan 数

矩阵连乘的动态规划算法

```
void MatrixChain(int p, int n,
         int * *m, int * *s)
{
   for (int i=1; i<=n; i++)
     m[i][i]=0;
     for (int r=2; r<=n; r++){
       for (int i=1; i<=n-r+1; i++){
         int j=i+r-1; \\ r是跨度
         m[i][j]= m[i+1][j]
                  +p[i-1]·p[i]·p[j];
         s[i][j]=i;
```

```
          for (int k=i+1; k<j; k++){
            int t= m[i][k]
                 +m[k+1][j]
                 +p[i-1]·p[k]·p[j];
                if (t< m[i][j]) {
                  m[i][j]=t;
                  s[i][j]=k; }
               }
            }
        }
     }
  }
```

- 算法的时间复杂度为$O(n^3)$

算法过程示例

6个矩阵连乘：P=[30,35,15,5,10,20,25]

$$m[2][5]=\min\begin{cases} m[2][2]+m[3][5]+p_1p_2p_5=0+2500+35\times15\times20=13000 \\ m[2][3]+m[4][5]+p_1p_3p_5=2625+1000+35\times5\times20=7125 \\ m[2][4]+m[5][5]+p_1p_4p_5=4375+0+35\times10\times20=11375 \end{cases}$$

$$=7125$$

计算 $m[i][j]$ 以及 $s[i][j]$ 的过程

$m[i][j]$

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

跨度增加方向

$s[i][j]$

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

回溯过程

```
void Traceback(int i, int j, int * * s)
{
  if (i= = j) return;
  Traceback(i, s[i][j], s);
  Traceback(s[i][j]+1, j, s);
   cout << "Multiply A" << i
         << "," << s[i][j];
   cout << "and A" <<(s[i][j] +1)
         << "," << j << endl;
}
```

- 以s[i][j]为元素的2维数组却给出了加括号的全部的信息。因为s[i][j]=k说明，计算连乘积A[i..j]的最佳方式应该在矩阵A_k和A_{k+1}之间断开，即最优加括号方式为

  (A[i..k])(A[k+1..j])。

  可以从 s[1][n]开始，逐步深入找出分点的位置，进而得到所有括号。

0/1背包问题

- 容量为C的背包，n件物品，第i件重量和价值分别是w_i和p_i, i=1, 2, …, n。要将这n件物品的某些件完整地装入背包中。
- 给出装包方法，使得装入背包的物品的总价值最大。
- **数学规划问题**：
- $$\max \sum_{1\le i\le n} p_i x_i$$ $$\text{s.t.} \quad \sum_{1\le i\le n} w_i x_i \le C, \quad x_i \in \{0,1\}, i = 1, 2, \cdots, n$$
- **最优子结构性质**：若 $y_1, y_2, \cdots, y_n$ 是原问题的最优解，则 $y_2, \cdots, y_n$ 是0/1背包问题的下述子问题：

$$\max \sum_{2\le i\le n} p_i x_i$$
$$\text{s.t.} \quad \sum_{2\le i\le n} w_i x_i \le c - y_1 w_1, \quad x_i \in \{0,1\}, i = 2, \cdots, n$$

的最优解。
- **目标值递推关系式**：$m[k][X] = \max\{m[k-1][X], m[k-1][X - w_k] + p_k\}$

分析装包决策及出现的各种可能状态

- 有 k 件物品往背包里装时，背包中物品重量和价值可能出现的各种情况

$$k=0:\quad WP_0=\{(0,0)\}$$

$$k=1:\quad WP_1=\{(0,0),(w_1,p_1)\}$$

$$k=2:\quad WP_2=\{(0,0),(w_1,p_1),(w_2,p_2),(w_1+w_2,p_1+p_2)\}$$

$$k=3:\quad WP_3=\left\{\begin{matrix}(0,0), & (w_1,p_1), & (w_2,p_2), & (w_1+w_2,p_1+p_2), \\ (w_3,p_3), & (w_1+w_3,p_1+p_3), & (w_2+w_3,p_2+p_3), & (w_1+w_2+w_3,p_1+p_2+p_3)\end{matrix}\right\}$$

- WP_k 与 WP_{k+1} 之间的关系

$$WP_{k+1}=WP_k\cup\left((w_{k+1},p_{k+1})+WP_k\right),\ k=1,2,\cdots,n-1$$

- 点对(w,v)溢出，若w > C；点对(w,v)覆盖点对(w′,v′)，若w ≤ w′ ，且v ≥ v′
- 整理 WP_k ：抛弃溢出点对，剔除被覆盖点对，然后按照重量值从小到大给点对编号。此时，价值必也是依编号递增的。
- 算法流程：

$$WP_0=S^0\to S^0\cup\left((w_1,p_1)+S^0\right)\xrightarrow{\text{整理}}S^1\to S^1\cup\left((w_2,p_2)+S^1\right)\xrightarrow{\text{整理}}S^2$$

$$S^2\to\cdots\to S^{n-1}\cup\left((w_n,p_n)+S^{n-1}\right)\xrightarrow{\text{整理}}S^n$$

例子：

- $n=4,(w_1,w_2,w_3,w_4)=(2,3,4,4),(p_1,p_2,p_3,p_4)=(1,2,5,6),\ \ C=6$

$$S^0=MP_0=\{(0,0)\},(w_1,p_1)=(2,1),$$
$$MP_1=\{(0,0),(2,1)\},\quad S^1=MP_1,(w_2,p_2)=(3,2),$$
$$MP_2=\{(0,0),(2,1),(3,2),(5,3)\},\quad S^2=MP_2,(w_3,p_3)=(4,5),$$
$$MP_3=\{(0,0),(2,1),(3,2),(5,3),(4,5),(6,6),(7,7),(9,8)\},$$
$$S^3=\{(0,0),(2,1),(3,2),(4,5),(6,6)\},(w_4,p_4)=(4,6),$$
$$MP_4=\{(0,0),(2,1),(3,2),(4,5),(6,6),(4,6),(6,7),(7,8),(8,11),(10,12)\},$$
$$S^4=\{(0,0),(2,1),(3,2),(4,6),(6,7)\}$$

- 回溯求解：

$$(6,7)\in S^4\setminus S^3,\Rightarrow x_4=1;\quad (6,7)-(4,6)=(2,1)$$
$$(2,1)\in S^3\cap S^2,\Rightarrow x_3=0;\quad (2,1)\in S^2\cap S^1,\Rightarrow x_2=0;$$
$$(2,1)\in S^1\setminus S^0,\Rightarrow x_1=1;$$

- 最优解为 $x=(1,0,0,1)$ ，最优值为 7。

0/1背包问题的动态规划算法

```
DKnapsack(w, p, c, n, m) //数组w, p中的
   //元素分别代表各件物品的重量和价
   //值，n是物品个数，c代表背包容量
  float p[1..n], w[1..n], B[1..m], V[1..m],
         ww, pp, c;
  integer F[0..n], l, h, i, j, next;
  F[0]:=1; B[1]:=0; V[1]:=0;
  s:=1; t:=1; //S^0的首和尾
  F[1]:=2; next:=2; // B,V中的第一个空位
  for i to n-1 do
   k:=s;
   u:=最大指标r， 使得s≤ r ≤ t,
      而且B[r]+w_i ≤ c;  //抛弃溢出点对
   for j from s to u do
     (ww, pp):=( B[j]+w_i, V[j]+p_i);
    while k ≤ t && B[k] < ww do
       B[next]:=B[k]; V[next]:=V[k];
       next:=next+1; k:=k+1;
    end{while}
•    if k ≤ t && B[k]=ww then
•      pp:=max(V[k], pp);  k:=k+1;
•    end{if}   //更换一点对
•    if pp > V[next-1] then
•     (B[next], V[next]):=(ww, pp);
•     next:=next+1;  //决定放入
•    end{if}
•    while k ≤ t && V[k]<V[next-1] do
•       k:=k+1;  //剔除被淹没点对
•    end{while}
   end{for}
   while k ≤ t do // 将S^{i-1}剩者并入S^i
•    (B[next], V[next]):= (B[k], V[k]);
•     next:=next+1; k:=k+1;
   end{while}
•   s:=t+1; t:=next-1; F[i+1]:=next;
•   //为S^{i+1}赋初始位置
  end{for}
  traceparts // 逐一确定最优解分量
end{DKnapsack}
```

算法复杂度

- 算法DKnapsack 的主要工作是产生诸S^i。在$i > 0$的情况下，每个S^i由S^{i-1}和$(w_i,p_i)+S^{i-1}$ 归并整理而成，因此 $|S^i| \le 2|S^{i-1}|$ (在最坏情况下，没有序偶会被清除) 所以

$$\sum_{1\le i\le n-1} |S^i| = \sum_{1\le i\le n} 2^{i-1} = 2^n - 1$$

- 由此知，算法DKnapsack的空间复杂度是 $O(2^n)$。
- 由S^{i-1}生成S^i需要$\Theta(|S^{i-1}|)$ 的时间，因此，计算 $S^0, S^1, \cdots, S^{n-1}$ 总共需要的时间为

$$\sum_{1\le i\le n} |S^{i-1}| \le \sum_{1\le i\le n} 2^{i-1} = 2^n - 1$$

- 算法DKnapsack的时间复杂度为 $O(2^n)$ 。
- 如果物品的重量w_i和所产生的效益值p_i都是整数，那么，S^k中的元素(b,v)的分量 b 和v 也都是整数，且 $b \le c, v \le \sum_{1\le k\le i} p_k$ 。又 S^i 中不同的元素对应的分量也都是不同的，故

$$|S^i| \le 1 + \sum_{1\le k\le i} p_k$$

$$|S^i| \le 1 + \min\left\{\sum_{1\le k\le i} w_k, c\right\}$$

- 此时算法DKnapsack的时间复杂度为 $O(\min\left\{2^n, nc, n\sum_{i=1}^{n} p_k\right\})$ 。

动态规划算法设计的基本步骤

- **分析最优解的结构**
  选定要解决问题的一个计算模型，其具有最优子结构性质
- **建立递推关系式**
  关于目标值最优值的递推计算公式，有时可能不是一个简单的解析表达式
- **设计求最优值的迭代算法**
  因为要使用已经处理过的子问题的计算结果，所以采用迭代方法，计算过程中需要保留获取最优解的线索，即记录一些信息。
- **用回溯方法给出最优解**
  把求最优值算法的计算过程倒回来，借用那里保留的信息就可追溯到最优解。

流水作业调度问题

- n个作业$\{1, 2, . . . , n\}$，在由两台机器M_1和M_2组成的流水线上完成加工。每个作业加工的顺序都是先在M_1上加工，然后在M_2上加工。M_1和M_2加工作业 i 所需的时间分别为a_i和 b_i，$1\le i \le n$。
- 流水作业调度问题要求确定这n个作业的最优加工次序，使得从第一个作业在机器M_1上开始加工，到最后一个作业在机器M_2上加工完成所需的时间最少
- **最优子结构性质** $N=\{1, 2, . . . , n\}$，$S\subseteq N$， $T(S,t)$，$T(N,0)$

当机器M_1开始加工S中的作业时，机器M_2可能正在加工其它的作业，要等待时间t后才可用来加工S中的作业。这种情况下流水线完成S中的作业所需的最短时间记为$T(S, t)$ 。一个最优调度π： $\pi(1), \pi(2), \ldots, \pi(n)$ 一加工顺序。往证

$$T(N,0) = a_{\pi(1)} + T(N\backslash\{\pi(1)\},b_{\pi(1)})$$

若π'是作业集$N\backslash\{\pi(1)\}$在机器M_2等待时间为$b_{\pi(1)}$情况下的一个最优调度，则加工

次序 $\pi(1), \pi'(2), \ldots, \pi'(n)$ 是作业集N的一个调度(机器M_2不需等待时间)，它所用的加工时间即为 $a_{\pi(1)} + T(N\backslash\{\pi(1)\},b_{\pi(1)})$，但π是最优调度，所以$T(N,0) \le a_{\pi(1)} + T(N\backslash\{\pi(1)\},b_{\pi(1)})$。另一方面，$T(N,0)= a_{\pi(1)} + T'$， T'是机器M_2等待时间为$b_{\pi(1)}$时，按调度π加工作业集$N\backslash\{\pi(1)\}$所用的时间，于是，$T(N\backslash\{\pi(1)\},b_{\pi(1)}) \le T'$。 得到 $T(N,0) \ge a_{\pi(1)} + T(N\backslash\{\pi(1)\},b_{\pi(1)})$。

目标值递推关系式

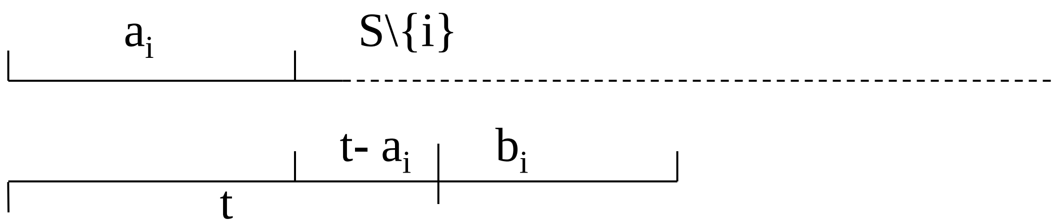
- 初始时机器空闲

$$T(N,0)=\min_{1\le i\le n}\{a_i+T(N\setminus\{i\},b_i)\}$$

- 一般情况

$$T(S,t)=\min_{i\in S}\left\{a_i+T\left(S\setminus\{i\},b_i+\max\{t-a_i,0\}\right)\right\}$$

- 其中 b_i+max(t-a_i,0)= b_i+max(t, a_i)-a_i是安排作业集S\{i}时，机器M_2需要等待的时间。如下图所示



- 动态规划算法，时间复杂度 $O(2^n)$ ，Johnson改进算法$O(n\log n)$ 。
- Johnson不等式：作业i对作业j

$$\min\{a_i,b_j\}\le\min\{a_j,b_i\}$$

- Johnson命题：两个机器的流水作业调度π是最优调度当且仅当下述Johnson不等式满足 $\min\{a_{\pi(i)},b_{\pi(i+1)}\}\le\min\{a_{\pi(i+1)},b_{\pi(i)}\}$，$1\le i\le n-1$

Johnson命题的证明

- 如果调度π将作业i安排在作业j前，则

$$T(S,t)=a_i+T\left(S\setminus\{i\},b_i+\max\{t-a_i,0\}\right)=a_i+a_j+T(S\setminus\{i,j\},t_{ij})$$

其中 $t_{ij}=b_j+b_i-a_j-a_i+\max\{t,a_i+a_j-b_i,a_i\}$

- 交换作业i与作业j的加工顺序，则得到另一个调度π′，它所用时间为

$$T'(S,t)=a_j+T\left(S\setminus\{j\},b_j+\max\{t-a_j,0\}\right)=a_i+a_j+T(S\setminus\{i,j\},t_{ji})$$

- 其中 $t_{ji}=b_j+b_i-a_j-a_i+\max\{t,a_i+a_j-b_j,a_j\}$
- 如果作业i对作业j满足Johnson不等式，则 $t_{ij}\le t_{ji}$。可见，在一个调度π中，如果前面的作业对于后面的作业满足Johnson不等式，则必然会省时间。如果在调度π的加工顺序中，每个作业对于它后面的作业都满足Johnson不等式，则π必是最优调度。反之亦然。

Johnson算法

- **基本思想：**

1）令 $AB=\{i\mid a_i<b_i\},\quad BA=\{i\mid b_i\le a_i\}$

2）将 AB 中作业依 a_i 的非减次序排列

将 BA 中作业依 b_i 的非增次序排列

3）AB 中作业接 BA 中作业即构成满足 Johnson法则的最优调度。

- **算法伪代码**

```
FlowShop(a,b,n,p)  // 给作业排序，数组p
  //记录作业号的一个排列
 float a[1..n],b[1..n];
 integer c[1..n],d[1..n],p[1..n], n,j,k;
 j:=1; k:=n;
 for i to n do
   if a[i]<b[i] then
     d[j]:=i; c[j]:=a[i]; j:=j+1;
  else
     d[k]:=i; c[k]:=b[i]; k:=k-1;
   end{if}
 end{for}
 MergeSortL(c,1,k,q);
 MergeSortL(c,k+1,n,r);
 j:=q[0];
 for i to k do
   p[i]:=d[j];
   j:=q[j];
 end{for}
 j:=r[0];
 for i to n-k do
   p[n-i+1]:=d[k+j];
   j:=r[j];
 end{for}
end{FlowShop}
```

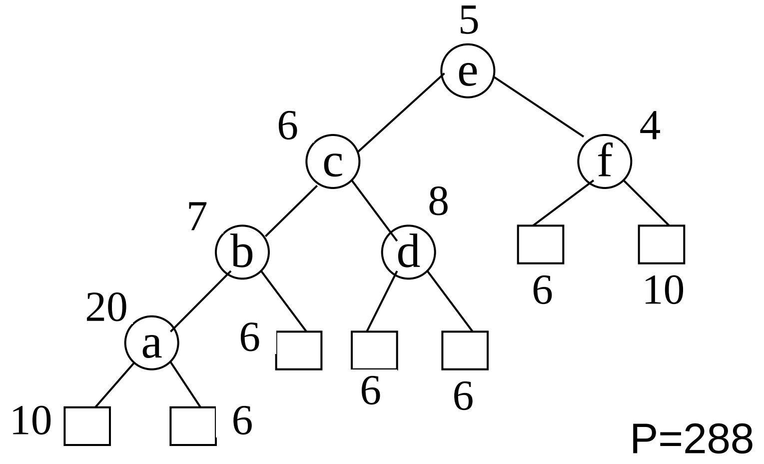
最优二叉搜索树

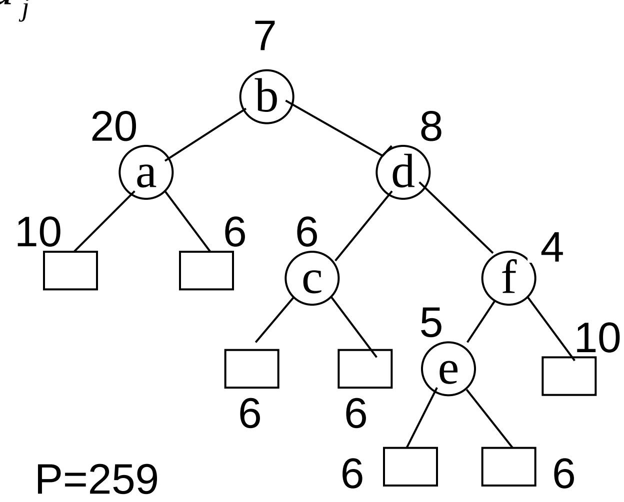
- **有序集** $S:\quad x_1 < x_2 < \cdots < x_n$，
- **二叉树 T**：一个内部节点存储一个数，一个叶节点代表一个区间：(x_i,x_{i+1})，内节点的值大于其左儿子节点的值，小于其右儿子节点的值。
- **存取概率分布**：x是数x_i的概率为b_i，位于区间(x_i,x_{i+1}) 的概率为a_i，约定x_0=-∞，x_{n+1}=+ ∞。

$$a_i \geq 0,\ \ 0 \leq i \leq n, b_j \geq 0, 1 \leq j \leq n;\ \sum_{i=0}^{n} a_i + \sum_{j=1}^{n} b_j = 1$$

- **平均路长**(搜索次数)：存储x_i的节点的深度为c_i，代表区间(x_i,x_{i+1})的叶节点的深度为d_i，平均路长为

$$p = \sum_{i=1}^{n} b_i(1+c_i) + \sum_{j=0}^{n} a_j d_j$$

- 



最优子结构性质

- **观点：** 二叉搜索树T的一棵含有节点 $x_i, x_{i+1}, \cdots, x_j$ 和叶节点 $(x_{i-1}, x_i), (x_i, x_{i+1}), \cdots, (x_j, x_{j+1})$ 的子树可以看作是有序集 $S(i, j): x_i < x_{i+1} < \cdots < x_j$ 关于全集为实数区间(x_{i-1}, x_{j+1})的一棵二叉搜索树，T自身可以看作是有序集 S： $x_1 < x_2 < \cdots < x_n$ 关于全集为整个实数区间 $(-\infty, +\infty)$ 的二叉搜索树。这样建立了二叉搜索树T与子搜索树T_{ij}的联系。根据存取概率分布，x 在子树T_{ij}上被搜索到的概率为

$$w_{ij} = \sum_{i-1 \le k \le j} a_k + \sum_{i \le k \le j} b_k$$

做为二叉搜索树的子问题，$\{x_i, x_{i+1}, \cdots, x_j\}$ 的存储概率分布为 $\{\bar{a}_{i-1}, \bar{b}_i, \cdots, \bar{b}_j, \bar{a}_j\}$ ，其中 $\bar{b}_k = b_k / w_{ij}$， $i \le k \le j$; $\bar{a}_h = a_h / w_{ij}$， $i - 1 \le h \le j$ 是条件概率

- **最优子结构性质**

设T_{ij}是最优二叉搜索树，根节点为x_m，左右子树分别为T_l，T_r。p_{ij}，p_l，p_r分别是二叉搜索树T_{ij}，T_l，T_r的平均路长，则T_l，T_r中节点的深度等于它们在T_{ij}中的深度减1。

$$p_{ij} = \sum_{i-1 \le k \le j} \bar{d}_k a_k / w_{ij} + \sum_{i \le k \le j} (\bar{c}_k + 1) b_k / w_{ij} \implies w_{ij} p_{ij} = \sum_{i-1 \le k \le j} \bar{d}_k a_k + \sum_{i \le k \le j} (\bar{c}_k + 1) b_k$$

目标值递归关系式

$$w_{i,m-1}p_l = \sum_{i-1\le k\le m-1}(\bar{d}_k - 1)a_k + \sum_{i\le k\le m-1}\bar{c}_k b_k\ ,\quad w_{m+1,j}p_r = \sum_{m\le k\le j}(\bar{d}_k - 1)a_k + \sum_{m+1\le k\le j}\bar{c}_k b_k$$

$$\Rightarrow\ w_{ij}p_{ij} = w_{ij} + w_{i,m-1}p_l + w_{m+1,j}p_r\ (\because \bar{c}_m = 0)$$

由于T_l是有序集$\{x_i,\cdots,x_{m-1}\}$的一棵二叉搜索树，故 $p_l \ge p_{i,m-1}$ 。若 $p_l > p_{i,m-1}$ 则用 $T_{i,m-1}$ 替换 T_l 可得到平均路长比T_{ij}更小的二叉搜索树。这与T_{ij}是最优二叉搜索树矛盾。所以， T_l是最优二叉搜索树。同理可证，T_r 也是一棵最优二叉搜索树。

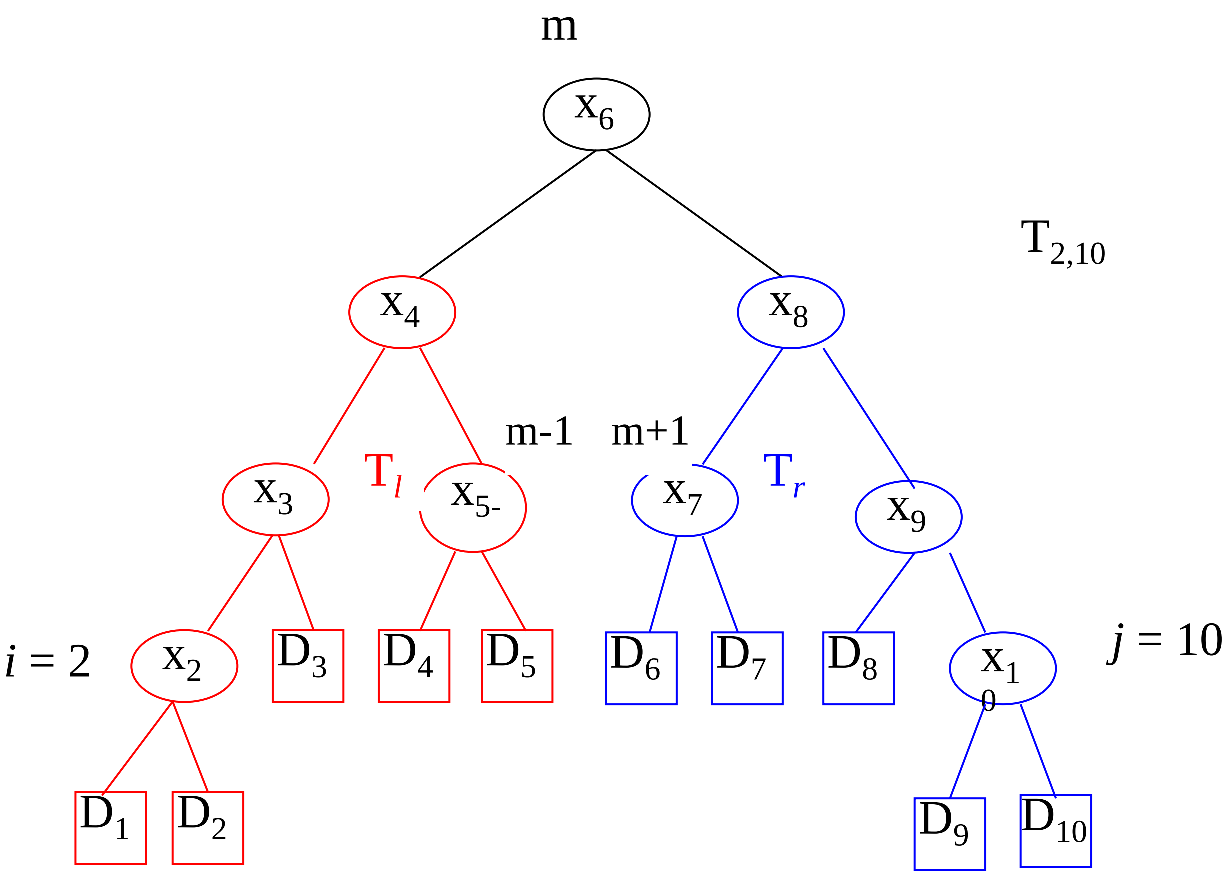
- **目标值递归关系式**

$$w_{ij}p_{ij} = w_{ij} + \min_{i\le k\le j}\{w_{i,k-1}p_{i,k-1} + w_{k+1,j}p_{k+1,j}\},\quad i \le j$$

- 令 $m(i,j)=w_{ij}p_{ij}$

$$m(i,j) = w_{ij} + \min_{i\le k\le j}\{m(i,k-1) + m(k+1,j)\},\quad i \le j$$

- $$m(i,i-1) = 0,\quad i = 1,2,\cdots,n$$



m
x6
T2,10
x4
x8
m-1
m+1
Tl
Tr
x3
x5-
x7
x9
i = 2
j = 10
x2
D3
D4
D5
D6
D7
D8
x10
D1
D2
D9
D10

最优二叉搜索树的动态规划算法

```
void OBSTree( int a, int b, int n, int
                    **m, int **s, int **w)
{
  for (int i = 0; i < n; i++) {
    w[i+1][i] = a[i];
    m[i+1][i] = 0;
  }
  for (int r = 0; r < n; r++) {
     for (int i = 1; i <= n-r; i++) {
        int j = i+ r ;
        w[i][j] = w[i][j-1] + a[j] +b[j];
        m[i][j] = m[i+1][j];
        s[i][j] = i;
        for (int k =i + 1; k<= j; k++) {
           int t = m[i][k-1] + m[k+1][j];
           if (t < m[i][j]) {
             m[i][j] = t;
             s[i][j] = k; }
          }
          m[i][j] + = w[i][j];
      }
  }
}
```

- 算法的时间复杂度为 $O(n^3)$

- s[i][j]保存最优子树的根顶点中元素， s[1][n]=k表明整个二叉搜索树的根节点是x_k，通过s[1][k-1]和s[k+1][n]找到它的两个儿子，此推