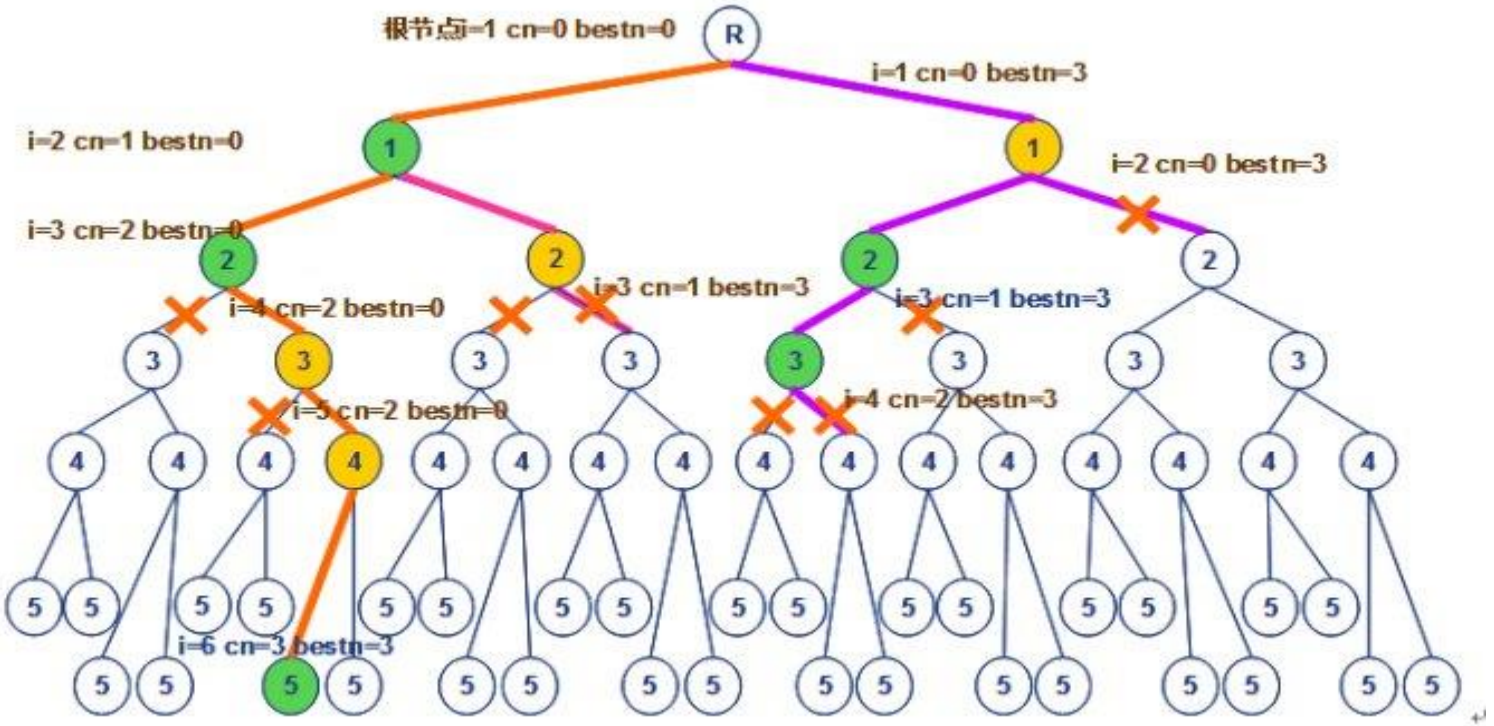
算法设计与分析

回溯法



主讲： 朱东杰 博士、硕导

地点： M楼305

电话/微信： 18953856806

Email： zhudongjie@hit.edu.cn

图搜索——回溯法

- **学习要点**
- 理解回溯法的深度优先搜索策略。
- 掌握用回溯法解题的算法框架
- （1）递归回溯
- （2）迭代回溯
- （3）子集树算法框架
- （4）排列树算法框架

图搜索——回溯法

- 通过应用范例学习回溯法的设计策略。
- （1）0-1背包问题；
- （2）旅行售货员问题
- （3）装载问题；
- （4）n后问题；
- （5）最大团问题；

- 课外
- （6）图的m着色问题
- （7）圆排列问题
- （8）电路板排列问题
- （9）连续邮资问题
- （10）批处理作业调度；
- （11）符号三角形问题

回溯法概念

- 回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。

- 回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

- 许多复杂的，规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。

回溯法基本思想

- 回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。回溯法适用于解一些组合数相当大的问题。
- 在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。
- 若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。
- 而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

用回溯法解题的一般步骤

- （1）针对所给问题，确定问题的解空间：

首先应明确定义问题的解空间，问题的解空间应至少包含问题的一个（最优）解。

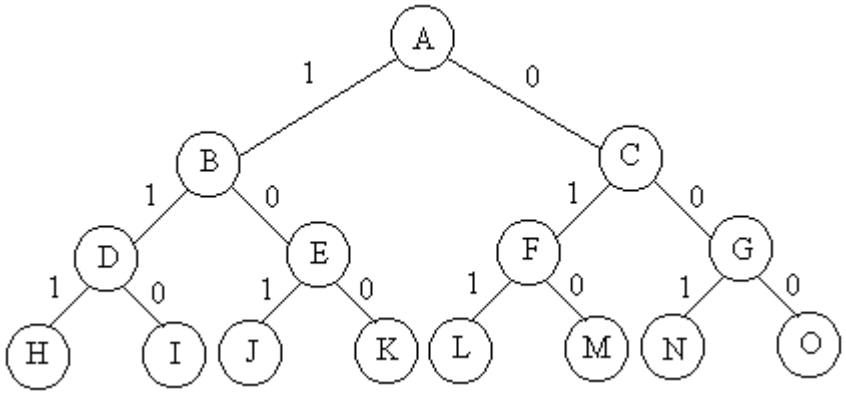
- （2）确定结点的扩展搜索规则

- （3）以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

问题的解空间

- 问题的解向量：回溯法希望一个问题的解能够表示成一个n元式(x1, x2, …, xn)的形式。
- 显约束：对分量xi的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。
- 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。



n=3时的0-1背包问题用完全二叉树表示的解空间

生成问题状态的基本方法

- 扩展结点:一个正在产生儿子的结点称为扩展结点
- 活结点:一个自身已生成但其儿子还没有全部生成的节点称做活结点
- 死结点:一个所有儿子已经产生的结点称做死结点
- 深度优先的问题状态生成法：如果对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点。在完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子（如果存在）
- 宽度优先的问题状态生成法：在一个扩展结点变成死结点之前，它一直是扩展结点
- 回溯法：为了避免生成那些不可能产生最佳解的问题状态，要不断地利用限界函数(bounding function)来处死那些实际上不可能产生所需解的活结点，以减少问题的计算量。具有限界函数的深度优先生成法称为回溯法

递归回溯

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。其中t为搜索的深度，框架如下：

```
//针对N叉树的递归回溯方法
1.void backtrack (int t)
2.{
3.    if (t>n) output(x); //叶子节点，输出结果，x是可行解
4.    else
5.       for i = 1 to k//当前节点的所有子节点
6.        {
7.            x[t]=value(i); //每个子节点的值赋值给x
8.            //满足约束条件和限界条件
9.          if (constraint(t)&&bound(t))
10.                backtrack(t+1);  //递归下一层
11.       }
12.}
```

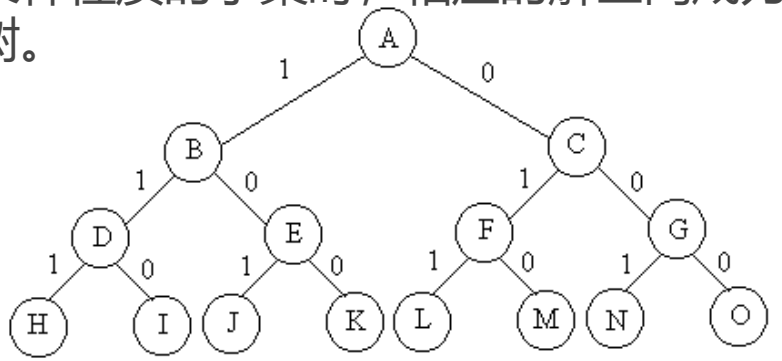
迭代回溯

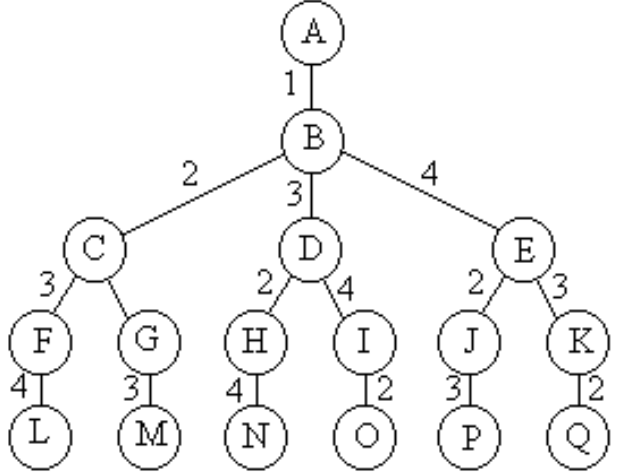
采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

```
//针对N叉树的迭代回溯方法
1.void iterativeBacktrack () {
2.    int t=1;
3.    while (t>0) {
4.       if(ExistSubNode(t)) //当前节点的存在子节点
5.       {
6.          for i = 1 to k  //遍历当前节点的所有子节点
7.          {
8.             x[t]=value(i);//每个子节点的值赋值给x
9.             if (constraint(t)&&bound(t))//满足约束条件和限界条件
10.              {   //solution表示在节点t处得到了一个解
11.                 if (solution(t)) output(x);//得到问题的一个可行解，输出
12.                 else t++;//没有得到解，继续向下搜索
13.              } } }
14.       else //不存在子节点，返回上一层
15.       {
16.          t--;
17.       } } }
```

子集树与排列树

所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间成为子集树。





所给的问题是确定n个元素满足某种性质的排列时，相应的解空间就是排列树。

遍历子集树需$O(2^n)$计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++) {
        x[t]=i;
        if (legal(t)) backtrack(t+1);
      }
}
```

遍历排列树需要O(n!)计算时间

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=t;i<=n;i++) {
        swap(x[t], x[i]);
        if (legal(t)) backtrack(t+1);
        swap(x[t], x[i]);
      }}
```

图的基本算法——深度优先搜索

0-1背包问题：

$$Q = \max_{(x_1, \ldots, x_n) \in X} \{ p_1 x_1 + p_2 x_2 + \ldots + p_n x_n \}$$

$$\text{其中} X = \{ (x_1, x_2, \ldots, x_n) \mid w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \leqslant C \}$$

例. N = 3时的0-1背包问题，考虑下面的具体例子。

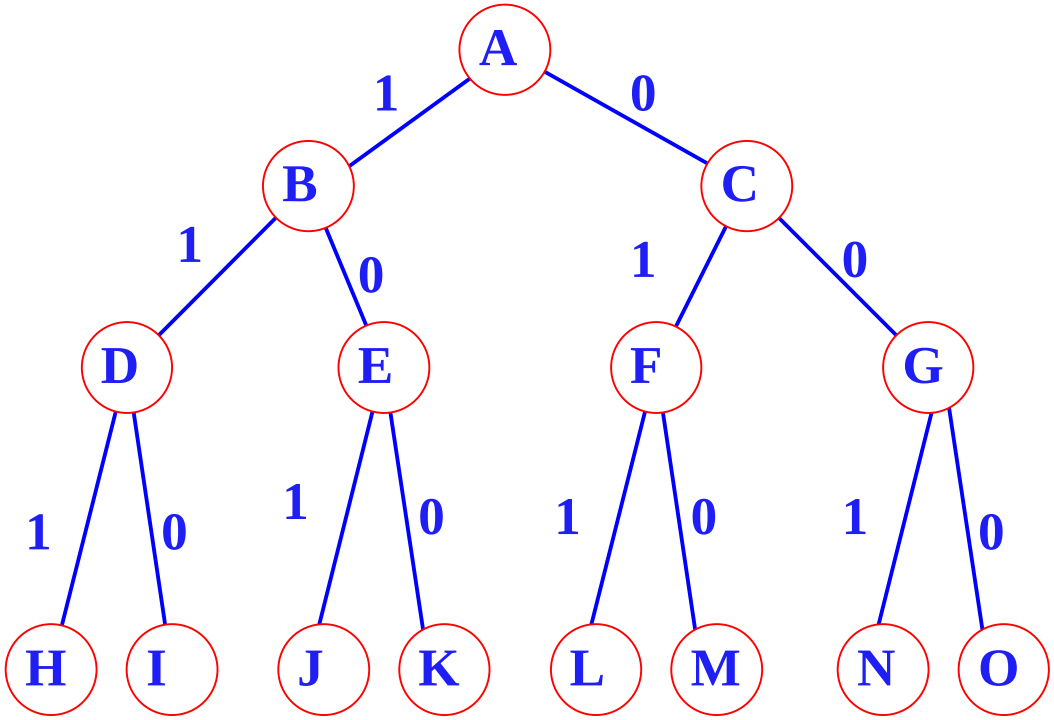
W = [16,15,15]，　P = [45,25,25]，　C = 30。

分析： 问题的解空间为{ (0,0,0)，(0,1,0)，(0,0,1)，(1,0,0)，(0,1,1)，(1,0,1)，(1,1,0)，(1,1,1) }

第22章 图的基本算法——深度优先搜索

问题参数： W = [16 , 15 , 15] , P = [45 , 25 , 25], C = 30 。

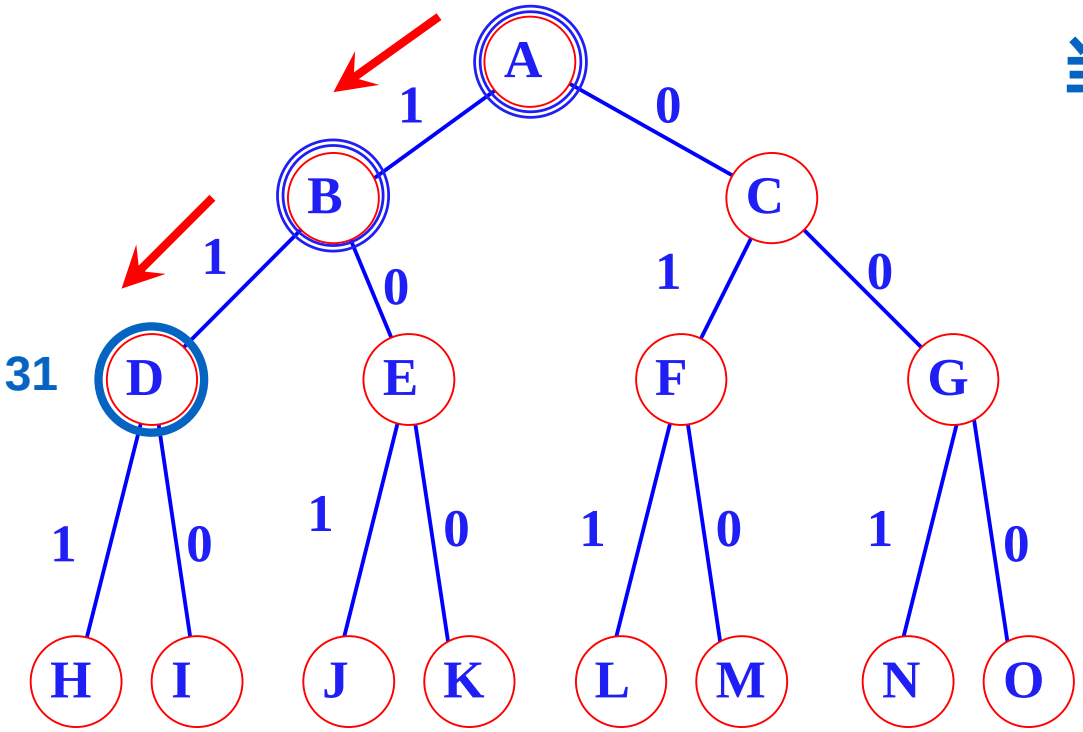
问题的解空间： { (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) }



图的基本算法——深度优先搜索

问题参数： W = [16 , 15 , 15] , P = [45 , 25 , 25], C = 30 。

问题的解空间： { (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) }



当前装载

70

节点堆栈

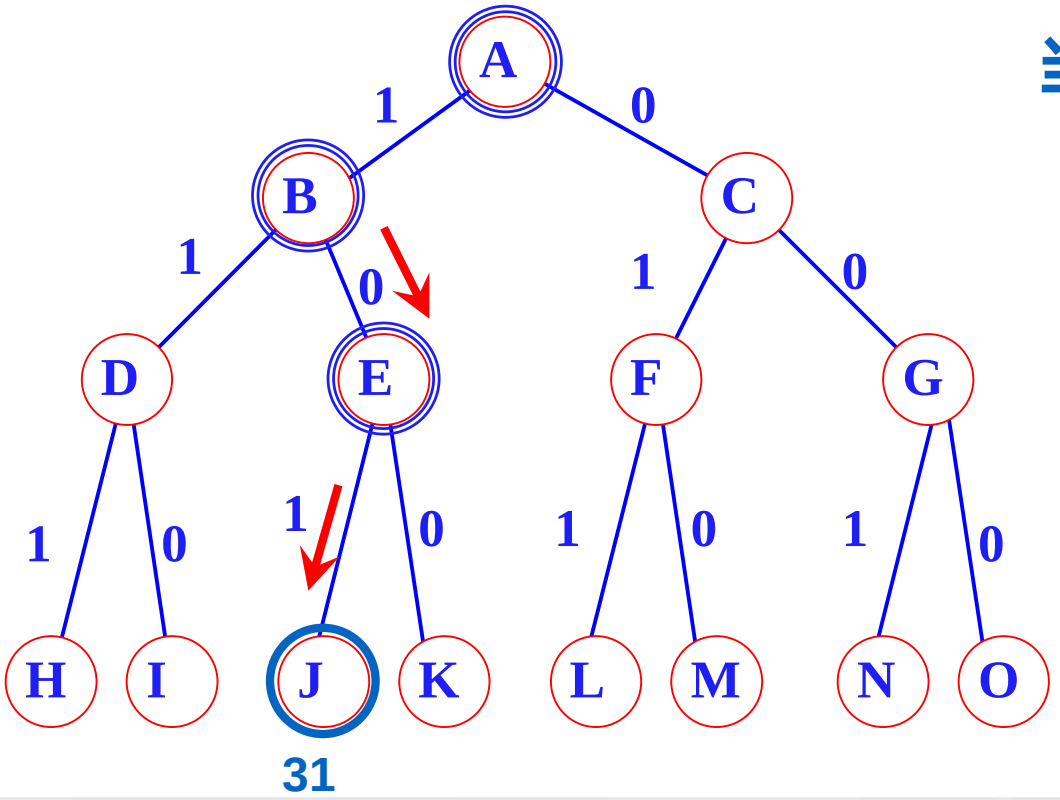
A

B

图的基本算法——深度优先搜索

问题参数： W = [16 , 15 , 15] , P = [45 , 25 , 25], C = 30 。

问题的解空间： { (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) }



当前装载

70

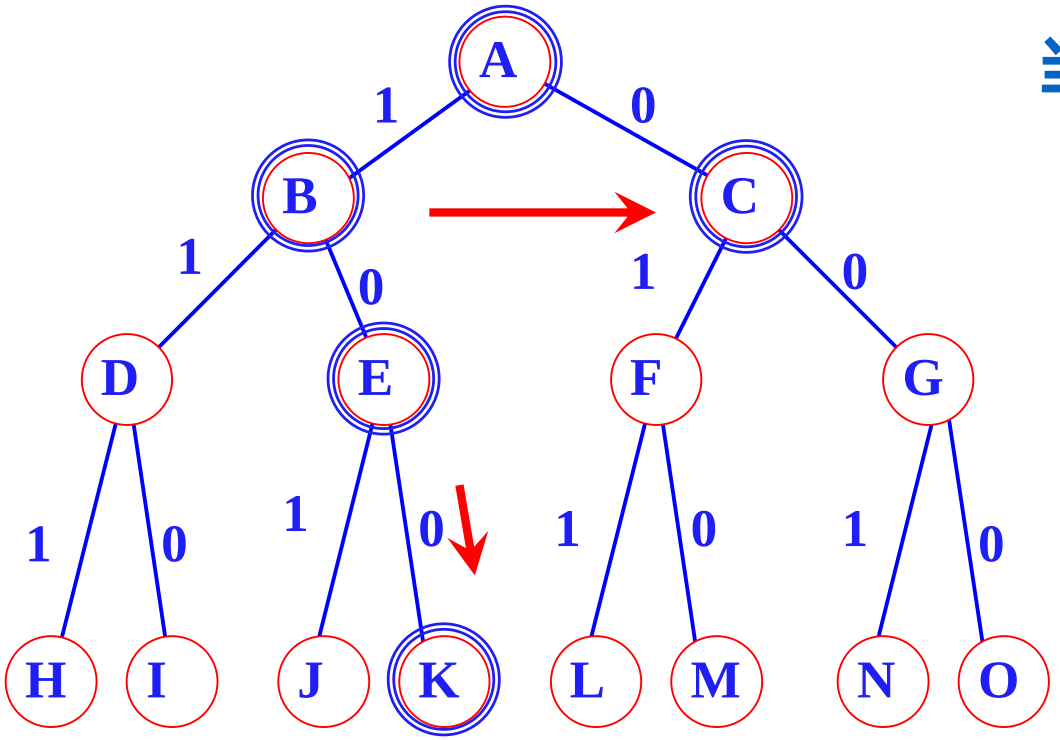
节点堆栈

A
B
E

图的基本算法——深度优先搜索

问题参数： W = [16 , 15 , 15] , P = [45 , 25 , 25], C = 30 。

问题的解空间： { (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) }



当前装载

45

节点堆栈

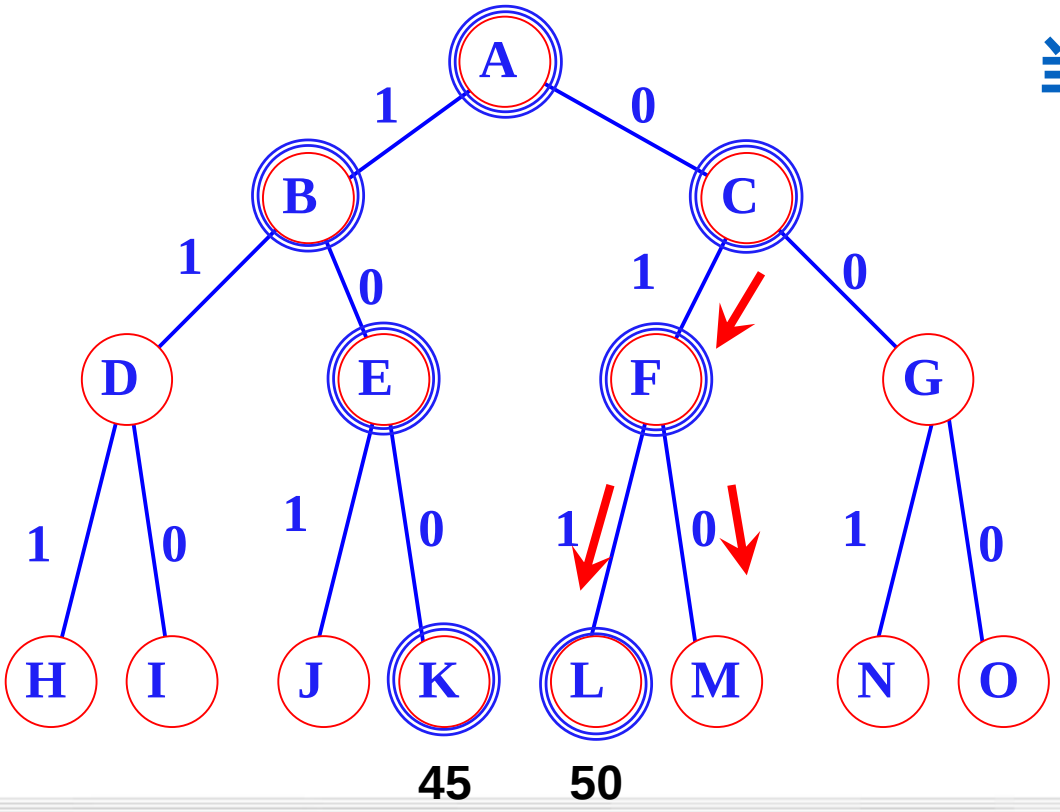
A

B

图的基本算法——深度优先搜索

问题参数： W = [16 , 15 , 15] , P = [45 , 25 , 25], C = 30 。

问题的解空间： { (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) }



当前装载

50

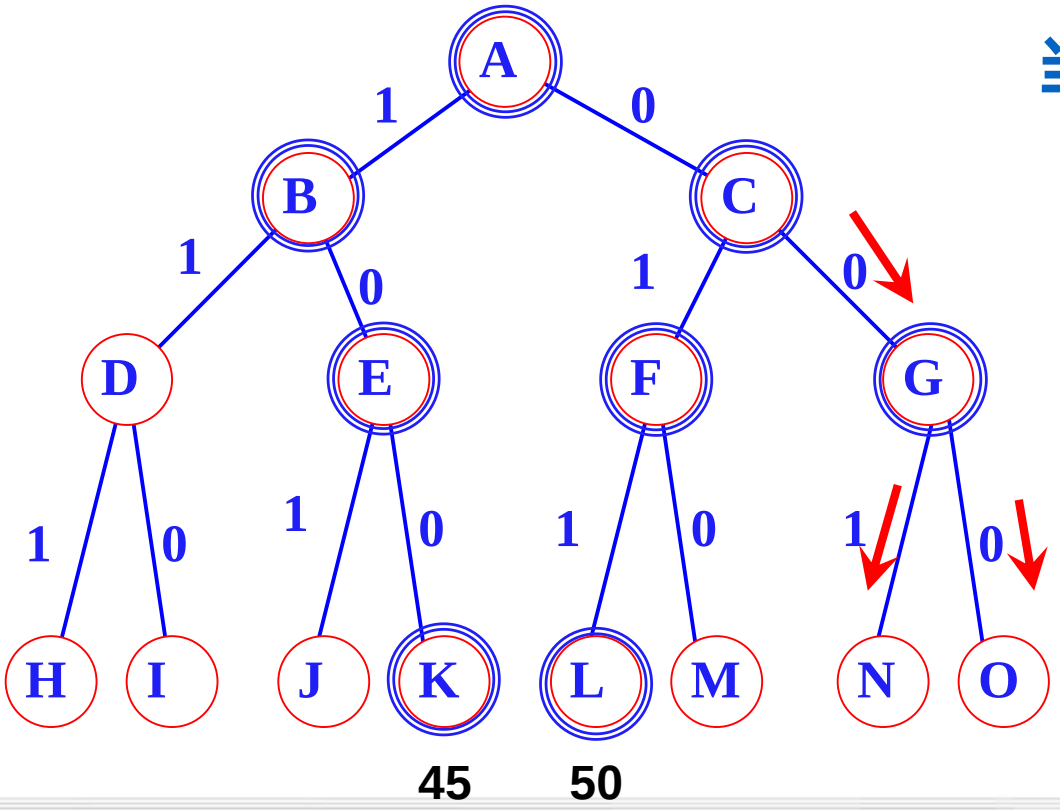
节点堆栈

A
C
F

图的基本算法——深度优先搜索

问题参数： W = [16 , 15 , 15] , P = [45 , 25 , 25], C = 30 。

问题的解空间： { (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) }



当前装载

25

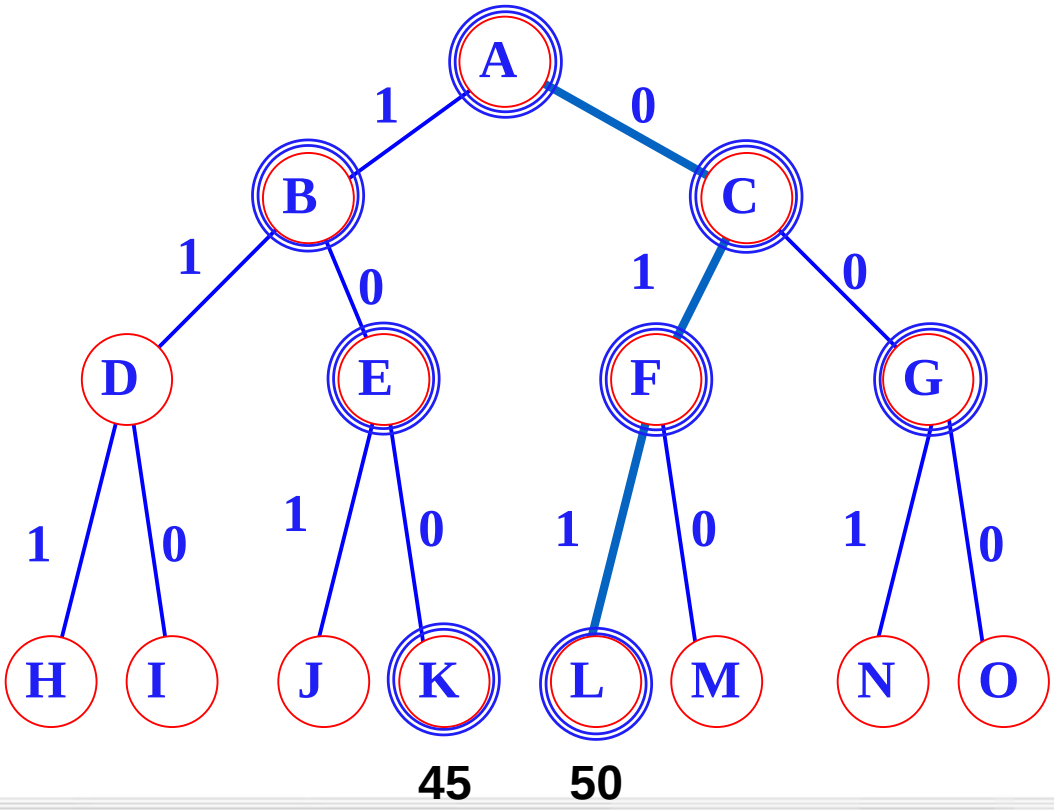
节点堆栈

C

图的基本算法——深度优先搜索

问题参数： W = [16 , 15 , 15] , P = [45 , 25 , 25], C = 30 。

问题的解空间： { (0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1) }

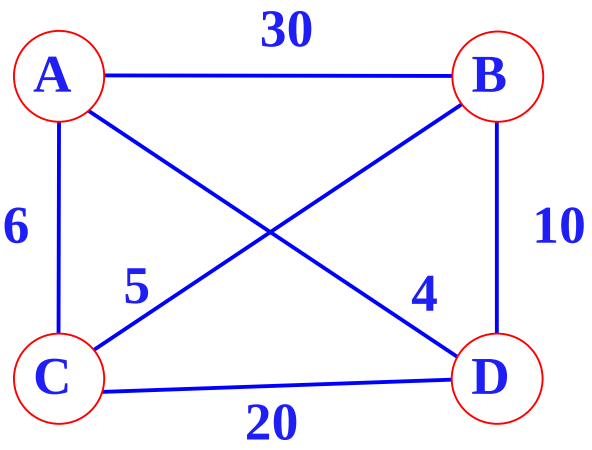


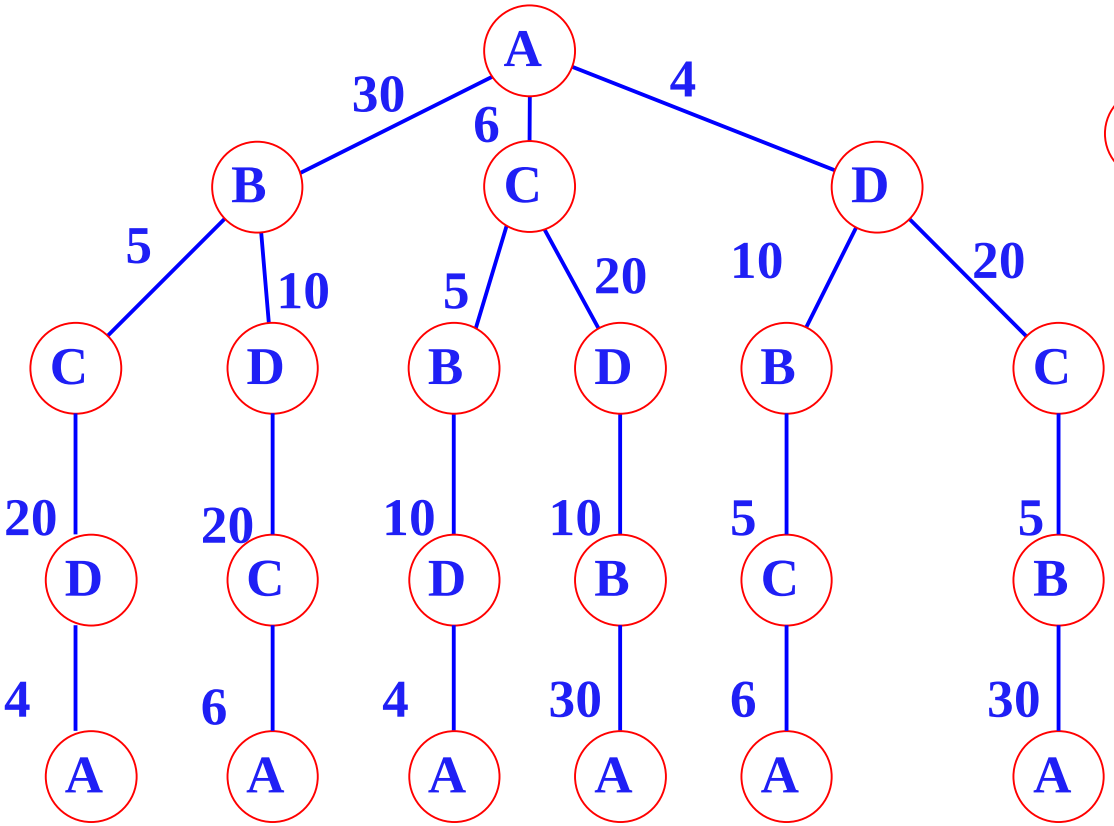
最优解为(0 , 1 , 1)

最大值为 50

图的基本算法——深度优先搜索

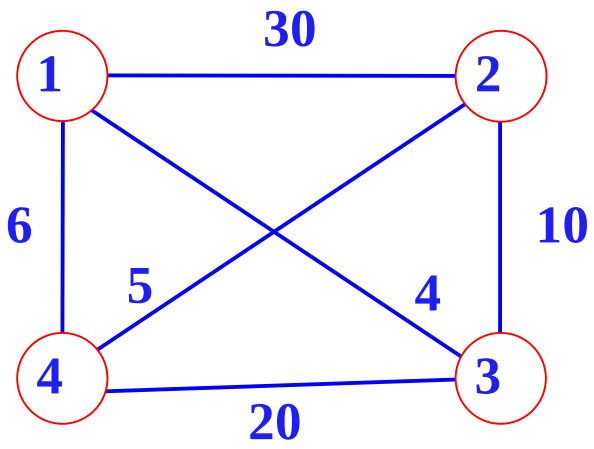
旅行售货员问题(TSP)分析：

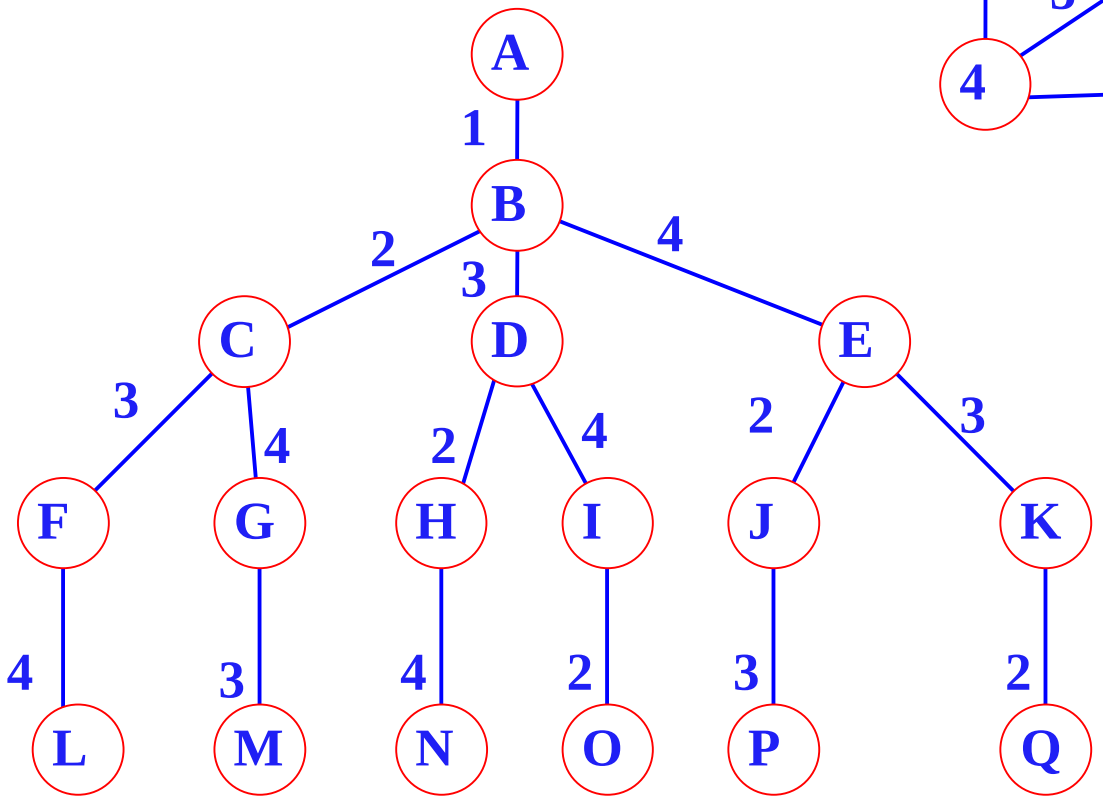




图的基本算法——深度优先搜索

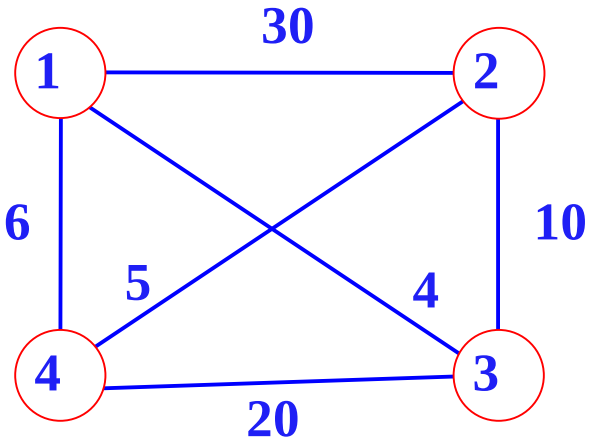
TSP问题分析：

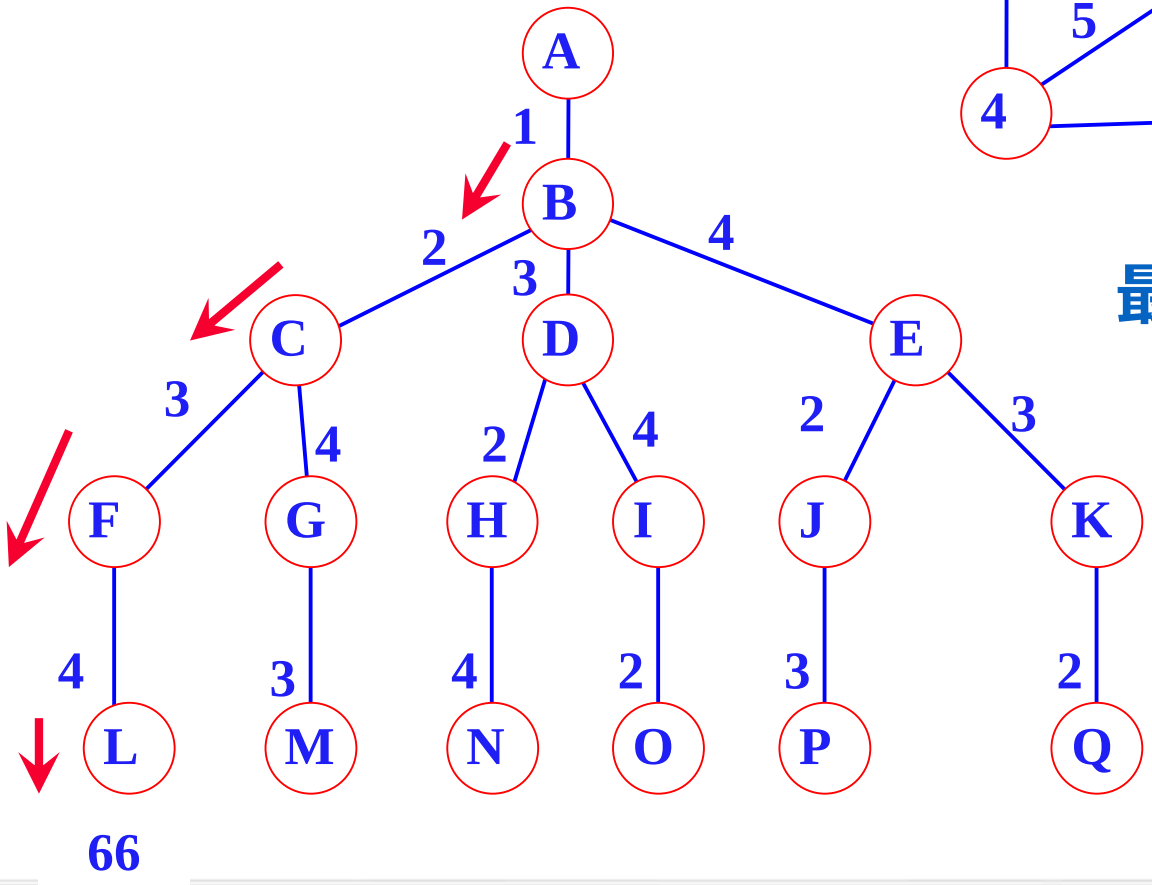




图的基本算法——深度优先搜索

TSP问题分析：





最短路径

66

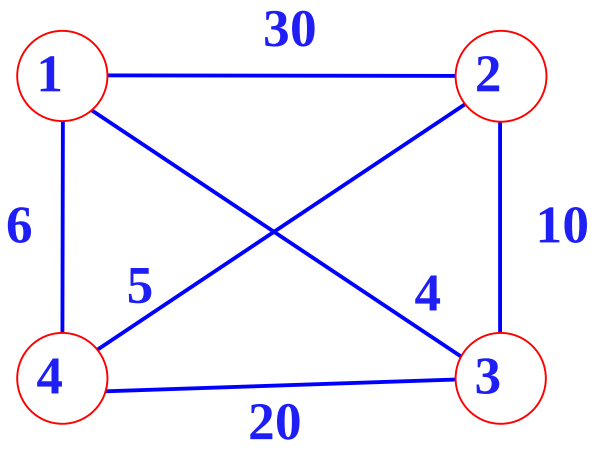
节点堆栈

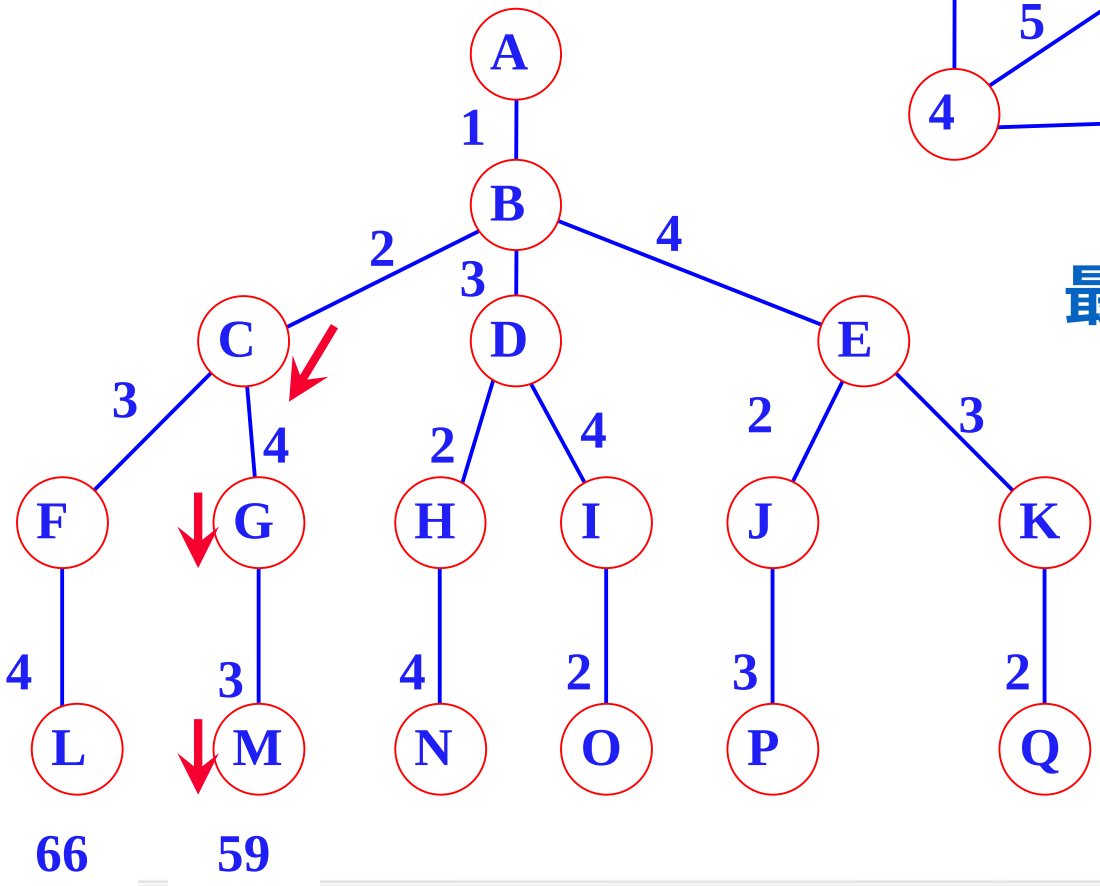
B

C

图的基本算法——深度优先搜索

TSP问题分析：





最短路径

59

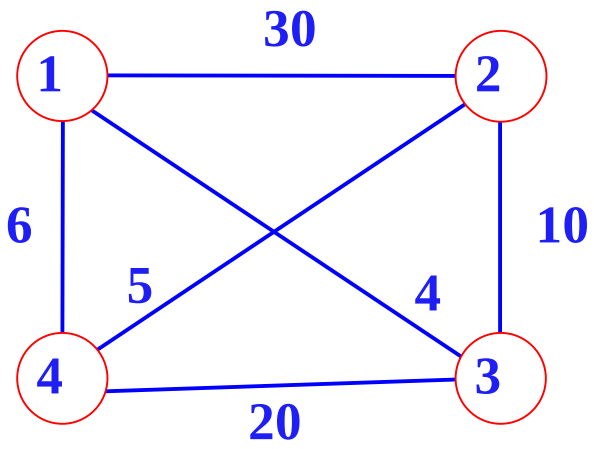
节点堆栈

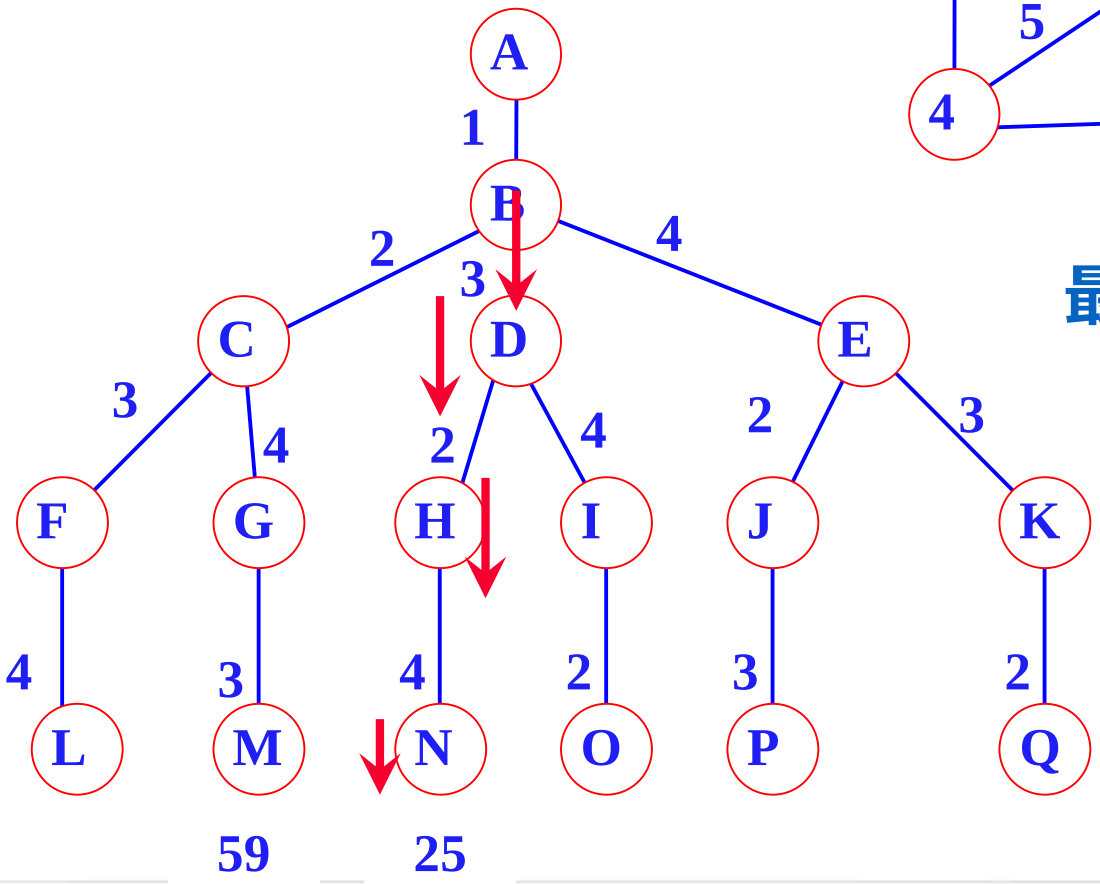
B

C

图的基本算法——深度优先搜索

TSP问题分析：

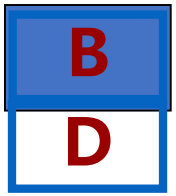




最短路径

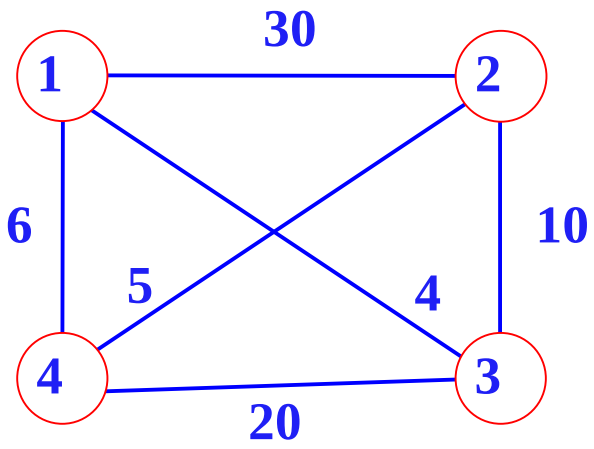
25

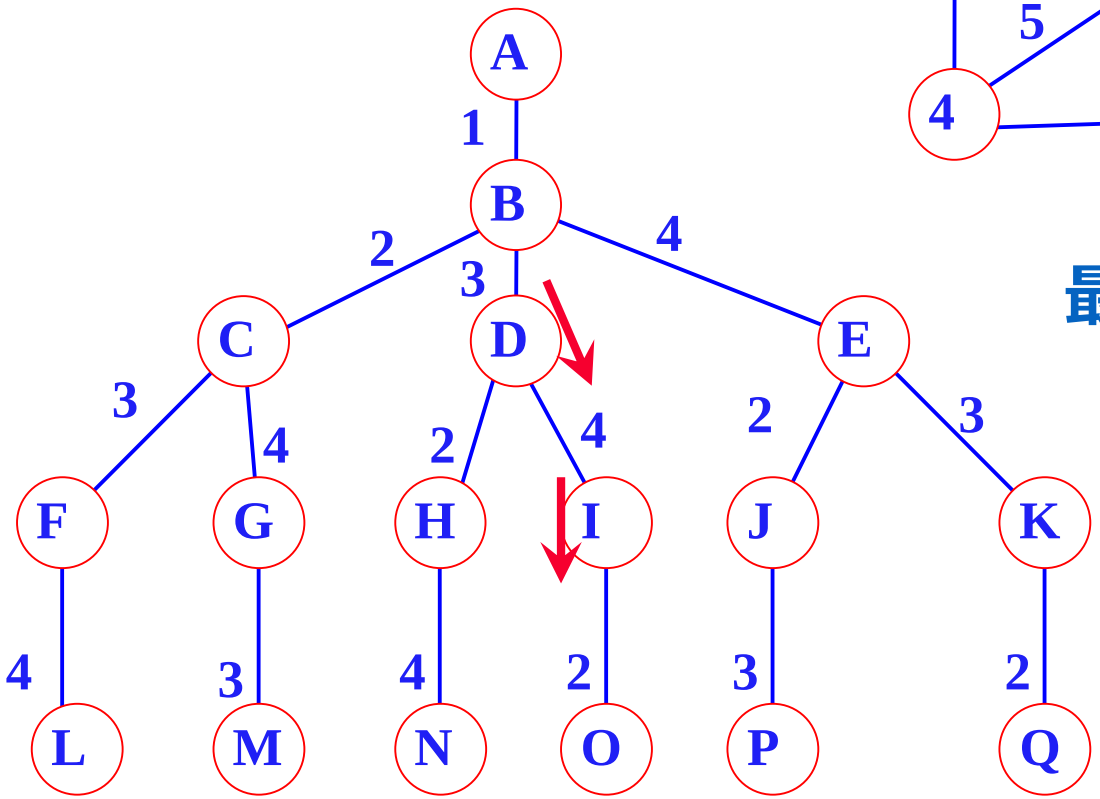
节点堆栈



图的基本算法——深度优先搜索

TSP问题分析：





最短路径

29

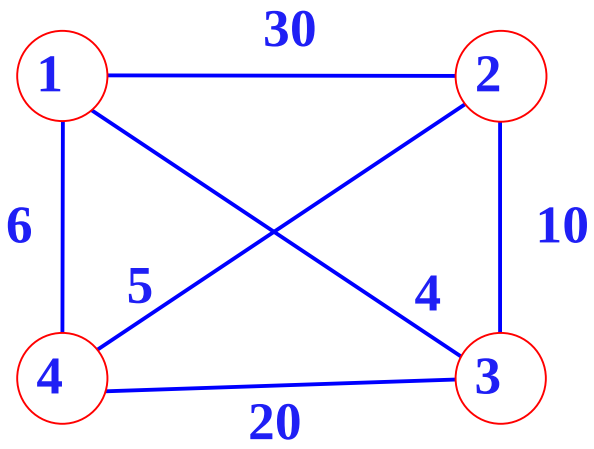
节点堆栈

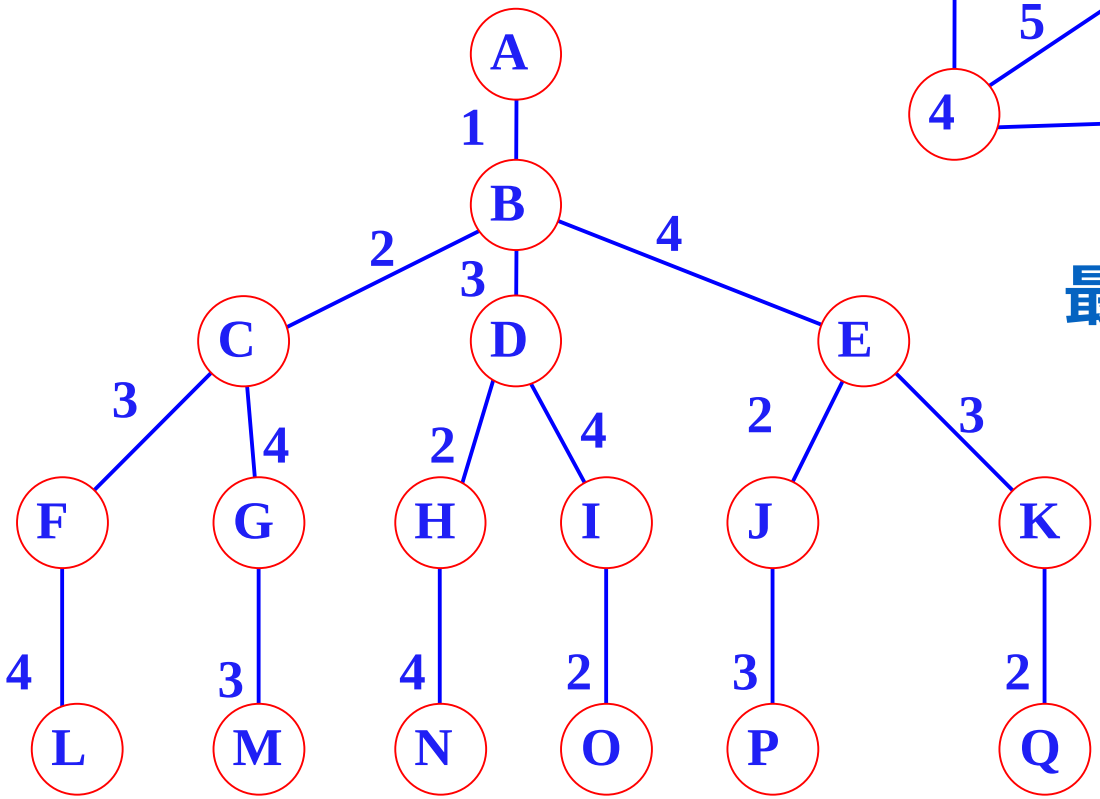
B

D

图的基本算法——深度优先搜索

TSP问题分析：





最短路径

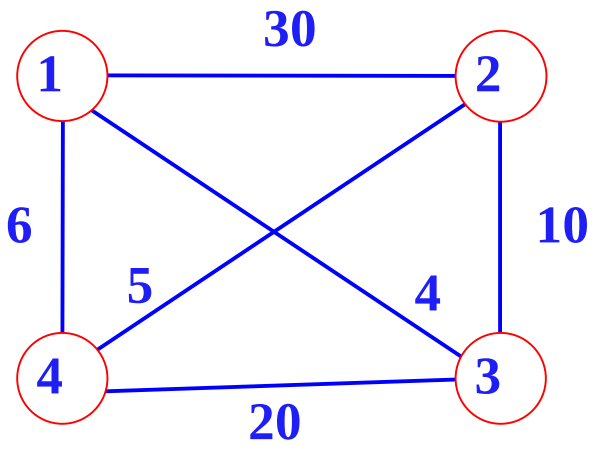
0

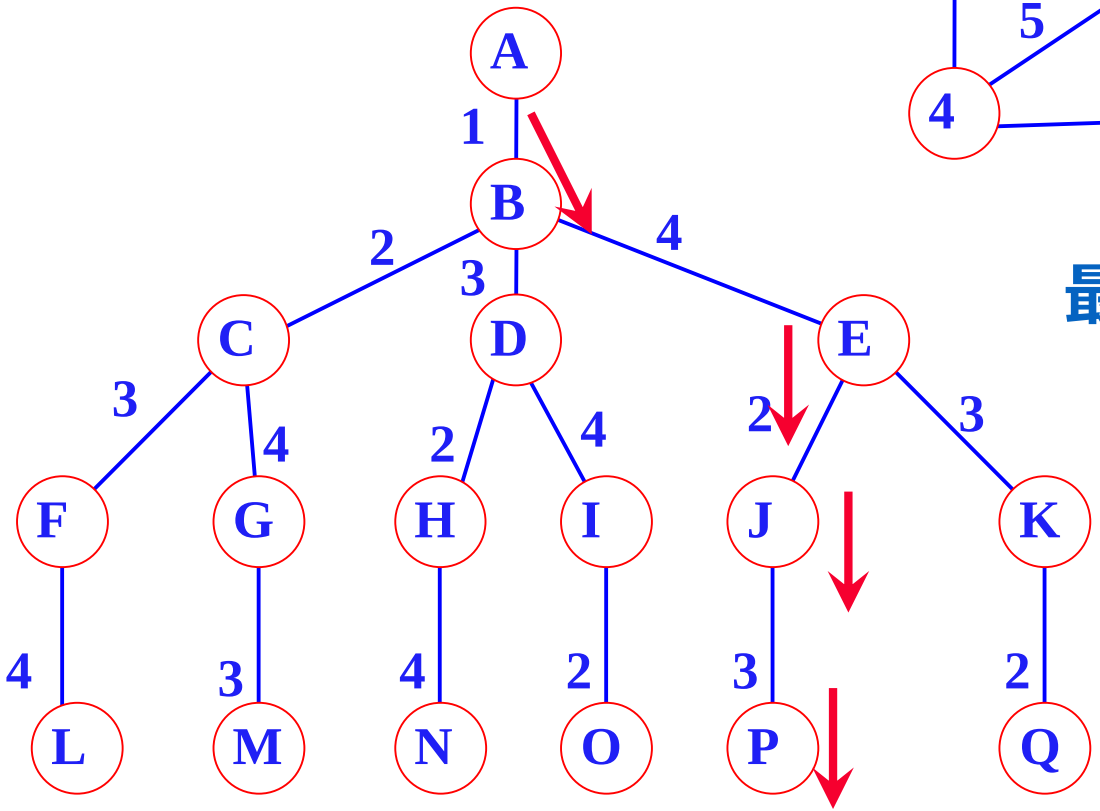
节点堆栈

B

图的基本算法——深度优先搜索

TSP问题分析：





最短路径

26

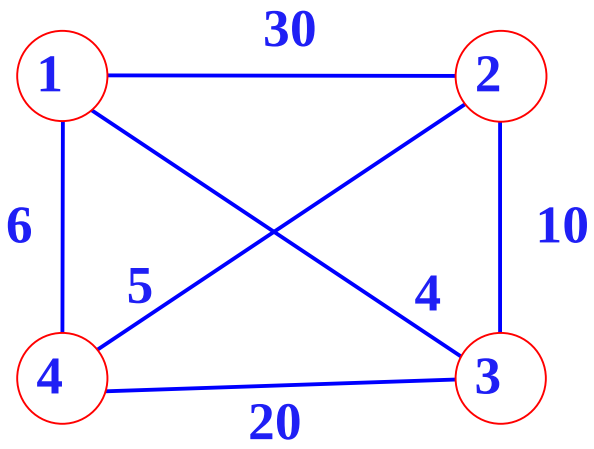
节点堆栈

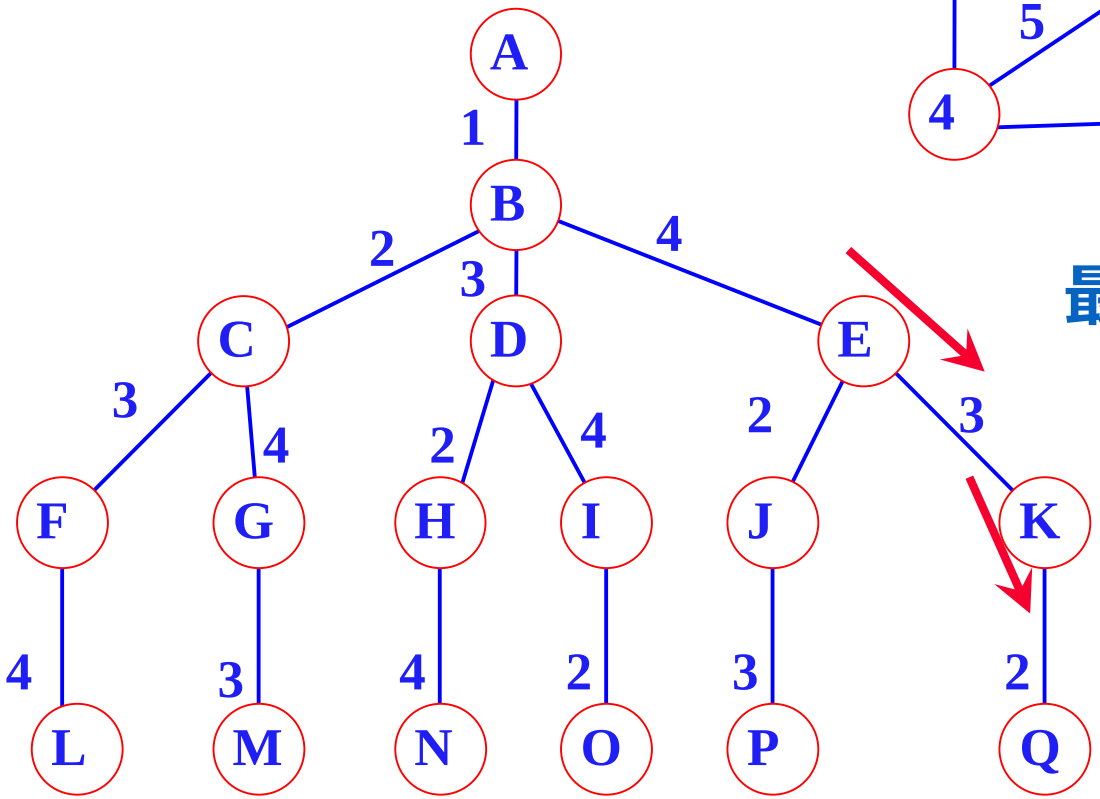
B

E

图的基本算法——深度优先搜索

TSP问题分析：





最短路径

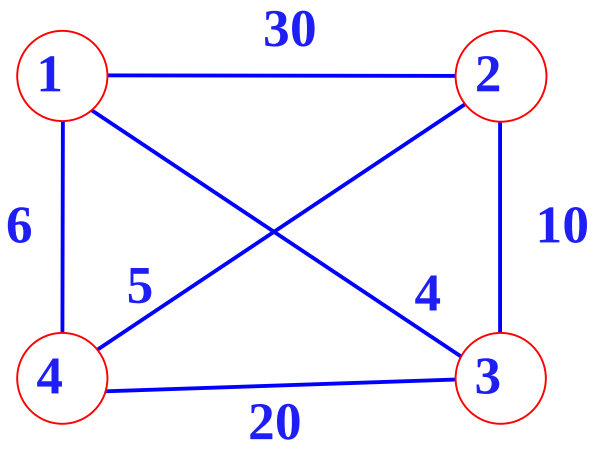
26

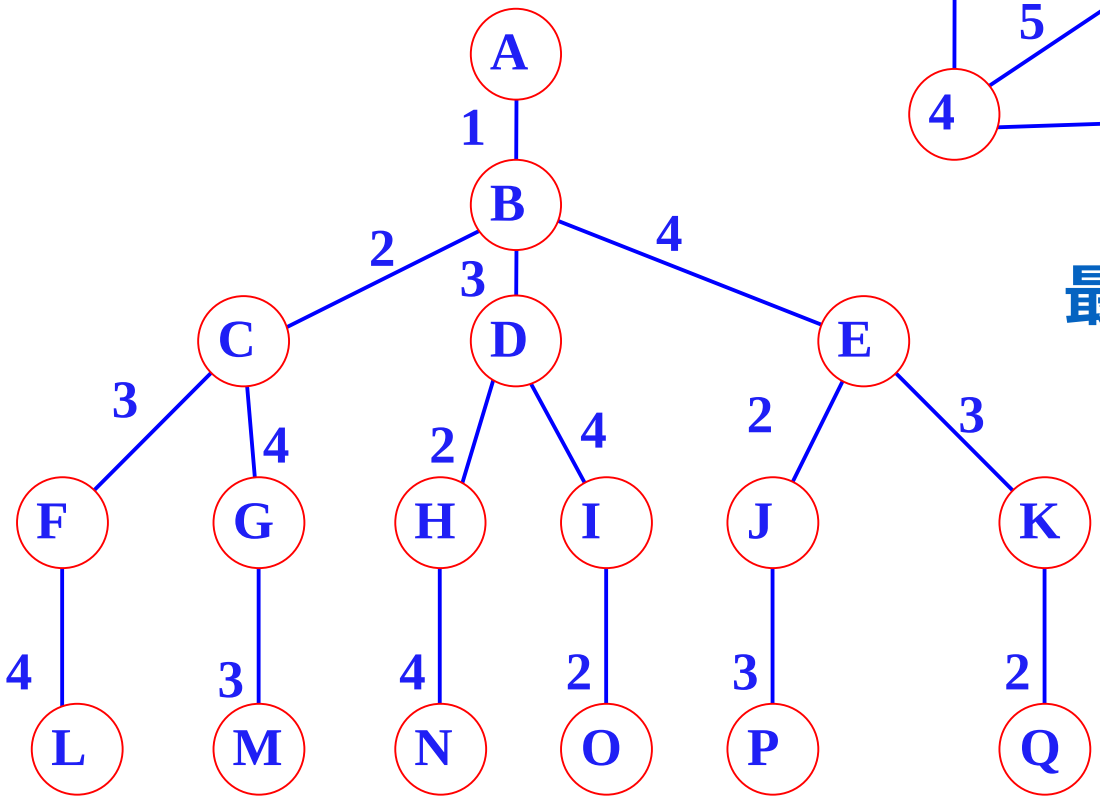
节点堆栈

E

图的基本算法——深度优先搜索

TSP问题分析：





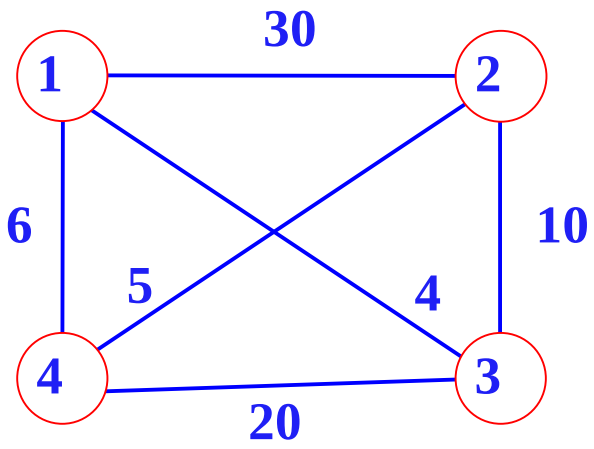
最短路径

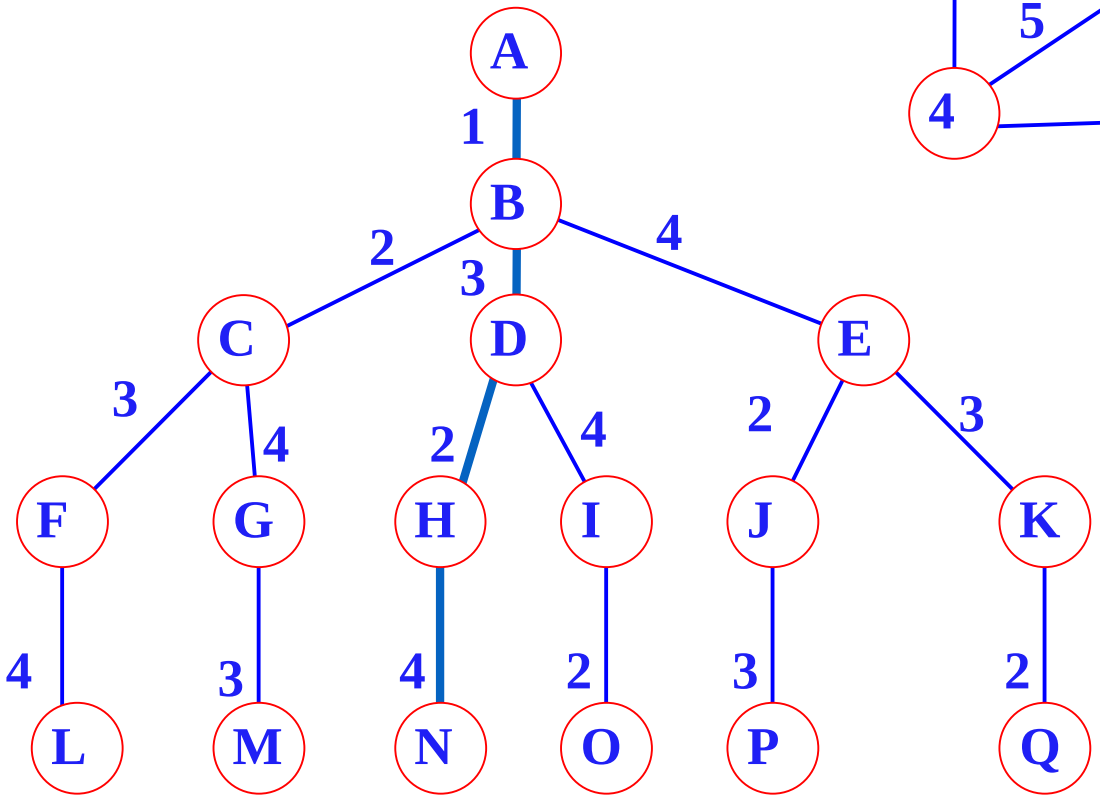
节点堆栈

空

图的基本算法——深度优先搜索

TSP问题分析：





最短路径

(1,3,2,4)

图的基本算法——深度优先搜索

回溯法的算法框架

回溯法通常包含一下三个步骤：

- ◆针对所给问题，定义问题的解空间。
- ◆确定易于搜索的解空间结构
- ◆以深度优先的方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

装载问题

有一批共n个集装箱要装上2艘载重量分别为c1和c2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这2艘轮船。如果有，找出一种装载方案。

容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1)首先将第一艘轮船尽可能装满；

(2)将剩余的集装箱装上第二艘轮船。

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近。由此可知，装载问题等价于以下特殊的0-1背包问题。

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

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

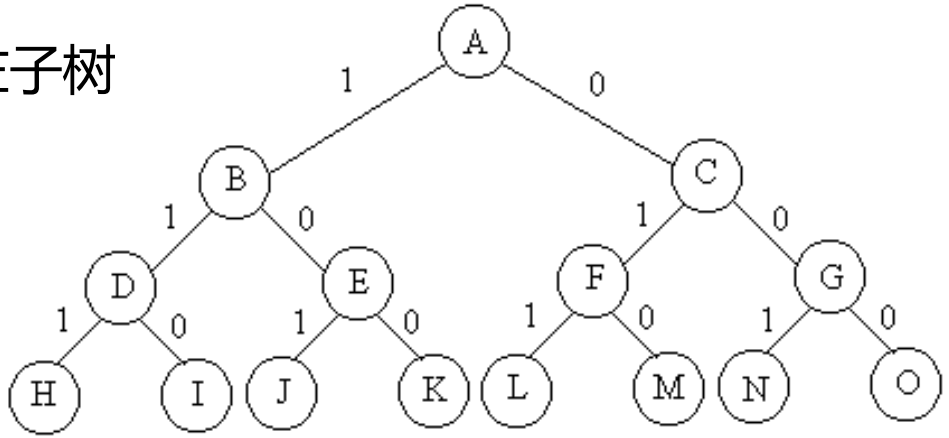
用回溯法设计解装载问题的$O(2^n)$计算时间算法。在某些情况下该算法优于动态规划算法。

装载问题

- 解空间：子集树
- 可行性约束函数(选择当前元素)：$\sum_{i=1}^{n} w_i x_i \le c_1$
- 上界函数(不选择当前元素)：

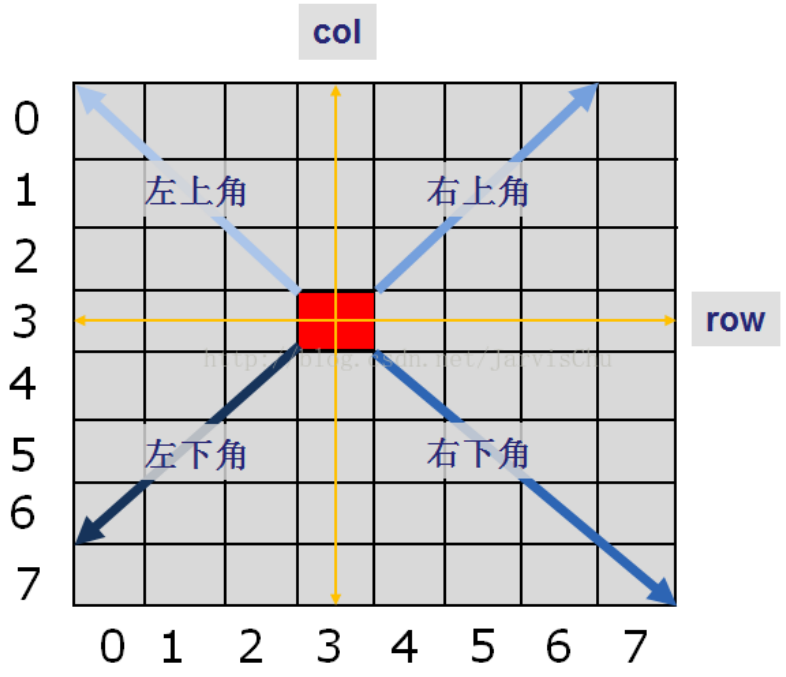
当前载重量cw+剩余集装箱的重量r≤当前最优载重量bestw

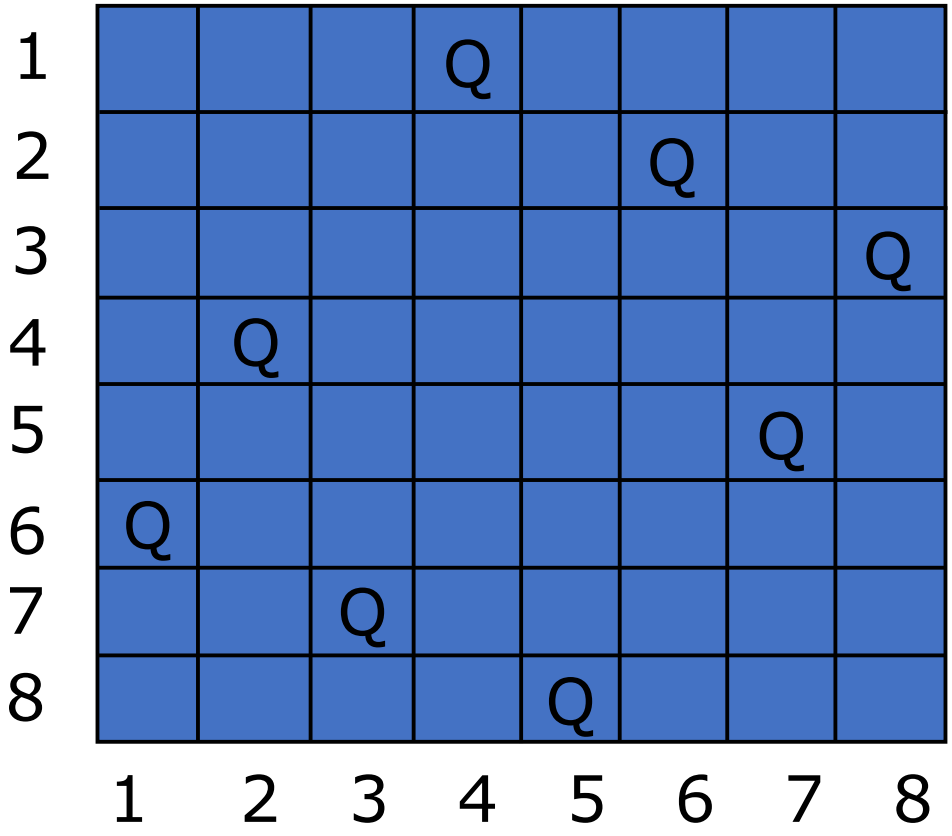
```
void backtrack (int i)
  {// 搜索第i层结点
    if (i > n)  // 到达叶结点
    更新最优解bestx,bestw;return;
    r -= w[i];
    if (cw + w[i] <= c) {// 搜索左子树
      x[i] = 1;
      cw += w[i];
      backtrack(i + 1);
      cw -= w[i];      }
    if (cw + r > bestw)  {
      x[i] = 0;  // 搜索右子树
      backtrack(i + 1);      }
    r += w[i];   }
```



n后问题

在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。





n后问题

- 解向量：$(x_1, x_2, \ldots, x_n)$
- 显约束：$x_i=1,2, \ldots ,n$
- 隐约束：
  1)不同列：$x_i \neq x_j$
  2)不处于同一正、反对角线：$|i-j| \neq |x_i-x_j|$

```
bool Queen::Place(int k)
{
  for (int j=1;j<k;j++)
    if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k])) return false;
  return true;
}

void Queen::Backtrack(int t)
{
  if (t>n) sum++;
    else
      for (int i=1;i<=n;i++) {
        x[t]=i;
        if (Place(t)) Backtrack(t+1);
      }
 }
```

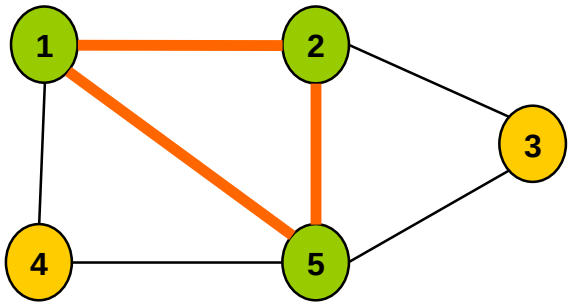
最大团问题（最大完全子图）

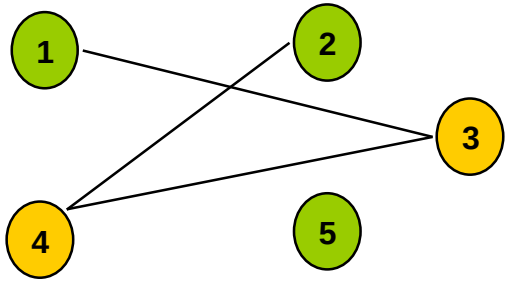
给定无向图G=(V，E)。如果U⊆V，且对任意u，v∈U有(u，v)∈E，则称U是G的完全子图。G的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中。G的最大团是指G中所含顶点数最多的团。

如果U⊆V且对任意u，v∈U有(u，v)∉E，则称U是G的空子图。G的空子图U是G的独立集当且仅当U不包含在G的更大的空子图中。G的最大独立集是G中所含顶点数最多的独立集。

对于任一无向图G=(V，E)其补图$\overline{G}$=(V1，E1)定义为：V1=V，且(u，v)∈E1当且仅当(u，v)∉E。

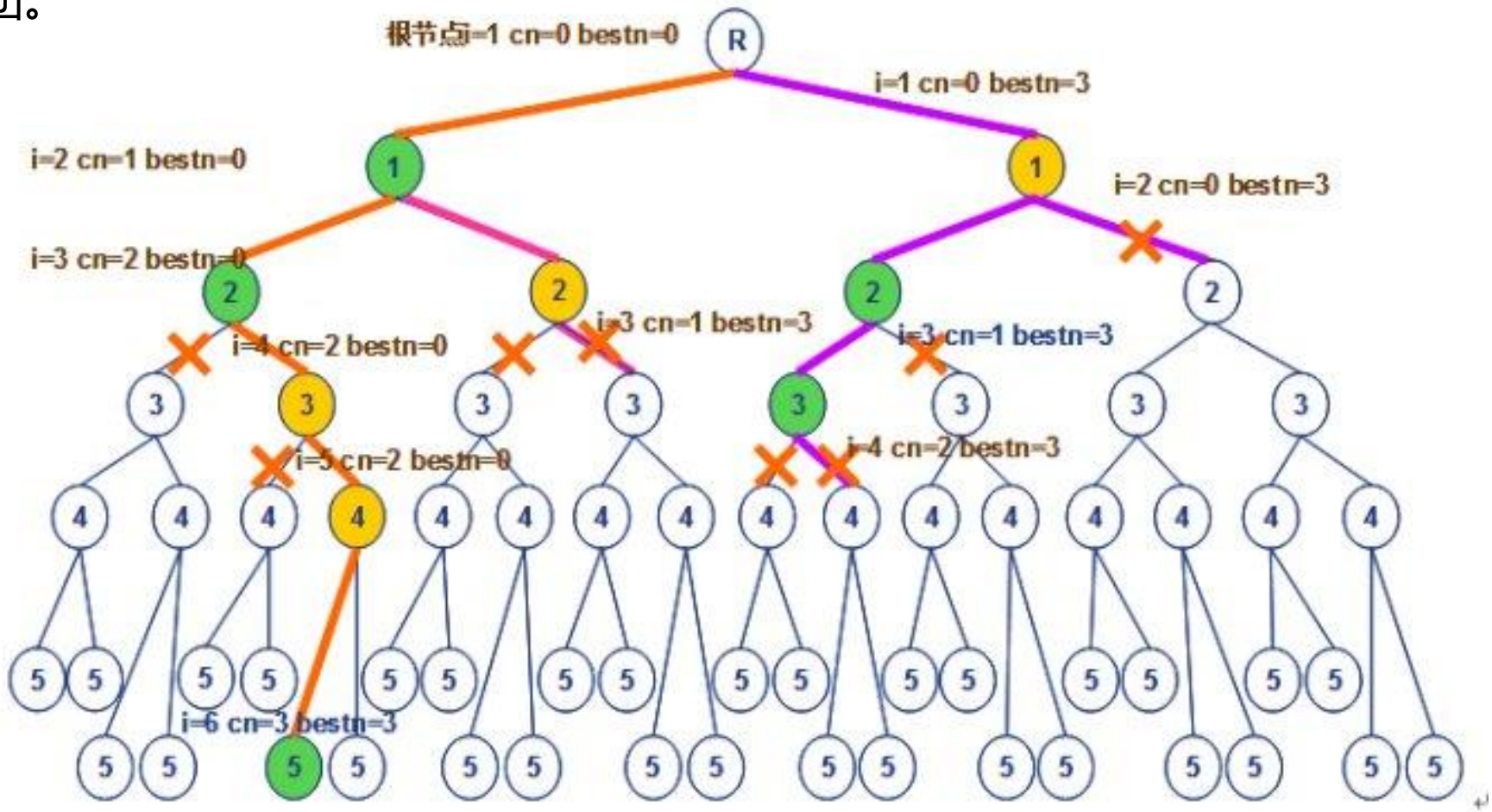
U是G的最大团当且仅当U是$\overline{G}$的最大独立集。





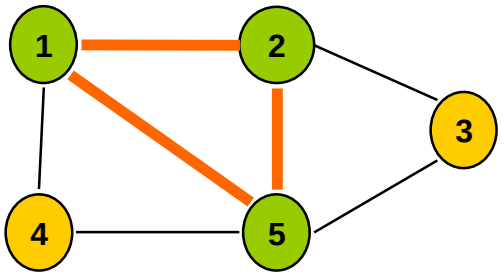
最大团问题

- 解空间：子集树
- 可行性约束函数：顶点i到已选入的顶点集中每一个顶点都有边相连。
- 上界函数：有足够多的可选择顶点使得算法有可能在右子树中找到更大的团。



最大团问题

```
void Clique::Backtrack(int i)
{// 计算最大团
   if (i > n) {// 到达叶结点
      for (int j = 1; j <= n; j++) bestx[j] = x[j];
      bestn = cn;   return;}
   // 检查顶点 i 与当前团的连接
   int OK
   for (int
      if (x[j
         // i
         OK
   if (OK) {// 进入左子树
      x[i] = 1;  cn++;
      Backtrack(i+1);
      x[i] = 0; cn--;}
   if (cn + n - i > bestn) {// 进入右子树
      x[i] = 0;
      Backtrack(i+1);}}
```



复杂度分析

最大团问题的回溯算法**backtrack**所需的计算时间显然为$O(n2^n)$。

进一步改进

- 选择合适的搜索顺序，可以使得上界函数更有效的发挥作用。例如在搜索之前可以将顶点按度从小到大排序。这在某种意义上相当于给回溯法加入了启发性。
- 定义Si={v_i,v_{i+1},…,v_n}，依次求出S_n,S_{n-1},…,S_1的解。从而得到一个更精确的上界函数，若cn+S_i<=max则剪枝。同时注意到：从S_{i+1}到S_i，如果找到一个更大的团，那么v_i必然属于找到的团，此时有S_i=S_{i+1}+1，否则S_i=S_{i+1}。因此只要max的值被更新过，就可以确定已经找到最大值，不必再往下搜索了。

回溯法的基本思想

(1)针对所给问题，定义问题的解空间；
(2)确定易于搜索的解空间结构；
(3)以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

常用剪枝函数：
用约束函数在扩展结点处剪去不满足约束的子树；
用限界函数剪去得不到最优解的子树。

用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为h(n)，则回溯法所需的计算空间通常为O(h(n))。而显式地存储整个解空间则需要$O(2^{h(n)})$或O(h(n)!)内存空间。

回溯法效率分析

通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：

(1)产生x[k]的时间；

(2)满足显约束的x[k]值的个数；

(3)计算约束函数**constraint**的时间；

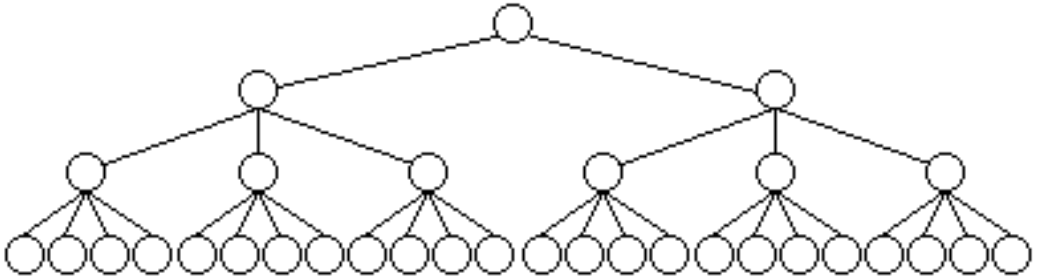
(4)计算上界函数**bound**的时间；

(5)满足约束函数和上界函数约束的所有x[k]的个数。

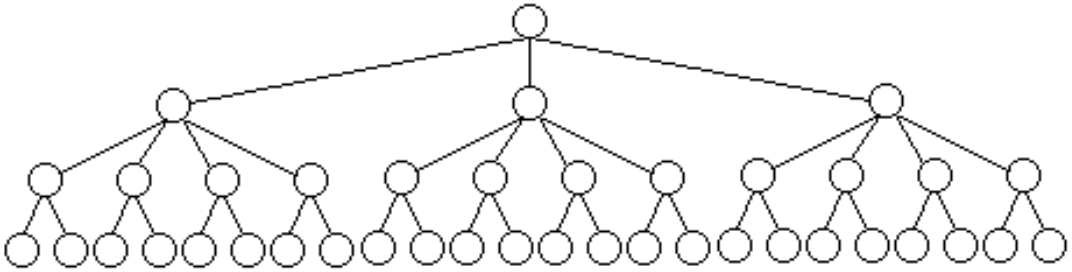
好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷

重排原理

对于许多问题而言，在搜索试探时选取x[i]的值顺序是任意的。**在其它条件相当的前提下，让可取值最少的x[i]优先**。从图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。



(a)



(b)

图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

图搜索——回溯法练习

- 通过应用范例学习回溯法的设计策略。
- （1）图的m着色问题
- （2）圆排列问题
- （3）电路板排列问题
- （4）连续邮资问题
- （5）批处理作业调度；
- （6）符号三角形问题