习题参考答案

1.1

最优解决例子：百米比赛中金牌获得者的判断方法，只能是跑最短时间的那个选手。

“近似”最优解例子：通常数学解题中用 π 的近似值 3.14 来代替 π。

1.2

在该算法中，当整数集 S 确定后，它的所有子集也就确定了，所以是否含有和为 m 的子集也就可以求出来，该算法是确定的，可行的。另外集合 S 的元素个数 n 是有穷的，而它的子集的总个数为 2^n，同样也是有穷的。所以算法也是有穷的。综上可得该算法满足算法的特点。

1.3

15 12 1 3 7 19 12 15
12 15 1 3 7 19 12 15
1 12 15 3 7 19 12 15
1 3 12 15 7 19 12 15
1 3 7 12 15 19 12 15
1 3 7 12 15 19 12 15
1 3 7 12 12 15 19 15
1 3 7 12 12 15 15 19

1.4

```
InsertSort(A)
1  for  j←2 to n do
2          key ← A[j]
3          i← j-1
4          while i > 0 and A[i] < key do     //修改地方
5                  A[i + 1]← A[i]
6                  i ← i -1
7          A[i + 1] ← key
8  return A
```

1.5

FindMax (A)	cost	times
1 max← A[1]	c_1	1
2 **for** j←2 **to** n **do**	c_2	n
3 **if** A[j]>max **then**	c_3	n-1

4 max←A[j] c_4 $\sum_{2}^{n} t_i$

$T(n) = c_1 + c_2 * n + c_3 * (n-1) + c_4 * \sum_{2}^{n} t_i$

当数组 A 的第一个元素为最大时该算法达到最佳，此时 $t_i = 0$ ：

$T(n) = c_1 + c_2 * n + c_3 * (n-1) = (c_2 + c_3) * n + c_1 - c_3 = \Theta(n)$

当数组 A 是递增时该算法达到最坏，此时 $t_i = 1$ ：

$T(n) = c_1 + c_2 * n + c_3 * (n-1) + c_4 * (n-1) = (c_2 + c_3 + c_4) * n + c_1 - c_3 - c_4 = \Theta(n)$

1.6

循环不变量：在 **for** 循环第 j 个迭代执行前，max 为当前数组 $A[1..j-1]$中的最大值

初始步：在第一轮迭代前，j=1，当前数组中只有一个元素 max =A[1]，此时 max 确是最大的，所以在初始化中不变量是正确的。

归纳步： 在 **for** 循环第 $j=k$ 个迭代执行前，max 是 $A[1,...,k-1]$ 中的最大值，此后执行第 $j=k$ 个迭代，将 max 与 $A[k]$ 进行比较并将较大的值赋给 max，所以 max 仍然是 $A[1,..., k]$中的最大值。

终止步：当 $j=n+1$ 时，for 循环终止，此时 max 是 A[1,...,n] 中的最大元素，即是全数组中的最大值，所以算法是正确的。

1.7

```
MaxIndex(A)
1  max←A[1]
2  j←1
3  for i←2 to n do
4     if max≤A[i] then
5        max←A[i]
6        j←i
7  return j
```

1.8

```
Exp(a, n)
1  i←1
2  pow←1
3  while i≤n do
4        pow←pow×a
5        i←i+1
6  return pow
```

循环不变量：当执行第 i 次循环前，pow 的值为 a^{i-1}

初始步：当 i=1，即在第一轮循环前，pow 为 1，若 n=0，则不循环，返回 1，若 n=1，则循环一次，pow=a，所以不变量正确

归纳步：当执行第 i=k 次循环前，假设 pow 的值为 a^{k-1}，当执行第 i=k 次循环后，则由算法可知，pow=pow*a= a^{k-1} *a= a^{k} ，所以不变量仍正确

终止步：当 i=n+1 时，循环退出，此时共执行了 n 次循环，所以结果为 a^n ，算法是正确的

1.9

Searchx(A, x)		cost	times
1	i←1	c_1	1
2	**while** x≠A[i] and i≤n **do**	c_2	t
3	i←i+1	c_3	t-1

//其中 t≤n+1

$$T(n) = c_1 + c_2 t + c_3(t-1)$$

当第一个元素就是指定数 x 时该算法达到最好情形，此时 t=1：

$$T(n) = c_1 + c_2 = \Theta(1)$$

当该数组中不含指定数 x 时该算法达到最坏情形，此时 t=n：

$$T(n) = c_1 + c_2(n+1) + c_3 n = \Theta(n)$$

正确性证明：

循环不变量：当执行第 i 次循环前，指定数 x 不在当前数组 A[1..i-1]中

初始步：当 i=1，即在第一轮循环前，当前数组为空，不包含指定数，所以循环不变量成立。

归纳步：当执行第 i=k 次循环前，指定数 x 不在当前数组 A[1..k-1]中，当执行第 i=k 次循环后，此时表明 x≠A[i]且 i≤n，否则，找找到了指定数。因此，在执行第 i=k+1 次循环前，指定数 x 不在当前数组 A[1..k]中，所以循环不变量成立。

终止步：当 i=n+1 时，循环退出，此时，表明指定数 x 不在当前数组 A[1..n]中。否则，在前面任一次循环退出，均找到了指定数，所以整个算法是正确的。

1.10

要使运行时间 $100*n^2$ 比 2^n 快，则需要：

$$100*n^2 \ < \ 2^n$$

通过计算得 $n \geq 15$

所以得 n 最小值为 15

1.11

要使插入排序优于合并排序，则：

$$8n^2 < 64n\lg n$$

$$\Rightarrow n/8 < \lg n$$

$$\Rightarrow 2^{n/8} < n$$

$$\Rightarrow 2 \le n \le 43$$ // 这两个值是通过计算得来的

实验题

1.11 完成 XOJ 如下题目：1000，1001，1002，1003，1004。

1000 A+B 1001 宅男健身计划 1002 C=?+? 1003 Sort 1004 Sort Ver.2

注：1000 1001 1003 必练习，1002 和 1004 可不做

参考答案

2.1

要证明 $f(n)=\Theta(g(n))$ 当且仅当存在正常数 c_1,c_2,n_0，使得当 $n\geq n_0$ 时，

$c_1g(n)\leq f(n)\leq c_2g(n)$。取 $c_1=3/5,c_2=1$ 代入不等式，由左不等式可得 n>=21/13,由右不等式可得 n>=0。所以取 $n_0=21/13$。所以当 $n\geq n_0,c_1=3/5,c_2=1$ 时，$f(n)=\Theta(g(n))$

2.2

因为 $f(n)\leq \max(f(n),g(n))$，$g(n)\leq \max(f(n),g(n))$

所以有 $f(n)+g(n)\leq 2\max(f(n),g(n))$

即 $\frac{1}{2}(f(n)+g(n))\leq \max(f(n),g(n))$

又因为 $\max(f(n),g(n))=f(n)$或者$g(n)$

所以 $\max(f(n),g(n))\leq f(n)+g(n)$

对于$n\geq n_0$,取$c_1=\frac{1}{2},c_2=1$,可得

$c_1(f(n)+g(n))\leq \max(f(n),g(n))\leq c_2(f(n)+g(n))$

故所证成立。

2.3

要证 $(n+a)^b=\theta(n^b)$ 当且仅当存在正常数 a_0,c_1,c_2，使得 $c_1n^b\leq(n+a)^b\leq c_2n^b$

先设$c_1=(\frac{1}{2})^b$，则由$c_1(n^b)\leq(n+a)^b\Leftrightarrow(\frac{1}{2}n)^b\leq(n+a)^b$

$\Leftrightarrow\frac{1}{2}n\leq n+a\Leftrightarrow-\frac{1}{2}n\leq a\Leftrightarrow n\geq -2a$

又因为$n\geq 0$，所以可取 $n_0=|2a|$，使得对于任何的a，当$n\geq n_0$时，

有 $(\frac{1}{2}n)^b\leq(n+a)^b$

而当$n_0=|2a|$时，$(n+a)^b\leq(n+|a|)^b\leq(n+\frac{n}{2})^b\leq(\frac{3}{2}n)^b\leq(2n)^b=2^bn^b$

所以 C_2可取2^b

所以当$c_1=(\frac{1}{2})^b, c_2=2^b, n_0=|2a|$时，可证$(n+a)^b=\theta(n^b)$

//根据极限来证也可以。

2.4

要证 $f(n)=\Theta(n^2)$ 当且仅当存在正常数 c_1,c_2,n_0，使得当 $n\geq n_0$ 时，$c_1n^2\leq f(n)\leq c_2n^2$，取 $c_1=a/2, c_2=2a$，代入左不等式可得：$\frac{a}{2}n^2+bn+c\geq 0$, 求出 $n\geq\frac{-b+\sqrt{|4ac-b^2|}}{a}$。代入右不等式可得：$an^2-bn-c\geq 0$, 求出 $n\geq\frac{b+\sqrt{|-4ac-b^2|}}{2a}$。取 $n_0=\max(\frac{-b+\sqrt{|4ac-b^2|}}{a},\frac{b+\sqrt{|-4ac-b^2|}}{2a},0)$。当 $n\geq n_0$ 时，$f(n)=\Theta(n^2)$

2.5

要证明 $f(n)=O(g(n))$ 当且仅当存在正常数 c,n_0，使得当 $n\geq n_0$ 时，有 $f(n)\leq cg(n)$，即 $n^2\leq cn^3$，求解得 $c\geq 1/n\geq 0$，所以对于所有的 $n\geq 0$，有 $f(n)=O(g(n))$

2.6

$2^{n+1}=O(2^n)$ 成立，因为存在正常数 $c\geq 2$，使得当 $n\geq 0$ 时，有 $2^{n+1}\leq c2^n$

$2^{2n}=O(2^n)$ 不成立，因为根据不等式 $2^{2n}\leq c2^n$ 计算可得 $c\geq 2^n$，当 n 无穷大时，c 不存在，所以不成立。

2.7

由 $f(n)=O(g(n))$ 可得存在正常数 c,n_0，当 $n\geq n_0$ 时，有 $f(n)\leq cg(n)$，可得 $1/cf(n)\leq g(n)$，所以存在一个正常数 $c_1=1/c$，当 $n\geq n_0$ 时，$c_1f(n)\leq g(n)$，所以 $g(n)=\Omega(f(n))$

2.8

要证明 $n^3=\Omega(n^2)$ 当且仅当存在正常数 c,n_0，使得当 $n\geq n_0$ 时，有 $cn^2\leq n^3$，求解得

$c \geq 1/n \geq 0$，所以对于所有的$n \geq 0$，有$n^3 = \Omega(n^2)$

2.9

按照定义证明即可，比较简单，略（充分性，必要性）

2.10

由已知可得存在正常数n_1, n_2, c_1, c_2，使得当$n \geq n_1$时，有$f_1(n) \leq c_1 g(n)$。当$n \geq n_2$时，有$f_2(n) \leq c_2 g(n)$。所以$f_1(n) \times f_2(n) \leq c_1 g(n) \times c_2 g(n) = c_1 c_2 (g(n) \times g(n))$，所以当$n_0 = \max(n_1, n_2)$，$c = c_1 c_2$时，$f_1(n) \times f_2(n) = \mathrm{O}(g(n) \times g(n))$

参考答案

3.1

只雇用一次的概率是：$1/n$

雇佣 n 次的概率是：$1/n$！

刚好雇佣两次的概率：

$$(n-1)!+(n-1)!+1!(n-2)!+.....+3!(n-4)!+2!(n-3)!+1!(n-2)!$$

3.2

最大值有可能在 n 个数的数组中的任一个位置出现，因此在每个位置出现的概率为 $1/n$

假设在位置 k 出现，则需要比较 k 次，因此总的比较次数为

$$T(n)=\sum_{k=1}^{n}k\frac{1}{n}$$

3.3

对题意的理解：

①一个由 n 个 PUSH、POP 和 MULTIPUSH 操作构成的序列。

②一个由 n 个 PUSH、POP、MULTIPOP 和 MULTIPUSH 操作构成的序列。

最坏情况下： 一次 MULTIPUSH 操作时间为 $O(n)$

n 个操作都为 MULTIPUSH 操作

3.4

令 c_i 表示第 i 个运算的费用，则有

$$c_i=\begin{cases}i & i\text{为2的整数幂}\\ 1 & \text{否则}\end{cases}$$

则运算序列 1 2 3 4 5 6 7 8…

费用依次为 1 2 1 4 1 1 1 8…

故 n 个运算的费用为

$$\sum_{i=1}^{n}c_i\le n+\sum_{j=0}^{\lg n}2^j=n+(2n-1)<3n$$

故每次运算的分摊费用 O(1)

3.5

给 Push，Pop 运算分摊费用为 2， 给复制栈运算的分摊费用为 0。每调用一次 Push 或 Pop 运算，均有 1 美元存款。因为堆栈的大小不会超过 k，因此实际的复制费用也不会超过 k，它将由复制的元素的存款来支付。在进行一次复制栈运算之前要进行 k 次的 Push 或 Pop 运算，因此存款至少为 k， 所以保证每次存款始终非负，即总的分摊费用为运算序列实际费用的一个上界，为 $2n$，即 O(n)。

3.6

令 c_i 表示第 i 个运算的费用，则有

$$c_i = \begin{cases} i & i\text{为2的整数幂} \\ 1 & \text{否则} \end{cases}$$

给每个运算的分摊费用为 $\hat{c}_i = 3$

如果 i 为 2 的整数幂，则用存储的钱支付 i
否则支付 1，存款 2。

运算序列	1	2	3	4	5	6	7	8…
分摊费用为	3	3	3	3	3	3	3	3…
实际费用依次为	1	2	1	4	1	1	1	8…
存款为	2	3	5	4	6	8	10	5

故 n 个运算的总分摊费用为 $\sum_{i=1}^{n} \hat{c}_i = 3n$

而 n 个运算的实际费用为

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n-1) < 3n$$

因此总分摊费用是实际费用的上界

故每次运算的分摊费用 $O(1)$，n 个运算的费用为 $O(n)$

3.7

定义 $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)\ (i \ge 1)$

则 $\Phi'(D_0) = \Phi(D_0) - \Phi(D_0) = 0$，$\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \ge 0$

分摊代价为：

$$\begin{aligned} \hat{c}_i' &= c_i + \Phi'(D_i) - \Phi'(D_{i-1}) \\ &= c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_{i-1}) - \Phi(D_0)) \\ &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \hat{c}_i \end{aligned}$$

3.8

令 $\phi(D_0) = 0$，定义

$$\phi(D_i) = \begin{cases} 0 & i\text{为2的整数幂} \\ 2 + \phi(D_{i-1}) & \text{否则} \end{cases}$$

其中 $i \ge 1$。注意到对任意的 $i \ge 0$，均有 $\phi(D_i) \ge 0$。如果 i 不是 2 的整数幂，则有

$$\hat{c}_i = 1 + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3$$

如果i是 2 的整数幂，例如对某个正整数k，$i = 2^k$，则有

$$\hat{c}_i = i + \Phi(D_i) - \Phi(D_{i-1}) = 2^k + 0 - \Phi(D_{2^k-1}) = 2^k - 2((2^k - 1) - 2^{k-1}) = 2$$

因此有$\hat{c}_i = O(1)$。

3.9

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$= 2 * n - S_n + S_0$

3.10

根据书上可以类似分析。

参考答案

4.1

将数组 A 的第 i 个元素与 x 比较，若相等则返回 i；否则递归比较第 i+1 个元素；若 i>n 则数组不含 x，返回 0；

```
Seachx(A, x, i)
1   if i>n    then    return 0
2   else if A[i]=x then
3              return i
4          else
5         return Searchx(A,x,i+1)
```

4.2

先求出第 i 个元素在数组中出现的个数 count，如果大于当前最大的个数 maxcount,则将 count 赋值给 maxcount，并将对应当前个数最多的元素 k 赋值为 $A[i]$；然后再用同样的方法递归比较第 i+1 个元素出现的个数 count 与 maxcount 的大小。

```
FindMostElements(A,i)
1   if i>n then return k and maxcount
2   else
3        count←0
4        for j=1 to n do
5             if A[j]=A[i] then
6                  count←count+1
7        if maxcount<count then
8             maxcount  ←count
9             k←A[i]
10       FindMostElements(A,i+1)
```

4.3

```
RecInsertSort(A, j)
1     if j=n+1 then return A
2     else
3          key←A[j]
4          i←j-1
5          while i>0 and A[i]>key do
6                A[i+1]←A[i]
7                i←i-1
8     A[i+1]←key
9     RecInsertSort(A,j+1)
```

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ T(n-1)+(n-1) & \text{if } n>1\end{cases}$$

求解过程同例 4.2，这里略。

4.4

当 n=1 时，只有一个元素时，只可能产生一种排列方式，直接输出 P[1]，显然是对 P[1]的全排列。

假设算法对 $n=j$ 个元素的全排列是正确的，即有 j!个全排列，当 $n=j$+1 时，由算法可知它是经过 for 循环将这 j+1 位上的元素一一与第一位的元素交换，余下按 $n=j$ 全排列，这样总共有 j!*（j+1）种排列，又因为作为第一元素的 j+1 个元素是互不相同的，

所以这 j+1 个不同元素分别作为第一元素而递归求解，由于假定 j 个元素的全排列由于包含元素不同以致全排列不会相同，所以这 $(j+1)*j!$ 种排列是互不相同的，而 $(j+1)*j!=(j+1)!$ 是对 $(j+1)$ 个元素的全排列，所以算法对 $n=j+1$ 个元素的全排列是正确的！

4.5

所谓字典序输出排列，指的是将排列由小到大依次进行输出。

4.6

因为 perm2(m)函数是从大到小的顺序将各个值排列到数组 P 中，所以在调用 perm2(m)时，m+1,m+2,...,n 已经被赋值到 P[1...n]中了，共有 n-m 项，所以 P 中包含 0 的项恰好只剩下 n-(n-m)=m 个。因为当将 m 每赋值给 P 中一个为 0 的项时，就执行一次 perm2(m-1)。因为有 m 个 0；因此递归调用 perm2(m-1)将恰好执行 m 次。

4.7

```
GeneratingPermuation()
1    for j←1 to n do
2         P[j] ←0
3    Perm2(1)

Perm2(m)
1    if m=n+1 then output P[1..n]
2    else
3        for j←1 to n do
4            if P[j]=0 then
5                P[j] ←m
6                Perm2(m+1)
7                P[j] ←0
```

4.8

```
GeneratingPermuation()
1    for j←1 to n do
2         P[j] ←0
3    Perm2(n)

Perm2(m)
1    if m=0 then output P[1..k]
2    else
3     for j←1 to n do
4           if P[j]=0 then
5               P[j] ←m
6               Perm2(m-1)
7               P[j] ←0
```

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=0\\ nT(n-1)+n & \text{if } n\geq 1\end{cases}$$

猜想 $T(n)\leq c(n!-1)$，代入可得

$$
\begin{aligned}
T(n) &= nT(n-1)+n \\
&\le n(c((n-1)!-1))+n \\
&= cn!-cn+n \\
&= c(n!-1)+c-cn+n \\
&\le c(n!-1)
\end{aligned}
$$

只要$c-cn+n<0$即可，当 $c>1$，且当 $n>c$ 显然成立。

4.9

要证$T(n)\le c(n-b)\lg(n-b)$，假设对$\lfloor n/2\rfloor$成立，则

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2\rfloor+17)+n \\
&\le 2c(\lfloor n/2\rfloor-b+17)\lg(\lfloor n/2\rfloor-b+17)+n \\
&\le 2c(n/2-b+17)\lg(n/2-b+17)+n \\
&\le c(n-2b+34)[\lg(n-2b+34)-\lg 2]+n \\
&\le c(n-2b+34)\lg(n-2b+34)+(1-c)n+2b-34 \\
&\le c(n-b)\lg(n-b)
\end{aligned}
$$

只要$b\ge 34, c\ge 1$即可证明。

4.10

因为$a=2,b=2,f(n)=n$，$n^{\log_b^a}=n=f(n)$，由公式法可得

$T(n)=\Theta(n\lg n)$

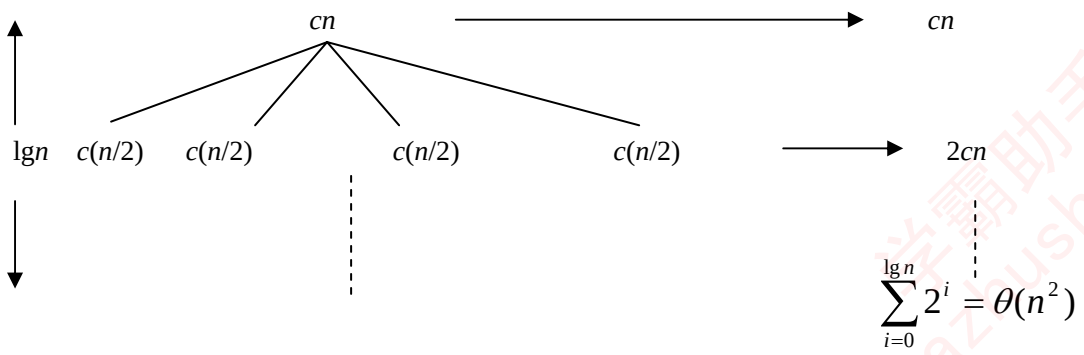
所以得证。

4.11

令$m=\lg n, S(m)=T(2^m)$，则有$T(2^m)=T(2^{m/2})+1$因此

$S(m)=S(m/2)+1$，用公式法可得$T(n)=\Theta(\lg\lg n)$。

4.12



替换法证明：

猜想$T(n) \le dn^2 - dn$，假设对$\lfloor n/2 \rfloor$成立，则：

$$T(n) = 4T(\lfloor n/2 \rfloor) + cn$$

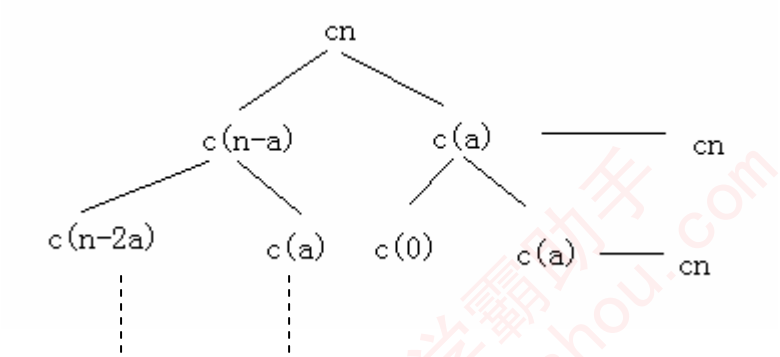
$$\le 4(d\lfloor n/2 \rfloor^2 - d\lfloor n/2 \rfloor + cn$$

$$\le dn^2 - 2dn + cn$$

$$\le dn^2 - dn + cn - dn$$

只要$cn - dn < 0$，即 $d>c$ 即可。

4.13



树的深度为$\lfloor n/a \rfloor$，且有$\lfloor n/a \rfloor$+1 层，每层代价都为 cn.

$$T(n) = T(n-a) + T(a) + cn$$

$$\le cn(\lfloor n/a \rfloor + 1)$$

$$\le cn(n/a + 1)$$

$$= (c/a)n^2 + cn$$

猜想$T(n) = \Theta(n^2)$

4.14

猜想：$T(n) \le dn\lg n$，代入原式得：

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

$$\le T(n/3) + T(2n/3) + cn$$

$$\le d(n/3)\lg(n/3) + d(2n/3)\lg)2n/3) + cn$$

$\le dn\lg n - dn(\lg 3 - 2/3) + cn$

$\le dn\lg n$

只要$-dn(\lg 3 - 2/3) + cn \le 0$，即$d \ge c/(\lg 3 - 2/3)$

4.15

(1) $T(n) = \theta(n^2)$

(2) $T(n) = \theta(n^2 \lg n)$

(3) $T(n) = \theta(n^3)$

4.16

不能。

假设用公式法，因为$a = 4, b = 2, f(n) = n^2 \lg n$，可得：

$n^{\log_b^a} = n^2 \le f(n)$，应用情形 3 得，存在$\varepsilon > 0$，使得$f(n) = \Omega(n^{\log_b^{a+\varepsilon}})$，

又因为情况 3 中要求：

$af(n/b) = 4f(n/2) = n^2 \lg n - n^2 \le cf(n) = cn^2 \lg n, c < 1$

解得：$1 - c \le 1/\lg n$，即

$c \ge 1 - 1/\lg n$ 1-c<=1/lgn

当$n \to \infty$时，$c \ge 1$与要求矛盾，所以不能用公式法。

猜想：$T(n) \le cn^2 \lg n \lg n$,代入得：

$$T(n) \le 4c(n/2)^2 \lg(n/2)\lg(n/2) + n^2 \lg n$$
$$= cn^2(\lg n - \lg 2)^2 + n^2 \lg n$$
$$= cn^2(\lg n - 1)^2 + n^2 \lg n$$
$$= cn^2 \lg n \lg n - 2cn^2 \lg n + cn^2 + n^2 \lg n$$

当$-2cn^2 \lg n + cn^2 + n^2 \lg n \le 0$时，即$c \ge 1/2$时，

$T(n) \le O(cn^2 \lg n \lg n)$

实验题

4.17 完成 XOJ 如下题目：1005，1071，1082。
1005 Complete Permutation 必做 1071 和 1082 题目改变，可删除
4.18 完成 POJ 如下题目：1007，1173，1187，1240，1426，2083，3601。

参考答案

5.1

将数组中相邻元素两两配对，用合并算法将它们排好序，构成 $n/2$ 组长度为 2 的排好序子数组段，然后将它们排序成长度为 4 的排好序子数组段，如此下去直到整个数组排好序。

例子：　5　2　8　4　6　1　3　7

S=1　(5)　(2)　(8)　(4)　(6)　(1)　(3)　(7)

S=2　(2　5)　(4　8)　(1　6)　(3　7)

S=3　(2　4　5　8)　(1　3　6　7)

(1　2　3　4　5　6　7　8)

IterMergeSort(*A*, *n*)

```
1       s←1
2       while s<n do
3           MergeAB(A, B, s, n) //合并 A 中大小为 s 的相邻子数组，存放到 B 中
4           s←s+s
5           MergeAB(B, A, s, n)   //合并 B 中大小为 s 的相邻子数组，存放到 A 中
6           s←s+s
```

5.2

$$C=\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}=\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

$$C=\begin{pmatrix} d_1+d_4-d_5+d_7 & d_3+d_5 \\ d_2+d_4 & d_1+d_3-d_2+d_6 \end{pmatrix}$$

$$d_1=(A_{11}+A_{22})(B_{11}+B_{22})$$
$$d_2=(A_{21}+A_{22})B_{11}$$
$$d_3=A_{11}(B_{12}-B_{22})$$
$$d_4=A_{22}(B_{21}-B_{11})$$
$$d_5=(A_{11}+A_{12})B_{22}$$
$$d_6=(A_{21}-A_{11})(B_{11}+B_{12})$$
$$d_7=(A_{12}-A_{22})(B_{21}+B_{22})$$

$$C=\begin{pmatrix} 12 & -36 \\ 2 & -35 \end{pmatrix}$$

5.3

算法思想：

1、先将问题划分为大小近似相同的两个子问题

2、对子问题递归调用该算法进行处理，递归出口为子问题只含一个元素，这时将其与 x 直接比较，若等于 x 则返回该元素在数组中的位置

3、由于给定数组是有序的，因而可以根据 x 的大小判断 x 位于那个子问题内，而将另一个不含 x 的子问题丢弃

Searchx(*A*, *p*, *r*, *x*)

```
1       if p>r then
2           return 0
```

```
3       else
4           q← (p+r)/2
5           if x=A[q] then
6                   return q
7           else   if   x< A[q] then
8                       return    Searchx(A, p, q-1, x)
9                   else
10                      Searchx(A, q+1, r, x)
```

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ T(\lfloor n/2 \rfloor)+\Theta(1) & \text{if } n\geq 2.\end{cases}$$

由公式法可得 $T(n)=O(\lg n)$

5.4

算法思想：

1、先将数组 A 划分为大小近似相等的两个子数组，令 $k=n/2$，首先做 k 次比较 $a[i]$和 $a[k+i]$，把小的放前面，大的放后面。这样经 k 次比较后我们有 $a[i]<a[k+i]$

2、用 k-1 次比较找出 $A[1..k]$中最小者，再用 k-1 次比较找出 $A[k+1..n]$中最大者。找出的这两个数即为所求的最小值和最大值。

3、总比较次数为 $3k-2=3n/2-2$

```
MaxMin(A,n)
1    k←n/2
2    for i←1 to k do
3        if A[i]>A[k+i] then
4            A[i] ↔A[k+i]
5    min ← A[1]
6    for i←2 to k do
7        if min>A[i] then
8                min←A[i]
9    max←A[k+1]
10   for i←k+2 to n do
11       if   max<A[i] then
12               max←A[i]
13   return (min,max)
```

5.5

算法思想：

1、先将问题划分为大小近似相等的两个子问题

2、对子问题递归调用该算法进行处理，递归出口为子问题只含一个元素，这时元素的和就为该元素自己

3、原问题结果为这两个子问题之和

```
Sum(A, p, r)
1       if p=r then
2               return A[p]
```

```
3       else
4           q←(p+r)/2
5           sum1←Sum(A, p, q)
6           sum2←Sum(A, q+1, r)
7           return  (sum1+sum2)
```

5.6

算法思想：

1、先将问题划分为大小近似相等的两个子问题

2、对子问题递归调用该算法进行处理，递归出口为子问题只含一个元素，若该元素等于 x，则返回 x 的出现次数 1，若该元素不等于 x，则返回 0

3、原问题结果为这两个子问题所得结果之和

```
Countx(A, p, r, x)
1   if p=r then
2       if A[p]=x then
3           return 1
4       else
5           return 0
6   else
7       q←(p+r)/2
8       return  (Countx(A, p, q, x)+Countx(A, q+1, r, x))
```

$$T(n)=\begin{cases}\Theta(1) & \text{if } n=1\\ 2T(\lfloor n/2 \rfloor)+\Theta(1) & \text{if } n\geq 2.\end{cases}$$

利用公式法可得 $T(n)=\Theta(n)$

5.7

利用二分搜索的思想。

```
BinarySearch(A,left, right, ix);

BinarySearch(A, p, r,x1,x2,y1,y2)
1   if x1>x2 then
2       x1↔ x2
3   beswap←1
4   while p≤r do
5       q←(p+r)/2
6       if A[q]<x1 then   p←q+1;
7       else if A[q]=x1 then
8           y1←q;
9           y2←BinarySearch(A,q+1,r,x2);
10      else if A[q]>x1 and A[q]<x2 then
11          y1←BinarySearch(A,p,q-1,x1);
12          y2←BinarySearch(A,q+1,r,x2);
```

```
13        else if A[q]=x2 then
14            y2←q;
15            y1←BinarySearch(A,l,q-1,x1);
16        else
17                r←q-1;
18  if beswap=1 then
19                y1↔y2
```

5.8

5.9

算法思想：

1、选择一个元素 $a[r]$作为支点，将数组 A 划分成三个部分：A1、A2、A3，它们分别包含小于、等于和大于 $a[r]$的元素

2、根据数组的长度可以判断第 k 小元素出现在三个数组中的哪一个中，对于包含第 k 小元素的那个数组递归调用算法进行处理，丢弃不含的两部分

Selectimin(A, p, r, k)

1 **if** $p = r$

2 **then return** $A[p]$

3 $q \leftarrow$Partition (A, p, r)

4 $i \leftarrow q - p + 1$

5 **if** $k = i$ // the pivot value is the answer

6 **then return** $A[q]$

7 **elseif** $k < i$

8 **then return** Selectimin ($A, p, q-1, k$)

9 **else return** Selectimin ($A, q+1, r, k-i$)

5.10

参见上一习题答案

5.11

有两种办法，可使用栈，也可以不使用栈，网上都可以搜到，这里略。

5.12

算法思想：

1、利用算法 QuickSort 中的划分过程 Partition 将数组 A 划分成两部分，假设划分过程

返回支点元素 $A[q]$，则 A1=$A[1..q-1]$、A2=$A[q+1..r]$，其中 A1 中元素均比 $A[q]$小，A2 中元素均比 $A[q]$大

2、若 $q=k+1$，则 A1 就为前 k 个最小元素

若 $q>k+1$，则可将 A2 丢弃，继续递归调用算法在 A1 中寻找前 k 个最小元素

若 $q<k+1$，则 A1 为结果的一部分，结果的另一部分为递归调用算法在 A2 中寻找前 $k-q+1$ 个最小元素

算法过程同习题 5.9

5.13

5.14

5.15

算法思想：

1、首先从 T1 和 T2 的根节点开始判断，如果两二叉树的根节点相同，则两二叉树有可能相同

2、递归调用算法，分别判断根的左右子树是否相同，如果根节点相同且左右子树都分别相同，则这两个二叉树相同

```
IsEqual(T1,T2)
1.  if T1=Null and T2=Null then
2.      return true
3.  else if T1≠Null and T2≠Null then
4.      if T1->data=T2->data then
5.          return IsEqual(T1->lchild,T2->lchild) and IsEqual(T1->rchild,T2->rchild)
8.      else
9.          return flase
10.     end if
11. else
12.     return false
13. end if
```

5.16

算法思想：

1、要计算一棵二叉树的高，先分别计算其左右子树的高度，二叉树的高度等于左右子树高度中大的再加 1

2、在计算左右子树高度时递归调用算法，递归出口为遍历到叶子节点时，其高度为 1

```
GetHeight(T)
1.  if T=Null then return 0
2.  else
3.      return max{GetHeight(T－>lchild),
4.                 GetHeight(T－>rchild)}+1
5.  end if
```

5.17

类似习题 5.9 求第 k 大元素，也可以如下求解。

```
FindSecond(A, p, r)
1   if   p=r then return (-∞, A[r])
3   else if r- p =1 then
4            if A[p]<A[r] then return (A[p], A[r])
5            else return (A[r], A[p])
6       else if r- p >1 then
7                 q←(p+r)/2
8                 (x1,y1)←FindSecond(A, p, q)
9                 (x2,y2)←FindSecond(A, q +1, r)
10                if y1>y2 then
11                     y←y1    // 子数组的最大元素
12                     x←max(x1,y2) // 子数组的次大元素
13                else
14                     y←y2
15                     x←max(x2,y1)
16                return (x, y)
17          else
18                return(-∞,-∞)
```

5.18

实验题

5.19 编程序实现残缺棋盘游戏算法，并能用图形演示。

5.20 分别将插入排序、选择排序、合并排序和快速排序编程实现，并使用实验分析方法比较各种算法的效率。

5.21 完成 XOJ 如下题目：1004，1007，1017，1018，1022，1057。

5.22 完成 POJ 如下题目：1002，1947，2082，2282，2299，2318，2379，2418，2726，3122。

3.1

习题参考答案

6.1

$$F(n) = F(n-1) + F(n-2)$$
$$\le 2F(n-1) \le 2^2 F(n-2)$$
$$\le \cdots \le 2^n F(0) = O(2^n)$$

6.2

```
PrintStations(l, j)
1    i← l*
2    l* ← li[j]
3    PrintStations(l, j-1)
4    print "line " i", station " j
```

6.3

分析：

可以不记录 $f_i[j]$，而用两个变量 $f1$ 和 $f2$ 来保存原程序中的 $f1[j-1]$ 和 $f2[j-1]$，并在一轮循环中用 $f1$、$f2$ 来计算 $f1[j]$ 和 $f2[j]$即新的 $f1$、$f2$。因而可以节省 $2n-2$ 个空间，在最后一轮循环后即 j=n 后，使之仍然能够计算出 f^* ，并且仍然能够构造出最快装配路线。 $f1$、$f2$ 保存的为 $f1[n]$ 和 $f2[n]$。根据这两个值，就能计算出 f^* 。

6.4

	1	2	3	4	5	6
6	2010	1950	1770	1860	1500	0
5	1655	2430	930	3000	0	
4	405	330	180	0		
3	330	360	0			
2	150	0				
1	0					

	1	2	3	4	5	6
6	2	2	4	4	5	-
5	4	2	4	4	-	
4	2	2	3	-		
3	2	2	-			
2	1	-				
1						

((A1A2)((A3A4)(A5A6)))

6.5

```
MatrixChainMultiply(Ai…j, s, i, j)
1    if i=j then
2        return Ai
3    else
4        q←s[i,j]
5        A1 ← MatrixChainMultiply(Ai…q, s, i, q)
6            A2 ← MatrixChainMultiply(Aq…j, s, q+1, j)
7        return    MatrixMultiply(A1, A2 )
```

6.6

6.7

备忘录方法是一种自顶向下的高效动态规划方法，它可能包含递归过程，并在第一次解决一个子问题时将结果记录到一个表中，在下一次遇见该子问题时，只需查表而无需再解决一次，因而当某个算法有重叠子问题时，能很高效地解决问题。而 MergeSort 算法的每个子问题都是相互独立的，不存在重叠子问题，因而使用备忘录方法并不能提高效率。

6.8

参考课件

6.9

```
PrintLCS(c, X, Y, i, j)
1       if i = 0 or j = 0 then
2            return 0
3       if X[i]=Y[j] then
4            PrintLCS(c, X, Y, i -1, j -1)
5            print x[i]
6       else if c[i-1, j] >= c[i, j-1] then
7                    PrintLCS(c, X, Y, i-1, j )
8            else
9                    PrintLCS(c, X, Y, i, j -1)
```

6.10

```
MemoizedLCSLength(X, Y, m, n)
1            for i←0    to    m    do
2                    for j← 0    to    n do
3                            if i=0 or j=0 then
4                                    c[i, j] ← 0
5                            else
 6                                   c[i, j] ← -1
 7              LookupLCSLength(X, Y, m, n)
 8              return        c and b
LookupLCSLength(X, Y, i, j)
1       if    c[i, j]≥o then
2              return    c[i, j]
3       if x[i]=y[j] then
4              q← LookupLCSLength(X, Y, i-1, j-1)
5              c[i, j] ← q+1
6              b[i, j] ← "↖"
7      else
8              q1← LookupLCSLength(X, Y, i-1, j-1)
9              q2←LookupLCSLength(X, Y, i-1, j-1)
10             if    q1≤q2 then
11                           c[i, j] ← q1
12                           b[i, j] ← "↑"
13             else
```

```
14                  c[i, j] ← q2
15                  b[i, j] ← "←"
```

6.11

反证法。

假设存在一个不以 A 开始的最长公共子序列，设为 Zk=z1z2…z3 则 Zk 不包括 Xm 和 Yn 的开头字符 A。现在将 A 加入到 Zk 的头部，得到的序列 A Zk 必定为 Xm 和 Yn 的公共子序列，且 A Zk 的长度为 k+1，这就与题设中 Zk 为最长公共子序列相矛盾了。

6.12

```
X=(x1, x2, ……xn)
b[i]= max 1≤k≤i-1 {b[k]}          xk≥xi
      max 1≤k≤i-1 {b[k]}+1        xk≤xi
LSC(X)
1     b[1] ←1
2     for   i←2 to n do
3           k←0
4           for j←1 to i-1 do
5                 if    xj≤xi and k<b[j]   then
6                       k←b[j]
7           b[i] ←k+1
8     return   MAX(b)
```

6.13

6.14

p[*i*, *w*]：当背包容量为 *w* 时，可以从物品 1 到 *i* 获得的最大价值

(1)不装入物品 *i*　　p[*i*, *w*]=p[*i*-1, *w*]

(2)装入 1 个物品 i　　p[*i*, *w*]=p[*i*-1, *w*-w_i]+v_i

(3)装入 2 个物品 i　　p[*i*, *w*]=p[*i*-1, *w*-2w_i]+2v_i

递归方程为：

p[*i*, *w*]= max{p[*i*-1, *w*], p[*i*-1, *w*-w_i]+v_i , p[*i*-1, *w*-2w_i]+2v$_i$}

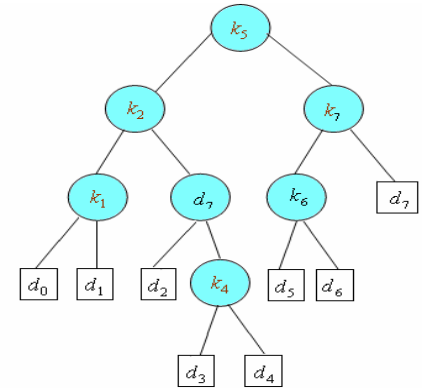
```
012knapsack(W, v, w)
1   for   i←2 to n do
2     for   w← 0 to W do
3       if   2wi<=w then
4           p[i,w] ←max{p[i-1,w],p[i-1,w-wi]+vi, p[i-1, w-2wi]+2vi}
5       else   if   wi<=w   then
6                  p[i, w] ←max{p[i-1, w], p[i-1, w-wi]+vi }
7            else
8                  p[i, w] ←p[i-1, w]
```

6.15

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.1	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

最优代价为 $E(T)$=2.96

最优二叉树：



6.16

当用直接公式计算时，则求 $w[i, j]$ 需要的时间为 $O(n)$，OptimalBST 算法中有两个 **for** 嵌套的循环和嵌套在这两个外循环的两个独立的内循环。所以算法的时间复杂度为：

$$T(n) = O(nn(n+n)) = O(n^3)$$

6.17

```
OptimalBST(p, q, n)
1   for i←1 to n + 1 do
2               e[i, i– 1] ← qi-1
3               w[i, i– 1] ← qi-1
4   for i←1 to n do
5               root[i,i] ←i
6   for l← 1 to n do
7               for i ← 1 to n–l + 1 do
8                       j ←i + l–1
9                       e[i, j ]←∞
10                      w[i, j ] ← w[i, j–1] + pj+qj
11                      for r←root[i,j-1] to root[i+1,j] do
12                              t ← e[i, r–1] + e[r + 1, j ] + w[i, j ]
13                              if t < e[i, j] then
14                                      e[i, j] ← t
15                                      root[i, j ] ←r
16  return e and root
```

6.18

不妨设 $a_1 \le a_2 \le \cdots \le a_n$，假设用 $a_1, a_2, \cdots, a_i$ 硬币找钱数 j 的最少硬币个数为 $c[i, j]$，

设找钱序列为 $x_1, x_2, \cdots, x_i$，其中 x_k 表示面值 a_k 需要的个数。则最优子结构性质为：

假设 $x_1, x_2, \cdots, x_i$ 是最优找钱序列，则 $x_1, x_2, \cdots, x_{i-1}$ 是只用面值 $a_1, a_2, \cdots, a_{i-1}$ 找钱 $j - x_i a_i$ 的一个最优找钱序列。

由最优子结构性质可得如下递归方程

$$c[i, j] = \min_{0 \le k \le (j/a_i)} (k + c[i-1, j - k a_i])$$

初始条件为 $c[i,0] = 0, i = 1,...,n$

$$c[1, j] = \begin{cases} \infty & j \bmod a_1 \neq 0 \\ j / a_1 & otherwise \end{cases}$$

```
FindMoney()
1   for i←1 to n do
2       c[i,0]←0
3   for i←1 to j do
4       if i≥a[1] then
5           if i%a[1]=0 then c[1,i]←i/a[1] else c[1,i]←Max
6       else
7           c[1,i]←Max
8   for i←2 to n do
9       for k←1 to j do
10          if k≥a[i] then
11              if c[i,k-a[i]]+1 < c[i-1,k] then c[i,k]←c[i, k-a[i]]+1
12              else c[i,k]←c[i-1,k]
13              if c[i,k]≥Max then c[i,k]←Max
14          else
15              c[i,k]←c[i-1,k]
16  k←j
17  s←n
18  if c[s, k]<Max then
19      while k>0 do
20              if c[s,k]=c[s-1,k] then s←s-1
21              else
22                  x[s]←x[s]+1;
23                  k←k-a[s]
```

6.19

问题是需要求出将字符串 A 转换为字符串 B，所需要的最少的字符运算数。请设计一个有效算法，对任给的 2 个字符串 A 和 B，输出将字符串 A 转换为字符串 B 所需的最少字符运算数。

设两个字符串为 $A_i =< a_1, a_2, \cdots, a_i >$ 和 $B_j =< b_1, b_2, \cdots, b_j >$，令 $c[i, j]$ 表示将 A_i 转换

为 B_i 所需要的最小的字符运算数，则有下面三种情况：

1)删除 A_i 一个字符 a_i，则 $c[i, j]= c[i\text{-}1, j]+1$；

2)插入一个字符 b_j，则 $c[i, j]= c[i, j\text{-}1]+1$；

3)将一个字符改写为另一个字符，如果 $a_i=b_j$，则需要字符运算 $c=0$，否则需要 $c=1$，

因此 $c[i, j]= c[i\text{-}1, j\text{-}1]+c$

综上所述，递归方程为 $c[i, j]=\min\{ c[i\text{-}1, j]+1, c[i, j\text{-}1]+1, c[i\text{-}1, j\text{-}1]+c \}$

详细的算法过程如下

```
TransformString(A, B)
1    for i←0 to m do
2          c[i,0] = i
3    for j←0 to n do
4          c[0,j] = j
5    for i←1 to m do
6        for j = 1 to n do
7            c[i,j]←Max
8            c[i,j]←min{c[i,j], c[i – 1,j] + 1}
9            c[i,j]←min{c[i,j], c[i,j - 1] + 1}
10           if  a_i = b_j  then c←0
11           else c←1
12           c[i,j]←  min{c[i,j], c[i – 1,j - 1] + c}
```

6.20

设 n 堆货物从左到右编号为 1,2,⋯,n，每堆货物的货物数分别为 A1,A2,⋯,An，n 堆货物的合并可以有许多不同的方式，每一种合并方式都对应于 A1,A2,⋯,An 的一种完全加括号方式。这样类似矩阵链的乘积问题求解，令合并子问题 $A_i, A_{i+1}, \cdots, A_j$ 所需要的最小总代价为 $m[i, j]$，则有如下递归方程：

$$m[i, j]=\begin{cases} 0 & \text{if } i=j \\ \min\limits_{i\le k\le j}\{m[i,k]+m[k+1, j]+\sum\limits_{r=i}^{j} A_r\} & \text{if } i<j \end{cases}$$

详细的伪代码略。

6.21

算法思想：从三角形顶至三角形底的最优路径包含了该路径上各点至三角形底的最优路径。令 a[i,j]表示数字三角形某点的数字，设 f[i,j]表示数字 a[i, j]至三角形底的最优路径，则

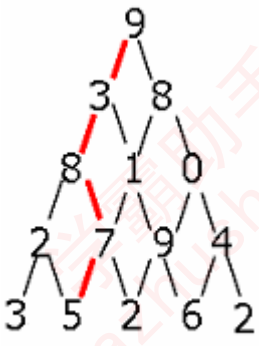


图 6.21

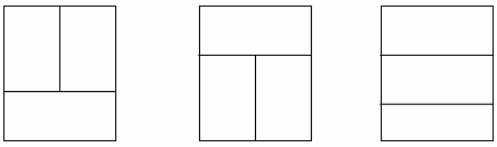
$$f(i,j)=\begin{cases} a[i,j] & i=n \\ \max\{f[i+1,j],f[i+1,j+1]\}+a[i,j] & i<n \end{cases}$$

详细的伪代码略。

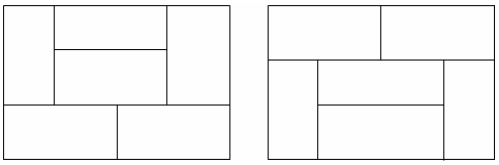
6.22

由题可求出切割布条的方法有两种：

一种是占用的大小，这种切割法有三种方案（该算法中有考虑上下次序），如下图所示：



另外一种是占用的大小，这种切割法有两种方案，如下图所示：



对整条布的切割其实可看为是先用的方法切割或先用的方法切割，余下的布条用同样的方法切割。令 $F(n)$表示将 $3\times n$ 的布条按照要求切割的种数，则递归方程为：

$$F(n)=\begin{cases} 1 & n=1 \\ 3 & n=2 \\ 3F(n-2)+2F(n-4) & n>2 \end{cases}$$

详细的伪代码略。

6.23

对整条路的走法与对部分路的走法是一样的，设 $f(n)$表示为走 n 米的走法总数，则递归方程为：

$$f(n)=\begin{cases} 1 & n=1 \\ 2 & n=2 \\ 4 & n=3 \\ f(n-1)+f(n-2)+f(n-3) & n>3 \end{cases}$$

6.24

算法思想：先将这 n 个人分成两组(当 n 为偶数时，两组人数相等，当 n 为奇数时，两组人数相差为 1)，接下来按以下的递归过程调换两边的人直到两边的重量差为最小：

递归过程：1 先计算出两边的重量差 m；

2 查询两边中重量差最接近 m/2 的两个人，并把这两个人对调；

3 当 m=0 或原 m 的绝对值小于新 m 的绝对值，则说明两组的重量差达到最小。可推出递归。

实验题

6.25 分别实现矩阵链乘法的递归算法、动态规划算法及其备忘录算法，并用实验分析方法分析比较三种算法的效率。

6.26 完成 XOJ 如下题目：1010，1022，1023，1028，1029，1030，1031，1032，1033，1038，1041，1056，1059，1086。

完成 POJ 如下题目：1018，1050，1080，1088，1159，1160，1179，1276，1678，1742，2593，2614，2411，2778，320

参考答案

7.1

```
DPtaskSelect()
1   sort by finish time
2   for i←0 to n+1 do
3    for j←0 to i-1 do
4            c[i,j]  ←0
5   for i←0 to n+1 do
6        for j←i+1 to n+1 do
7            max1←0
8            for k←i+1 to j-1 do
9                if task[k].s≥task[i].f and task[k].f≤act[j].s then
10                   if c[i,k]+c[k,j]+1>max1 then
11                       max1←c[i,k]+c[k,j]+1
12                       c[i,j]  ←max1
13  return c[0,n+1]
```

7.2

最优子结构性质证明同书上。

贪心选择性质证明：

子问题定义 S_{ij} 同书上。设 $S_{ij} \neq \varnothing$ 且 a_m 为 S_{ij} 中开始时间最大的任务，即

$$s_m = \min\{s_k : a_k \in S_{ij}\}$$

(1) a_m 为 S_{ij} 中开始时间最大的任务，则 a_m 一定包含在子问题中最优解中

证明：假设 A_{ij} 为子问题 S_{ij} 的最优解，且将 S_{ij} 中任务按开始时间降序排列，且设 a_k 为 S_{ij} 中第一个任务，即 $A_{ij} = \{a_k, \cdots\}$

如果 $a_k = a_m$ 则 a_m 在 S_{ij} 的最优解中

如果 $a_k \neq a_m$ 则

将 a_k 用 am 替换掉得解 $A'_{ij} = \{A_{ij} - \{a_k\}\} \cup \{a_m\}$。因为 a_m 为开始时间最大的任务，所以 $s_m \geq s_k$，所以 A'_{ij} 中各任务相互兼容而且 $|A'_{ij}| = |A_{ij}|$。这样，我们得到了一个包含任务 a_m 的最优解 A'_{ij}，故所证成立。

(2) 子问题 S_{im} 是空集，即选择 a_m 后，只剩下唯一一个可能具有非空解的子问题 S_{mj}。

证明：假设 S_{im} 不是空集，即存在一个任务 $a_k \in S_{im}$，满足

$f_i > s_i \geq f_k > s_k \geq f_m > s_m$

即 $s_k > s_m$ 这与 a_m 是开始时间最大的任务相矛盾，故所证成立。

算法可以描述如下：

(1)原问题 $S_{0(n+1)}$ 按开始时间降序排列

(2)贪心选择 S_{ij} 中开始时间最大的任务 a_m

(3)将 a_m 加入到最优解中

(4)用同样的方法解决子集 S_{mj}

7.3

将 n 个活动看作是直线上的 n 个半闭活动区间 $[s_i, f_i)$ ，实际上就是求这 n 个半闭区间的最大重叠数，因为重叠的活动区间所对应的活动是互不相容的。若这 n 个活动区间的最大重叠数为 m，则这 m 个重叠区间所对应的活动互不相容，因此至少要安排 m 个教室来容纳这 m 个活动。

算法可以描述如下：

(1)先将活动按开始时间 s_i 升序排列 $S = (a_1, a_2, \cdots, a_n)$

(2)维持两个教室的队列：QFree QBusy，初始时两个队列都为空。线性扫描活动序列 S，调度一个新活动时，从 QFree 中选择一个教室，若 QFree 为空则打开一个新的教室。将正在使用的教室从 QFree 中取出放入 QBusy。若一个活动结束了，则将其占用的教室从 QBusy 中取出放入 QFree 中。调度完所有活动之后，QFree 中的教室个数就为所求。

正确性证明：

假设由算法求得：调度 n 个活动需要 m 个教室，则我们可以知道 n 个活动集合 S 中一定含有这样一个活动子集

$$S_{ij} = \{a_k \in S : s_i \leq s_k < f_i, f_k > f_j\} \text{且} | S_{ij} | \geq m$$

即 S_{ij} 表示在活动 a_i 开始到结束这段时间内，a_i 活动之后还有 m-1 个活动开始了但还未结束，且这些活动按开始时间升序排列。

因为若不含这样的活动子集，则 S 的所有活动子集的长度都小于 m，即 S 中某时段同时处于活动状态的活动个数至多为 m-1 个，那么算法得出的至多教室数也为 m-1 而不是 m。

由上可知 S 中一定含有长度至少为 m 且活动同时发生活动子集，即任何求解的算法得到的教室数都等于或大于 m

因此算法得出的解 m 为最优的

7.4

同 0/1 背包问题的证明类似，略。

7.5

算法思想：按价值来贪心。

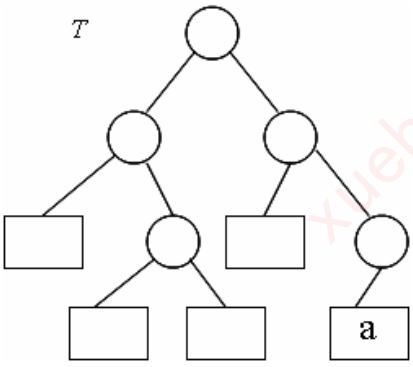
最优子结构性质类似书上证明。

贪心选择性质：

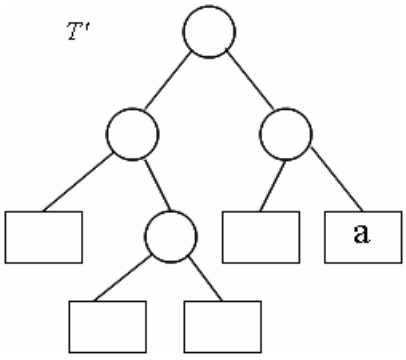
假设物品按价值大小降序排列，即 $v_1 \geq v_2 \geq \cdots \geq v_n$，且 $w_1 \leq w_2 \leq \cdots \leq w_n$，令 $V[i,w]$ 表示将物品 1 到物品 i 的物品装入载重量为 w 的背包中所能获得的最大价值，即子问题 (i,w) 的最大价值。按贪心选择策略，下一步要在物品 $i+1,\cdots,n$ 中选择满足条件 $w+w_k \leq W$ 且价值最大的物品 k。现在要证明这个物品 k 一定在某个最优解中。

假设物品 k 不在最优解中，则最优解 X ： $x_1\ x_2 \cdots x_k \cdots x_j \cdots x_n$ 中，有 $x_k = 0$，且有某个 $x_j = 1 (j > k)$，其中 $v_k \geq v_j$ 且 $w_k \leq w_j$。令 $x_k = 1$， $x_j = 0$，其它不变，我们可以得到另一个解 X' ： $x_1\ x_2 \cdots 1 \cdots 0 \cdots x_n$，这相当于从背包里取出物品 j 放入物品 k，由于 $v_k \geq v_j$ 且 $w_k \leq w_j$，显然新解 X' 的价值更大且没有违反约束条件，这与假设 X 是最优解矛盾。

7.6



(1)



(2)

反证法证。假设不满的二叉树 T 对应一种最优前缀编码，如图(1)所示。将图(1)转化为图(2)的二叉树 T'，除了字符 a 外，其他字符的深度均相同，此时有

$$\begin{aligned} B(T') &= \sum_{c \in C} f(c) d_{T'}(c) \\ &= \sum_{c \in C} f(c) d_T(c) - f(a) d_T(a) + f(a) d_{T'}(a) \\ &= B(T) - f(a) d_T(a) + f(a)[d_T(a) - 1] \\ &= B(T) - f(a) < B(T) \end{aligned}$$

所以 T' 是比 T 代价还小的解，这与 T 为最优解相矛盾。

7.7

反证法证。

假设我们将字符表按出现频度的单调递减顺序排序：$(c_1,c_2,...,c_n)$，其中$f(c_1)\geq f(c_2)...\geq f(c_n)$，对应于此字母表不存在编码长度单调递增的最优编码。即对所有的最优编码树T，至少存在这样一对字符c_i,c_j，有：

$$f(c_i)\geq f(c_j)，\ d_T(c_i)\geq d_T(c_j)$$

对T进行改造，将叶子c_i与c_j互换，可得树T'

$$\begin{aligned}B(T)-B(T')&=f(c_i)d_T(c_i)+f(c_j)d_T(c_j)-f(c_i)d_{T'}(c_i)-f(c_j)d_{T'}(c_j)\\&=f(c_i)d_T(c_i)+f(c_j)d_T(c_j)-f(c_i)d_T(c_j)-f(c_j)d_T(c_i)\\&=[f(c_i)-f(c_j)][d_T(c_i)-d_T(c_j)]\geq 0\end{aligned}$$

即$B(T)\geq B(T')$，这与T为最优解相矛盾。

7.8

7.9

贪心策略：
从大的面额开始除。

7.10

设客户i的等待时间为w_i，则$w_i=t_1+...+t_i$，则平均等待时间为$\sum w_i/n$，因此贪心策略为将服务时间由小到大排序，进行处理。

7.11

假设沿路有m个加油站：$1,2,\cdots,m$且两加油站i，j之间的距离为d_{ij}，如果在加油站i加油，则令$x_i=1$，否则$x_i=0$，则原问题可建模为

$$\min\sum x_i$$

且满足任意两加油站的之间的距离$d_{ij}\leq n$。

使用如下贪心策略：在油量有剩余的情况下尽可能多地行使路程，在经过某个加油站时，若剩余油量不足以行使到下一站，则必须停下来加油。
正确性证明：

假设算法得出需要停靠的站点为p个，分别为$O_1,O_2,\cdots,O_p$，则由贪心策略可知，

$$d_{O_iO_{i+1}}\leq n\text{且}d_{O_iO_{i+2}}>n$$

若贪心算法不能得到最优解，则存在$O_1,O_2,\cdots,O_q$，其中$q<p$且满足问题的约束条件，不妨假设$q=p-1$，即要在p个加油站的两个站点i,k之间取消一个站点j（详细地，可用归纳法证明），即

$O_1,\cdots,O_i,O_j,O_k\cdots,O_p$

$O_1,\cdots,O_i,O_k\cdots,O_q$

由于当取消O_j后，$d_{O_iO_k}>n$，此时即使在站点i加满油也无法行驶到站点k，所以$O_1,\cdots,O_i,O_k\cdots,O_q$不是一个可行解，这与它是最优解矛盾。

7.12

对于给定正整数x,a，函数$y=a^x,y=x^a$均为增函数，即

若$a_i>b_j$，则$a_i^{b_i}>a_i^{b_j}$

若$a_i>a_j$，则$a_i^{b_j}>a_j^{b_j}$

由上式可得$a_i^{b_i}>a_i^{b_j}>a_j^{b_j}$

贪心策略：每次都选择最大的a_i,b_i，其组成的$a_i^{b_i}$最大。

正确性证明：

设由贪心算法得到的回报为

$C=a_1^{b_1}a_2^{b_2}\cdots a_n^{b_n}$，其中$a_1\geq a_2\geq\cdots\geq a_n$，$b_1\geq b_2\geq\cdots\geq b_n$

假设贪心算法得到的回报不是最优的，则必定存在某个解，其回报为

$C'=a_1'^{b_1'}a_2'^{b_2'}\cdots a_n'^{b_n'}$

比C更优。其中$a_1',a_2',\cdots,a_n'$为$a_1,a_2,\cdots,a_n$的一个重排列，同理$b_1',b_2',\cdots b_n'$为$b_1,b_2,\cdots,b_n$的一个重排列

由于$a_1,a_2,\cdots,a_n$为按降序进行排列，则$a_1',a_2',\cdots,a_n'$中必定至少存在这样一对数(a_i',a_j')：

$$i<j \quad \text{且} \quad a_i'\leq a_j'$$

否则两个数列就相等了。

不妨设只存在一对这样的数(a_i',a_j')且先不考虑b_i'，即$b_i=b_i'$，在排列

$a'_1, a'_2, \cdots, a'_n$ 中，将 a'_i 和 a'_j 交换，可得排列 $a_1, a_2, \cdots, a_n$。

由 $C' = a_1'^{b'_1} a_2'^{b'_2} \cdots a_n'^{b'_n} = a_1'^{b'_1} \cdots a_i'^{b'_i} \cdots a_j'^{b'_j} \cdots a_n'^{b'_n}$

$$= a_1'^{b'_1} \cdots a_i'^{b'_i} \cdots a_j'^{b'_j} \cdots a_n'^{b'_n} (a_j'^{b'_i} a_i'^{b'_j}) / (a_j'^{b'_i} a_i'^{b'_j})$$

$$= a_1'^{b'_1} \cdots a_j'^{b'_i} \cdots a_i'^{b'_j} \cdots a_n'^{b'_n} (a_i'^{b'_i} a_j'^{b'_j}) / (a_j'^{b'_i} a_i'^{b'_j})$$

$$= C\ a_i'^{b'_i - b'_j} / a_j'^{b'_i - b'_j}$$

$$= C\ a_j^{b'_i - b'_j} / a_i^{b'_i - b'_j}$$

因为 $b_i \geq b_j$ 所以 $b_i - b_j \geq 0$，又因为 $a_i \geq a_j$，所以 $a_j^{b'_i - b'_j} / a_i^{b'_i - b'_j} \leq 1$，

$C' \leq C$，这与假设矛盾。故所证成立。

7.13

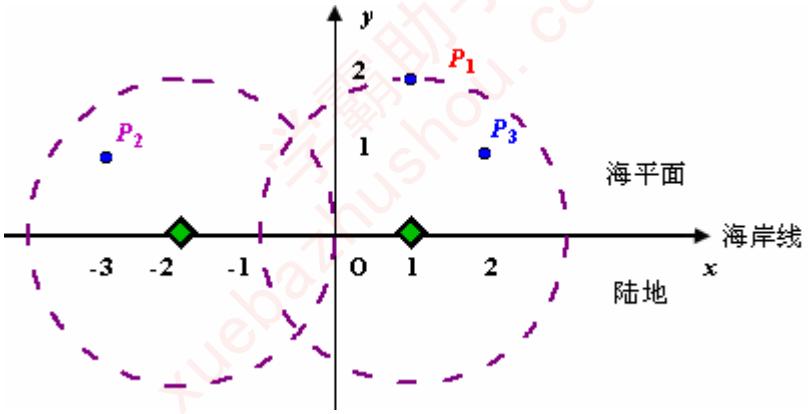


图 7.11

7.14

实验题

7.15 将背包问题分别用三种贪心算法实现，用实验分析方法分析哪个贪心算法更有效。

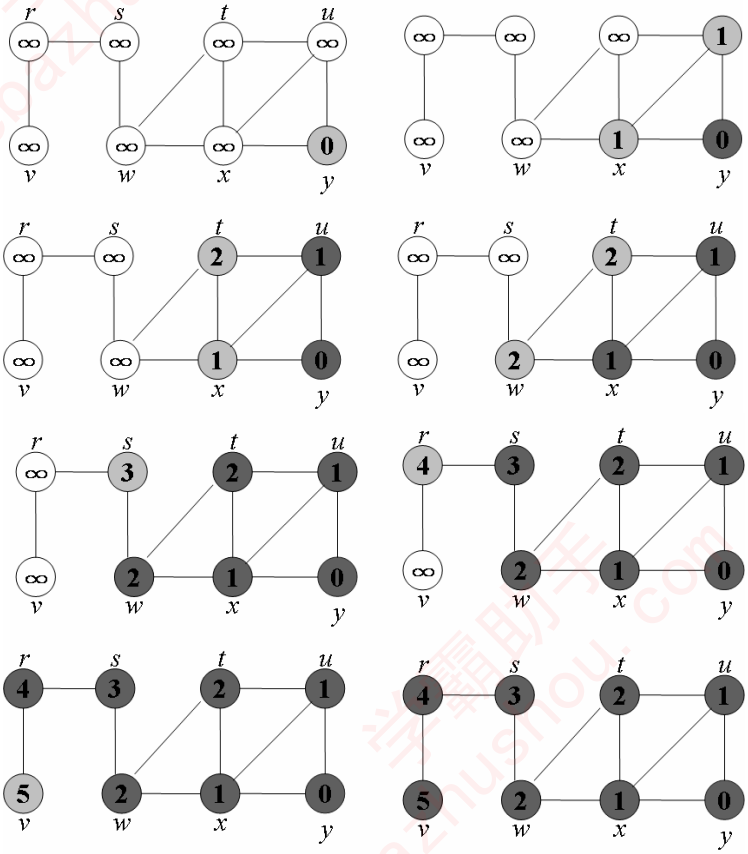
7.16 完成 XOJ 如下题目：1061，1062。

7.17 完成 POJ 如下题目：1017，1018，1042，1065，1083, 1089，1230，1328，1659，1716，1744，2751，3069，3687。

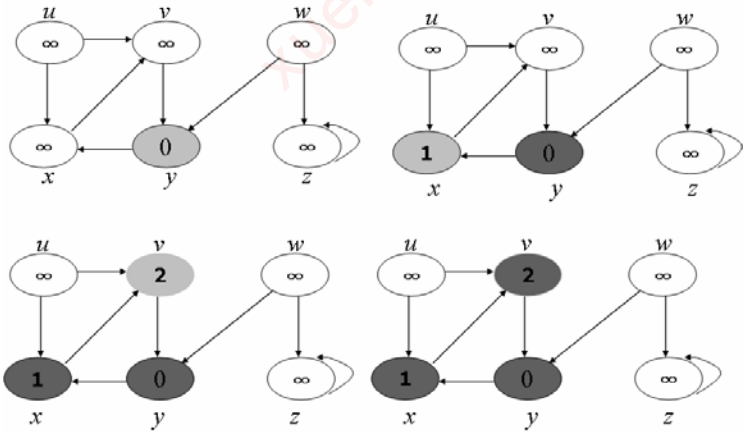
参考答案

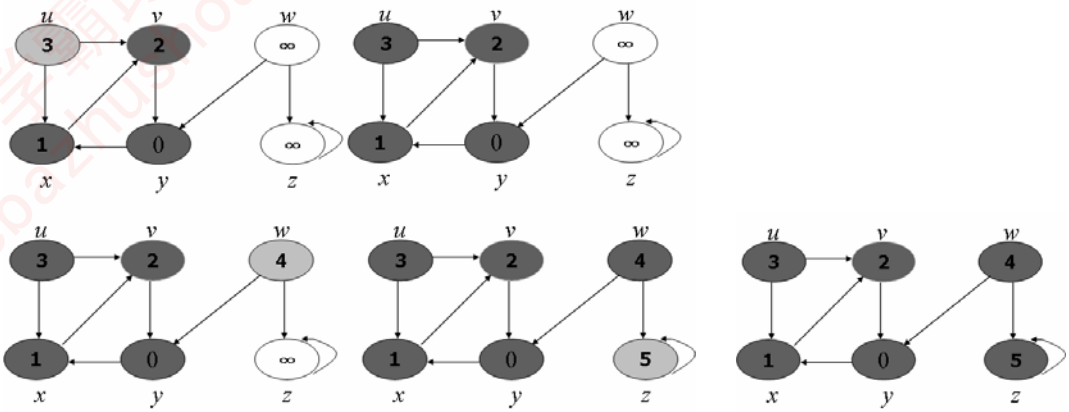
8.1 完成以下练习：

1) 对图8.3(a)从顶点 y 开始，类似图8.3产生一棵宽度优先生成树。



2) 对图8.4(a)从顶点 y 开始，类似图8.4产生一棵深度优先生成树。





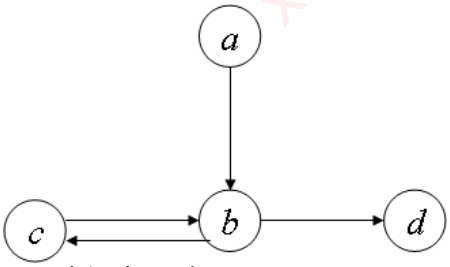
8.2 重写 DFS 算法，用数据结构栈来消除递归。

```
DFSstack(G)
1   for each vertex do
2       colour[u]←White
3   InitStack(s)
4   for each vertex do
5       if colour[u] =White then
6           clcour[u]←Gray
7           push(s,u)
8           while !empty(s) do
9                   pop(s,v)
10                  colour[v]←DarkGray
11                  for each u∈V do
12                      if colour[k]=White then
13                          colour[k]←Gray
14                          push(s,k)
```

8.3

如果用邻接表表示，则只需扫描每个顶点的邻接表，因此所需要的时间为 $O(|V|+|E|)$，如果用邻接矩阵表示，则实现非常简单，但是需要的时间为 $O(|V|^2)$。

8.4



以 a 为源点。则 $d(a)=1, d(b)=2, d(c)=3, d(d)=4, d(c)>d(d)$,且从 c 到 d 存在路径，但是 d 不是 c 的后继。

8.5

深度优先搜索无向图，只产生树边和后向边。注意到无向图不包含回路当且仅当深度优先搜索不产生后向边。利用深度优先搜索，若发现后向边，则说明有回路。

8.6

We have at our disposal an $O(V + E)$-time algorithm that computes strongly connected

components. Let us assume that the output of this algorithm is a mapping $scc[u]$, giving the number of the strongly connected component containing vertex u, for each vertex u. Without loss of generality, assume that $scc[u]$ is an integer in the set $\{1, 2, \ldots, |V|\}$.

Construct the multiset (a set that can contain the same object more than once) $T = \{scc[u] : u \in V\}$, and sort it by using counting sort. Since the values we are sorting are integers in the range 1 to $|V|$, the time to sort is $O(V)$. Go through the sorted multiset T and every time we Þnd an element x that is distinct from the one before it, add x to V_{SCC}. (Consider the Þrst element of the sorted set as .distinct from the one before it..) It takes $O(V)$ time to construct V_{SCC}.

Construct the set of ordered pairs $S = \{(x, y)$: there is an edge $(u, v) \in E$, $x = scc[u]$, and $y = scc[v]\}$.

We can easily construct this set in $\_(E)$ time by going through all edges in E and looking up $scc[u]$ and $scc[v]$ for each edge $(u, v) \in E$.

Having constructed S, remove all elements of the form (x, x). Alternatively, when we construct S, do not put an element in S when we Þnd an edge (u, v) for which $scc[u] = scc[v]$. S now has at most $|E|$ elements.

Now sort the elements of S using radix sort. Sort on one component at a time. The order does not matter. In other words, we are performing two passes of counting sort. The time to do so is $O(V + E)$, since the values we are sorting on are integers in the range 1 to $|V|$.

Finally, go through the sorted set S, and every time we Þnd an element (x, y) that is distinct from the element before it (again considering the Þrst element of the sorted set as distinct from the one before it), add (x, y) to E_{SCC}. Sorting and then adding (x, y) only if it is distinct from the element before it ensures that we add (x, y) at most once. It takes $O(E)$ time to go through S in this way, once S has been sorted.

The total time is $O(V + E)$.

8.7

令T是一个$|V| \times |V|$的矩阵，用来表示传递闭包，使得G中存在一条从i到j的路径时，$T[i, j] = 1$，否则$T[i, j] = 0$，初始化T如下

$$T[i,j] = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

当边(u, v)增加到G，T能够更新如下

TRANSITIVE-CLOSURE-UPDATE(u, v)

```
1  for i ← 1 to |V| do
2      for j ← 1 to |V| do
3          if T[i, u] = 1 and T[v, j] = 1 then
4              T[i, j] ← 1
```

这表示增加边的效果是产生一条每个已经到达u的顶点到每个可能已经从v可达顶点的一条路径。值得注意的是算法设置$T[u, v] \leftarrow 1$是因为初始值$T[u, u] = T[v, v] = 1$。

由于只有两个for循环，因此时间复杂度为$O(|V|^2)$。

8.8

Prim(G, v) //以为 v 起点

```
1  for i←0 to |V| do
```

```
2       closest[i]←v //点 i 到已完成顶点集 U 的最小值顶点
3       lowcost[i]←w(v,i)//U 中顶点到 V-U 中顶点的最小值
4   for i←0 to |V|-1 do
5       min←∞
6       for j←0 to |V| do
7           if lowcost[j] !=0 and lowcost[j]<min then
8               min←lowcost[j]
9               k←j         // (closest[k],k)是最小值的边
10      lowcost[k]←0
11      for j←0 to |V| do
12          if w(k, j) !=0 and w(k,j)<lowcost[j] then
13              lowcost[j]←w(k,j)
14              closest[j]←k
```

8.9

算法思想：把修改的边加入到最小生成树里，这样在树中就会产生了一条含有修改边的回路。在这个回路中去掉权值最大的，就可以得到一颗新的最小生成树。

8.10 证明 PrimMST 算法的正确性。

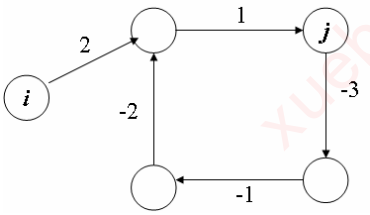
类似书上证明。略。

8.11 是否存在这样的图，对这些图，PrimMST 算法要慢于 KruskalMST 算法？

Prim 算法的时间复杂度为 O(|*E*|lg|*V*|), Kruskal 算法的时间复杂度为 O(|*E*|lg|*E*|),要使 Prim 算法慢于 Kruskal 算法，即要求|*E*|lg|*V*|>|*E*|lg|*E*|，可得 *V*>*E*。又因为这两个算法是用来求图的最小生成树，所以 *E*≥*V*-1。综上所述，当 *E*=*V*-1 时，Prim 算法要慢于 Kruskal 算法。故存在这样的图。

8.12

存在负权回路



8.13 修改 BellmanFood 算法，使得对任意顶点 v，当从源点到 v 的某些路径上存在一个负权回路时，则设置 $d[v] = -\infty$。

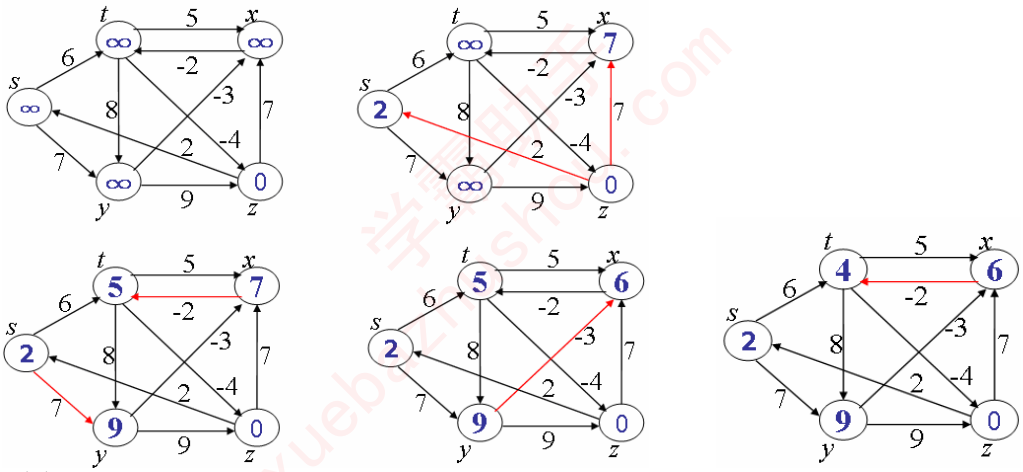
```
BellmanFord(G, w, s)
1   InitializeSingleSource(G, s)
2   for i ← 1 to |V| - 1 do
3       for each edge (u, v)∈ E do
4               Relax(u, v, w)
5   for each edge (u, v) ∈ E do
6       if d[v] > d[u] + w(u, v) then
7               d[v]←-∞
8               return False
9   return True
```

8.14

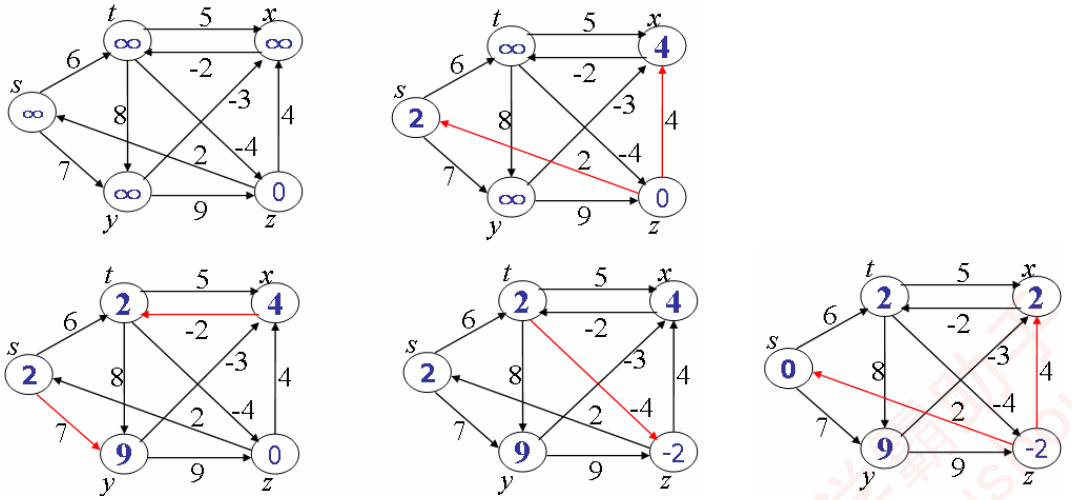
```
Pathnum(G, w)
1  Count←0
2  for each vertex v∈V do
3      BellmanFord(G, w, v)

BellmanFord(G, w, s)
1  InitializeSingleSource(G, s)
2  for i←1 to |V| - 1 do
3      for each edge (u, v)∈E do
4              if d[u]+w(u,v)≠∞ and π[v]≠u then
5                      count←count+1
6                      d[v]←d[u]+w(u,v)
7                      π[v]←u
```

8.15 令顶点 z 为源点，对图 8.10(a)所示的有向图运行 BellmanFord 算法。在每趟运行中，按图中的顺序对边进行松弛，显示每一趟运行后的 d 值和 π 值。现在把边 (z,x) 的权值改为 4 并把 z 作为源点再运行该算法的结果。

(1)



(2)

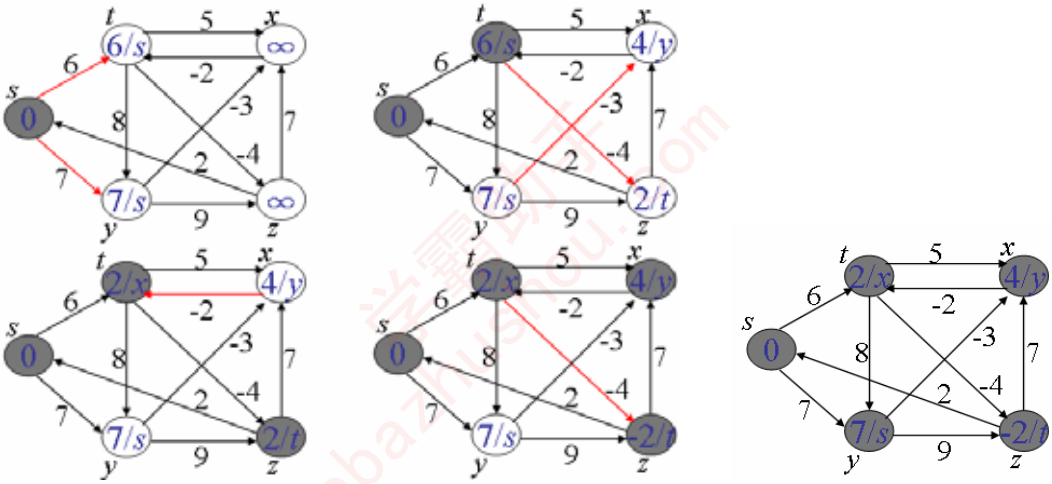


8.16

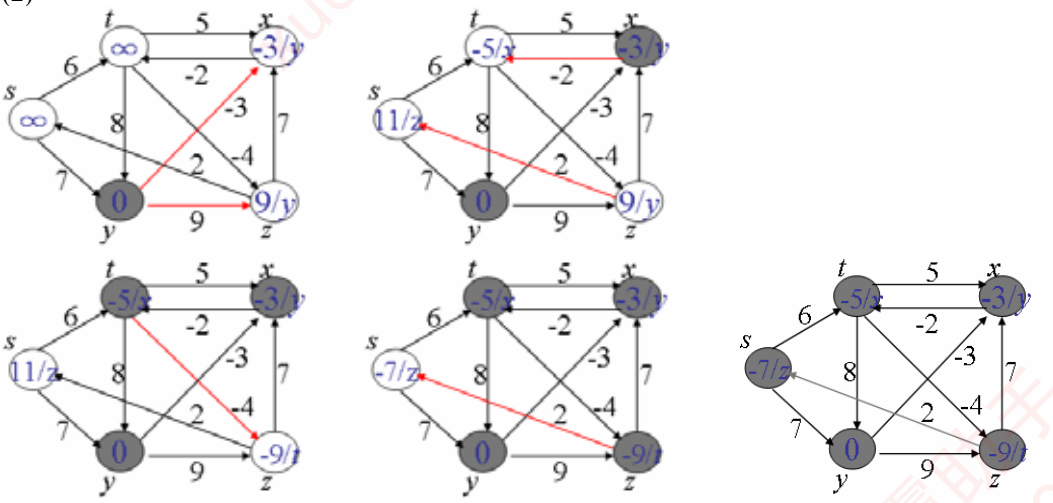
```
BellmanFordM1(G,w, s)
1    InitializeSingleSource(G, s)
2    changes ← True
3    while changes = True do
4            changes ← False
5            for each edge (u, v)∈E do
6                    RelaxM(u, v,w)
RelaxM(u, v,w)
1    if d[v] > d[u] + w(u, v) then
2            d[v]←d[u] + w(u, v)
3            π[v]←u
4            changes←True
```

8.17 令顶点 s 和顶点 y 分别为源点，对图 8.10(a)所示的有向图运行 Dijkstra 算法，类似图 8.11 所示的方式，给出 **while** 循环的每次迭代后的 d 和 π 值以及集合 S 中的顶点。

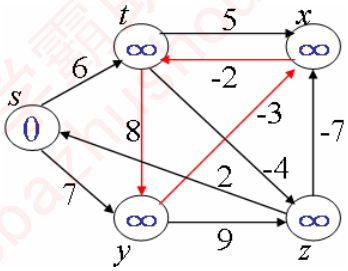
(1) 以 s 为源点(黑色点表示该点属于集合 S，圆圈中列出此时各个点的 d 值和 π值)



(2)



8.18 给出一个带负权边的有向图的简单实例，说明 Dijkstra 算法计算该例子会产生错误的结果。如果允许图中边的权为负，说明定理 8.8 的证明不能成立的原因。



上图中存在一条负权回路，所以从 S 到 Z 不存在最短路径。但是利用 Dijkstra 算法时却能找到一条错误的路径。

定理 8.8 证明不成立的原因是：当图中的边存在负权时，则有可能存在负权回路。

则图中含有该负权回路的所有路径，都没有最短路径。不妨设该路径为从 s 到顶点 u，即 $\delta(s, u) = -\infty$。

但是由于从 s 到顶点 u 存在着路径，又 **while** 循环是有限次的，所以最后得到的 $d(s, u) \neq -\infty$,即 $d(s,u) \neq \delta(s, u)$ ，故定理不成立。

8.19 利用合计方法分析 Dijkstra 算法的时间复杂度。

第一行的初始化部分的运行时间为 O(|*V*|)，**while** 循环的循环体需要执行|*V*|次，循环体中每次 ExtractMin 操作需要 O(lg|*V*|)时间，而对于第 7 至 8 行总运算需要 O(|*E*|)时间，所以算法的总运行时间为 O(|*V*|+|*V*|lg|*V*|+|*E*|)=O(|*V*|lg|*V*|)

8.20 假设将 Dijkstra 算法的第 4 行改为：

4 **while** $|Q|>1$

这一修改使得 **while** 循环执行 $|V|-1$ 次而不是 $|V|$ 次，算法是否仍然正确？

算法仍然正确。令 u 是队列 Q 中剩余的顶点。如果 u 不能从 s 可达，则有 $d[u]=\delta(s,u)=\infty$。如果 u 从 s 可达，则有一条路径 p，它从 s 经过 x 再直接到 u，由于 $d[x]=\delta(s,x)$，按照最短路径松弛性质，可得 $d[u]=\delta(s,u)$。故算法正确。

8.21

只需要修改 Dijkstra 算法，使它能够求解最大化路径上每条边可靠性的乘积问题。具体可以修改如下：

(1) 将队列运算 ExtractMin 改为 ExtractMax

(2) 在松弛运算步，+改为×

例如，对于 Relax 可以修改如下

RelaxReliability*(u, v, r)*

1 **if** $d[v] < d[u] \cdot r(u, v)$ **then**

2 $d[v] \leftarrow d[u] \cdot r(u, v)$

3 $\pi[v] \leftarrow u$

另一种解法是，对任意边 (u,v)，权值修改为 $w(u,v)=\lg r(u,v)$，然后运行 Dijkstra 算法，最可靠路径即为从 s 到 t 的最短路径，路径的可靠性为路径上边可靠性的乘积。

8.22

Observe that if a shortest-path estimate is not ∞, then it's at most $(|V| - 1)W$. Why? In order to have $d[v] < \infty$, we must have relaxed an edge (u, v) with $d[u] < \infty$. By induction, we can show that if we relax (u, v), then $d[v]$ is at most the number of edges on a path from s to v times the maximum edge weight. Since any acyclic path has at most $|V| - 1$ edges and the maximum edge weight is W, we see that $d[v] \le (|V|-1)W$. Note also that $d[v]$ must also be an integer, unless it is ∞.

We also observe that in Dijkstra's algorithm, the values returned by the ExtractMin calls are monotonically increasing over time. Why? After we do our initial $|V|$ Insert operations, we never do another. The only other way that a key value can change is by a DecreaseKey operation. Since edge weights are nonnegative, when we relax an edge (u, v), we have that $d[u] \le d[v]$. Since u is the minimum vertex that we just extracted, we know that any other vertex we extract later has a key value that is at least $d[u]$.When keys are known to be integers in the range 0 to k and the key values extracted are monotonically increasing over time, we can implement a min-priority queue so that any sequence of m Insert, ExtractMin, and DecreaseKey operations takes $O(m + k)$ time. Here.s how. We use an array, say $A[0 . . k]$, where $A[j]$ is a linked list of each element whose key is j . Think of $A[j]$ as a bucket for all elements with key j . We implement each bucket by a circular, doubly linked list with a sentinel, so that we can insert into or delete from each bucket in $O(1)$ time.

We perform the min-priority queue operations as follows:

• Insert: To insert an element with key j , just insert it into the linked list in $A[j]$. Time: $O(1)$ per Insert.

• ExtractMin: We maintain an index min of the value of the smallest key extracted. Initially, min is 0. To Þnd the smallest key, look in $A[min]$ and, if this list is nonempty, use any element in it, removing the element from the list and returning it to the caller. Otherwise, we rely on the monotonicity property (and that there is no IncreaseKey operation) and increment min until we either Þnd a list $A[min]$ that is nonempty (using any element in $A[min]$ as before)or we run off the end of the array A (in which case the min-priority queue is empty).Since there are at most m Insert operations, there are at most m elements in the min-priority queue. We increment min at most k times, and we remove and return some element at most m times. Thus, the total time over all ExtractMin operations is $O(m + k)$.

• DecreaseKey: To decrease the key of an element from j to i , Þrst check whether $i \le j$, ßagging an error if not. Otherwise, we remove the element from its list $A[j]$ in $O(1)$ time and insert it into the list $A[i]$ in $O(1)$ time.Time: $O(1)$ per DecreaseKey.

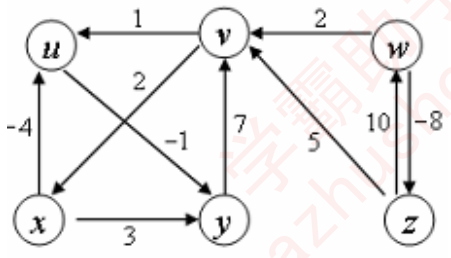
To apply this kind of min-priority queue to Dijkstra's algorithm, we need to let $k = (|V| - 1)W$, and we also need a separate list for keys with value ∞. The number of operations m is $O(|V|+|E|)$ (since there are $|V|$ Insert and $|V|$ ExtractMin operations and at most $|E|$ DecreaseKey operations), and so the total time is $O(|V| + |E| + |V||W|) = O(|V||W| + |E|)$.

8.23 根据递归方程（8.2）和（8.3），写出动态规划算法的伪代码，并分析其时间复杂度。

```
Shortestpaths(G,W)
1   for i=1 to |V| do
2       for j= 1 to |V| do
3           if i=j then  l_ij^0 ← 0
4           else   l_ij^0 ← ∞
5   for m=1 to |V|-1 do
6       for i=1 to |V| do
7           for    j=1 to |V| do
8               min ← ∞
9               for   each k∈adi[j] do
10                  if   l_ik^(m-1) + w_kj < min  then  min ← l_ik^(m-1) + w_kj
11              l_ij^m ← min
```

8.24 修改 FloydWarshall(W)算法，以便构造出最优解。

```
FloydWarshall(W)
1   D^(0) ← W
2   s[1..|V|,1..|V|]←0
```



8.16

```
3    for k ← 1 to |V| do
4        for i ← 1 to |V| do
5            for j ← 1 to |V| do
6                if d_ij^(k-1) < d_ik^(k-1) + d_kj^(k-1) then d_ij^(k) ← d_ij^(k-1)
7                else
8                    d_ij^(k) ← d_ik^(k-1) + d_kj^(k-1)
9                    s[i, j] ← k
10
11   return D^(|V|) and s
```

8.25 给定图 8.16，运行 FloydWarshall 算法，并写出外层循环中每次迭代所生成的矩阵 $D^{(k)}$。

k 为 0 时的矩阵:

0	32767	-1	32767	32767	32767
-4	0	3	32767	32767	32767
32767	32767	0	7	32767	32767
1	2	32767	0	32767	32767
32767	32767	32767	2	0	-8
32767	32767	32767	5	10	0

k 为 1 时的矩阵:

0	32767	-1	32767	32767	32767
-4	0	-5	32763	32763	32763
32767	32767	0	7	32767	32767
1	2	0	0	32767	32767
32767	32767	32766	2	0	-8
32767	32767	32766	5	10	0

k 为 2 时的矩阵

0	32767	-1	32767	32767	32767
-4	0	-5	32763	32763	32763
32763	32767	0	7	32767	32767
-2	2	-3	0	32765	32765
32763	32767	32762	2	0	-8
32763	32767	32762	5	10	0

k 为 3 时的矩阵

0	32766	-1	6	32766	32766
-4	0	-5	2	32762	32762
32763	32767	0	7	32767	32767
-2	2	-3	0	32764	32764
32763	32767	32762	2	0	-8
32763	32767	32762	5	10	0

k 为 4 时的矩阵

0	8	-1	6	32766	32766
-4	0	-5	2	32762	32762
5	9	0	7	32767	32767
-2	2	-3	0	32764	32764
0	4	-1	2	0	-8
3	7	2	5	10	0

k 为 5 时的矩阵

0	8	-1	6	32766	32758
-4	0	-5	2	32762	32754
5	9	0	7	32767	32759
-2	2	-3	0	32764	32756
0	4	-1	2	0	-8
3	7	2	5	10	0

k 为 6 时的矩阵

0	8	-1	6	32766	32758
-4	0	-5	2	32762	32754
5	9	0	7	32767	32759
-2	2	-3	0	32764	32756
-5	-1	-6	-3	0	-8
3	7	2	5	10	0

8.26 由于要计算 $d_{ij}^{(k)}$，$i,j,k=1,2,\cdots,|V|$，所以 FloydWarashall 算法的空间要求为 $O(|V|^3)$。

给出仅仅去掉所有上标所得的算法

```
FloydWarshall(W)
1   D ← W
2   for k ← 1 to |V| do
3       for i ← 1 to |V| do
4           for j ← 1 to |V| do
5               if d_ij < d_ik + d_kj then d_ij ← d_ij
6               else    d_ij ← d_ik + d_kj
7   return D
```

证明该算法是正确的，而且所需空间仅为 $O(|V|^2)$。

当把所有上标去掉时，则原递归方程变为：

$d_{ij} \leftarrow \min(d_{ij},\ d_{ik}+d_{kj})$，计算该算式包含的情况有以下三种：

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k)})$

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k)}+d_{kj}^{(k-1)})$

如果从节点 i 到 k 存在最短路径，则在这条路径的中间节点中一定不包含 i 或 k。假设有包含，说明存在着回路，当把回路去掉时就可以得到一条更短的路径，与已知的矛盾。所以 $d_{ik}^{(k)}=d_{ik}^{(k-1)}$，$d_{kj}^{(k)}=d_{kj}^{(k-1)}$，所以以上三种情况仍可全部转化为：$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$，即与原递归方程相等。所以可以去掉上标，得到的结果仍是正确的。在空间上由于只要保存 d_{ij} 即可。所以需要的空间为 $O(|V|^2)$

8.27

只要把 FloydWarshall 算法多迭代一次，检查 d 值的变化。如果有负权回路，则某些最短路径的权和会变小。如果没有负权回路，则 d 值不会改变。

8.28

(1)

$$\phi_{ij}^{k}=\begin{cases}0 & other\\ \phi_{ij}^{k-1} & d_{ij}^{k-1}\le d_{ik}^{k-1}+d_{kj}^{k-1}\\ k & d_{ij}^{k-1}>d_{ik}^{k-1}+d_{kj}^{k-1}\end{cases}$$

(2)修改 FloydWarshall 算法类似 8.25.见 8.25

(3) 与矩阵链乘相似之处是都指出了把原问题分成两部分的分割位置。

```
PrintPath(Q,i,j)
1   if Q[i, j]=0 then
2           if w[i, j]≠∞ then   print(i);   print(j);   return
3           else output "there is no path from i to j"; return
4   else
5       PrintPath(Q,i,Q[i, j])
6       PrintPath(Q,Q[i, j], j)
```

实验题

8.29 编程实现求解最短路径问题的类似矩阵乘法的动态规划算法和 FloydWarshall 算法，并用实验分析方法比较两种动态规划算法。

8.30 完成 XOJ 如下题目：1073，1074，1075，1076，1077，1078，1040，1043，1049，1050，1063。

8.31 完成 POJ 如下题目：1062，1094，1125，1158，1161，1178，1181，1364，1639，1665，1679，1860，2394，2421，2553，2728，3009，3026，3259，3660。

参考答案

习题

9.1 定义 9.4 定义的流和，满足流的三个性质吗？如果满足，请证明，如果不满足，哪一个性质最有可能被违背。

不满足。给定流网络 $G=(V, E)$，设 f_1 和 f_2 为 $V\times V$ 到 R 上的函数。定义如下：对所有 u，$v\in V$

$$(f_1 + f_2)(u,v) = f_1(u,v) + f_2(u,v)$$

容量约束性质可能被违背

反对称性质：

$$\begin{aligned}(f_1 + f_2)(u,v) &= f_1(u,v) + f_2(u,v)\\ &= -f_1(v,u) - f_2(v,u)\\ &= -(f_1(v,u) + f_2(v,u))\\ &= -(f_1 + f_2)(v,u)\end{aligned}$$

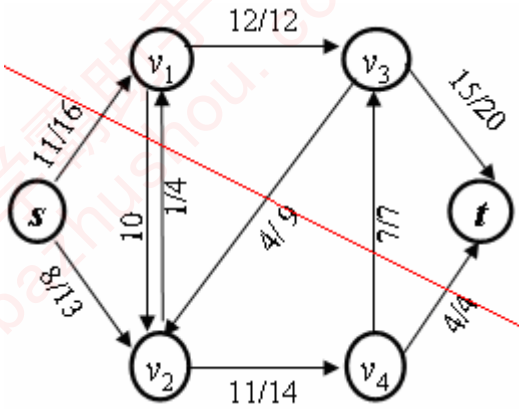
流守恒性质：

$$\begin{aligned}\sum_{v\in V}(f_1 + f_2)(u,v) &= \sum_{v\in V}(f_1(u,v) + f_2(u,v))\\ &= \sum_{v\in V} f_1(u,v) + \sum_{v\in V} f_2(u,v)\\ &= 0 + 0 = 0\end{aligned}$$

9.2 亚当教授有两个孩子，不幸地是两个孩子互不喜欢，他们不仅拒绝一同上学，而且甚至不愿意走过对方当天走过的街区。两个孩子对他们在拐角处交叉的路径并不会产生问题。幸运地是，教授的房子和学校都是在拐角处，但是他并不确定是否该把他的两个孩子送到同一所学校。教授有镇上的一份地图。试说明如何将决定两个孩子是否可以上同一所学校的问题建模为一个最大流问题。

每个拐角设为一个顶点，如果在顶点 u 和 v 之间有街道，则画边(u,v)和(v,u)，每条边的容量设为 1。教授家设为源点，学校设为汇点。如果存在一条大小为 2 的流，就可以确定两个孩子可以上同一所学校。

9.3 在图 9.2(a)中，通过割$(\{s, v_2, v_4\},\{v_1, v_3, t\})$的流是多少？该割的容量是多少？



流量为 19，容量：31

9.4 证明引理 9.2。

9.5 证明对任意一对顶点 u 和 v、任意的容量函数 c 和流 f，有

$$c_f(u,v)+c_f(v,u)=c(u,v)+c(v,u)$$

$$c_f(u,v)+c_f(v,u)=c(u,v)-f(u,v)+c(v,u)-f(v,u) \quad \text{（按照定义）}$$

$$=c(u,v)+c(v,u) \quad \text{（反对称性质）}$$

9.6 给定一个网络 $G=(V,E)$，证明 G 的最大流总可以被至多由 $|E|$ 条增广路径所组成的序列找到。（提示：找出最大流后再确定路径。）

9.7 若一个网络中所有的容量值都不同，则存在一个唯一的最小割，它把源点和汇点分割开。该结论正确吗？如果正确，请证明；否则，请说明理由。

9.8 说明如何有效地在一个给定的剩余网络中，找到一条增广路径。

9.9 设计一个有效的算法，它在一个给定的有向无回路图中寻找具有最大瓶颈容量的增广路径。

9.10 设计一个有效算法以找出一个给定的有向无回路图的层次图(level graph)。

9.11 针对多个源点和多个汇点的问题，扩展流的性质和定义。证明多个源点和多个汇点的流网络的任意流均对应于通过增加一个超级源点和超级汇点所得到的单源点单汇点的流，且流值相同。反之亦然。

9.12 在最大流的应用一节中，我们通过增加具有无限容量的边，把一个多源点多汇点的流网络转换为单源点单汇点的流网络。证明如果初始的多源点多汇点网络的边具有有限的容量，则转换后所得到的网络的任意流均为有限值。

9.13 假定多源点多汇点问题中，每个源点 s_i 产生 p_i 单位的流，即 $f(s_i,V)=p_i$。同时假定每个汇点 t_j 消耗 q_j 单位的流，即 $f(V,t_j)=q_j$，其中 $\sum p_i=\sum p_j$。说明如何把寻找一个流 f 以满足这些附加条件的问题转化为在一个单源点单汇点流网络中寻找最大流的问题。

9.14 有向图 $G=(V,E)$ 的一个路径覆盖是一个顶点不相交路径的集合 p，且 V 的每一个顶

点仅属于 p 的一条路径。路径可以开始和结束于任意顶点，且长度也为任意值，包括 0。G 的一个最短路径覆盖是指包含尽可能少的路径数的路径覆盖。

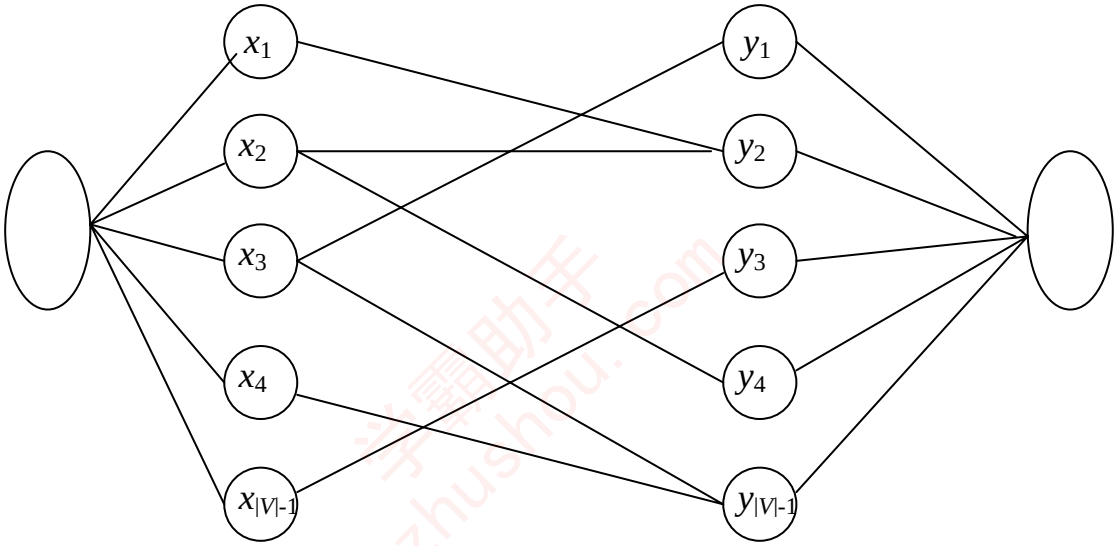
1） 设计一个有效算法以找出有向无回路图 $G=(V,E)$ 的一个最短路径覆盖（提示：假设 $V=\{1,2,\cdots,|V|\}$，构造图 $G'=(V',E')$，其中，

$$V'=\{x_0,x_1,\cdots,x_{|V|}\}\cup\{y_0,y_1,y_2,\cdots,y_{|V|}\},$$

$$E'=\{(x_0,x_i):i\in V\}\cup\{(y_i,y_0):i\in V\}\cup\{(x_i,y_j):(i,j)\in E\}$$

然后运行最大流算法）。

2） 说明所给的算法是否适用于包含回路的有向图。



9.15 证明推论 9.3。

9.16 证明定理 9.10。

9.17 证明定理 9.13 结论的第二部分，G 含有一条增广路径，则 G' 含有一条增广路径。

9.18 史密斯教授评审了 n 篇论文，他将每篇论文的评审结果，分别用 n 个信封将信（评审结果）装好，准备第二天邮寄出去。不幸的是，史密斯调皮的儿子小史密斯当晚就把这 n 封信都拿出了信封，玩过之后，他不知道如何将拿出的信正确地装回信封中了。着急的小史密斯只有求助于你，假设他所提供的 n 封信依次编号为 $1,2,\cdots,n$ 且 n 个信封也依次编号为 $1,2,\cdots,n$。小史密斯唯一能提供的信息是：第 i 封信肯定不是装在信封 j 中。请设计一个有效的算法，帮助小史密斯尽可能多地将信正确地装回信封中。

9.19 在东方，男女相爱讲究的是缘分。只要月下老人做媒，就能千里姻缘一线牵，无论他们身处何地。而在西方，只能借助爱神丘比特之箭，射中的两个男女才能相爱。假定丘比特之箭的射程为 d，给定 n 对男女，已知每个男女的位置 (x,y)，以及男女之间的缘分值，请设计一个算法，使得每对男女射中一次，且每一对被射中的男女之间的缘分值的和最大。注意箭的轨迹只能是一条直线。

9.20 给定一个无向图 $G=(V,E)$，M 是图 G 的一个匹配，如图 9.26 所示。试写出算法

GeneralGraphMatching(G)求该图最大匹配的过程。

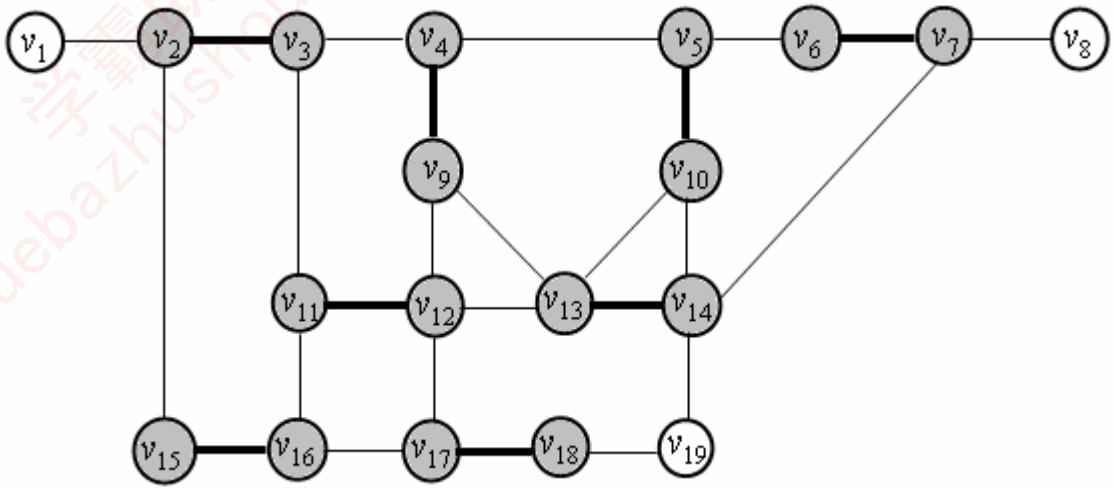


图 9.26

实验题

9.21 编程实现求解最大流问题的最短路径增广算法、Dinic 算法以及 MPM 算法，并用实验分析方法比较三种算法的效率。

9.22 完成 XOJ1089。

9.23 完成 POJ 如下题目：1087，1149，1273，1274，1325，1459，1637，1698，1719，2195，2239，2516，2771，3020，3041。

2.1

习题参考答案

习题

11.1

算法运行时间为$O(nW)$。这似乎是在多项式时间内解决了 0/1 背包问题，但是该算法并非严格意义上的多项式时间算法。它的运行时间依赖背包载重量的大小。一个多项式时间算法应该仅仅依赖于物品的数目，而不是他们本身重量或价值的大小。因此，DPKnapsack(I,W) 算法是一个伪多项式时间算法。

11.2

最优化问题 LongPathLenght 可以在多项式时间内解决，显然可以在多项式时间内判定 LongestPath。反过来，如果 LongestPath 能够在多项式时间内判定，假设答案是“是”则我们可以逐步减少 k 的值，然后再调用判定算法。否则，我们可以增加 k 的值，然后再调用判定算法，由于$0<k<|V|$，调用判定算法的次数是有限次，因此，可以在多项式时间里解决最优化问题 LongPathLenght。

11. 3

注意这里 TSP 是判定问题，可以类似书上图着色问题及上题类似求解，略。

11.4

简单，略。

11.5

考虑如下算法：

```
RunSlow(n)
1    s←a
2    for i←1 to n do
3        s←Concatenate(s, s)
```

调用O(n)个子程序，每个花线性时间。第一次调用Concatenate(s, s)，连接两个a，花时间O(2n)，第2次调用连接大小各为2的字符串，花时间O(2^2n)，总共n次调用，所用的时间为

O(2n)+ O(2^2n)+…+ O(2^nn)= $\sum_{k=1}^{n} 2^k n$，时间复杂度显然不是多项式时间复杂度。如果常数次调用，则仍然是多项式时间。

11. 6

只要找到该问题的一个多项式时间验证算法即可。主要理解同构的概念。

设$G_1=(V_1,E_1)$，$G_2=(V_2,E_2)$表示两个图，若存在双射函数$f:V_1\to V_2$，使得$(u,v)\in E_1$当且仅当$(f(u),f(v))\in E_2$，则两图同构。

设输入为：$G_1=(V_1,E_1)$和$G_2=(V_2,E_2)$，证书为函数f，只需验证是否有$(u,v)\in E_1$当且仅当$(f(u),f(v))\in E_2$。如果图用邻接矩阵存储，可在$O(|V|^2)$时间内验证。我们也可以构造一个非确定性算法：

GraphIsOmorphism(G1, G2)

1 **if** G1 does not have the same number of vertices as G2 **then return** false.
2 nondeterministically guess a permutation (bijection) f of m vertices.
3 **for** each pair of vertices (u, v) in G1 **do**
4 verify that (u, v) is an edge in G1 if and only if (f(u), f(v)) is an edge of G2
5 **if** all edges agree, **then return** true
6 **else return** false

11. 7

如果HamCycle∈ P，首先注意到对回路中的每个顶点精确地有两条边与之相连。可以如下找汉密尔顿回路:选择一个顶点$v \in V$，令E_v表示所有与v相连的边的集合。找到一对边$e_1, e_2 \in E_v$使得$G' = (V, E - E_v) \cup \{e_1, e_2\}$含有汉密尔顿回路，这个能够在多项式时间里通过尝试所有可能的一对边来实现。对图G'，类似上述过程求解。最终在多项式时间里得到一个图$H = (V, C)$，其中C为所求的一个汉密尔顿回路。

11.8

要证明 HamPath 为 NP 问题，只要找到一个多项式时间验证算法，具体如下：
输入为<G, u, v>,证书为一个顶点序列 v1,v2,…….vn,
1 检查是否有 v1=u,vn=v;
2 检查该序列是否包含了图中所有的顶点；
3 检查序列中任意两个相邻点在图中是否存在边；
上述三步显然可以在多项式时间内完成，得证。

11.9

由于停机问题是不可判定的，因此它不是 NP 问题，按照定义，显然它也不是一个 NPC 问题。对于 NPC 问题 SAT，我们可以构造一个多项式时间转换算法：任给一个输入命题公式，该算法对该公式枚举其变元的所有赋值，如果存在赋值使其为真，则停机，否则进入无限循环。这样，判断公式是否可满足便转化为判断以公式为输入的算法是否停机。因此，NPC 问题 SAT 可以多项式时间约简到停机问题，所以，停机问题是难问题。

11.10

如果有$L_1 \leq_P L_2$，且$L_2 \leq_P L_3$，则存在多项式可计算得函数f_1和f_2，使得对任意的$x \in \{0, 1\}^*$，$x \in L_1$当且仅当$f_1(x) \in L_2$，$x \in L_2$当且仅当$f_2(x) \in L_3$。函数$f_2(f_1(x))$是多项式可计算的且满足$x \in L_1$当且仅当$f_2(f_1(x)) \in L_3$，故所证成立。

11.11

两个差不多，这个更直接。

11.12

因为对于具有 m 个命题变元的公式ϕ，构造真值表需要2^m行，显然是指数级的，因此

不能导致多项式约简。

11.13

因为我们只需要检查任一个子句是否可满足，从而可决定整个子句是否可满足。

11.14

令这个判定算法为 A，其时间复杂度为 $O(n^k)$。任给一个命题公式 ϕ，我们调用 A(ϕ)，如果返回假，则停止，否则说明 ϕ 为真，我们可继续如下：令 $x_1=1$，我们得到一个命题公式 ϕ_1，如果 A(ϕ_1)返回假，则我们只需要令 $x_1=0$ 即可，否则 $x_1=1$。重复上面过程。如果有 n 个变元，上述过程只要重复 n 次，就可以找到可满足赋值。这相当于调用 n 次 A(ϕ)，故仍然是多项式时间复杂度 $O(nn^k)=O(n^{k+1})$。

11.15

算法思想：先根据给定的 2-CNF 构造有向图 G(V,E)，对每一个正文字及相应的负文字各设置一个顶点。假设公式中有 n 个独立的正文字，则|V|=2n。对于每个子句执行如下操作，假设子句为(x∨y)，则添加一条从¬x 到 y 的边及一条从¬y 到 x 的边。我们可得如下结论：

2-CNF 是不可满足的，当且仅当存在一个文字 x，在图 G 中存在从 x 到¬x 及从¬x 到 x 的路径。

证明： 设 ρ 是给定的 2-CNF 的一个真值指派。

一般地，对于某个正文字 x，设 ρ(x)=T ，若要使给定的 2-CNF 为真，则 x 指向的变元也必须为 T，依此类推。而 ¬x=F，指向 ¬x 的变元也必须为 F，依此类推。

所以若给定的 2-CNF 可满足，必不存在从 x 到 ¬x 的路径，同理，不存在从 ¬x 到 x 的路径。

实现：通过深度优先搜索求图 G 的强连通分支，然后在每个强连通分支里判断是否同时存在一个正文字及相应的负文字。

分析：

构造图：O(|*V*|+|*E*|)

求强连通分支：O(|*V*|+|*E*|)

判断：O(|*V*|)

11.16

If we did not require the vector x to have integer values, then this is the linear programming problem and is solvable in polynomial time. This one is more difficult. As usual it is easy to show that 0-1 INT is in NP. Just guess the values in x and multiply it out.A reduction from 3-SAT finishes the proof. In order to develop the mapping from clauses to a matrix we must change a problem in logic into an exercise in arithmetic. Examine the following chart. It is just a spreadsheet with values for the variables x_1, x_2, and x_3 and values for some expressions formed from them.

Expressions	Values							
X_1	0	0	0	0	1	1	1	1
X_2	0	0	1	1	0	0	1	1
X_3	0	1	0	1	0	1	0	1
$+ X_1 + X_2 + X_3$	0	1	1	2	1	2	2	3
$+ X_1 + X_2 - X_3$	0	-1	1	0	1	0	2	1
$+ X_1 - X_2 - X_3$	0	-1	-1	-2	1	0	0	-1
$- X_1 - X_2 - X_3$	0	-1	-1	-2	-1	-2	-2	-3

Above is a table of values for arithmetic expressions. Now we shall interpret the expressions in a logical framework. Let the plus signs mean *true* and the minus signs mean *false*. Place *or*'s between the variables. So, $+x_1 + x_2 - x_3$ now means that

x_1 is *true*, *or* x_2 is *true*, *or* x_3 is *false*.

If 1 denotes *true* and 0 means *false*, then we could read the expression as $x_1=1$ *or* $x_2=1$ *or* $x_3=0$.

Now note that in each row headed by an arithmetic expression there is a minimum value and it occurs exactly once. Find exactly which column contains this minimum value. The first expression row has a zero in the column where each x_i is also zero. Look at the expression. Recall that $+x_1 + x_2 + x_3$ means that at least one of the x_i should have the value 1. So, the minimum value occurs when the expression is *not satisfied*.

Look at the row headed by $+x_1 - x_2 - x_3$. This expression means that x_1 should be a 1 or one of the others should be 0. In the column containing the minimum value this is again not the case.

The points to remember now for each expression row are:

a) Each has *exactly* one column of minimum value.
b) This column corresponds to a nonsatisfying truth assignment.
c) Every other column satisfies the expression.
d) All other columnms have higher values.

Here is how we build a matrix from a set of clauses. First let the columns of the matrix correspond to the variables from the clauses. The rows of the matrix represent the clauses - one row for each one. For each clause, put a 1 under each variable which is not complemented and a -1 under those that are. Fill in the rest of the row with zeros. Or we could say:

$$a_{i,j} = \begin{cases} 1 \text{ if } v_j \in \text{ clause } i \\ -1 \text{ if } \overline{v_j} \in \text{ clause } i \\ 0 \text{ otherwise} \end{cases}$$

The vector b is merely made up of the appropriate minimum values plus one from the above chart. In other words:

b_i = 1 - (the number of complemented variables in clause i).

The above chart provides the needed ammunition for the proof that our construction is correct. The proper vector x is merely the truth assignment to the variables which satisfies all of the clauses. If there is such a truth assignment then each value in the vector Ax will indeed be greater than the minimum value in the appropriate chart column.

If a 0-1 valued vector x does exist such that $Ax \geq b$, then it from the chart we can easily see that it is a truth assignment for the variables which satisfies each and every clause. If not, then one of the values of the Ax vector will always be less than the corresponding value in b. This means that the that at least one clause is not satisfied for any truth assignment.

Here is a quick example. If we have the three clauses:

$$(x_1, \overline{x_3}, x_4), (\overline{x_2}, \overline{x_3}, x_4), (x_1, x_2, x_3)$$

then according to the above algorithm we build A and b as follows.

$$\begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & -1 & -1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix} \geq \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix}$$

Note that everything comes out fine if the proper values for the x_i are put in place. If x_3 is 0 then the first entry of Ax cannot come out less than 0 nor can the second ever be below -1. And if either x_2 or x_1 is 1 then the third entry will be at least 1.

11.17

将子集合问题约简到 Partition，即 SubsetSum$\leq_P$ Partition：

任给子集合问题的一个实例 $S=\{x_1,x_2,\cdots,x_n\}$ 以及一个正整数 t，我们可以构造一个划分实例：令 $s=\sum_{i=1}^{n}x_i$，令 $x_{n+1}=3s-t$，$x_{n+2}=3s-(s-t)=2s+t$。便可以得到划分实例：$S=\{x_1,x_2,\cdots,x_n,x_{n+1},x_{n+2}\}$。容易验证如果子集合的实例有一个"yes"解当且仅当划分实例有一个"yes"解。

11.18

将 Partition 问题约简到 BinPacking，即 Partition$\leq_P$ BinPacking：

任意给定 Partition 问题的一个实例，即一个物品的集合 $S=\{s_1,\cdots,s_i,\cdots,s_n\}$，其中 s_i 为正整数，我们可以如下构造一个 BinPacking 实例，物品集合一样，箱子的重量 $W=\frac{1}{2}\sum_{i=1}^{n}s_i$，箱子的个数 k=2。

容易验证划分的实例有一个“yes”答案当且仅当 BinPacking 的实例有一个“yes”答案。

11.19

有两种证明方法：一种是将 Partition$\leq_P$ ParallelScheduling problem with two machine，然后再约简到 ParallelScheduling。另一种办法是直接将 Partition$\leq_P$ ParallelScheduling。

11.20

用动态规划算法求解，其时间复杂度为 $O(nt)$，当 t 表示成一元形式，是一个常数，因此整个算法时间复杂度是多项式。

11.21

HamCycle$\leq_P$ 最长简单回路问题，事实上，前者是后者的一个特例。

习题参考答案

习题

12.1 将装载问题改进的递归算法改写为迭代回溯算法。

```
Backtrack()
1   bestw←0, i←1,flag1[1]←0,flag2[1]←0, cw←0
2   while i≥1 and i≤n+1 do
3               if   i=n+1    then
4                       if cw>bestw then bestw←cw ; i←i-1
5               else
6                       if c(i)≤W and flag1[i]=0 and flag2[i]=0 then
7                               cw←cw+w[i];flag1[i]←1
8                               flag1[i+1]←0;flag2[i+1]←0;i←i+1
9                       else if c(i)≤W and flag1[i]=1 and flag2[i]=0 then
10                              cw←cw-w[i];flag2[i]←1;flag1[i+1]←0
11                              flag2[i+1]←0;i←i+1
12                      else if c(i)≤W and flag1[i]=1 and flag2[i]=1 then
13                              i←i-1
14                      else if c(i)>W and flag1[i]=0 and flag2[i]=0 then
15                              flag1[i]←1;flag2[i]←1;flag1[i+1]←0
16                              flag2[i+1]←0;i←i+1
17                      else if c(i)>W and flag1[i]=1 and flag2[i]=1 then
18                              i←i-1
```

12.2

证明：先在这个棋盘上做一个与棋盘重合的坐标系，则棋盘上的皇后等同于坐标系上的点。在这个坐标系中的两个点在同一对角线当且仅当它们的连线的斜率为 1 或-1。先有两个皇后i, j，它们的坐标分别为$(i, x_i), (j, x_j)$，则这两点的斜率为$(x_i - x_j)/(i - j)$，要使这两皇后在同一对角线上，当且仅当它们连线的斜率为 1 或-1，即$(x_i - x_j)/(i - j)$ 等于 1 或-1，展开，所证成立。

12.3

```
 8queen(k)
1   t←false
2   for i←1 to 8 do
3           x[k] ←i
4           if place(k) then //  第 k 个皇后能放在 i 位置上
5                   if k=8 then
6                       t←true; output x
7                   else
8                           t1←8queen(k+1)
9                           if t1 then t←true
10  return t
```

12.4

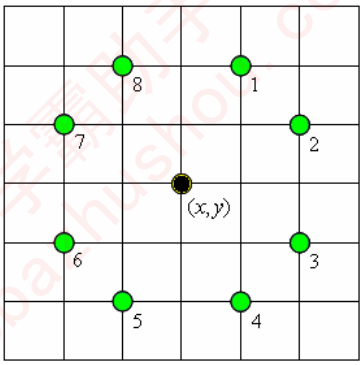
```
Nqueens(k)
1   x[1]←0
2   m←1
3   num←0
3   while m>0 do
4         flag[m]←0
5         while x[m]≤n-1 d0
6               x[m]← x[m] +1
7               if place(m)=true then
8                     flag[m]←1; num←num+1
9                     if m=n then
10                          if num≥k then return true
11                    else      m← m+1
12                          x[m] ← 0;   flag[m] ←0
13          if flag[m]=1 then   num←num-1
14          m ← m-1
15    return false
```

12.5

```
Backtrack(i)
1   if i>n then
2         for j←1 to n do
3               if x[j]=1 then output j
4   else
5         x[i]←1
6         Backtrack(i+1)
7         x[i]←0
8         Backtrack(i+1)
```

12.6

首先将起点作为当前位置，按照象棋马的移动规则，搜索有没有可以移动的相邻位置；如果有可以移动的相邻位置，则移动到其中的一个相邻位置，并将这个相邻位置作为新的当前位置，按同样的方法继续搜索通往终点的路径；如果搜索不成功，则换另外一个相邻位置，并以它作为新的当前位置继续搜索通往终点的路径。马走日字，当马一开始在黑点(x, y)时，它下一步可以到达的点有八个，分别是方向 1：$(x+1, y+2)$，方向 2：$(x+2, y+1)$，方向 3：$(x+2, y-1)$，…，如图所示。引进增量数组，dx[]= (1, 2, 2,…)，dy[] = (2, 1, -1,…)，现在只需要知道方向 k，下一步的位置就是$(x+dx[k], y+dy[k])$。



令 flag[0..8;0..8]表示棋盘，并初始化为 0，表示这些位置均未跳过。令(x0,y0)为起始位置，当前位置点为(x, y)。route 数组纪录访问路线。

```
JumpHorse(i)
1   for k←1 to 8 do
2     if x + dx[k]≥0 and x + dx[k] < 8 and y + dy[k]≥0 and y + dy[k] <8 then
3           route[i]←k
4           x ← x + dx[k];   y← y + dy[k]
5           if x=x0 and y=y0 then
6                 output(i); return
7           else
8                 if flag[x, y]=0 then
9                       flag[x, y]←1
10                      JumpHorse(i+1)
11          x←x - dx[k];   y←y - dy[k]
```

12.7

```
3SAT(i)
1   if i>m then output
2   else
3         for k=0 to 1 do
4               x[i]←k; t←false
5               for j=1 to n do
6                     if c[j]=1 then t←true return
7               if   t=false   then   3SAT(i+1)
```

还可以利用约束函数，若某个子句中的变元均已经取值，但是某个子句仍为假，则可以剪支。

12.8

```
Hamiton(i)
1   if i = n then
2         if   w[x[n-1],x[n]]≠∞  and w[x[n],1]≠∞  then
3                     output x[i]
4   else   for   j ← i   to   n-1   do
5         if   w[x[i-1],x[j]]≠∞  then
6                           x[i] ↔x[j]
```

```
7                       Hamiton(i +1)
8                           x[i] ↔x[j]
```

12.9

```
TSP()
1   while i>0 do
2       j←i+1
3       while   j≤n do
4           if w[x[i],x[j]]≠∞  and cw+ w[x[i], x[j]] < bestw then
5               x[i+1] ↔x[j];       cw ←cw+ w[x[i],x[i+1]]
6               i←i+1;   j←i+1
7               if i=n then
8                   if     w[x[n],1]≠∞  then
9                       if cw + w[x[n],1] < bestw then
10                                          bestw←cw+ w[x[n],1]
11                                          for k ← 1 to n do
12                                              bestx[k] ← x[k]
13                  else   j←j+1
14      cw←cw-w[x[i-1],x[i]];   w[x[i-1],x[i]] ← ∞
15      i←i-1
```

12.10

令 $x[i]$表示雇员 i 做做第 $x[i]$工作，显然解空间可以构造成一棵排列树。

```
BacktrackPerm(i)
1   if   i>n   then
2       sum←0
3       for j←1 to n do
4           sum←sum+c[j, x[j]]
5       if sum<bestc then
6           bestc←sum
7           for j←1 to n do
8               bestx[j] ←x[j]
9   else
10      for j←i to n do
11          x[i] ↔ x[ j]
12          BacktrackPerm(i+1)
13          x[i] ↔ x[ j]
```

12.11　用回溯法求解并行机调度问题（见第 11 章）。

令 $x[i]$表示将第 $i(1\le i\le n)$个任务分配给第 $x[i]$台机器。解空间为一棵 m 叉树(总共 m 台机器)。

```
BacktrackMachine(i)
1   if   i>n   then
2       temp←0
3       for j←1 to m do
4           if length[j]>temp then temp←length[i]
5       if temp<bestc then
6           bestc←temp
7           for j←1 to n do
8               bestx[j] ←x[j]
9   else
10      for j←1 to m do
11          length[j] ←length[j]+t[i]
12          x[i] ←j+1
13          if length[j]<bestc then
14              BacktrackMachine(i+1)
15          length[j] ←length[j]-t[i]
```

12.12

```
Backtrackjob(i)
1     if   i > n   then
2           if   t<bestcthen
3               bestc←t
4               for   j ← 1 to n do
5                   bestx[j] ← x[j]
6     else   for   j ← i   to   n   do
7           if t≥r[x[j]]   then
8               t←t+p[x[j]]+q[x[j]]
9               if t< bestc   then
10                  x[i] ↔ x[j]
11                  Backtrackjob(i +1)
12                  x[i] ↔x[j]
13              t←t-p[x[j]]-q[x[j]]
```

12.13

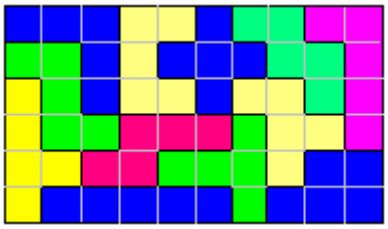


图 12.18

本题实现起来比较繁琐，可以不做。

12.14

```
BacktrackPerm(i)
```

```
1   if   i>n   then
2       sum←0
3       for j←1 to n do
4           sum←sum+P[j, x[j]] * Q[x[j], j]
5       if bestc<sum then
6           bestc←sum
7           for j←1 to n do
8               bestx[j] ←x[j]
9   else
10          for j←i to n do
11              x[i] ↔ x[ j]
12              if place(i) then
13                  BacktrackPerm(i+1)
14              x[i] ↔ x[ j]
```

其中 $x[i]$表示男队员 i 和女队员 $x[i]$配对。place(i)测试如果不同男队员和同一个女队员 x[i]配对，则违反约束条件，返回 false。

12.15

令锯后共 n 段，每段的长度为 $x[i]$，总长度为 S

```
Findmin(S,n,x[n])
1       for i←1 to S do
2           if S% i =0 then
3               if Ping(S,i,1)=true then return i
Ping(s,i,k)
1   if s=0 then return true
2   else
3       j←k; ss←0
4       for j←k to n do
5           if ss+x[j]≤i then
6               x[k] ↔x[j]
7               ss←ss+x[k]
8               if ss=i then t←Ping(s-i, i, k+1)
9                   if t then return true
10                  else    ss←ss-x[k]
11                          x[k] ↔x[j]
12  return   false
```

12.16　请为零件切割问题的回溯算法设计一个考虑浪费的剪支函数或者改进切割的过程。有不同的办法，请搜索相关文献。

实验题

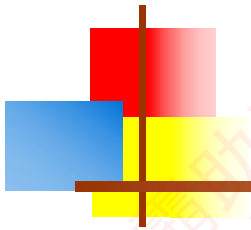
12.17　对旅行商问题，分别用动态规划法和回溯算法求解，用实验分析方法分析哪个算法更有效。

12.18　完成 XOJ 如下题目：1008，1027，1039，1063，1064。

12.19　完成 POJ 如下题目：1010，1011，1020，1062，1167，1190，1085，1753，2078，2488，2677。

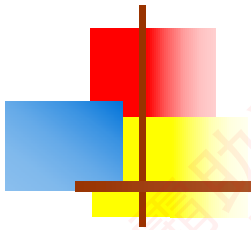
分支限界(Branch and bound)

- 基于广度优先搜索的一种穷举算法
- 尽可能的利用剪支技术



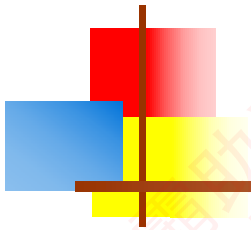
引言 (Introduction)

- 与回溯法一样，分支限界是搜索一个解空间，而这个解空间通常组织成一棵树。
- 常见的树结构为子集树和排列树。
- 回溯以深度优先搜索一棵树，而分支限界常常以广度优先或最小耗费优先的方法搜索这棵树。
- 解空间树.
- 分支限界法常用于解最优化问题.
  - 确定所求问题的上下界
  - 在每个节点使用限界函数来屏蔽节点或是扩展节点.
  - 然后,使用目前为止最好的解来帮助剪枝,直到所有顶点被遍历或是被剪掉.



Ideas

- 分支限界法也可以说是对回溯法的一个改进.
- 假如我们在考虑一个最小化问题时. 我们的想法是使所有的可能目标函数值必须维持在上界(目前为止最好的解的目标函数值)与下界之间. 如果在某一数量的决策之后（转移操作），我们到达一个节点，在这个节点上我们得到的下界大于或等于上界，那么就没有必要在扩展这个节点既不需在延伸这个分支。
- 对于最大化问题规则正好相反：一旦上界小于或等于先前确定的下界，那么就剪掉这个枝。



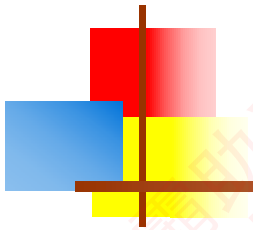
- 首先，分支限界是对最优化问题可行解进行剪枝的一个方法。
- 将搜索集中在有希望得到解的分支上。也就是说，在基于上下界和可以得到最优解的基础上,扩展分支,理由是发展这样的分支可以得到更好的上下界，从而可以剪去更多的分支
- 总之，不要单纯以DFS或BFS来进行搜索，而要结合起来进行搜索.

- 分之限界需要的步骤如下:

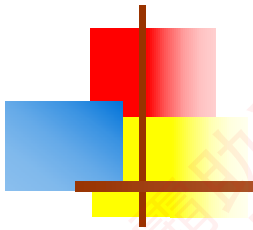
1.对所求的问题定义一个解空间。这个解空间至少包含这个问题的最优解。

2.对解空间进行组织，以便能更好的搜索。比较常见组织方式有图或是一棵树。

3.以广度优先搜索的方式搜索解空间，使用限界函数来避免那些不能得到解的子空间



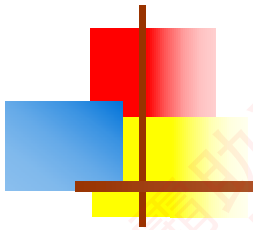
- 分支限界是有系统的搜索一个解空间的另一个方法。首先在扩展节点的扩展方法上，它不同于回溯 。
- 每个活节点变成扩展节点只有一次。当一个节点变成扩展节点时，我们展开从它可到达的所有节点。其中那些不能得到可行解的节点去掉（成为死节点）,把剩下来的节点加到活节点的表中，然后，从这个表中选一个节点作为下一个扩展节点。



- 从活节点的表中选一个节点并扩展它。这个扩展操作持续到找到解或这个表为空为止。
- 有两种常用的方法来选择下一个扩展节点：

  1) 先进先出 (FIFO)

  这个方法是按节点放进表中的次序从活节点表中选择节点。这个活节点表可被看作一个队列。使用广度优先来搜索这个棵树

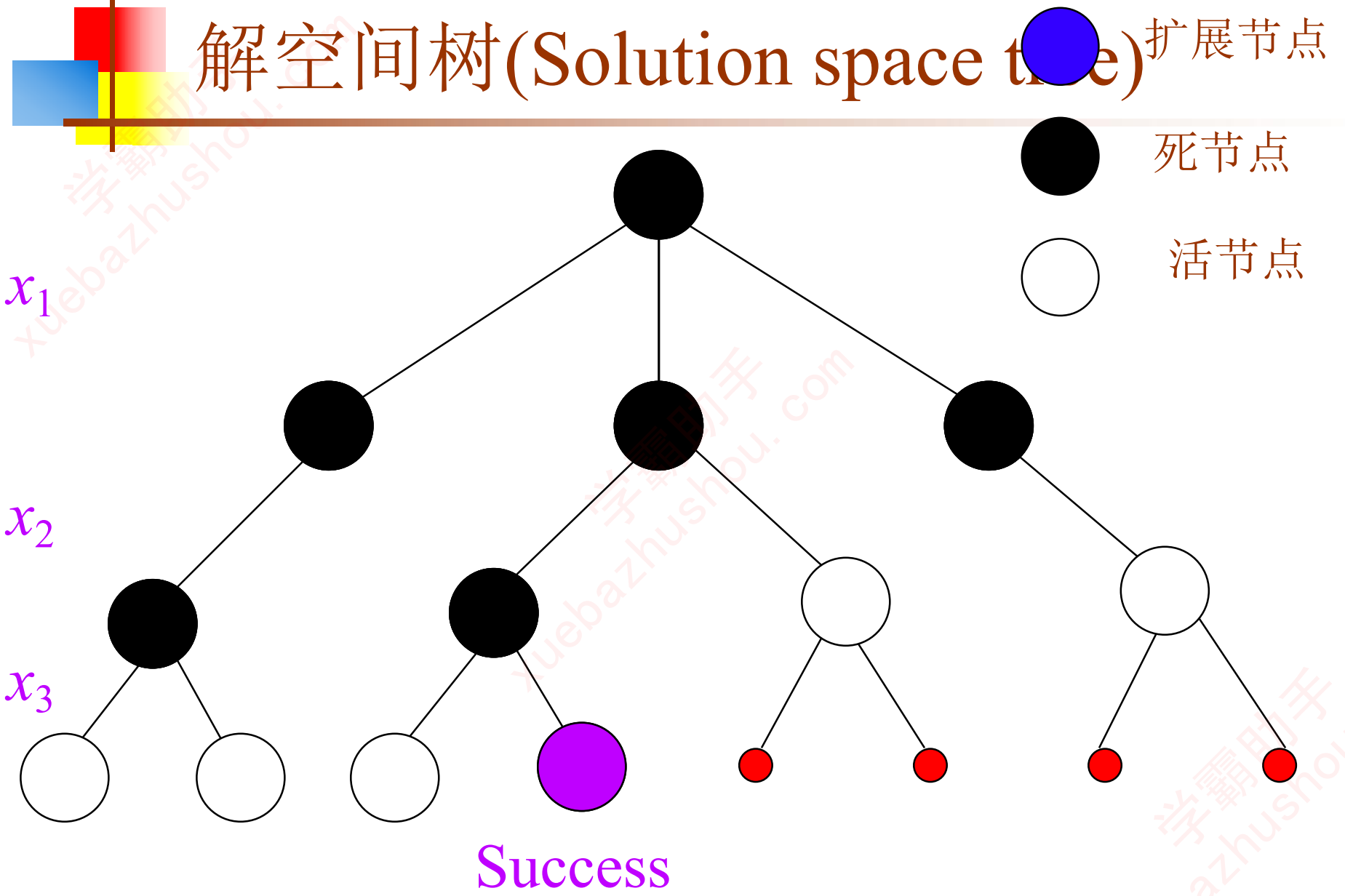


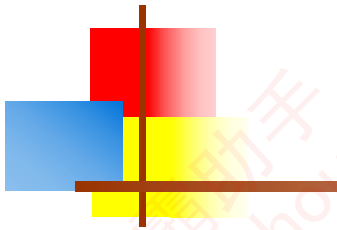
注意分支限界的FIFO不同于广度优先的FIFO在于 它不搜索那些不可行的节点。

2) 最小花费 ,最大收益

用一个优先队列代替一个FIFO队列。这个方法与每个节点的花费或收益相关。如果我们搜索最小花费的解时，那么活节点表就用一个最小堆来表示。下一个活节点是花费最少的节点。如果我们要得到最大收益的解时，活节点表可用一个最大堆来表示。下一个扩展节点为最大收益的节点。

解空间树(Solution space tree)





- 使用分支限界至少需要注意以下4点:

  怎么样计算上界 (极大值问题)

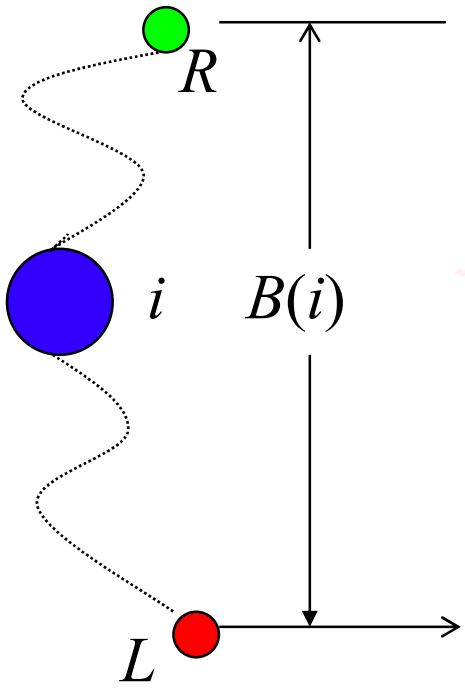
  怎样计算下界 (极小值问题)

  怎样为下一个分支操作选择一个节点.

  怎样扩展一棵搜索树 (BFS, DFS, ...)

- 充分利用限界函数和约束函数来剪去无效的枝并把搜索集中在可以得到解的分支上.

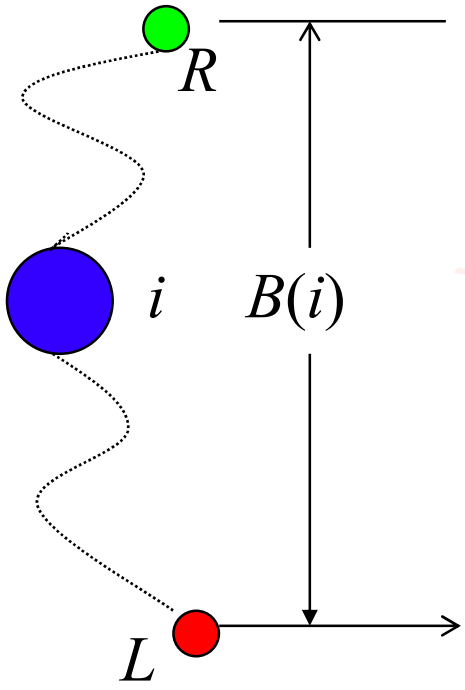
- 通常, 对极大值问题, 我们对扩展节点 i计算上界 $B(i)$. 如果目前保存的最大目标值不比 $B(i)$小, 那么进行剪枝，否则继续.



从根节点R通过 i到叶子 L的任一决策序列的目标函数值不能比 $B(i)$大,所以如果 $B(i)\leq$bestc, 那么它表明搜索扩展节点 i 不会成功, 因此对扩展节点i进行剪枝.

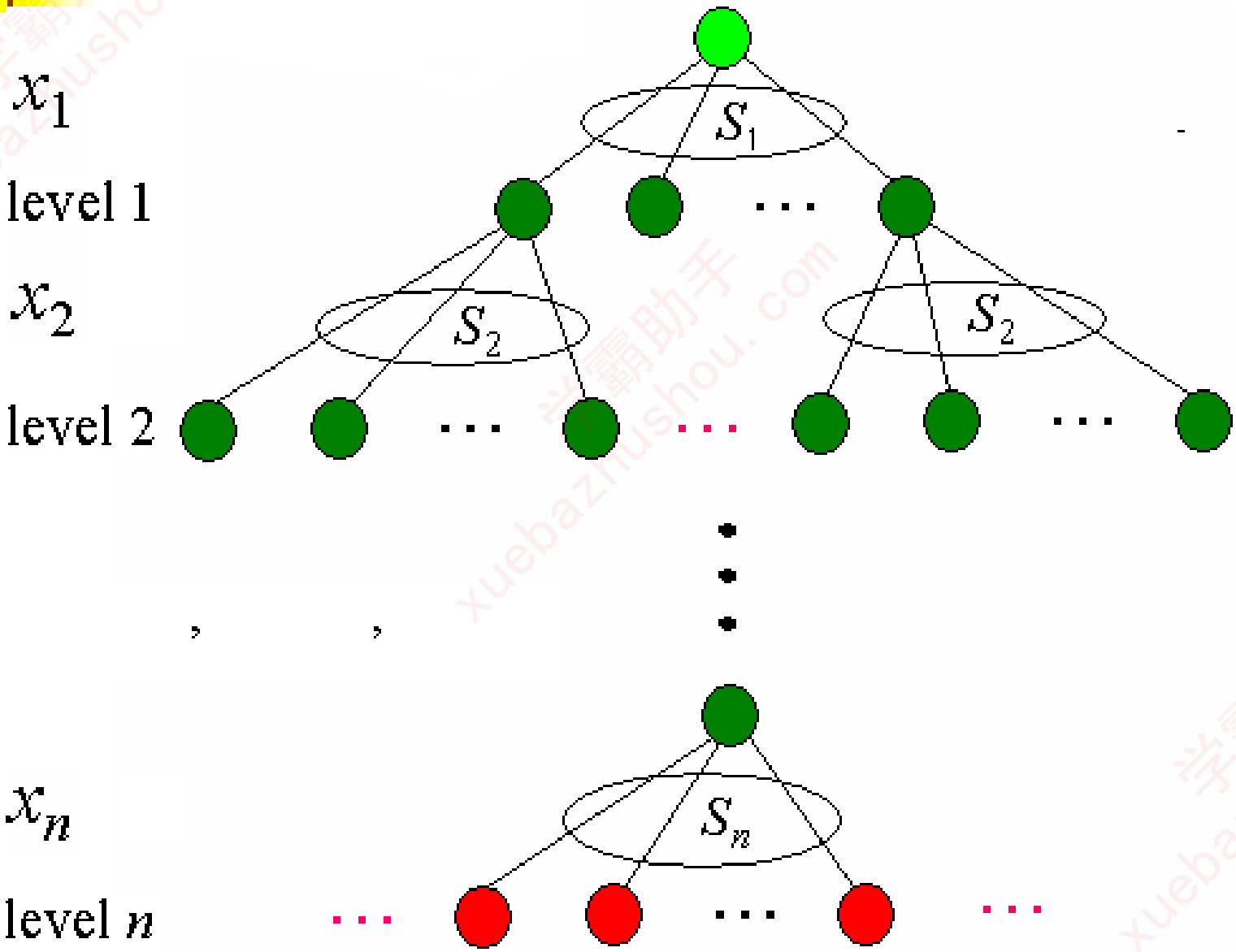
.

- 通常, 极小值问题, 我们对扩展节点 i计算下界 $B(i)$. 如果目前保存的最大目标函数值不比 $B(i)$大, 那么进行剪枝，否则继续.

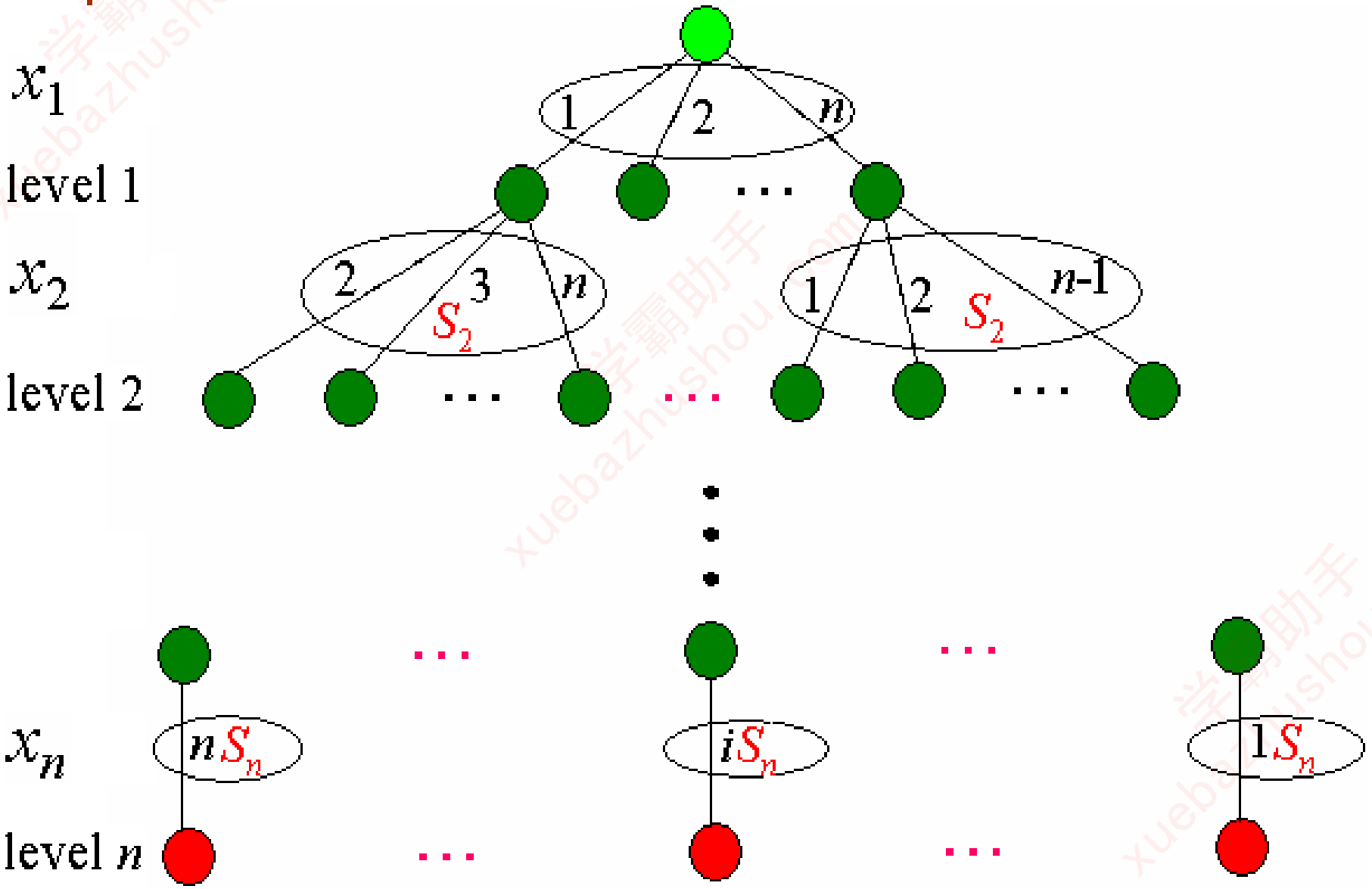


从根节点R 到叶子 L 通过 i的任一决策序列的目标函数值不能比$B(i)$小, 所以如果 $B(i) \geq$bestc, 那么它表明搜索扩展节点 i 不会成功, 因此对扩展节点i进行剪枝.

子集树(Subtree)



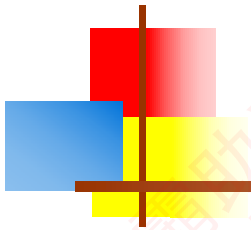
排列树(Permutation tree)



2. 装载问题(Container Loading problem)

- 问题描述
- 有n个集装箱要装上船，集装箱i的重量为w_i。船的最大负载为W。
- 装载问题是在保证不沉船的条件下,在船上装尽可能多的集装箱。
- 解空间

  假设用向量 $(x_1,x_2,\ldots,x_n)$表示解, 这里 $x_i \in \{0,1\}$, $x_i=1$ 表示集装箱 i 被装到船上, $x_i=0$ 表示集装箱i没有被装到船上.



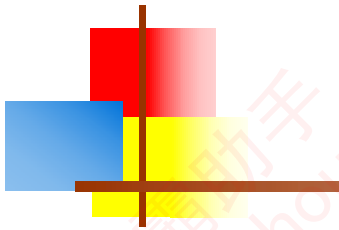
- 装载问题通常可以如下描述:

$$\max \sum_{i=1}^{n} w_i x_i$$

- 解空间树

  子树有个 2^n 叶子

- 在第j层 ,解空间的成员由 x_j划分.



- 约束函数

用$cw(i)$ 当前扩展节点的重量, 即

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

那么约束函数为:

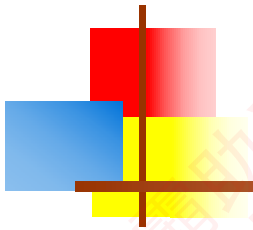
$$C(i)=cw(i-1)+w_i$$

- 剪枝

如果 $C(i)>W$ 那么第i-1层扩展节点的1分支节点不放入活节点表。

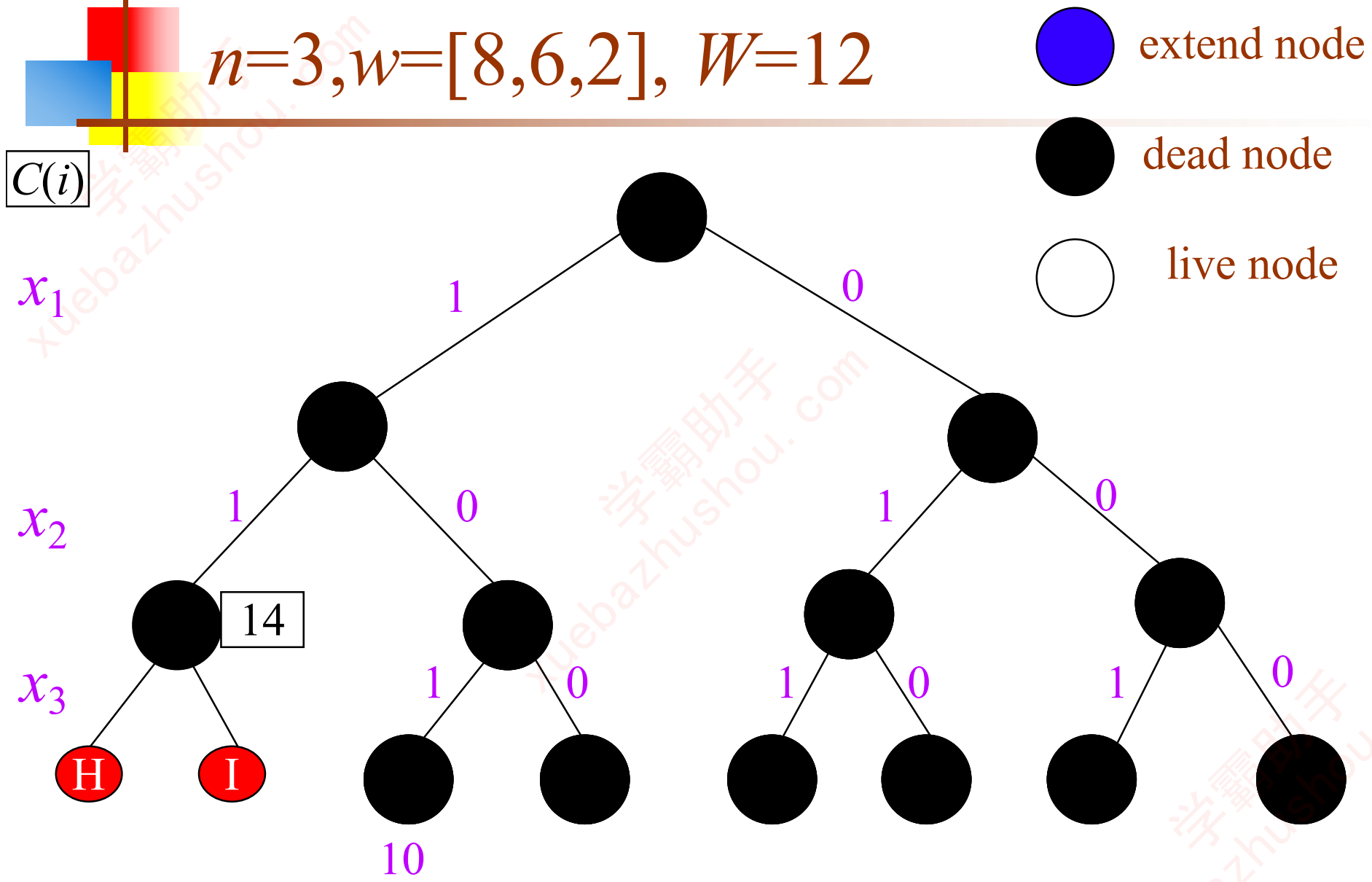
基于约束函数的FIFO(FIFO using constraint function)

```
FIFOMaxLoading(w, W, n)
1   i ← 1
2   Enqueue(Q, -1)
3    cw ← 0;  bestw ← 0
4   while Ø do
5      if C(i)≤W  then
6          SaveQueue(Q, C(i), bestw, i)
7      SaveQueue(Q, cw, bestw, i)
8      cw←Dequeue(Q)
9      if cw = -1 then
10            if Ø then return bestw
11         Enqueue(Q, -1)
12         cw←Dequeue(Q)
13         i ← i+1
14   return bestw
```



```
SaveQueue(Q, wt, bestw, i)
1  if i=n then
2        if wt>bestw then
3              bestw←wt
4  else
5        Enqueue(Q, wt)
```

n=3,w=[8,6,2], W=12



Congratulations!-we made a success, so slowly!

Improved FIFO(FIFO的改进)

- 充分利用限界函数:

$$B(i) = C(i) + r(i)$$

其中, $r(i)$ 剩下集装箱的重量, 即:,

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

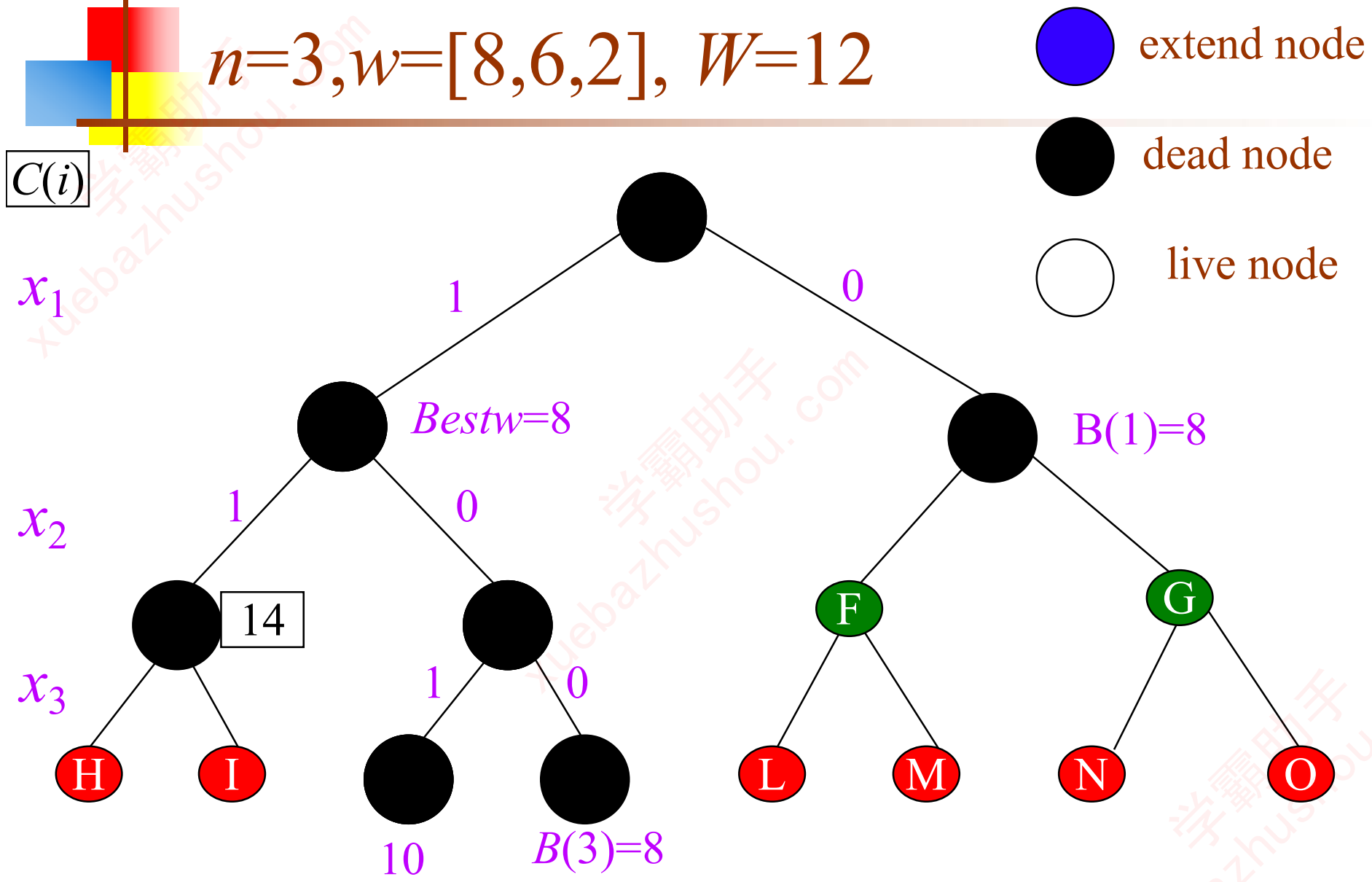
- 剪枝

如果 $B(i) \le \text{bestw}$,那么停止搜索第 i层 和以下的层否则继续搜索. bestw 表示到目前的最好重量值.

改进的FIFO(Improved FIFO)

```
ImprovedFIFOMaxLoading(w, W, n)
1  i←1
2  Enqueue(Q, -1)
3    cw ← 0;  bestw ← 0;  r ←0
4    for j ← 2 to n do r ← r+w[j]
5    while Q≠Ø do
6            if C(i)≤W then
7                        if C(i)>bestw then bestw←C(i)
8                        if i<n then Enqueue(Q, C(i))
9            if B(i)>bestw and i<n then Enqueue(Q, cw)
10           cw←Dequeue(Q)
11     if cw = -1 then
12       if Q = Ø then return bestw
13               Enqueue(Q, -1)
14         cw←Dequeue(Q)
15         i ← i+1
16         r ← r-w[i]
17   return bestw
```

n=3,w=[8,6,2], W=12



Queen 1 2 3 Congratulations! We made a success, so fast!

构造最优解(Construction optimal solution)

```
SolutionFIFOMaxLoading( )
1  i ← 1
2  Enqueue(Q, -1)
3  cw ← 0; bestw ← 0; r ← 0
4    for j ← 2 to n do r ← r+w[j]
5    while Q≠Ø do
6      if C(i)≤W then // x[i] = 1
7            SaveQueue(Q, C(i), i, bestw, E, bestE, bestx, 1)
8     if B(i)>bestw then
9          SaveQueue(Q, cw, i, bestw, E, bestE, bestx, 0)
10    E←Dequeue(Q)
11    if E= -1 then
12          if Q = Ø then return bestw
13        Enqueue(Q, -1); E←Dequeue(Q); i←i+1;  r←r-w[i]
14    cw←E.weight
15 for j←n-1 downto 1 do
16   bestx[j]←bestE.Lchild
17     bestE←bestE.parent
18 return bestw
```

```
SaveQueue(Q, wt, i, bestw, E, bestE, bestx, ch)
1  if i=n then
2        if wt>bestw then
3             bestE ← E
4             bestw ← wt
5             bestx[n] ← ch
6  else
7        b.weight ← wt
8        b.parent ← E
9        b.LChild ← ch
10       Enqueue(Q, b)
```

最大收益(Max-profit branch-and-bound)

- 活节点表是一个最大优先队列. 每个节点i都有一个上界 $B(i)$. 这个上界值是到节点i为止装上船的重量加上剩下集装箱的重量 ,即: $B(i)=C(i)+r(i)$
- 其中, $C(i)$, $r(i)$跟前的定义一样.
- 活节点变成扩展节点是以$B(i)$递减的次序选择.注意如果i是上界重量的一个节点,那么它的子树中没有比上界重量更重的解.
- 如果与叶子节点相关的重量等于它的上界重量,那么当一个叶子成为扩展节点时(以最大收益分支限界),没有其它的活节点能产生比叶子节点更重的重量. 因此,我们可以结束最优装载搜索

```
MaxCostLoading( )
1   i ← 1
2   r[n] ← 0
3   for j ←n-1 downto 1 do   r[j] ←r[j+1]+w[j+1]
4   while i ≠ n+1 do
5         if C(i)≤W then
6                 AddLiveNode(Q, E, C(i)+r[i], 1, i+1)
7         AddLiveNode(Q, E, cw+r[i], 0, i+1)
8         N←ExtractMax(Q)
9         i ←N.level
10        E ←N.ptr
11        cw ←N.weight-r[i-1]
12  for j ← n downto 1 do
13        bestx[j] ←E.Lchild
14      E ←E.parent
15 return cw
```

AddLiveNode(Q, E, wt, ch, lev)

1 $b.parent \leftarrow E$

2 $b.LChild \leftarrow ch$

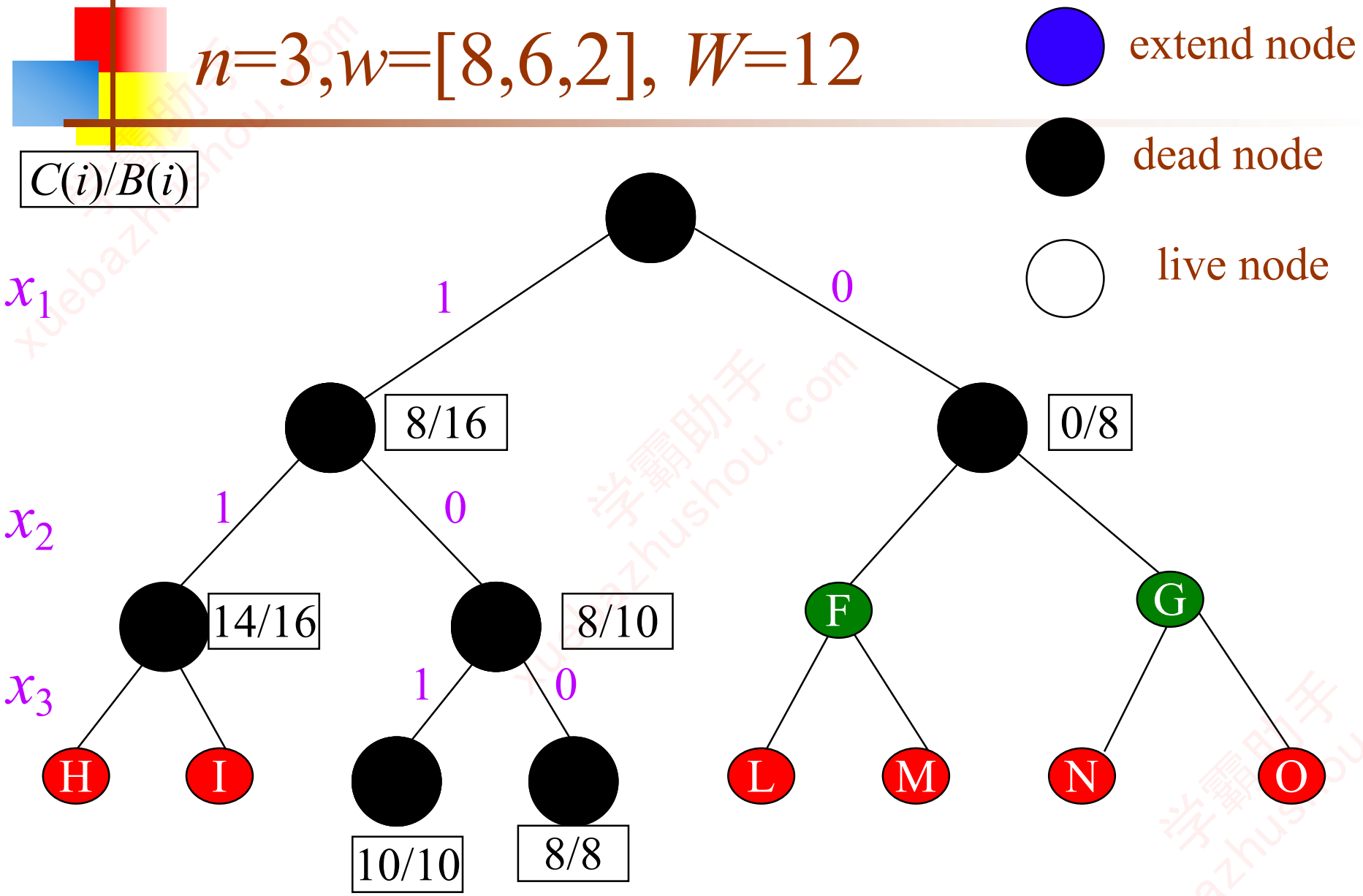
3 $N.weight \leftarrow wt$

4 $N.level \leftarrow lev$

5 $N.ptr \leftarrow b$

6 Insert(Q, N)

n=3,w=[8,6,2], W=12



3. 0/1 Knapsack problem

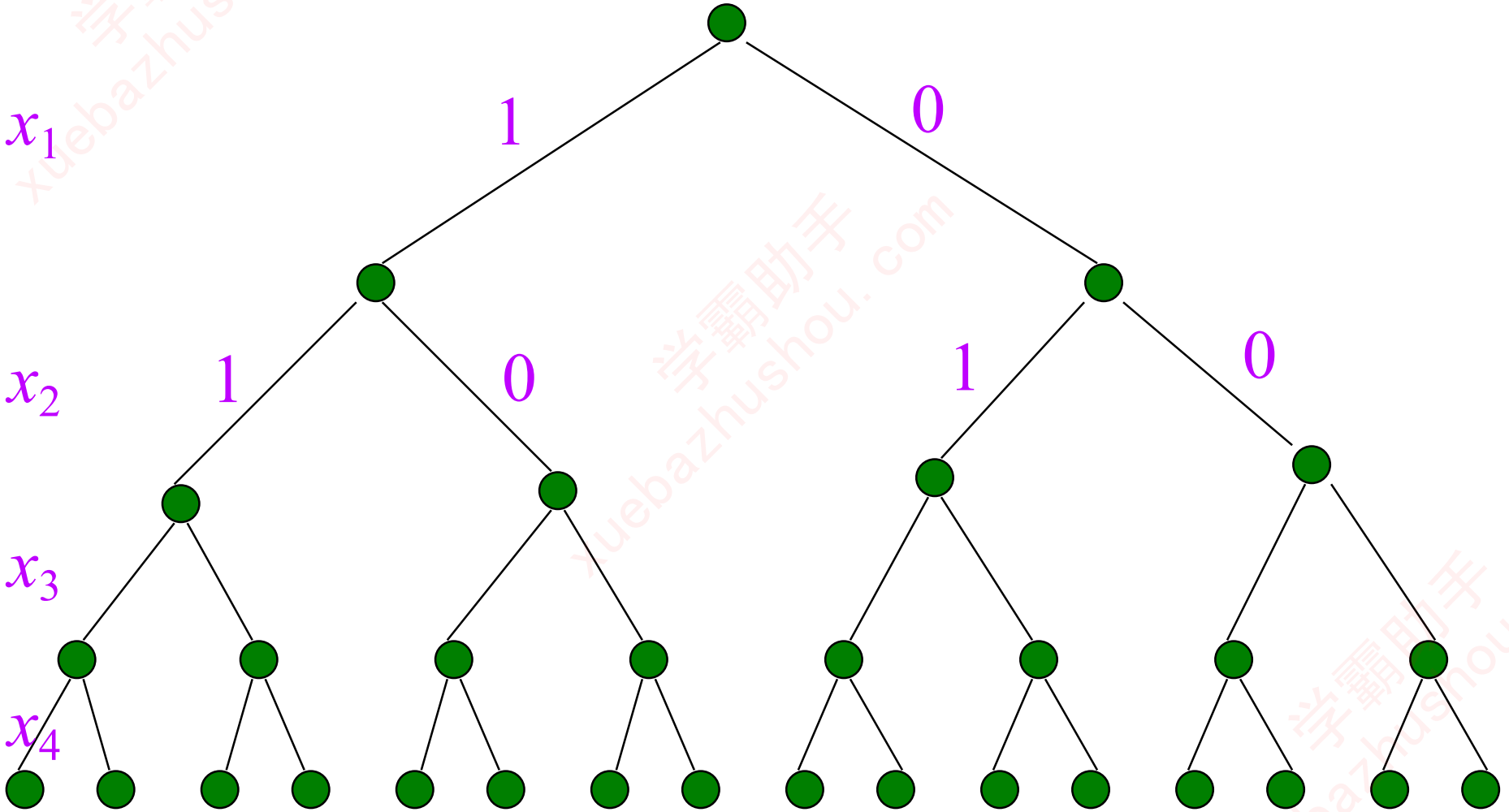
- 问题描述

  有n 个物体,物品i有重量w_i，价值为v_i，这个背包总的容量大小为W。0-1背包问题是从物品的集合中选择最大价值的一个子集，使其容量小于等于W。

- 解空间

  假设解用向量 $(x_1,x_2,\ldots,x_n)$表示, $x_i \in \{0,1\}$, $x_i=1$ 表示i放入包中, $x_i=0$ 表示没有放入包中.

Subtree: n=4



- 约束函数:

用$cw(i)$ 到达第i层时扩展节点的重量, 即

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

那么,约束函数为:

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

- 剪枝

如果$C(i)>W$,那么停止搜索第 i层及该节点以下的层搜索,否则继续搜索.

最大收益(Max-profit branch and bound)

- 限界函数

对在第i层的扩展节点N, 在N的收益为:

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

其中, $cv(i)$, $r(i)$ 定义同前的.

如果 $B(i) \le \text{bestv}$, 那么停止搜索以 N为根节点的子树, 节点 N 成为死节点. 否则继续.

同时,有最大收益的活节点优先扩展.

- 跟前面一样, $r(i)$ 的计算如下:

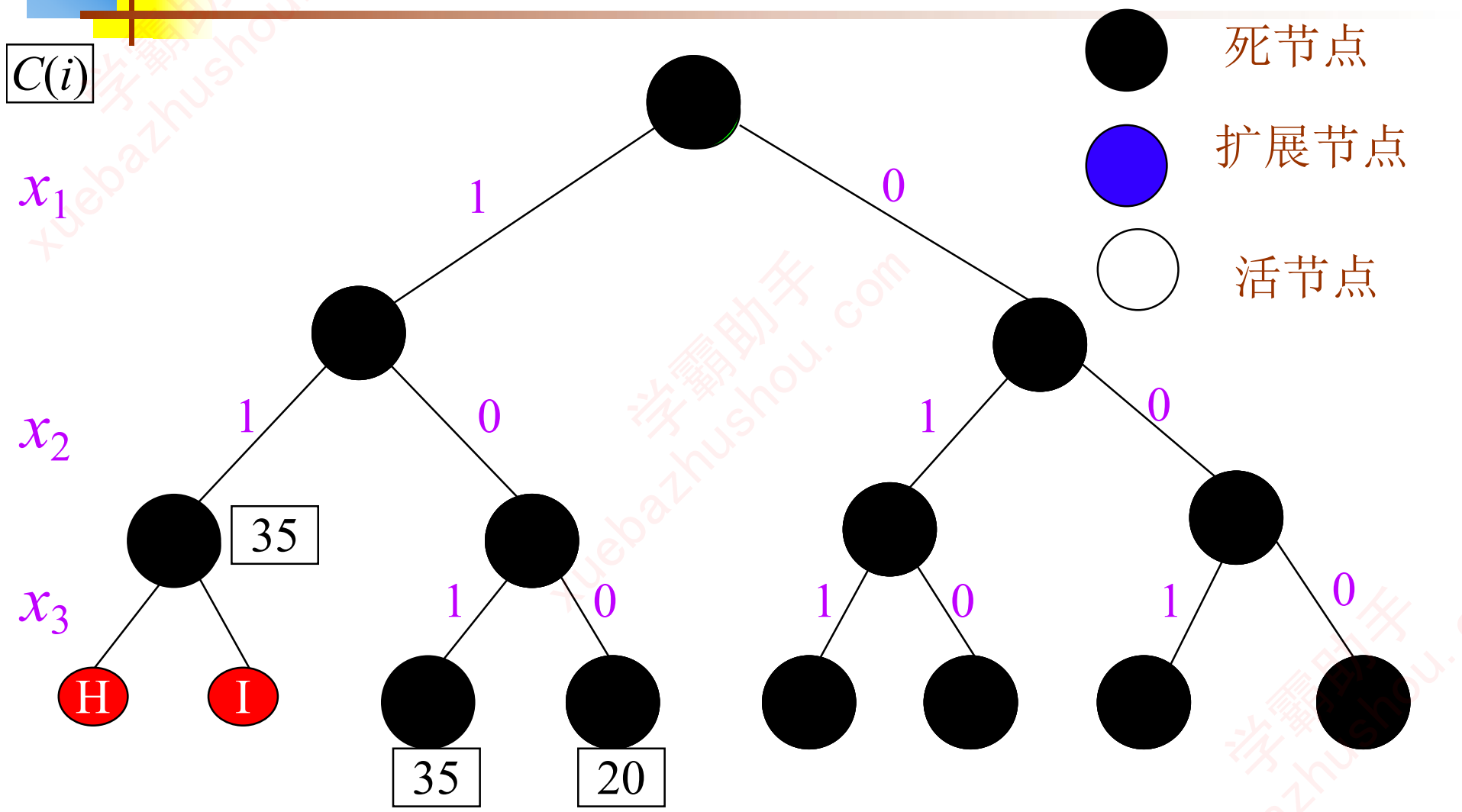
首先, 按value/weight的比值提前进行降序排序, 即

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

在第 i层时, 背包还能装的容量为 W-$cw(i)$, 由贪心策略把剩下的物品放入包中直到物品k不能在放进去为止,得到:

$$r(i) = \sum_{j=i+1}^{k-1} v_j + (W - cw(i) - \sum_{j=i+1}^{k-1} w_j)(\frac{v_k}{w_k})$$

n=3,W=30,w=[20,15,15],v=[40,25,25]



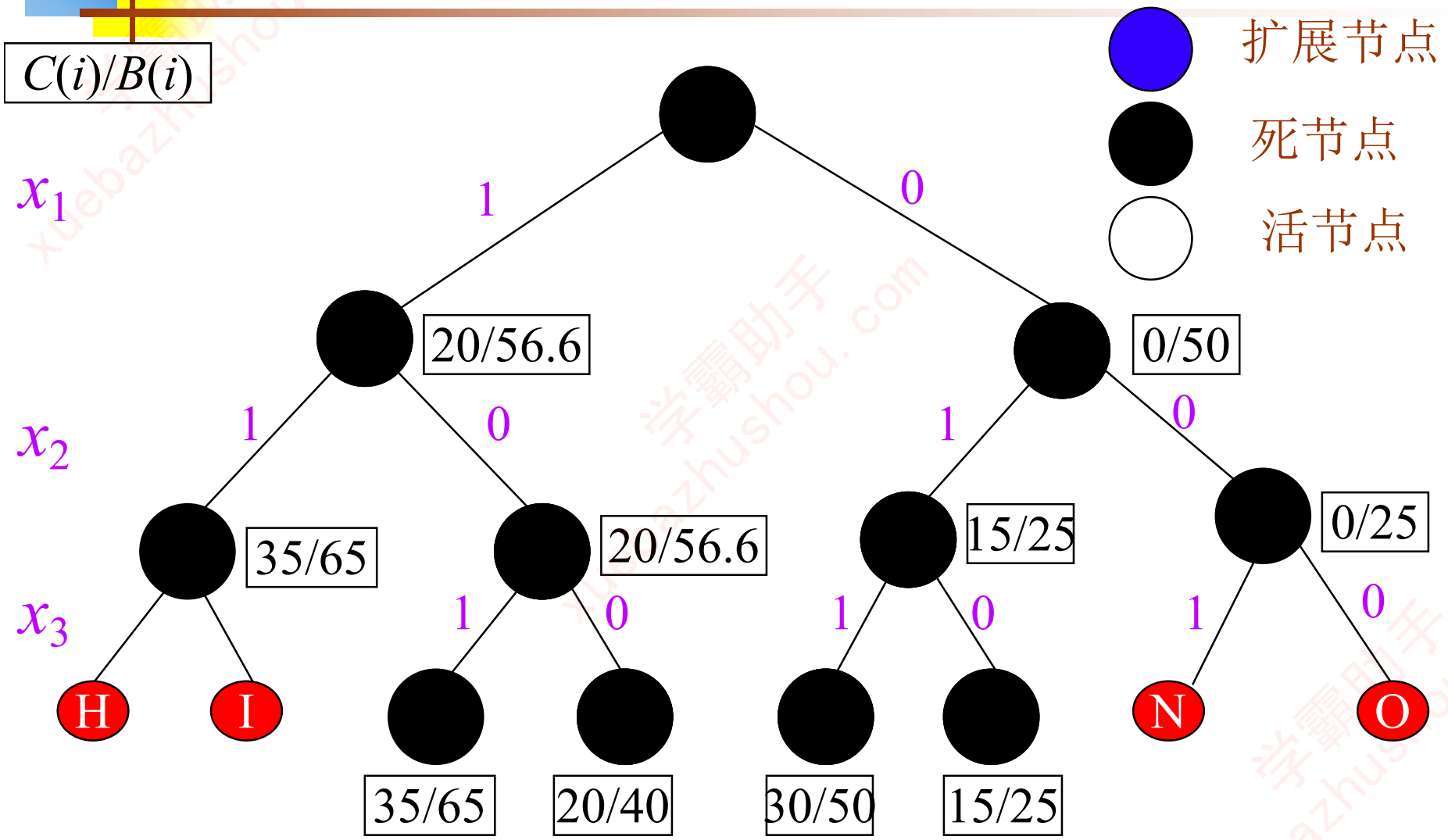
FIFO

```
FIFOKnapsack()
1   Enqueue(Q, -1)
2   i ← 1; cw ← 0; cv ← 0; bestv← 0; r ← 0
3   for j ← 2 to n do  r ← r+v[j]
4   while Q≠Ø do
5       if  C(i)≤W then
6               SaveQueue(Q, cv+v[i], cw+w[i], i, bestv, E, bestE, bestx, 1)
7       if  B(i)>bestv then
8               SaveQueue(Q, cv, cw, i, bestv, E, bestE, bestx, 0)
9     E←Dequeue(Q)
10    if  E=-1  then
11            if Q=Ø then return bestv
12              Enqueue(Q, -1)
13            E←Dequeue(Q)
14            i ← i+1
15      r ← r-v[i];  cw ← E.weight;  cv ← E.value
16  for j ← n-1 downto 1 do
17          bestx[j] ← bestE.Lchild
18        bestE ← bestE.parent
19  return bestv
```

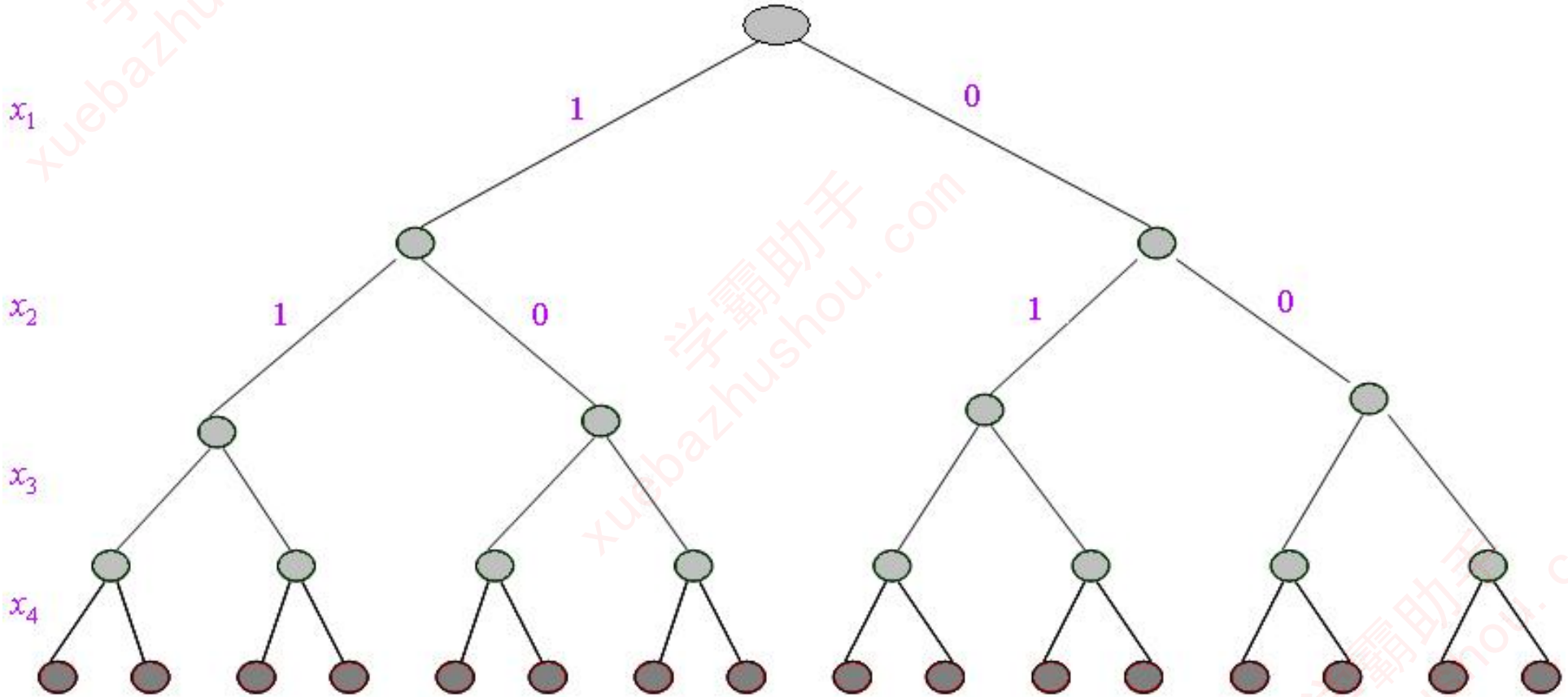
Max-profit branch and bound

```
MaxProfitKnapsack()
1   i ← 1
2   uv ← B(1);  bestv ←0
3   while i≠n+1 do
4      if  C(i) ≤W then
5           if cv+v[i]>bestv then  bestv ← cv+v[i]
6           AddLiveNode(uv, cv+v[i], C(i), 1, i+1)
7         uv ← B(i)
8      if  B(i)>bestv  then
9         AddLiveNode(B(i), cv, cw, 0, i+1)
10     N←ExtractMax(Q); E←N.ptr;  cw←N.weight
11     cv ← N.value; uv ← N.upvalue;  i ← N.level
12   for  j ←n  to 1 do
13     bestx[j] ←E.LChild;  E ← E.parent
14 return bestv
```

n=3,W=30,w=[20,15,15], v=[40,25,25]



4. SAT problem

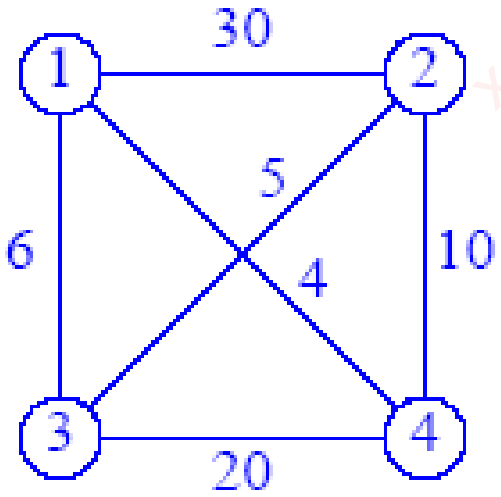


```
MaxProfitSAT()
1    i ← 1
2    while i≠m+1 do
3         cv←ok(i)
4         if cv >0 then
5              AddLiveNode(cv, 1, i+1)
6         cv←ok(i)
7         if cv >0 then
8              AddLiveNode(cv, 0, i+1)
9         N←ExtractMax(Q); i ← N.level
10   for j ←m downto 1 do
11        bestx[j] ←E.LChild; E ← E.parent
```

```
ok(i)
1    cn←0
2    for j←1 to n do
3         if C_j =0 then return 0
4         else if C_j =1 then cn←cn+1
5    return cn
```

5. Traveling Salesperson Problem

- 问题描述
- 有*n*个点的网络图（有向或是无向），问题是要找一条包含所有顶点一次且仅一次的最小代价的回路。
- 在旅行商问题中，我们要找一条最小代价的回路.

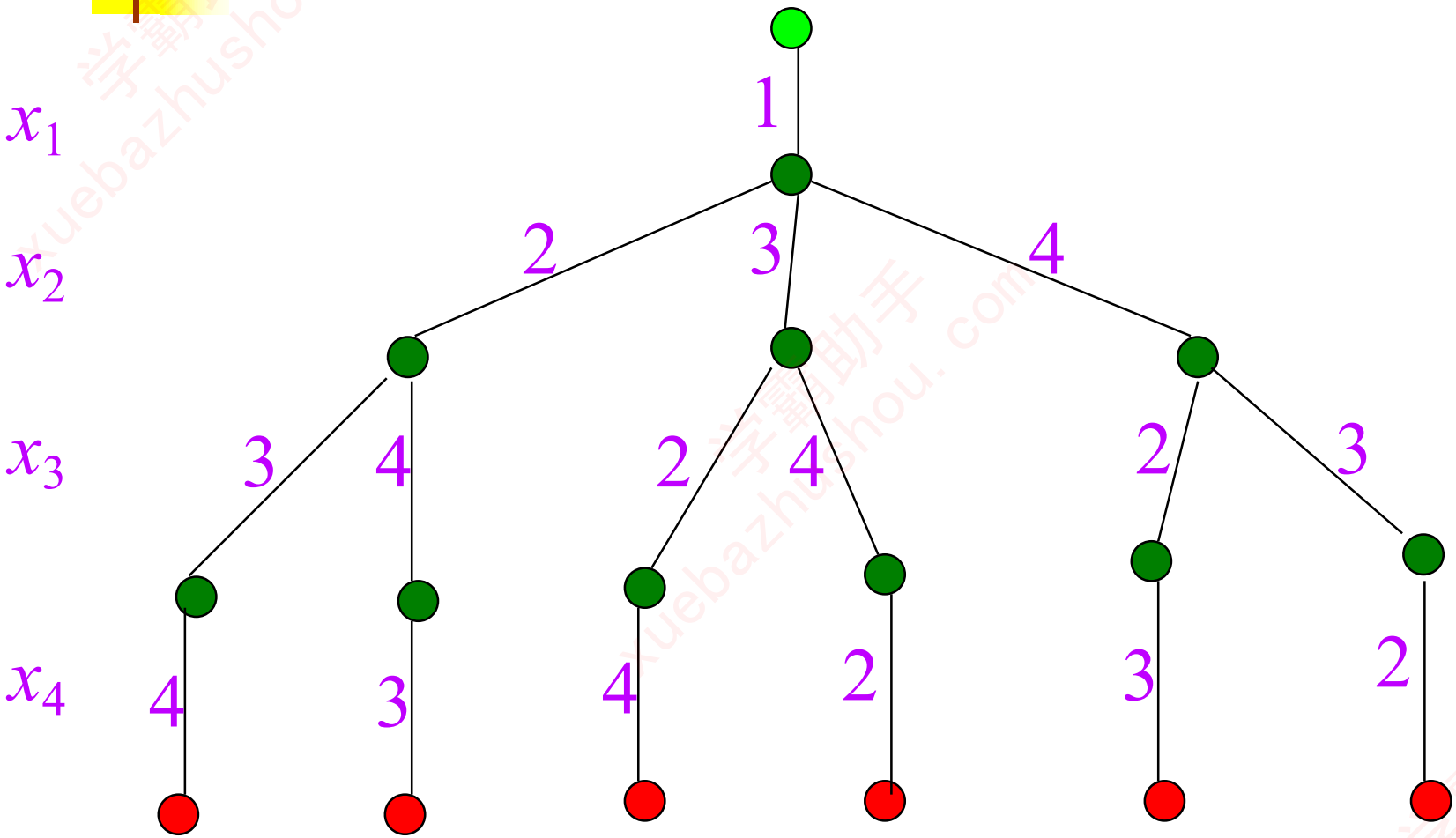


在这个图中的一些路线为1,2,4,3,1; 1,3,2,4,1; 1,4,3,2,1.其中, 1,3,2,4,1 的花费是25, 是最小的花费.

- 解空间
- 既然旅行路线是包含所有顶点的一个环，我们可以选择从任意一点开始(并作为结尾)。
- 假设把第一个顶点作为开始和结尾。每次的旅行路线可以描述为序列 1, x_2,...,x_n, 1其中 x_2,...,x_n 是(2, 3, ..., n)的一个排列.所有可能的路线可以被描述为一棵排列树，在这棵树中根到叶子决定着一条路线 .

  解空间大小为 3!

排列树(Permutation tree: n=4)



- 数据结构

用 $w[i,j]$ 表示点 i 和点 j的距离或是花费, 如果$w[i,j]=\infty$ 表示它们之间没有边.

- 限界函数 $cc(i)=\sum_{j=2}^{i} a[x[j-1]][x[j]]$

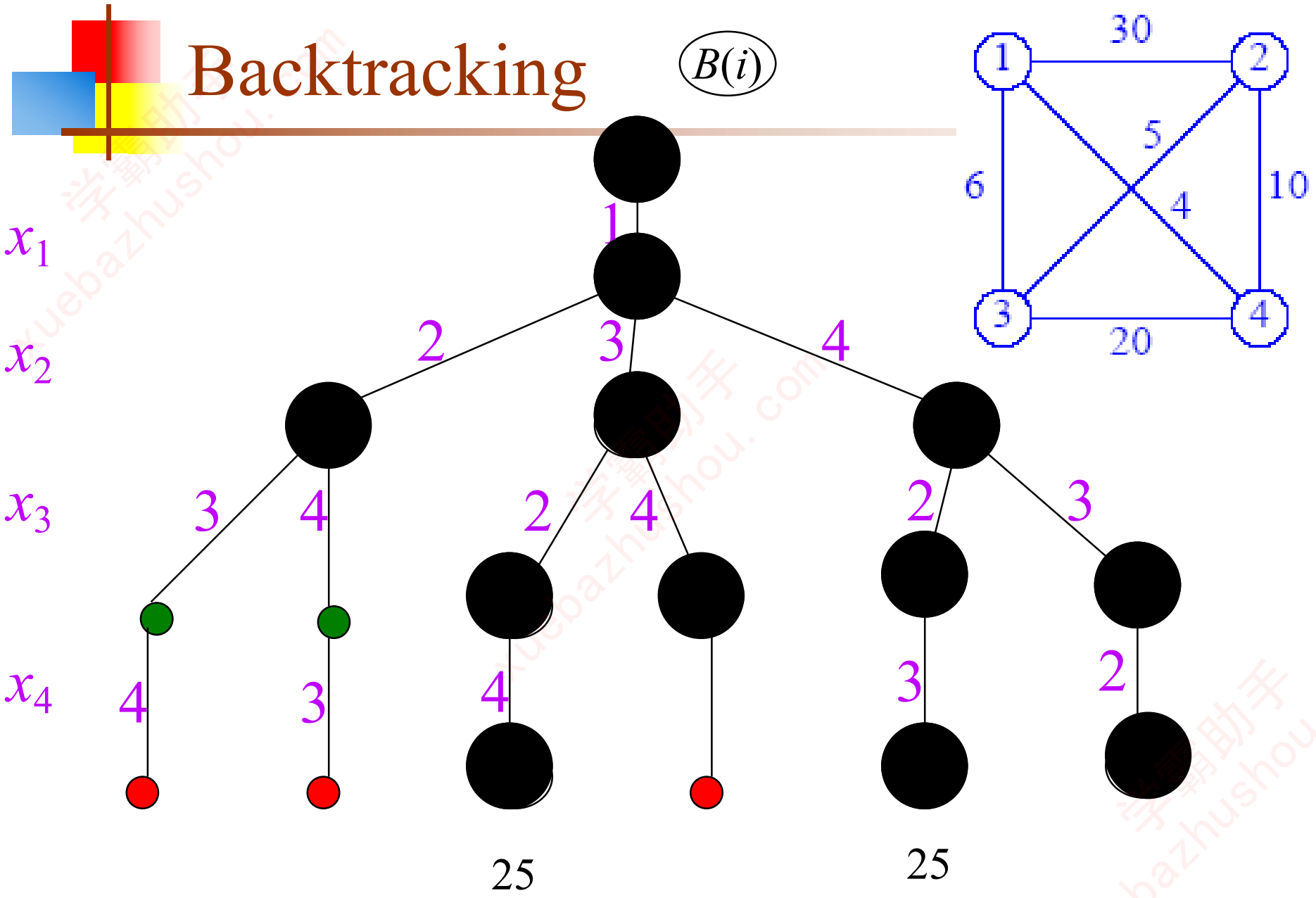
扩展节点i的代价为:

$B(i)=cw(i\text{-}1)+w[x[i\text{-}1],x[i]]$,

如果 $B(i)>$bestc,则不用把$x[i]$放入活节点表，否则放入。

同时,有最小花费的活节点优先扩展.

Backtracking



- 限界函数的进一步改进

扩展节点i的花费可表示如下:

$B(i)=cw(i)+rw(i)$

其中, $cw(i)$ 跟前面定义一样, $rw(i)$是从剩下的点到其它的最小花费的总和, 即:

$$rw(i)=\sum_{j=i+1}^{n}\min_{i<k<n,k\neq j}\{w[x[j],x[k]]\}$$

如果$B(i)\geq$bestc,则不用把$x[i]$放入活节点表，否则放入。

同时,有最小花费的活节点优先扩展.

```
MinWeightTSP()
1  MinSum ← 0
2  for i ← 1 to n do
3      Min ← ∞
4      for j ← 1 to n do
5          if  w[i, j] ≠∞ and w[i, j]< Min  then  Min ←w[i, j]
6      if Min = ∞ then return ∞
7      MinOut[i] ← Min
8      MinSum ← MinSum+Min
9  for i ← 1 to n  do  E.x[i] ← i
10 E.s←1;  E.cw←0;  E.rw←MinSum;   bestw←∞
11 while E.s < n do
12      if E.s = n-1 then
13          if w[E.x[n-1], E.x[n]] ≠∞ and w[E.x[n], E.x[1]] ≠∞ and
E.cw+ w[E.x[n-1], E.x[n]]+w[E.x[n], E.x[1]]< bestw then
14              bestw←E.cw+ w[E.x[n-1], E.x[n]]+w[E.x[n], E.x[1]]
15              E.cw ← bestw;  E.lw ← bestw
```

```
16            E.s ←E.s+1
17            Insert(Q, E)
18      else for i ← E.s + 1 to n do
19          if  w[E.x[E.s], E.x[i]] ≠ ∞ then
20              cw ← E.cw + w[E.x[E.s], E.x[i]]
21              rw ← E.rw - MinOut[E.x[E.s]]
22              B(i) ← cw + rw
23              if B(i) < bestw then
24                  for  j ← 1 to n do  N.x[j] ← E.x[j]
25                  N.x[E.s+1] ← E.x[i]
26                  N.x[i] ← E.x[E.s+1]
27                  N.cw ← cw;   N.s ← E.s + 1
28                  N.lw ←B(i);   N.rw ← rw
29                  Insert(Q, N)
30      E←ExtractMin(Q)
31  if bestw = ∞ return ∞
32  for i ←1 to n do  bestx[i] ←E.x[i]
33  return bestw
```

```
21    else for i ← E.s + 1 to n-1 do
22            if  a[E.x[E.s]][E.x[i]] != ∞ then
23               cc ← E.cc + a[E.x[E.s]][E.x[i]]
24               rcost ← E.rcost - MinOut[E.x[E.s]]
25               b ← cc + rcost
26               if (b < bestc || bestc = ∞ ) then
27                  for  j ← 0 to n-1 do  N.x[j] ← E.x[j]
28                  N.x[E.s+1] ← E.x[i]
29                  N.x[i] ← E.x[E.s+1]
30                  N.cc ← cc;     N.s ← E.s + 1
31                  N.lcost ← b;   N.rcost ← rcost
32                  H.Insert(N)
33            delete E.x
34    if (bestc = ∞) return ∞
35    for i ←0 to n-1  do  v[i+1] ←E.x[i]
```

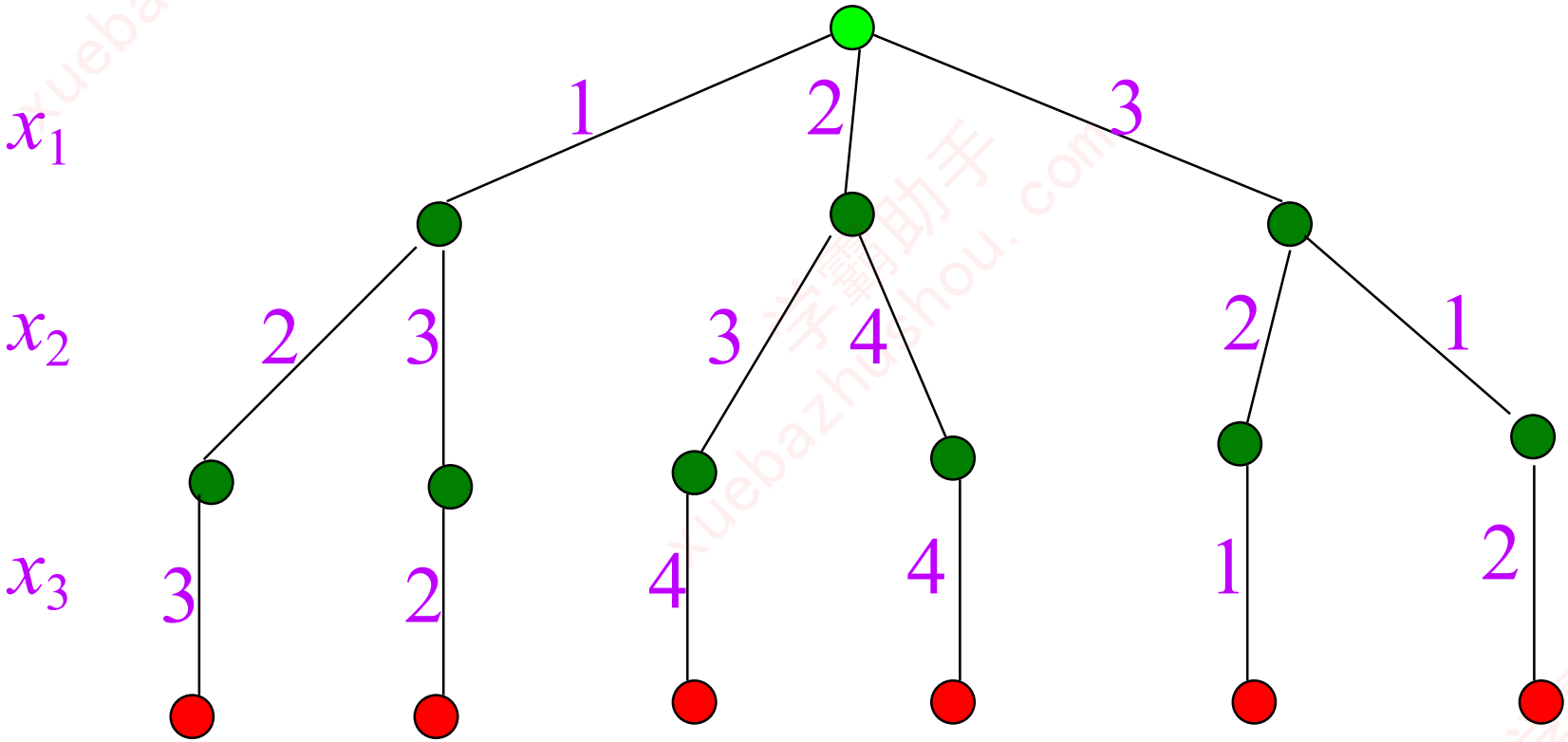
6. 流水车间调度问题

- 问题

给定 n 个工作 $J=(j_1,j_2,\ldots,j_n)$, 每个工作有两个分别在两台机器上进行处理的任务.一台机器一次只能处理一项任务,并且操作一旦开始就必须进行下去.此外，只有等到机器1上正在进行的任务完成以后，机器2才能对相同工作的另一个任务进行处理,也就是说，每个工作只能在机器1和机器2上轮流进行。 每个工作 i 在第 j 台机器上需要一个处理时间 $t[i,j]$.

Permutation tree: n=3

解空间可以看作是一棵排列树



限界函数的改进

用 $x=\{x[1],x[2],\ldots,x[i]\}$表示操作已到扩展节点i的作业的集合。那么，

$$f=\sum_{j=1}^{i}F[x[j],2]+rf(i)$$

计算第二项的值是很困难的，我们可以估计它的最小下界。？

- 考虑以下两点:

1)剩下的每个操作将陆续的有机器1和机器2没有等待的完成，即一个工作在机器1完成，一个在极其完成而没有等待，那么，

$$rf1(i)=\sum_{j=i+1}^{n}(F[x[i],1]+(n-j+1)t[x[j],1]+t[x[j],2])$$

此时有

$$rf(i)\geq rf1(i)$$

2) 机器2没有等待时间，即在机器2完成一个作业后，它能没有等待的处理下一个作业,此时有

$$rf2(i)=\sum_{j=i+1}^{n}(\max\{F[x[i],2],F[x[i],1]+\min_{i\le k\le n}t[x[k],1]\}+(n-j+1)t[x[j],2])$$

显然有 $rf(i)\ge rf2(i)$

对情况 1), 按非递减对作业排序$t[j,1]$, 那么我们可以得到更小$rf1(i)'$, 即, $rf1(i)'\le rf1(i)$

对情况 2),按非递减对作业排序$t[j,2]$, 得到的更小为 $rf2(i)'$, 即,

$$rf2(i)'\le rf2(i)$$

那么,有:

$$f = \sum_{j=1}^{i} F[x[j],2] + rf(i)$$

$$\geq \sum_{j=1}^{i} F[x[j],2] + \max\{rf1(i), rf2(i)\}$$

$$\geq \sum_{j=1}^{i} F[x[j],2] + \max\{rf1(i)', rf2(i)'\}$$

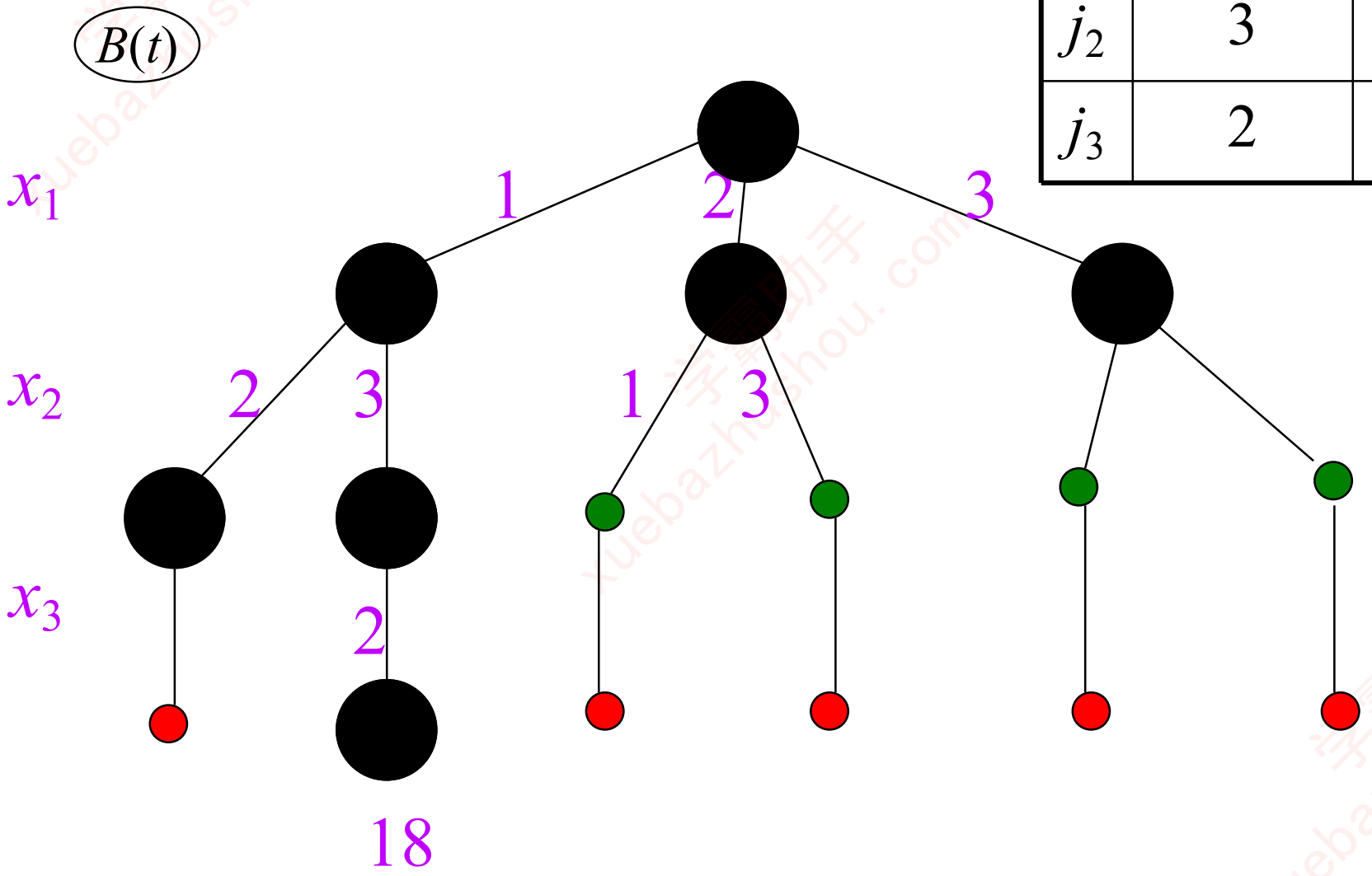
扩展节点i的值

$$B(i) = \sum_{j=1}^{i} F[x[j],2] + \max\{rf1(i)', rf2(i)'\}$$

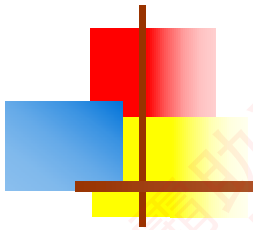
如果 $B(i)\geq$bestf,则不用将节点i放入活节点表，否则放入，与此同时，权值$B(i)$最小的活节点优先扩展

分支定界

t_{ij}	机器1	机器2
j_1	2	1
j_2	3	1
j_3	2	3



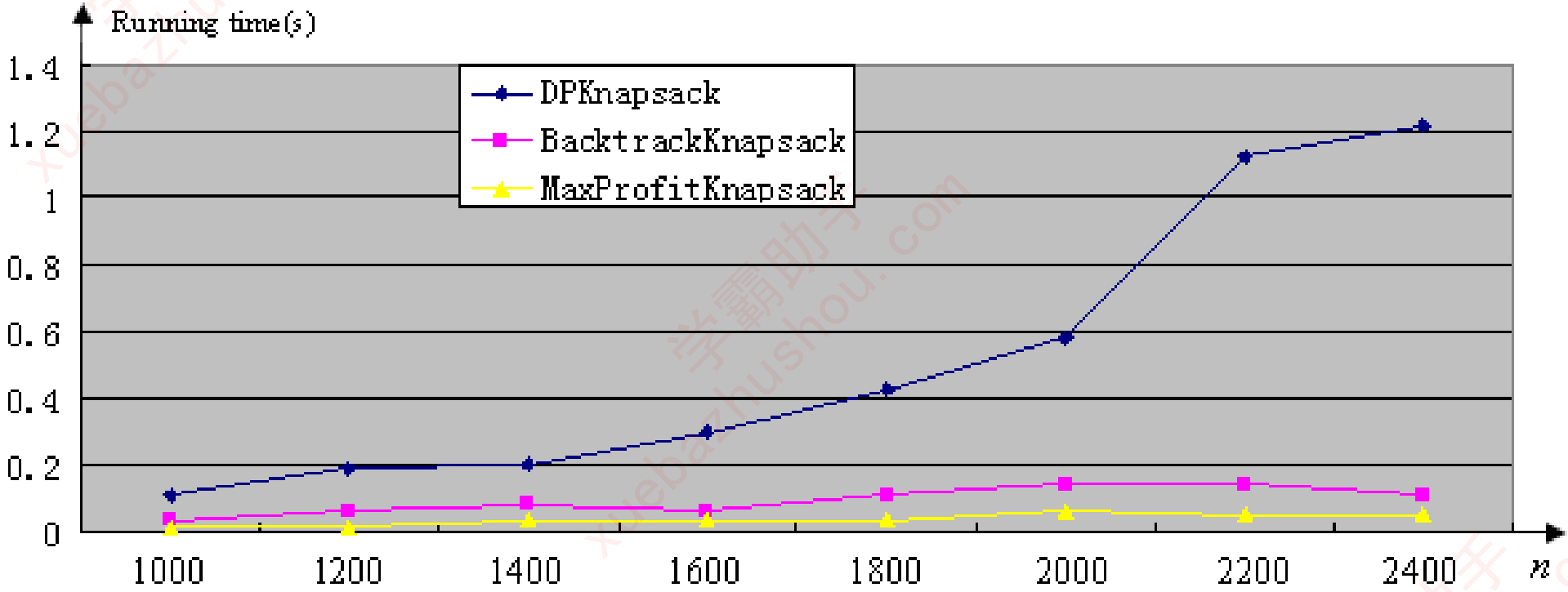
```
MinCostFlowShop()
1  E.s←1; E.f ← 0; E.f2 ← 0; E.f1 ← 0
2  for j←1 to n do E.x[j] ←j
3  bestf←∞
4  while E.s≤n do
5     if E.s=n then
6        if E.f<bestf then
7           bestf←E.f
8           for i←0 to n do bestx[i] ←E.x[i]
9           return bestf
10    else  for i←E.s+1 to n do
11             for j←1 to n do
12                                N.x[j] ←E.x[j]
13             N.x[E.s + 1] ← E.x[i]
14             N.x[i] ← E.x[E.s + 1]
15             f1 ← E.f1 + t[N.x[E.s+1], 1]
16             if E.f2 > f1 then  f2 ← E.f2 + t[N.x[E.s + 1], 2]
17             else f2 ← f1 + t[N.x[E.s + 1], 2]
18          N.f ← E.f + f2
19          N.s ← E.s + 1
20          N.f2 ← f2
21          N.f1 ← f1
22          if  B(i) < bestf then  Insert(Q, N)
23    E←ExtractMin(Q)
24 return bestf
```

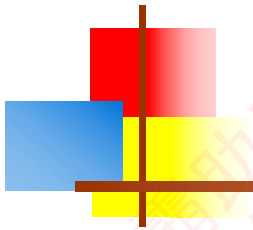


13.7 0/1 knapsack实验

表13.1 DPKnapsack、BacktrackKnapsack和MaxProfitKnapsack的实验结果

n	1000	1200	1400	1600	1800	2000	2200	2400
DPKnapsack	0.109	0.187	0.203	0.296	0.421	0.578	1.125	1.218
BacktrackKnapsack	0.031	0.063	0.078	0.063	0.11	0.14	0.14	0.109
MaxProfitKnapsack	0.015	0.015	0.031	0.031	0.031	0.062	0.046	0.046
Optimal value	282000	414610	455339	607732	748955	940129	1305502	1312372





作业

262-263