算法设计与分析

第5章 回溯法
（2）

谢晓芹
哈尔滨工程大学计算机科学与技术学院

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
  - 递归回溯
  - 迭代回溯
  - 子集树算法框架
  - 排列树算法框架
- 应用范例
  - 旅行售货员问题
  - 批处理作业调度
  - n后问题
  - 图的m着色问题

回溯法的基本思想

- 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。
- 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))。
  - 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。

子集树　　排列树

旅行售货员问题

- 给定n个顶点的带权图G=(V，E),图中各边的权为正数，图中的一条周游路线是包括V中的每个顶点在内的一条回路，一条周游路线的费用是这条路线上所有边的权之和。

哈密顿回路：经过图中所有顶点一次且仅一次的回路

- 旅行售货员问题(Traveling Salesperson)：必须访问n个城市，恰好访问每个城市一次，并最终回到出发城市。是要在图G中找出一条有最小费用的周游路线。此问题是NP完全问题。

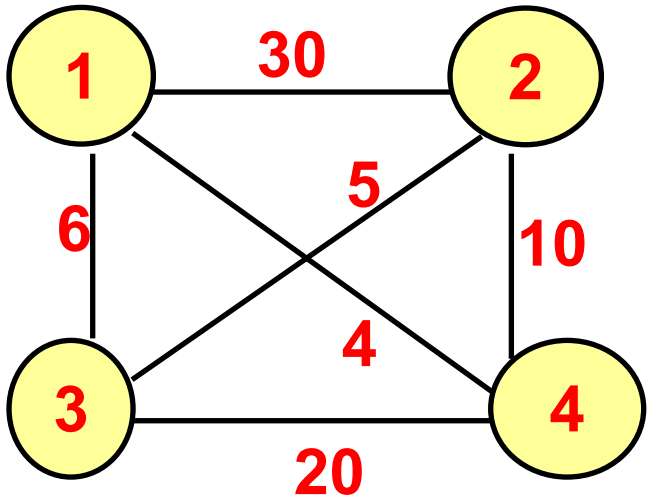
NP完全问题有一种令人惊奇的性质，即如果一个NP完全问题能在多项式时间内得到解决，那么NP中的每一个问题都可以在多项式时间内求解，即P=NP。目前还没有一个NP完全问题有多项式时间算法。

旅行售货员问题

- 形式化描述
  - 设G=(V，E)是一个带权图，图中各条边的耗费$c_{ij} > 0$。当$(i,j) \notin E$时，定义$c_{ij}=\infty$，设$|V|=n > 1$
  - 图中的一条周游路线是包括V中的每个节点在内的一条有向回路
  - 等价于求图的最短哈密尔顿回路问题
- 旅行售货员问题就是要在G中找出一条有最小耗费的周游路线
  - 输入：$G=(V,E)$, $c_{ij} > 0,(i,j) \in E$
  - 输出：从图中任一顶点v_i出发，经图中所有其他顶点一次且只有一次，最后回到同一顶点v_i的最短路径(即最小耗费的周游路线)

旅行售货员问题

- 实例: 如图所示, |V|=4



- 步骤1 定义解空间

如: 三条周游路径 (解空间)

[1,2,4,3,1]

[1,3,2,4,1]

[1,4,3,2,1]

……

旅行售货员问题

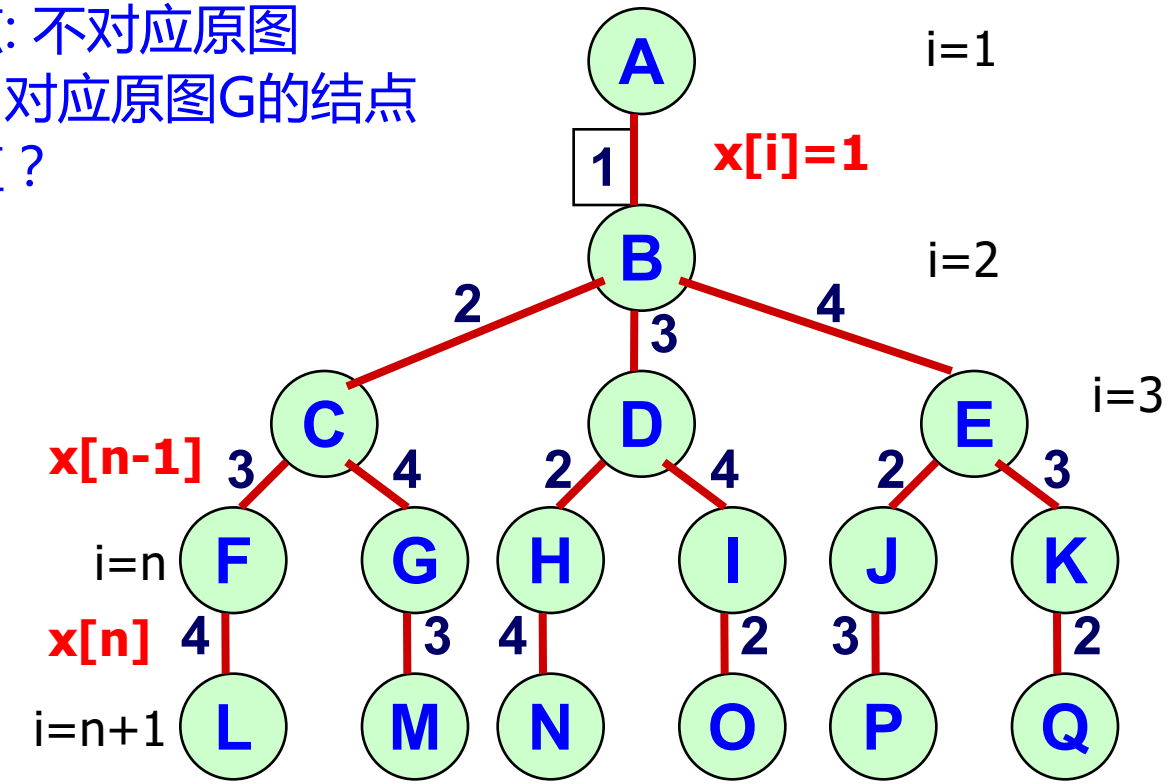
- 实例
- 步骤2 确定解空间结构

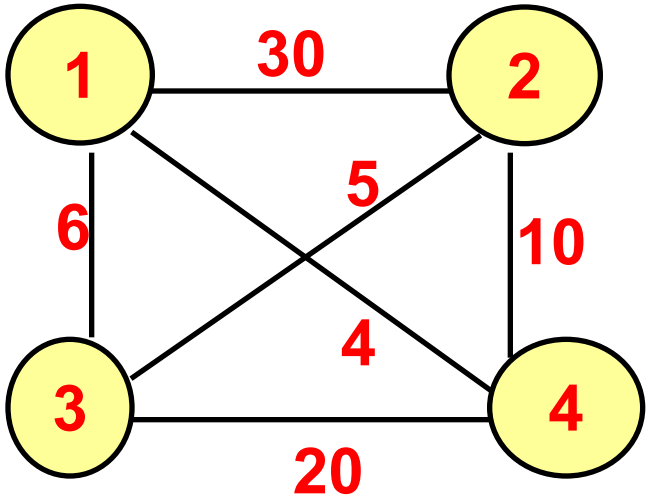
解空间树

结点: 不对应原图

边：对应原图G的结点

权值？





✓ **最优解：**

1 , 3 , 2 , 4 , 1

旅行售货员问题

- 定义（排列树）
  当所给的问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树
- 排列树通常有 n! 个叶结点
  遍历解空间树需要 Ω (n!)的计算时间
- 举例
  - 搜索空间为 (1, 2, 3,…,n-1,n),
    (2, 1, 3,…,n-1,n),
    ……
    (n, n-1,…,3,2,1)
  - 第1个元素有n种选择，第2个元素有n-1种选择,…,第n个元素有1种选择，共n!个状态

排列树的递归回溯实现框架

```
void backtrack (int t)        t:扩展结点编号
{
    if (t>n) output(x);
    else
        for (int i=t; i<=n; i++) {
            swap(x[t], x[i]);
            if (constraint(t)&&bound(t))
                backtrack(t+1);
            swap(x[t], x[i]);   //回溯
        }
}
```

t/n: 在当前扩展结点处未搜索过的子树的起始编号和终止编号

bound()=true 在当前扩展结点处x[1:t]取值未使目标函数越界，否则剪枝

constraint()=true 在当前扩展结点处x[1:t]取值满足约束条件，否则剪枝

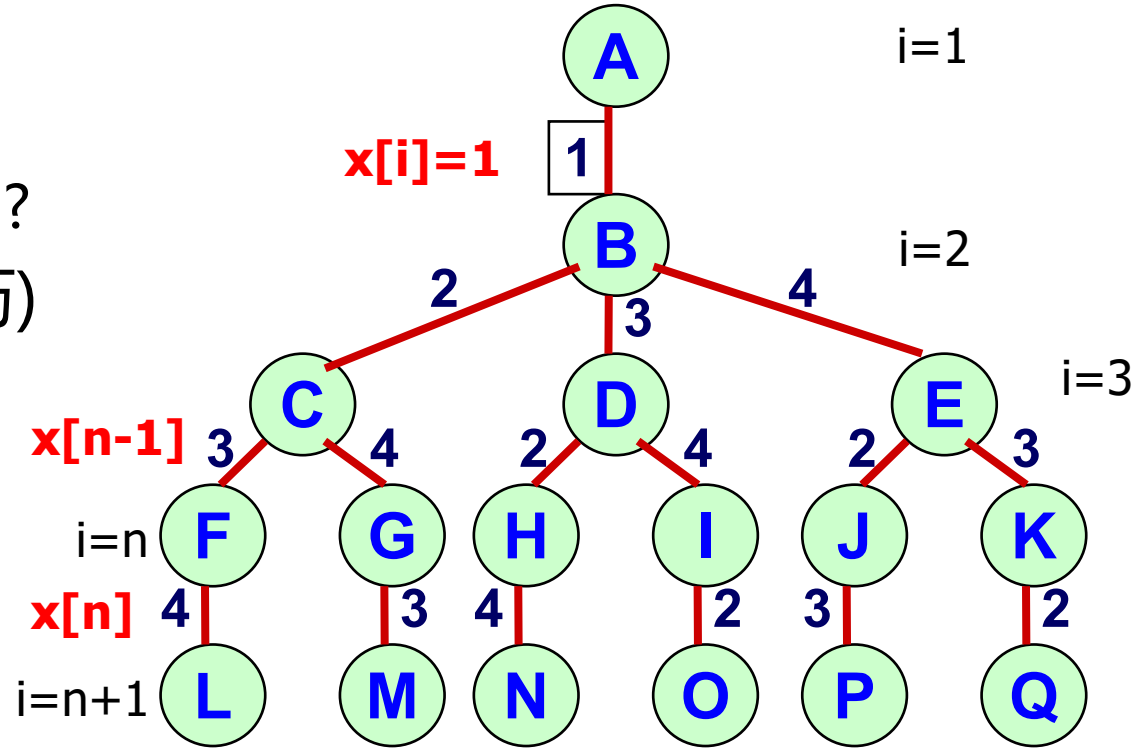
旅行售货员问题

- 步骤3 深度优先搜索遍历过程+ 剪枝
  - 原问题： backtrack(2)
  - 1. 何时找到解？
    - 层数 i=n时, x[n-1]到x[n]有边, 且x[n]到x[1]有边,则有解
  - 2. 解是否最优？
    - 回路费用是否优于当前最优费用bestC
    - cc+a[x[n-1], x[n]]+a[x[n],[1]]< bestC ?
  - 3. 在中间过程如何处理 （ i<n ） (遍历)
    - 判断是否可能为解？
      - x[i-1]到x[i] 是否有边？ 无边则不是解
    - 是否可能得到最优解？
      - x[1..i]费用<bestC，可能, 否则剪枝

剪枝

cc+a[x[i-1]x[i]]<bestC

Eg: n=4



旅行售货员问题

```
void Traveling<Type>::Backtrack(int i) {
   if (i == n) {
      if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
                                 (cc+a[x[n-1]][x[n]]+a[x[n]][1]<bestc ||bestc == NoEdge)) {
         for (int j=1; j <= n; j++)
            bestx[j] = x[j];
         bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];}
      }
      else {
         for (int j=i; j <= n; j++) {
            // 是否可进入x[j]子树
            if (a[x[i-1]][x[j]] != NoEdge && (cc+a[x[i-1]][x[j]]<bestc || bestc == NoEdge)) {
               // 搜索子树
               Swap(x[i], x[j]);
               cc += a[x[i-1]][x[i]];
               Backtrack(i+1);
               cc -= a[x[i-1]][x[i]];
               Swap(x[i], x[j]);
            }
         }
      }
   }
}
```

是否为解？

是否为解？

是否能得到最优解

1.何时找到解

2 是否最优解？

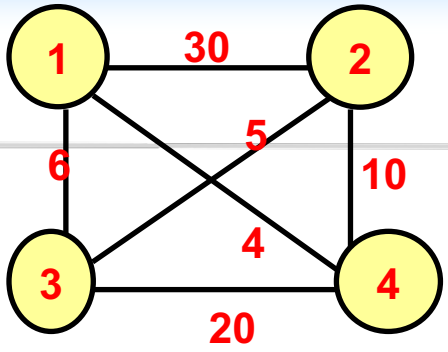
保存最优解
更新最优值

3 中间过程

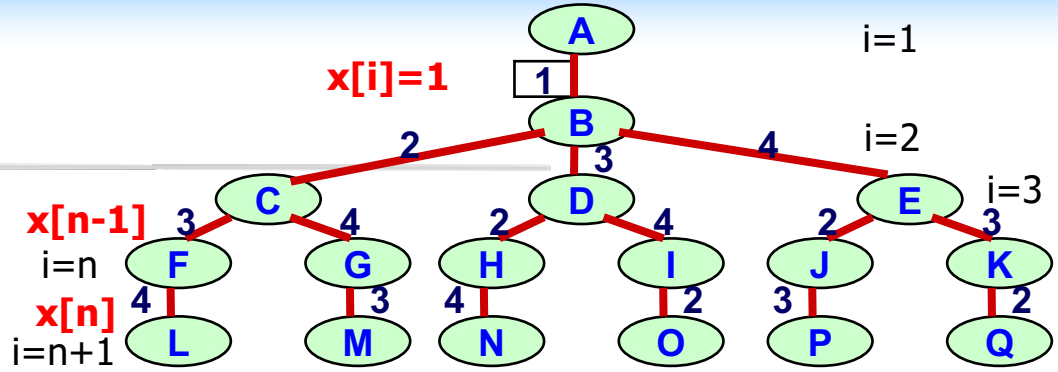
深度搜索

旅行售货员问题

- 回溯法搜索过程



n=4



活结点表	扩展结点	死结点	处理过程	当前代价
A	A		A→B	BestC=noEdge
AB	B		B→C,D,E	Cc=0
ABC	C		C→F,G	
ABCF	F		F→L, i=4时要判是否为解	bestC=59
ABC	(bt)C		C→G	
ABCG	G		G→M, i=4时要判是否为解	
ABC	C	C		
AB	(bt)B		B→D	
ABD	D		D→H,I	
ABDH	H	H	H→N, , i=4时要判是否为解	bestC=25
ABD	(bt)D		(续见下页)	

C: a[1,2]有边

F: a[2,3]有边

F: a[3,4]有边, (4,1)有边, bestC=59, **可行解(1,2,3,4,1)**

G: a[2,4]有边, 1,2,4 = 40<59 ✓

G: a[4,3]有边, (3,1)有边, 40+20+6=66>59 X

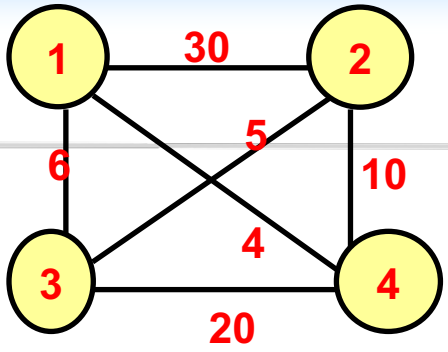
D: a[1,3]有边, 6<59 ✓

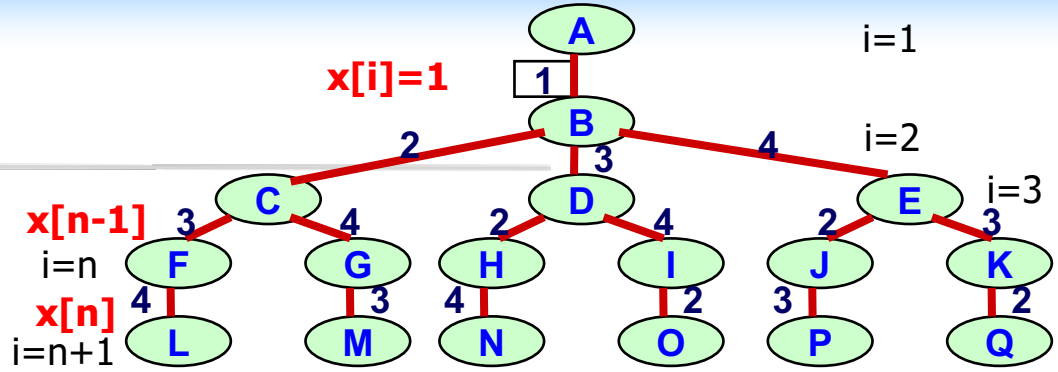
H: a[3,2]有边, 6+5=11<59 ✓

H: a[2,4]有边, (4,1)有边, 11+(2,4)+(4,1)=25<59 bestC = 25, **(1,3,2,4,1)**

旅行售货员问题

- 回溯法搜索过程





活结点表	扩展结点	死结点	处理过程	当前代价
ABD	(bt)D		D→I	
AB	(bt)B		B→E	
ABE	E		E→J,K	
ABEJ	J	J	J→P, i=4时要判是否为解	
ABE	(bt)E		E→K	
ABEK	K	K	K→Q, i=4时要判是否为解	
ABE	(bt)E	E		
AB	(bt)B	B		
A	(bt)A	A		
空	结束			

I: a[3,4]有边, 6+20=26>25 X

E: a[1,4]有边, 4<25 ✓

J: a[4,2]有边, 4+(4,2)=4+10=14<25 ✓

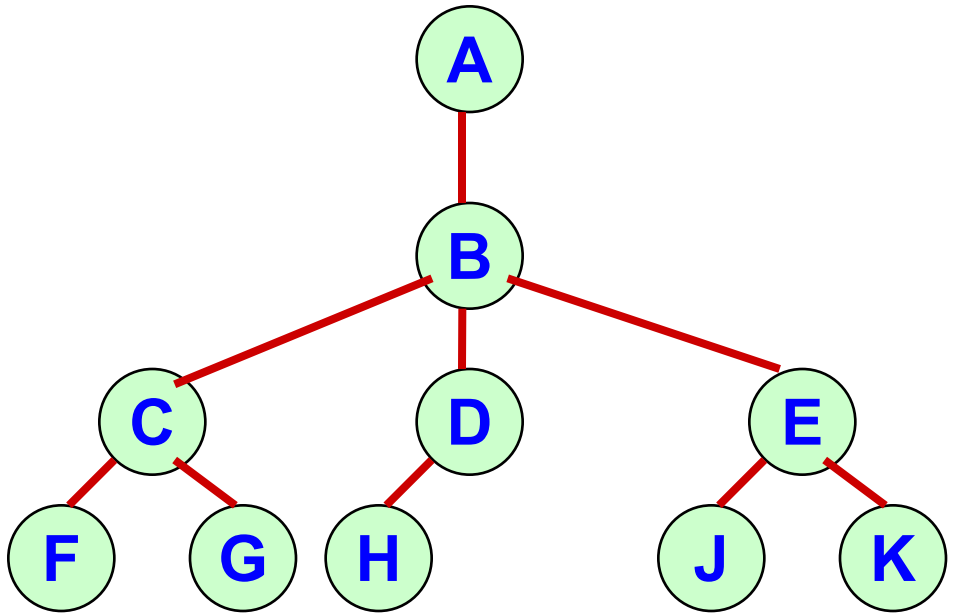
J: (2,3)和(3,1)有边,14+(2,3)+(3,1)=25=25
不更新bestC

E: a[4,3]有边,4+20=24<25 ✓

K: (3,2)和(2,1)有边,
24+(3,2)+(2,1)=59>25 X

旅行售货员问题

- 回溯法搜索过程中的扩展结点表顺序为
  - ABCFCGCBDHDBEJEKEBA
- 实际生成的树为：



旅行售货员问题

- 算法分析
  - Backtrack()：最坏情况下需要O((n-1)!)
  - 每次更新bestx：需要O(n)

从根到叶子的扫描过程，路径最长为n

  - Time=O(n!)

装载问题

- 有一批共n个集装箱要装上2艘载重量分别为c_1和c_2的轮船，其中集装箱i的重量为w_i，且$\sum_{i=1}^{n} w_i \leq c_1 + c_2$
- 装载问题
  - 确定是否有一个合理的装载方案可将这n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

装载问题

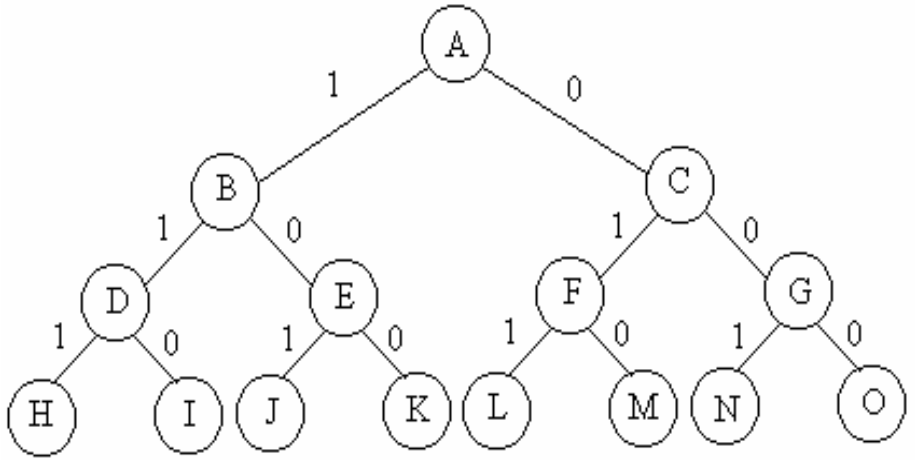
- 转换问题
  - 将第一艘轮船尽量装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近第一艘轮船的载重量。
- 由此可知，装载问题等价于以下特殊的0-1背包问题。

$$\max \sum_{i=1}^{n} w_i x_i \ , \quad \sum_{i=1}^{n} w_i x_i \le c_1 \ , \quad x_i \in \{0,1\}, \ 1 \le i \le n$$

- 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
- 解空间树： 子集树



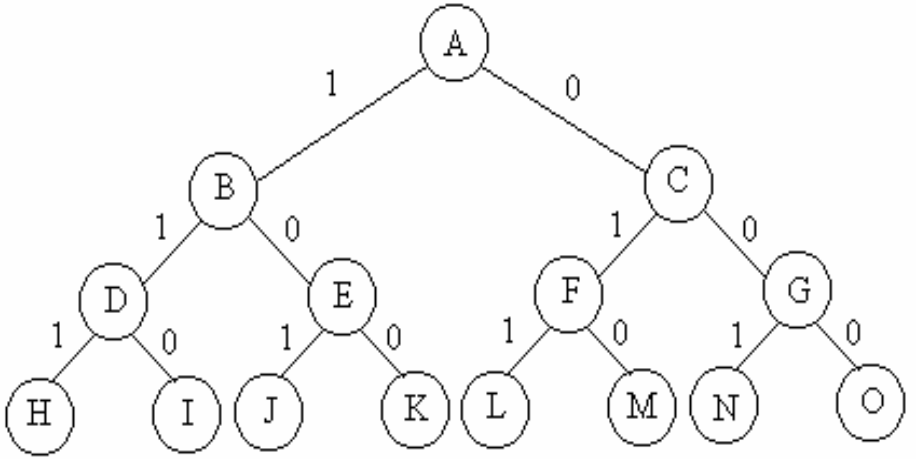
- 剪枝函数
  - 可行性约束函数(选择当前元素)： $\sum_{i=1}^{n} w_i x_i \leq c_1$ (约束条件剪去“不可行解”的子树)
  - 上界函数(不选择当前元素)： (限界条件剪去不含最优解的子树)
    - 当前载重量cw + 剩余集装箱的重量r≤ 当前最优载重量bestw
    - 保证算法搜索到的每个叶结点都是迄今为止找到的最优解

当前方案装的太少了

设Z是当前扩展结点, 以Z为根的子树中任一叶子结点的载重量不超过cw+r

装载问题

- 遍历过程
  - 何时找到解？ i>n
    - 叶子结点
    - 是否最优： 满足cw >bestw
  - 中间结点的处理？ i≤ n
    - 左孩子 x[i]=1，检查可行性约束 cw+w[i] ≤ c_1 继续深入
    - 右孩子x[i]=0，检查上界函数 cw+r>bestw 继续深入，否则剪枝



批处理作业调度

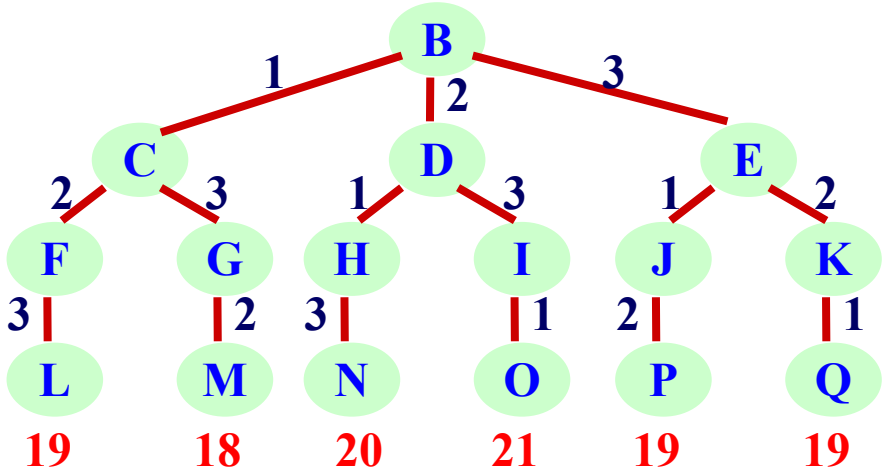
- 给定n个作业的集合$\{J_1, J_2, \ldots, J_n\}$，
  - 每个作业必须先由机器1处理，然后由机器2处理。
  - 作业J_i需要机器 j 的处理时间为t_{ji}
  - 对于一个确定的作业调度，设F_{ji}是作业i 在机器 j上完成处理的时间。
  - 所有作业在机器2上完成处理的时间和称为该作业调度的完成时间和。
- 批处理作业调度问题：对于给定的n个作业，制定最佳作业调度方案，使其完成时间和达到最小

批处理作业调度

- 实例

t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

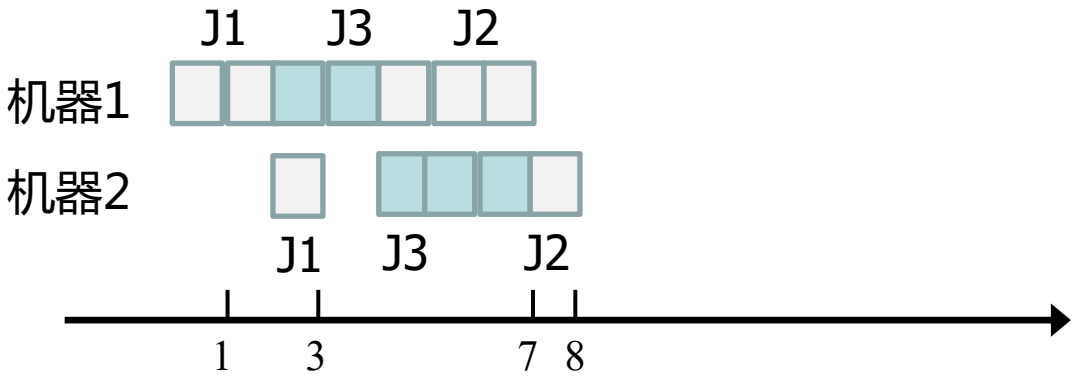
- 解空间
  - 这3个作业共有6种可能的调度方案
  - 1,2,3；1,3,2；2,1,3；2,3,1；3,1,2；3,2,1；
- 解空间结构： 排列树
  - 边对应具体的作业

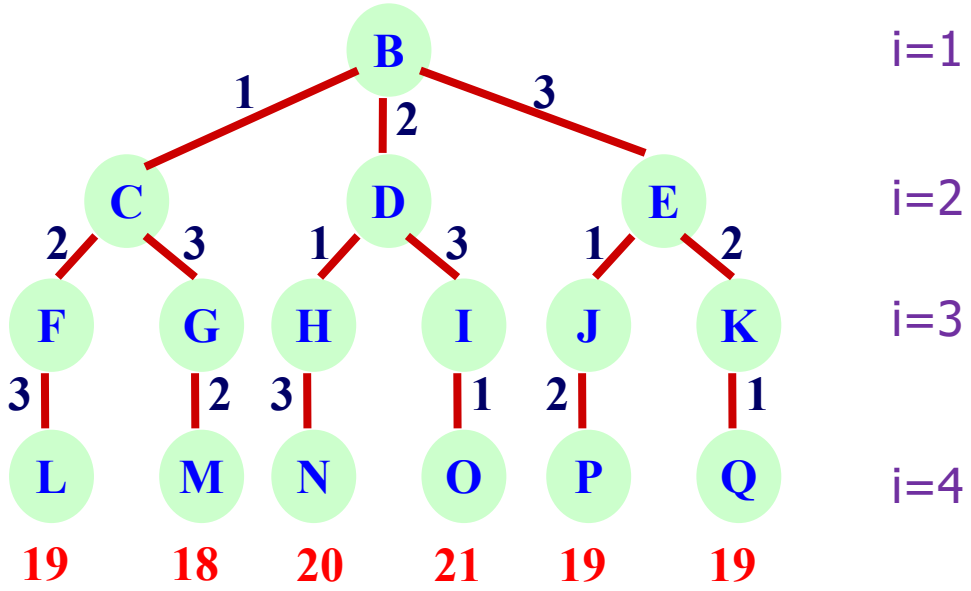


批处理作业调度

- 遍历过程
  - 何时找到解： i=n
    - 叶子结点, bestf=当前最小完成时间和
  - 中间过程： i<n
    - 剪枝： 当前完成时间和f >bestf　则剪枝

t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3



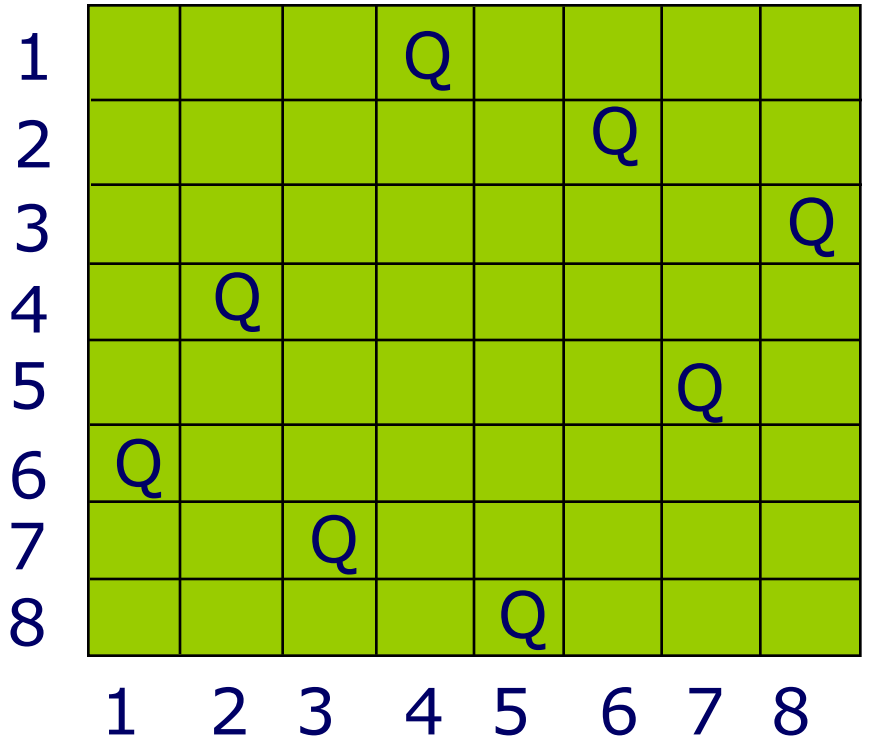


➢ 最佳调度方案是1,3,2，其完成时间和为18

思考：提前剪枝？

n后问题

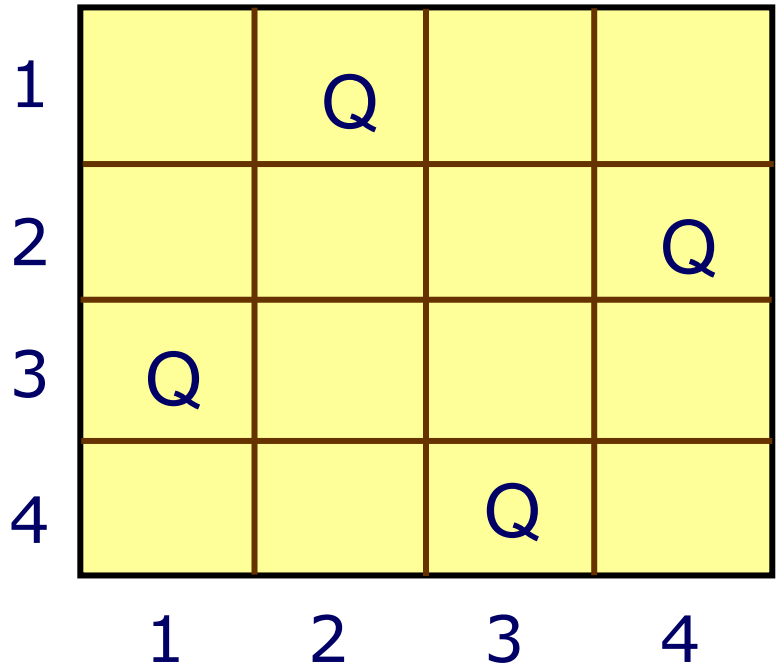
- 在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上



n后问题

- 实例
  - 4皇后
- 解向量：(x_1, x_2, x_3, x_4)
  - 解空间：$4^4 \Rightarrow 4!$ 种可能
- 显约束： x_i =1,2,3,4
- 隐约束：
  - 1)不同列： $x_i \neq x_j$
  - 2)不处于同一正、反对角线(不处于同一斜线)

    $|i-j| \neq |x_i - x_j|$
- 解空间树：4叉树



X_i：皇后i放在棋盘的第i行的第X[i]列

排列树问题

n后问题

- 剪枝函数
  - 可行性约束剪去不符合行、列、斜线约束的子树

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

找到的可行方案数

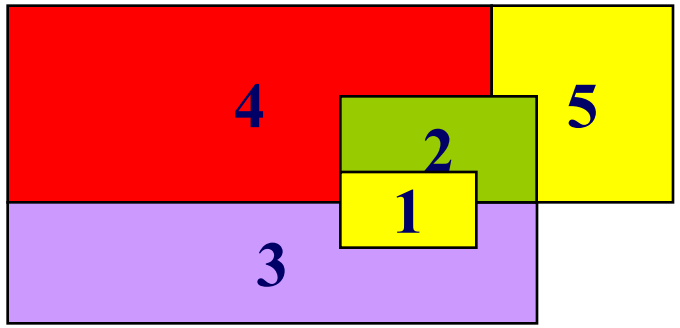
DFS
回溯法
改进？ 拉斯维加斯算法

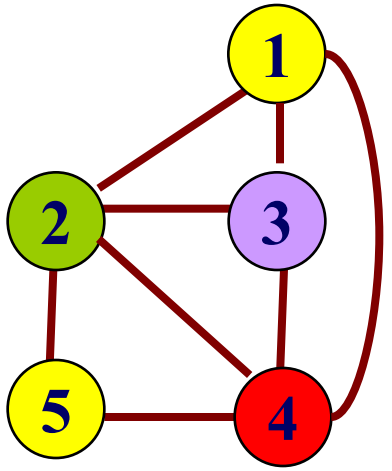
4. 对于每个孩子结点i
5. 检查可行性，深度优先递归搜索

```
Bool Queen::Place(int t){
1.    for (int j=1; j<k; j++){    //与前面k-1个皇后比较是否冲突
2.        if ((abs(k-j)==abs([x[i]-x[k])|| (x[j])==x[k])) return false;
3.     return true;
}
```

图的m着色问题

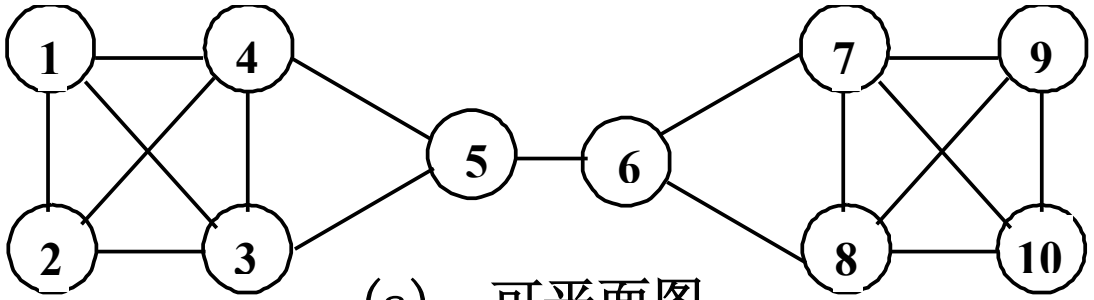
- 给定无向连通图G和m种不同的颜色。用这些颜色为图G的各顶点着色，每个顶点着一种颜色。是否有一种着色法使G中每条边的2个顶点着不同颜色。这个问题是图的m可着色判定问题。
- 若一个图最少需要m种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数m为该图的色数。求一个图的色数m的问题称为图的m可着色优化问题。



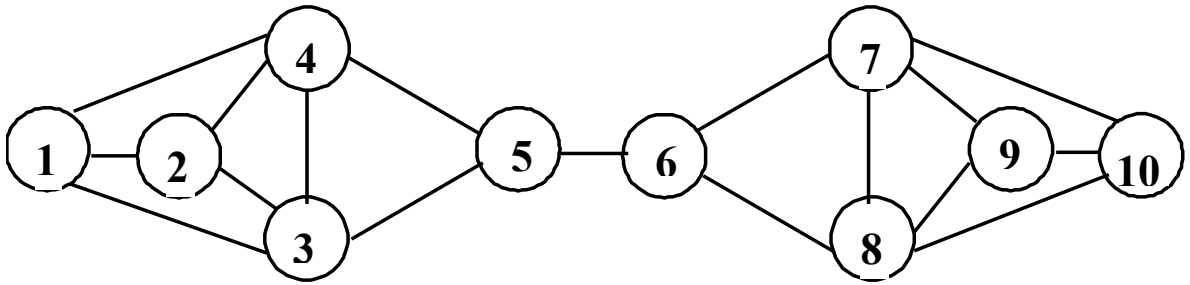


图的m着色问题

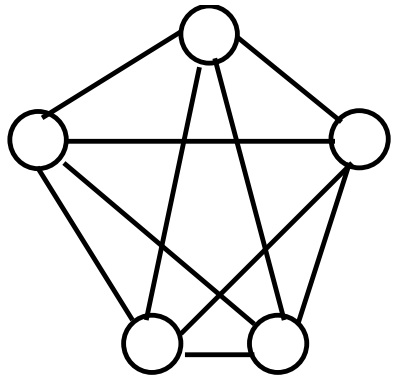
- 如果一个图的所有节点和边，都能用某种方式画在一个平面上且没有任何两边相交，则称这个图是平面图。
  - 例如，图 (a) 是一个可平面图, 因为它可以画成如图 (b)的形式
  - 实际上图a) 与图b)同构。图 (c)就是不可平面图。
  - 任何平面图都是可以4着色的



(a)　可平面图



(b)　图(a)的同构



(c)　不可平面图

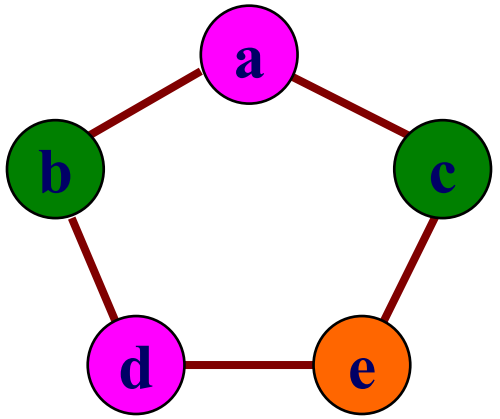
图的m着色问题

- 输入
  - 图G，m种颜色
- 输出
  - 如果不是m可着色就返回-1
  - 如果可着色，就返回所有不同着色法

- 回溯法求解

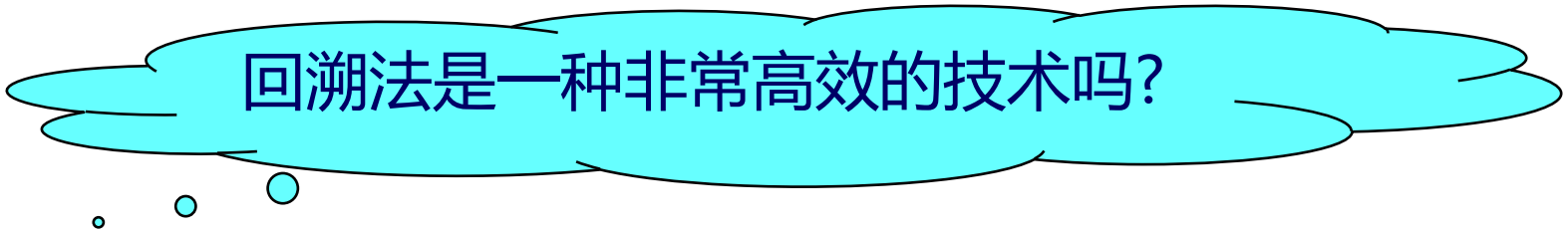
图的m着色问题

- 实例　　5顶点，3着色
- 解向量：$(x_1, x_2, x_3, x_4, x_5)$
  解空间：3^5种可能
- 显约束：　x_i =1,2, 3
- 隐约束：
  - 顶点i与已着色的相邻顶点颜色不重复
    (a[k][j]=1& x[j]!=x[k])
- 解空间树：3叉树　排列树



a[][]：图的邻接矩阵

讨论



- 克服困难性. 至少可以对某些组合难题的较大实例求解
- 最坏情况下,生成一个呈指数增长的状态空间中所有的可能解
- 缩小状态空间树规模的技巧
  - 利用组合问题的对称性
  - 把值预先分配给解的一个或多个分量
  - 预排序

回溯法效率分析

- 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。

由解空间结构确定
1,2,3

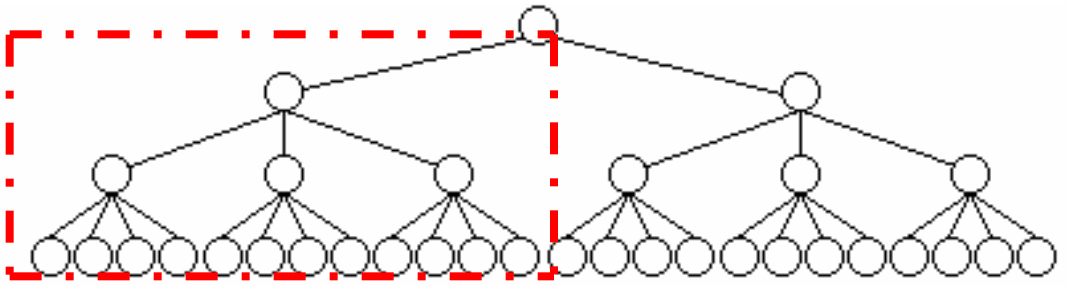
- 好的约束函数能显著地减少所生成的结点数
  - 但这样的约束函数往往计算量较大
  - 在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷
- 对于许多问题而言，在搜索试探时选取x[i]的值顺序是任意的。在其它条件相当的前提下，让可取值最少的x[i]优先

回溯法效率分析

- 实例

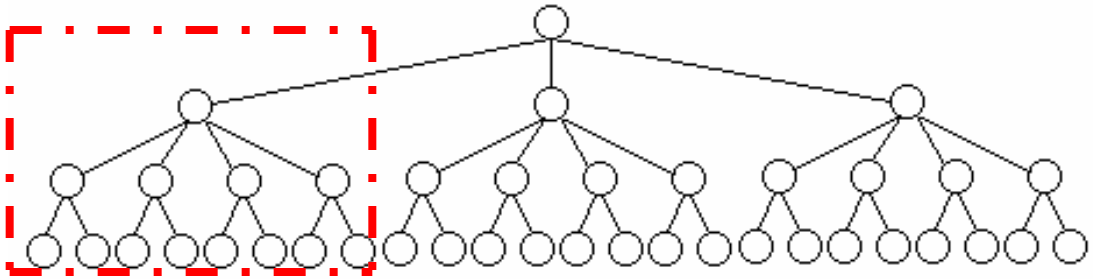
  图中关于同一问题的2棵不同解空间树

剪去12个元组



(a)

剪去8个元组



(b)

➢ 前者的效果明显比后者好

回溯法效率分析

- 回溯法的有效性
  - $O(p(n)2^n)$或$O(q(n)n!)$　　p(n)和q(n)是n的多项式
  - 当问题的规模较大时,也能用很少的时间求出问题的解
- 效率分析时的难点:
  - 生成结点的数目随问题的具体内容和结点的不同生成方式而变动
  - 同一问题的不同实例所产生的结点数也不同
  - 对于问题的具体实例很难预测回溯法的算法行为,难以估算解具体实例时产生的结点数

回溯法效率分析

如何估计出回溯算法的状态空间树的规模?

- 概率方法（ Monte Carlo ）
  - 在解空间树上生成一条从根到一个叶子的随机路径,并按照生成路径过程中不同选择的数量信息,来估计树的规模.
  - 设第i层的每个节点平均有c_i个子女(满足约束条件的子结点),则估计树中的结点数量为:
- $$1+ c_1 + c_1 c_2 +...+ c_1 c_2 ... c_n$$
- 提高估计的精确度
  - 选取若干条不同的随机路径(<20), 分别对各随机路径估计结点总数,然后再取这些结点总数的平均值

求所有解时,解空间中所有满足条件的结点都必须生成

回溯法效率分析

- 蒙特卡罗(Monte Carlo)算法
  - 一种计算方法。原理是通过大量随机样本，去了解一个系统，进而得到所要计算的值。
  - 在一般情况下可以保证对问题的所有实例都以高概率给出正确解,但是通常无法判定一个具体解是否正确.
  - 用于求问题的准确解
  - 缺点:
    - 无法有效的判定所得到的解是否肯定正确.

小结

- 回溯法主要用于以下困难的组合问题
  - 这些问题可能存在精确解,但无法用高效的算法求解
- 回溯法不同于穷举查找法
- 回溯法提供了一种有价值的解题方法

小结

- 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
  - 递归回溯
  - 迭代回溯
  - 子集树算法框架
  - 排列树算法框架
- 案例
  - 0-1 背包问题
  - 旅行商问题
  - 其它

思考题

- 最小生成树和旅行商问题？