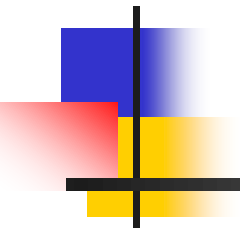


回溯法——通用的解题法

本章内容

- 搜索问题和算法
- 解空间
- 状态空间树
- 回溯法
- 应用
  - n 后问题
  - 子集和数问题
  - 货箱装船
  - 0/1 背包问题
  - 最大完备子图
  - 旅行商问题

搜索问题分类

- 我们考虑两类问题:

  存在性问题:

  - 求满足某些条件的一个或全部元组。
  - 如果存在这样的元组算法应返回Yes，否则返回No。
  - 这些条件称为约束条件.

  优化问题:

  - 给定一组约束条件,在满足约束条件的元组中求使某目标函数达到最大（小）值的元组。
  - 满足约束条件的元组称为问题的可行(feasible)解。

- 解决这类问题的最一般方法是使用搜索技术，即系统化的搜索解空间的技术。

搜索算法

- 穷举搜索(Exhaustive Search)
- 盲目搜索 (Blind or Brute-Force Search)
  - 深度优先(DFS)或**回溯搜索** (Backtracking)
  - 广度优先搜索(BFS)
  - 迭代加深搜索(Iterative Deepening)
  - **分支限界法**(Branch & Bound)
  - 博弈树搜索(α-β Search)
- 启发式搜索(Heuristic Search)
  - A*算法和最佳优先(Best-First Search)
  - 迭代加深的A*算法
  - B*， AO*，SSS*等算法
  - Local Search，GA等算法

搜索问题的形式化-解空间

- 用回溯法求解问题时，首先应明确定义问题的解空间。
- 问题的解空间至少应包含问题的一个(最有)解。
- 为了方便搜索整个解空间，需要将解空间组织起来。
- 搜索空间的三种表示方式：
  - 表序表示：搜索对象用线性表数据结构表示；
  - 显式图表示：搜索对象在搜索前就用图（树）的数据结构表示；
  - 隐式图表示：除了初始节点，其他节点在搜索过程中动态生成。主要应用于搜索空间大，难以全部存储的情形。
- 通常将解空间组织成树或者图的形式。

搜索空间的三种表示方式

- 对于解空间树，从树根到叶的任一路径表示解空间中的一个元素。
- 任何搜索算法都是一种系统地展开状态空间树的算法。
  - 状态空间树是在搜索过程中选择元组的各个分量时动态产生的树结构，是解空间树的子树。
  - 搜索算法并非事先将状态空间树存在计算机内再进行遍历, 而是通过一步一步展开状态空间树来找所求的解。
- 解空间不是唯一的,取决于算法的设计。

n后问题

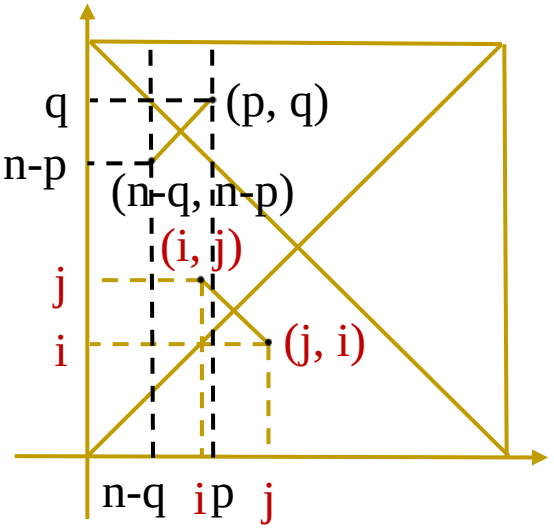
问题：在 n × n 格的棋盘上放置 n 个皇后，使任何 2 个皇后不放在同一行或同一列或同一斜线上.

解向量：$(x_1, x_2, \cdots, x_n)$

显约束：$x_i = 1, 2, \cdots, n$

隐约束：$x_i \neq x_j$ 并且 $|x_i - x_j| \neq |j - i|$

不同列　　不处于同一斜线

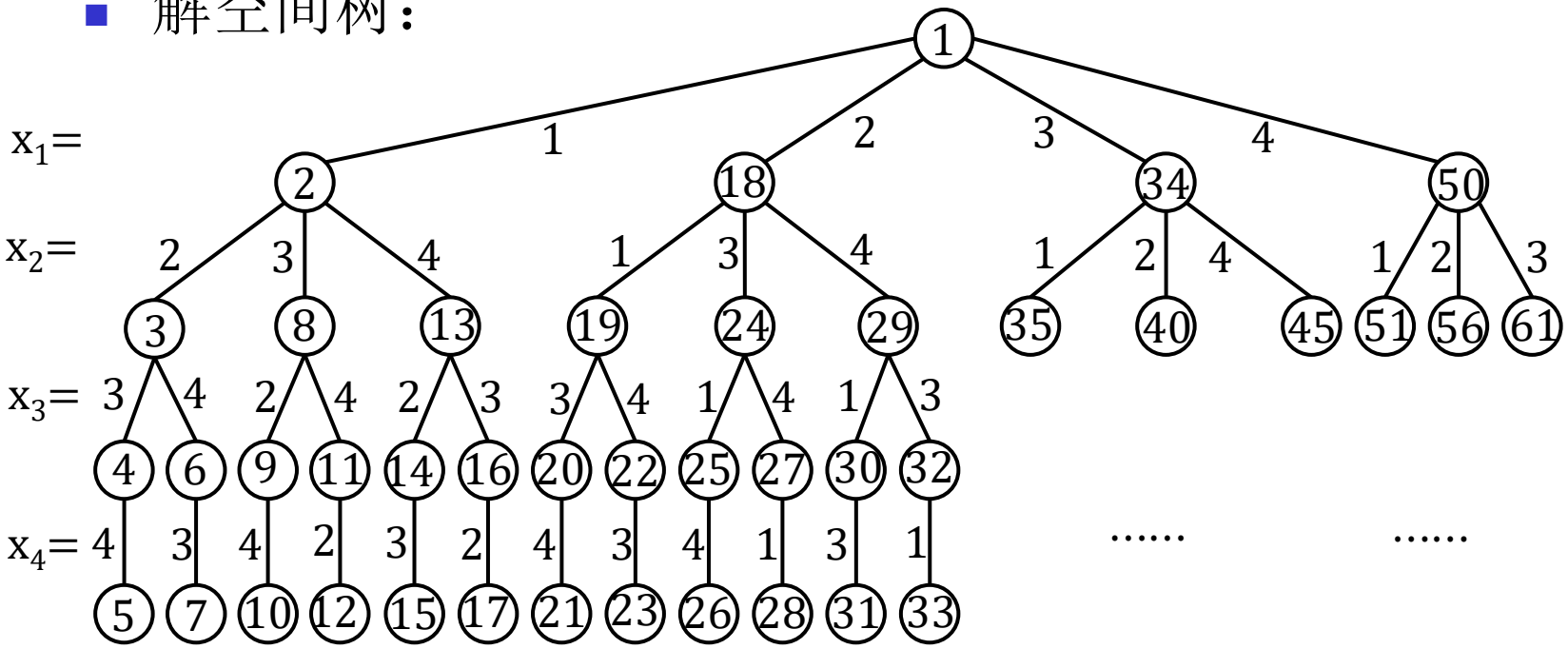


解空间：

- 显约束下问题的解空间由 n^n 个 n- 元组构成；
- 加入隐约束，每个 n-元组为 (1, 2, ···, n) 的一个排列，解空间的大小由 n^n 个元组减少到 n! 个元组。
- 很难直接利用隐约束生成解空间，可以用来限界。

4-皇后问题

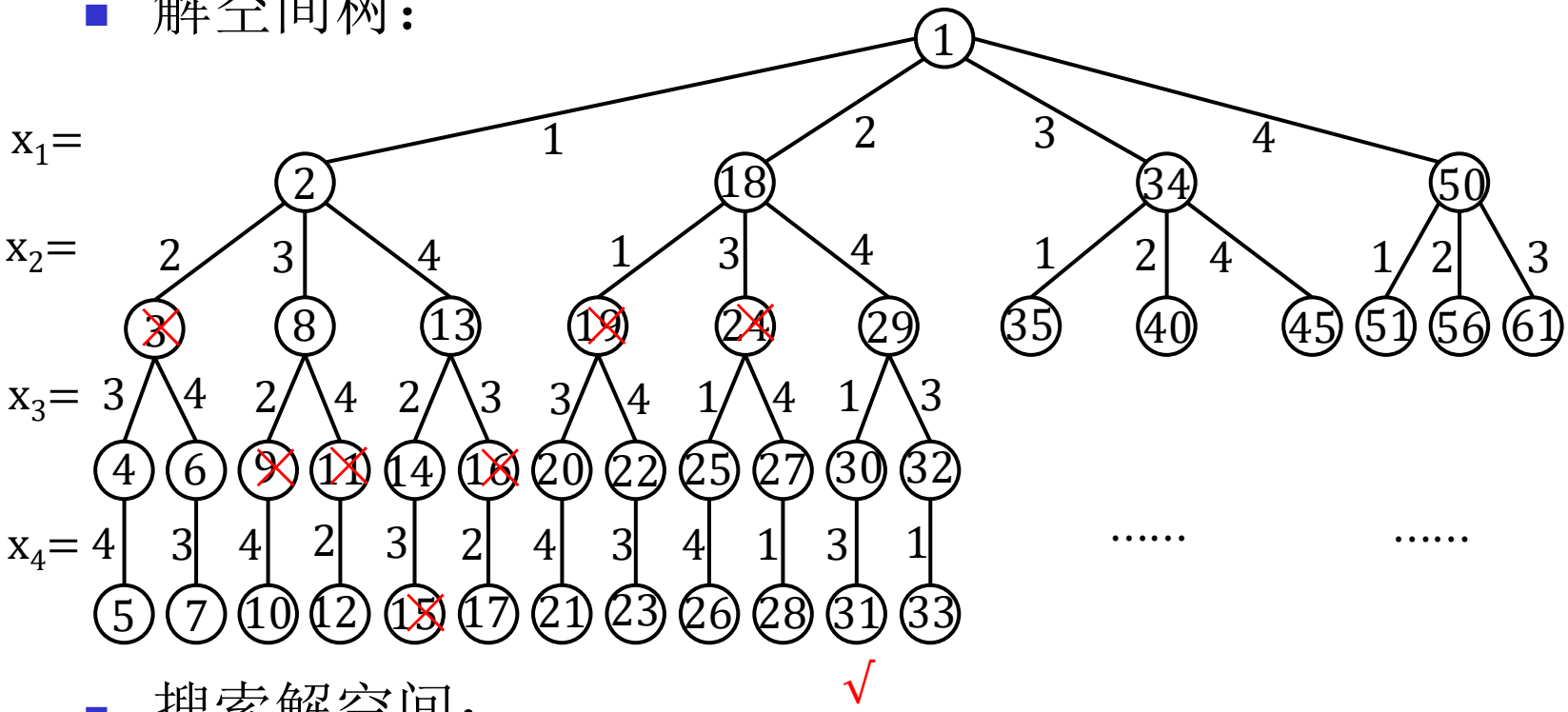
- 解空间树：



- 树的边由 x_i 的可能的取值标记。由 i 级到 i+1 级节点的边给出 x_i 的值。这种树也称为排列树。
- 从根节点到叶节点的一条路径对应解空间的一个元组.

4-皇后问题

- 解空间树：



- 搜索解空间：

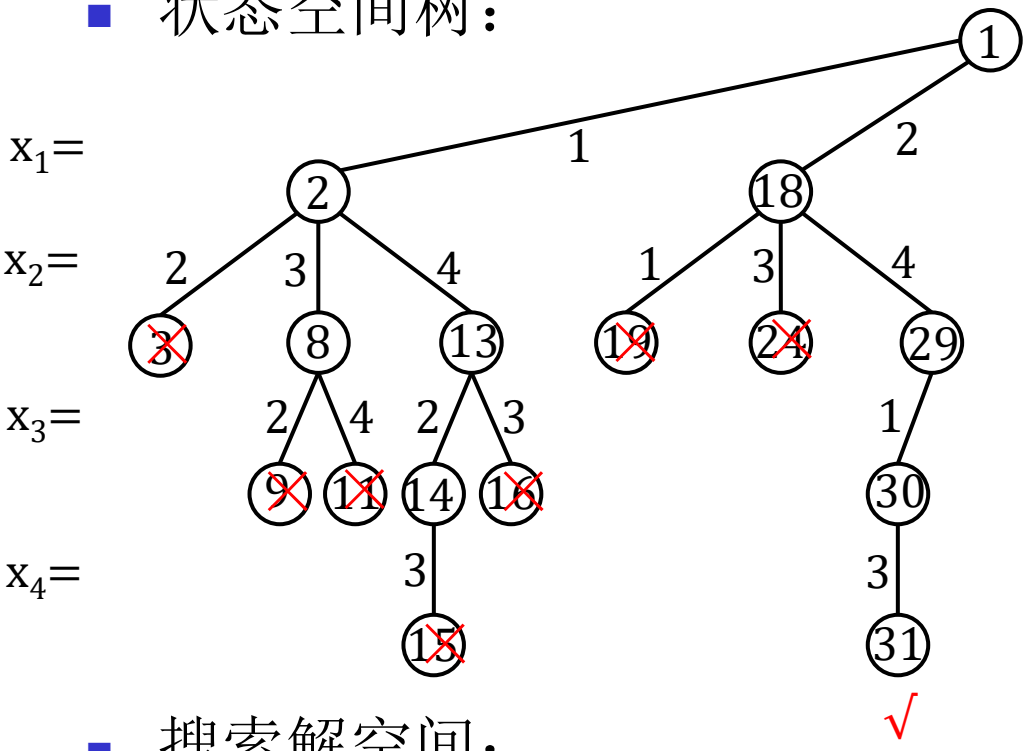
X1			
	X2	X2	
	X3		X3

X1			
	X2	X2	X2
	X3		
		X4	

	X1		
X2		X2	X2
X3			
		X4	

4-皇后问题

- 状态空间树：



- 搜索解空间：

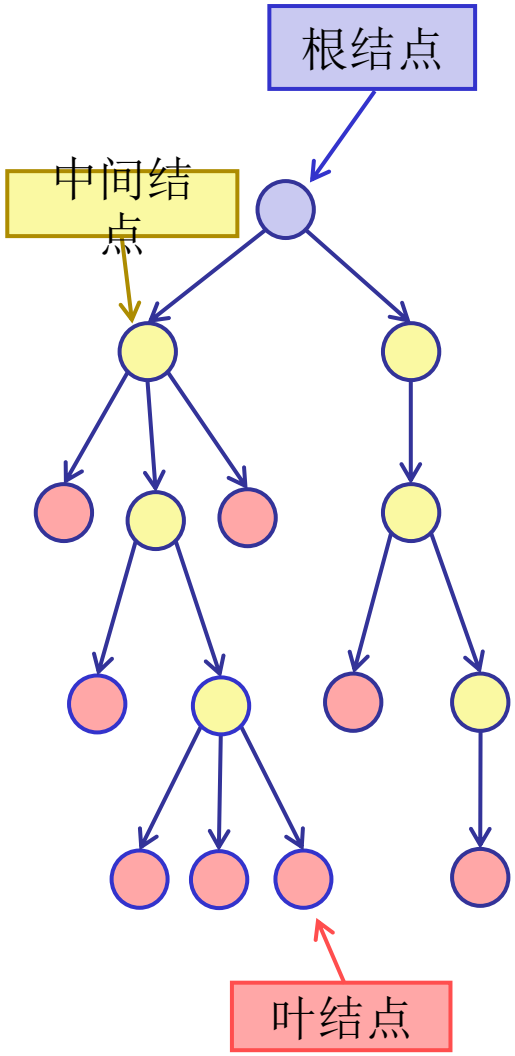
X1			
	X2	X2	
	X3		X3

X1			
	X2	X2	X2
	X3		
		X4	

	X1		
X2		X2	X2
X3			
		X4	

状态空间树

- 状态空间树上的节点可以分为三种：
  - 活结点：已展开了部分子节点，但所有子结点尚未全部展开的结点.
  - 死结点:被限界或已展开了所有子结点的结点.
  - 扩展结点：当前正在展开子结点的活结点.
- 展开过程，通过使用**约束函数**在扩展结点处剪去不满足约束的子树，从而降低搜索算法的时间和空间复杂度。
- 限界函数必须简单。



针对 n 叉树的递归回溯方法

```
1.  void backtrack (int t){
    // 形参 t 表示递归深度，即当前扩展结点在树中的深度
2.      if (t > n) {output (x);} // 到达叶子结点，将结果 x 输出
3.      else {
4.          for (int i = f(n, t); i<=g(n, t); i ++ ) {
            // 遍历结点t的所有子结点
5.            x[t] = h[i]; // h[i]是当前扩展点处x[t]的第i个可选值
6.            if (constraint (t) && bound (t))
              // 如果不满足剪枝条件，则继续遍历
7.              backtrack (t + 1);
8.          } }
9.  }
```

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 (f(n,t) <= g(n,t)) { // 遍历结点t的所有子结点
5.           for (int i = f(n,t); i <= g(n,t); i ++) { x[t] = h(i);
6.            if (constraint(t) && bound(t)) {  // 剪枝
7.                 if (solution(t)) { output(x);  }
                   // 找到问题的解，输出结果
8.                 else t ++;  // 未找到，向更深层次遍历
9.             }  }  }
10.       else {   t--;   }
11.     } }
```

回溯法的解题步骤

- 回溯法的解题步骤：
  - 问题的解表示成 $(x_1, x_2, \cdots, x_n)$ 的形式。
  - 针对所给问题，定义问题的解空间(显约束)。
  - 确定易于搜索的解空间结构。
  - 从根结点出发，以深度优先方式搜索解空间。
  - 在搜索过程中用剪枝函数避免无效搜索(隐约束)。
    - 假定 $(x_1, x_2, \cdots, x_{k-1})$ 已经确定。
    - 通过限界函数 $B(x_1, x_2, \cdots, x_{k-1})$ 判断哪些 x_k 的取值不能导致问题的解，从而停止展开该子节点。
- 回溯法是一种能避免不必要搜索的穷举式搜索法，适用于解一些解空间相当大的问题。

回溯法

- **回溯法**：加限界的深度优先展开状态空间树。
- **深度优先展开方法**：一个 E- 结点展开自己的一个子结点后，如果这个子节点不是叶子节点，就让该子结点成为新的 E- 结点，继续展开。这相当于对状态空间树做深度优先搜索。
- **优点**：在搜索过程中动态产生问题的状态空间。在任何时刻，算法只保存从根节点到当前扩展结点的路径。如果空间树中从根节点到叶节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))。
- **分枝限界法**：一个结点一旦成为 E- 结点，则它将展开其全部子节点，之后自己变成死结点。
  - 在分枝一限界法中要维持一个**活结点表**的结构, 存放已展开但还未成为 E 结点的那些结点.

n后问题

```
1.  bool Queen::Place(int k)
2.  {
3.   for (int j=1;j<k; j++)
4.     if ((abs(k-j)==abs(x[j]-
           x[k]))||(x[j]==x[k]))
5.        return false;
6.   return true;
7.  }
```

```
1.  void Queen::Backtrack(int t)
2.  {
3.     if (t>n) sum++; 到达叶子节点
4.       else
5.          for (int i=1;i<=n;i++)
6.          {
7.          x[t]=i;
8.          if (Place(t))  剪枝
9.              Backtrack(t+1);
10.         }
11.  }
```

- 8后问题的解为 (4, 6, 8, 2, 7, 1, 3, 5)。

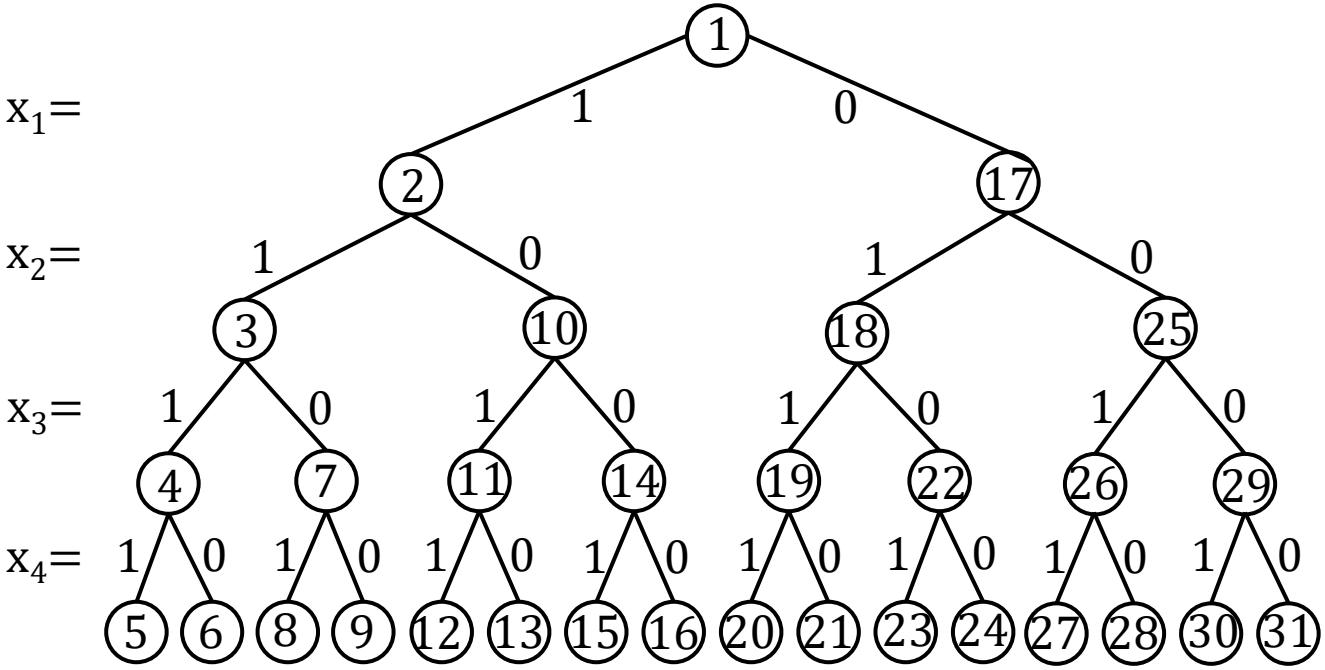
子集和数的判定问题

- 已知正数 M 和集合 $W = \{w_i \mid w_i > 0, 1 \le i \le n\}$，要求找出 W 的所有子集，使得子集内所有元素之和等于 M。
- 例如：$n = 4$，$M = 31$，$W = (11, 13, 24, 7)$。则满足要求的子集是 (11, 13, 7) 和 (24, 7)。
- 用 w_i 的下标 i 构成的元组表示一个解，则这两个解可表示为 (1, 2, 4) 和 (3, 4)。
- 元组(1, 2, 4) 和 (2, 1, 4) 代表同一子集。为了方便，限制元组分量按升序排列，即不考虑元组 (2, 1, 4)。
- 解的表示方法不唯一。

子集和数的判定问题

- 解也可以由一个 n- 元组 $(x_1, x_2, \cdots, x_n)$ 表示，其中 x_i 是 0/1 变量：
  - $x_i = 0$ 表示没有选择 w_i；
  - $x_i = 1$ 表示选择 w_i，
- 上例中的解可以表示为 (1,1,0,1) 和 (0,0,1,1)。
- 这种表示方法下，解的元组长度是固定的。
- 显式约束条件：$x_i \in \{0, 1\}$，$1 \le i \le n$。
- 隐式约束条件：$\sum_{1 \le i \le n} w_i x_i = M$。
- 解空间的大小为 2^n 个元组。

子集和数判定问题的解空间树



最大子集和数问题

- **子集和数的判定问题**：给定 n 个正数 w_i 和另一个正数 M，找出 $\{w_i, i=1, \cdots, n\}$ 中所有使得和数等于 M 的子集.
- **最大子集和数问题**：找不超过 M 的和数最大的子集.
- 当 w_i 和 M 为正整数时前者有解当且仅当后者有解。
- 最大子集和数问题是**背包问题**的特例：$w_i = v_i$。

最大子集和数问题

- 用长度固定的元组来设计一种回溯算法。
- 解向量的元素 x_i 取 1 或 0 值，表示子集是否包含 w_i。
- 限界函数的一种简单选择是：当且仅当

$$\sum_{1\le i\le k} w_i x_i + \sum_{k+1\le i\le n} w_i < M$$

时，$B(x_1, x_2, \cdots, x_k)$=true，停止展开节点 $(x_1, x_2, \cdots, x_k)$

- 此时继续展开不可能得到可行解。

强化限界函数

- 如果已将W(i)按非降次序排列, 则可以进一步限界如下: 如果 $\sum_{1\le i\le k} w_i x_i \neq M$，并且 $\sum_{1\le i\le k} w_i x_i + w_{k+1} > M$，则继续展开 $x_1, x_2, \cdots, x_k$ 不可能得到答案结点。
- 新的限界函数：$B(x_1, x_2, \cdots, x_k) = \text{ture}$ 当且仅当

$$\sum_{1\le i\le k} w_i x_i + \sum_{k+1\le i\le n} w_i < M$$

或者

$$\sum_{1\le i\le k} w_i x_i + w_{k+1} > M$$

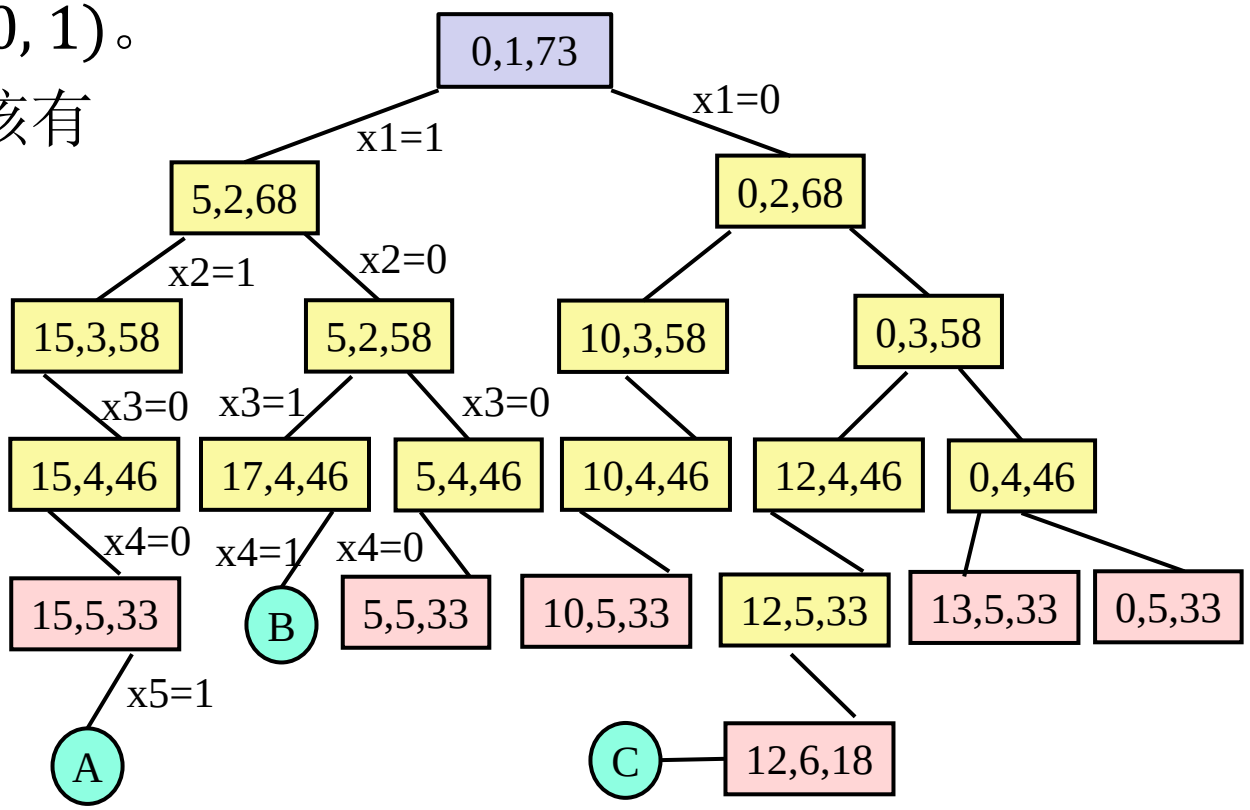
这时停止产生子节点 $(x_1, x_2, \cdots, x_k)$ 及其子树。

子集和数问题的回溯法

- 设 $x=(x_1,\cdots,x_{k-1})$ 为当前 E 节点，$S+r\geq M$ 成立；
- $S=w_1x_1+\cdots+w_{k-1}x_{k-1}$，$r=w_k+\cdots+w_n$。
- 如果 $S=M$，则找到了一个和数为 M 的子集，回溯找其它解。
- 如果 $S+w_k+w_{k+1}<M$，展开左子节点：
  - $x_k\leftarrow 1$，$S\leftarrow S+w_k$，$r\leftarrow r-w_k$，$k\leftarrow k+1$
- 如果 $S+w_k+w_{k+1}>M$，则 $r\leftarrow r-w_k$，并展开右子节点：
  - 如果 $S+r<M$ 或 $S+w_{k+1}>M$，停止展开右子节点并回溯；
  - 否则，$x_k\leftarrow 0$，$r\leftarrow r-w_k$，$k\leftarrow k+1$；
- 回溯：$k\leftarrow k-1$ (回到父节点)。

子集和数问题的回溯法例

- $n = 6$，$M = 30$，$W(1:6) = (5, 10, 12, 13, 15, 18)$。
- 图中矩形结点内的数字是 s, k, r 的值.
- 圆形结点表示一个答案节点,其对应的子集和数 = M。 这些节点是：$A = (1, 1, 0, 0, 1, 0)$，$B = (1, 0, 1, 1, 0, 0)$ 和 $C = (0, 0, 1, 0, 0, 1)$。
- 解空间树应该有 $2^6 - 1 = 63$ 个节点；
- 状态空间树只有 20 个矩形结点。



货箱装船问题

- 前面做过的货箱装船问题为：给定载重量为 c 的货船，找一种装船的方法，使得装载的货箱数目最多。
- 贪心法可以得到最优解！
- 现对该问题做一些改动：有两艘船和 n 个货箱；第一艘船的载重量是 c_1，第二艘船的载重量是 c_2，w_i 是货箱 i 的重量且 $\sum w_i \leq c_1 + c_2$，$1 \leq i \leq n$。
- 是否有一种可将所有 n 个货箱全部装走的方法(可行解)？若有的话找出该方法。

货箱装船问题

- 当 $n=3$，$c_1=c_2=50$，$w=[10,40,40]$ 时，可将货箱 1、2 装到第一艘船上，货箱 3 装到第二艘船上。
- 如果$w=[20,40,40]$，则无法将货箱全部装走。
- 当 $c_1=c_2$ 且 $\sum w_i=2c_1$ 时，该问题等价于分划问题：
  - 能否将一组非负整数分成两组，使得每组内的数之和相等？NP难度问题！
- 两船装船问题可以转化为求解优化问题：极大化第一只船的装箱重量。
- 如果剩余货箱能装入第二只船，则找到了一个可行解，否则无解。

货箱装船问题

- 与“极大化第一只船的装箱重量”等价的优化问题：

$$\begin{cases} \text{Maxmize } \sum w(i)x(i) \\ \text{s.t. } \sum w(i)x(i) \le c_1 \text{ and } x(i) \in \{0,1\} \end{cases}$$

- 背包问题 ($v_i = w_i$)，可用动态规划求解。
- 最大子集和数问题，也可用回溯法求解。

货箱装船问题限界条件

限界方法1：设 cw 为当前已装的重量，

$$cw = \sum_{0<j<i} w_j x_j$$

如果 $cw + w(i) > c_1$，则杀死该(左)子节点.

限界方法2：设 bestw 为当前最优装箱重量，r 为未装货箱的总重量，如 $cw + r \leq bestw$，则停止展开该节点。

- cw + r 为该节点对应的子问题的优化值的上界。
- 如果能找到更好的上界，限界效果会更好。

- 两种限界同时使用.
- Bestw 初始值为 -∞。
- 时间复杂度$O(2^n)$。
- 在某些情况下该算法优于动态规划算法。

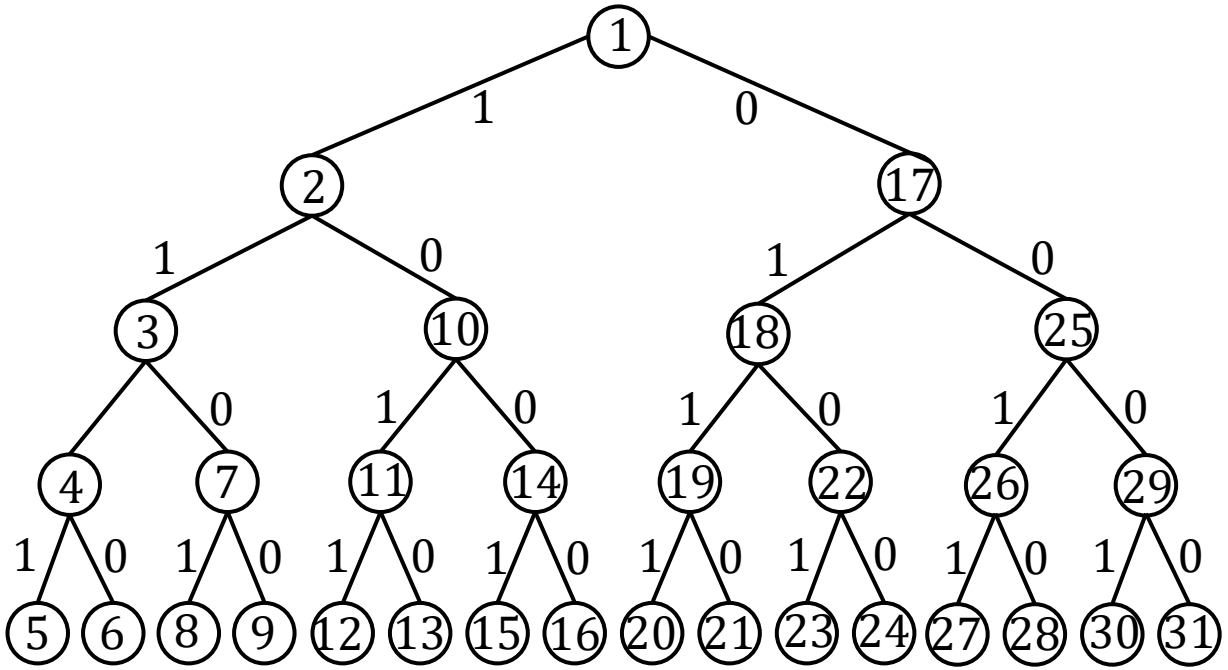
货箱装船问题的回溯算法

- 设 $x = (x_1, \cdots, x_{k-1})$ 为当前 E 节点, $cw + r > bestw$ 成立;
- 展开左子节点:
  - 如果 $cw + w_k > c_1$, 则停止展开该左子节点,

    $r \leftarrow r - w_k$, 并展开右子节点;
  - 否则, $x_k \leftarrow 1$, $cw \leftarrow cw + w_k$, $r \leftarrow r - w(k)$, $x = (x_1, \cdots, x_k)$为新的 E 节点;
- 展开右子节点:
  - 如果 $cw + r \leq bestw$, 停止展开该右子节点, 并回溯到最近的一个活节点;
  - 否则, 令 $x_k = 0$, $x = (x_1, \cdots, x_k)$ 为新的 E 节点.
- 回溯: 从 $i = k-1$ 开始, 找 $x_i = 1$ 的第一个 i; 修改 cw 和 r的值: $cw \leftarrow cw - w_i$, $r \leftarrow r - w_i$.
- 如果 $k = n$ 且 $cw + w_n > c_1$, 则:
  - 如果 $cw > bestw$, $bestw \leftarrow cw$, $x_n \leftarrow 0$;
  - 否则, 回溯.
- 否则 (即 $cw + w_n \leq c_1$),
  - $cw \leftarrow cw + w_n$, 如果$cw > bestw$, $bestw \leftarrow cw$, $x_n \leftarrow 1$;
  - 否则, 回溯.
- 当回溯过程到达根节点时算法结束.
- 因为左子节点和父节点有相同的 cw + r 值, 所以算法在展开左子节点时不检查限界条件2.

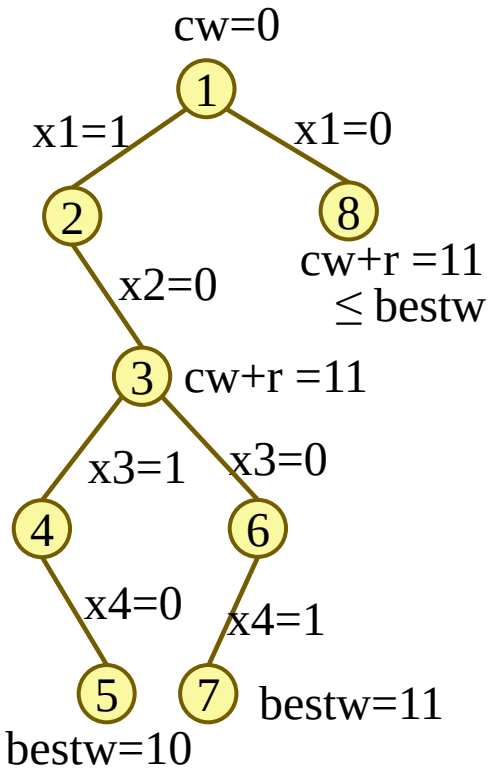
货箱装船问题例

- N = 4，w = [8, 6, 2, 3]，$c_1 = 12$。
- 解空间树：



展开的部分状态空间树

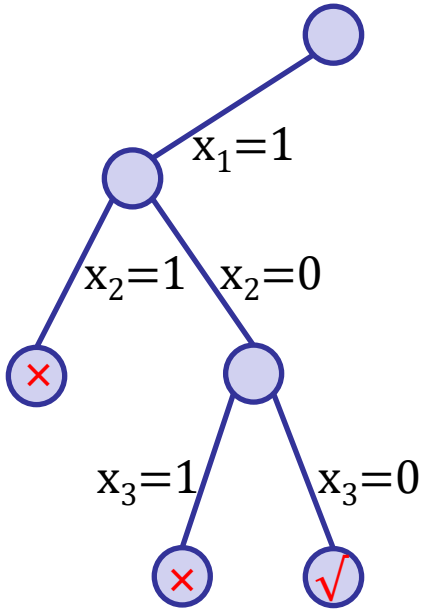
- $n = 4, w = [8, 6, 2, 3], c_1 = 12$。
- 状态空间树见右图。
- 程序中可以使用一个数组 bestx[] 存放达到最优解的路径。
- 数组 x[] 存放回溯法当前使用的路径.
- 回溯时, 如果 x[i]=1 说明还可以展开右子节点; 如果 x[i]=0 说明所有子节点都展开了, 该节点成为死节点, 应向上一级节点回溯.

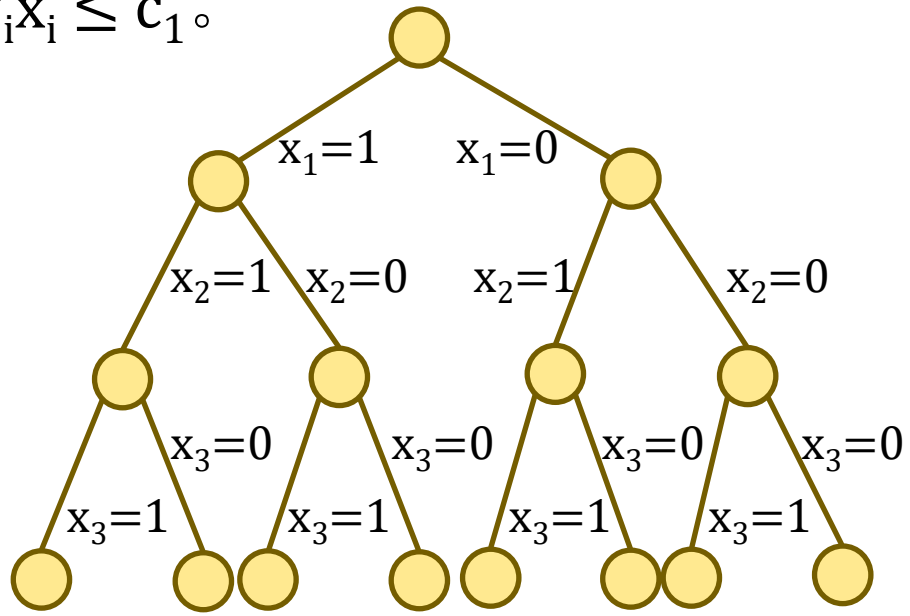


- 因为左子节点和父节点有相同的 cw + r 值, 所以算法在展开左子节点时不检查限界条件 2

0/1背包问题

- n = 3，w = [20, 15, 15]，p = [40, 25, 25]，C = 30。
- 约束函数：$\sum_{0\le i\le n} w_i x_i \le c_1$。
- 状态树：





解空间树

- 贪心解 (1, 0, 0)，贪心值为 40。

0/1背包问题的回溯算法

- 按密度降序对物品排序。
- Bestp ← −∞ 是算法目前获得的最优效益值
- 设 $X = (x_1, \cdots, x_{k-1})$ 为当前 E 节点， bound > bestp 成立。
- bound(X) = cp + r 为 X 对应的子树优化效益值的上界。
  - $cp = x_1p_1 + \cdots + x_{k-1}p_{k-1}$ 是当前获得的效益值。
  - r 为剩余待选物品的连续背包问题的优化效益值 (贪心法得到)。
- 展开左子节点：
  - 如果 $cw + w_k \le c$，则装入物品 k，且
  - $cw \leftarrow cw + w_k$, $cp \leftarrow cp + p_k$,
  - $x_k \leftarrow 1$
- 否则, 展开右子节点：
  - 如果 bound ≤ bestp，则停止产生右子树,
  - 否则, $x_k \leftarrow 0$, 并令 $(x_1, \cdots, x_k)$ 为 E 节点。

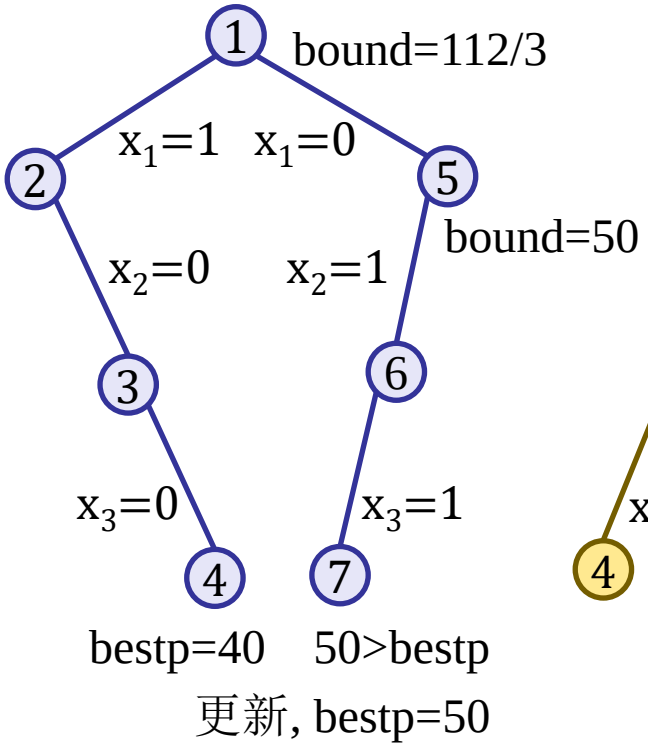
0/1背包问题的回溯法

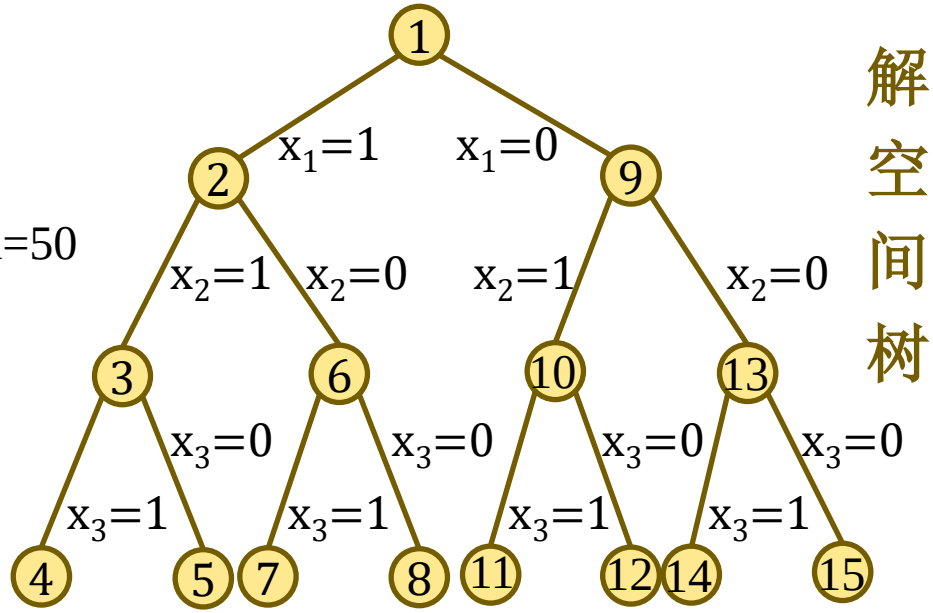
- 回溯算法 (或者说所有搜索算法)的关键在于剪枝，就是怎么减少搜索可能解的数量。
- x_k 的每个取值负责一个物品放进去或者不放的状态。
- 在第 k 个循环里面验证这个可能解是不是最优的。是就做点什么，不是，就继续执行。
- 问题：
  - 首先，k 不是事先知道的，
  - 第二，即使知道 k 多大，也没用，万一是10000呢？
- 回溯的细节类似装船问题。

0/1背包问题

- n = 3，w = [20, 15, 15]，p = [40, 25, 25]，C = 30。
- 优化解 (0, 1, 1)，优化值为 40。
- bound 是假设物品可拆，例如 $40 + (2/3) \cdot 25 = 112/3$，当比40大时要继续走，每一步都要算bound。

状态空间树

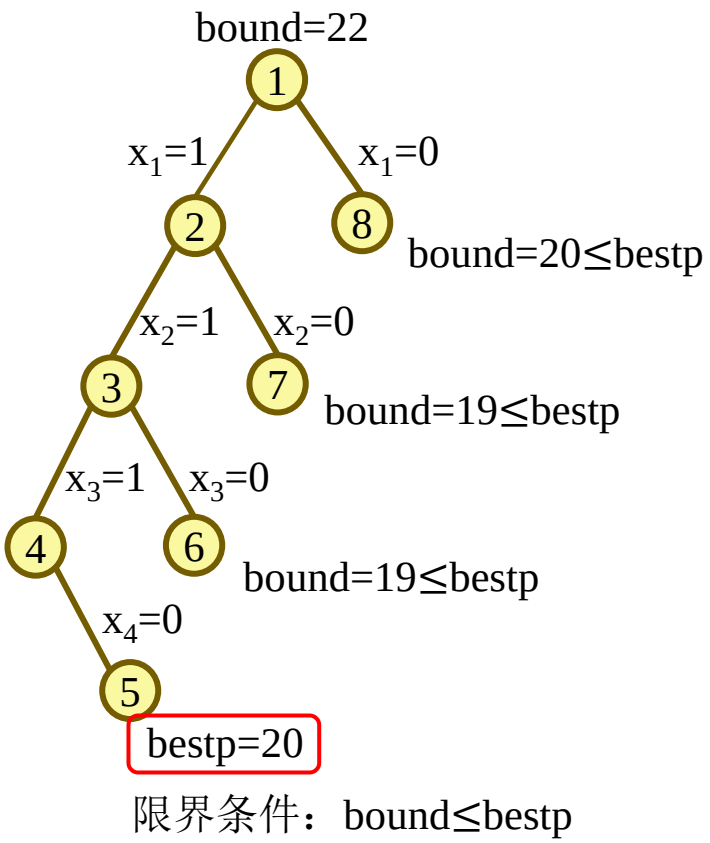




解空间树

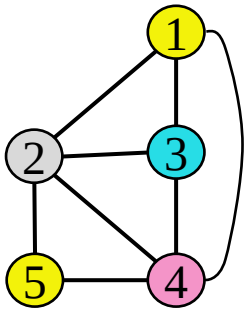
0/1背包问题例

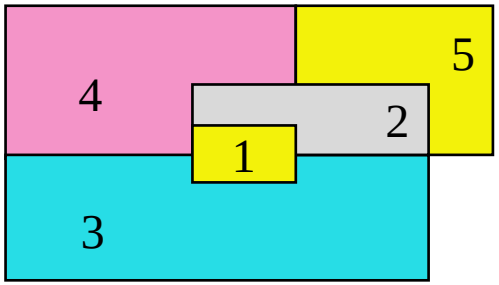
- n = 4, c = 7, p = [9, 10, 7, 4], w = [3, 5, 2, 1]。
- p/w = [3, 2, 3.5, 4]。
- 按密度排列为: (4, 3, 1, 2), w'=[1,2,3,5], p'=[4,7,9,10]
- 展开的状态空间树见右图。
- 在结点 5 处得到 bestp = 20；结点 6 处 bound = 19, 故限界掉；类似,结点7, 8 也被限界掉.
- 其解为 x' = [1,1,1,0]
- 回到排序前为 x = [1,0,1,1].



图的m着色问题

- 给定无向连通图 G 和 n 种不同的颜色。
- 图的 m 可着色判定问题：是否有一种着色法，使 G 中每条边的 2 个顶点着不同颜色。
- 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着色问题的贪心解法

本解法称 Welch Powell 法。

1. 将 G 的结点按照度数递减的次序排列。
2. 用第一种颜色对第一个结点着色，并按照结点排列的次序 对与前面着色点不邻接的每一点着以相同颜色。
3. 用第二种颜色对尚未着色的点重复步骤 2。
4. 用第三种颜色继续这种作法，直到所有点着色完为止。

四色问题

- 1852年，毕业于伦敦大学的弗南西斯·格思里来到一家科研单位搞地图着色工作时，发现了一种有趣的现象：“看来，每幅地图都可以用四种颜色着色，使得有共同边界的国家都被着上不同的颜色。”
- 1872年，英国当时最著名的数学家凯利正式向伦敦数学学会提出了这个问题，于是四色猜想成了世界数学界关注的问题，与费马猜想相媲美。
- 多年来，虽然已证明用 5 种颜色足以对任一幅地图着色，但是一直找不到一定要求多于 4 种颜色的地图。

四色问题

- 四色问题是图的 m 着色问题的一个特例。
- 1976年 爱普尔、黑肯和考西在美国伊利诺斯大学的两台不同的电子计算机上，用了1200小时，做了100亿次判断，终于证明了 4 种颜色足以对任何地图着色。
- 不过很多数学家并不满足于计算机取得的成就，仍致力于寻找一种简洁明快的书面证明方法。

图的m着色问题回溯算法

解向量：$(x_1, x_2, \ldots, x_n)$ 表示顶点 t 所着颜色 x_t。

可行性约束函数：顶点 t 与已着色的相邻顶点颜色不重复。

方案：

1.X[n] 存储 n 个顶点的着色方案，可以选择的颜色为 1 到 m。

2.若 t 与所有其它相邻顶点无颜色冲突，则继续为下一顶点着色；否则，回溯，测试下一颜色。

3.否则，依次对顶点 t 着色 (1 ～ m)。

4.若 t > n 则已求得一个解，输出着色方案即可。

图的m着色问题回溯算法

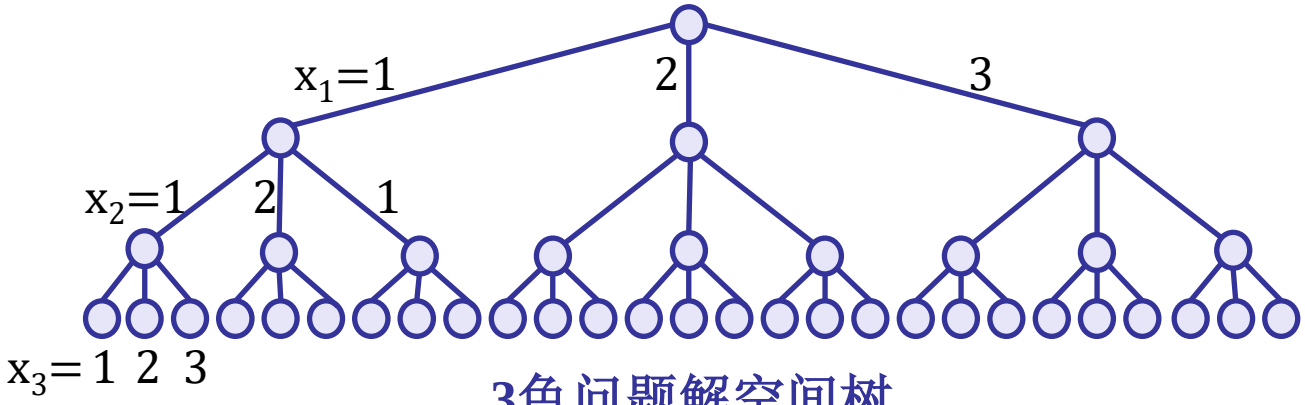
```
1.  void X::Backtrack(int t) {//X[n] 存储顶点的着色方案
2.    if (t>n) {//若 t > n 则已求得一个解，输出着色方案
3.       sum++;
4.       for (int i=1; i<=n; i++)
5.          cout << x[i] << ' ';
6.          cout << endl; }
7.     else  //依次对顶点 t 着色 ( 1 ～ m )
8.       for (int i=1;i<=m;i++) {
9.          x[t]=i;
10.         if (Ok(t)) Backtrack(t+1); }}
11. bool X::Ok(int k){// 检查颜色可用性
12.    for (int j=1;j<=n;j++)
13.       if ((a[k][j]==1)&&(x[j]==x[k])) return false;
14.    return true;}
```

图的m着色问题回溯算法

复杂度分析

- 图的 m 可着色问题的解空间树中，中间结点个数为 $\sum_{0\le i\le n-1} m^i$。
- 对于每一个中间结点，在最坏情况下，用 ok 检查这个结点的每一个儿子所相应的颜色可用性需耗时$O(mn)$。
- 因此，回溯法总的时间耗费是

$$\sum_{0\le i\le n-1} m^i(mn) = nm \cdot \frac{m^n-1}{m-1} = O(nm^n)$$

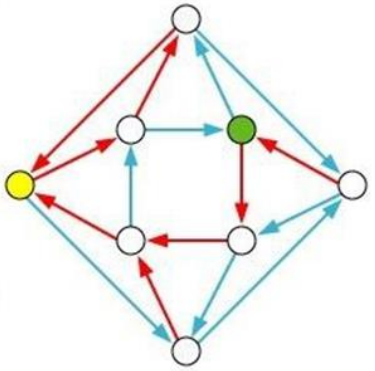


3色问题解空间树

路线着色问题

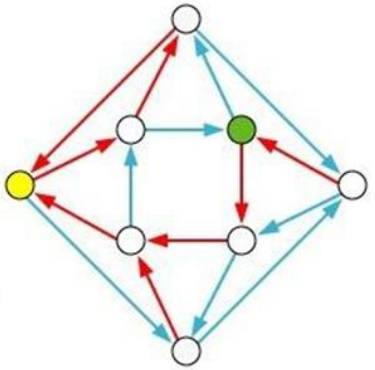
- 1970年犹太裔美国数学家、计算机专家本杰明-韦斯和他在 IBM 工作的同事罗伊-艾德勒于首先提出来**路线着色问题**。
- 一个人来到他从未造访过的小镇上，驾着车到处寻找他朋友的家，即使连路名都没有。朋友说，别担心，他会指示他如何到达，先向左，再向右，接着向左……”
- 猜想：如果路线的数量有限，人们应该能画出一张地图，标上不同的颜色，把人引导到某一目的地。
- 韦斯和艾德勒花了八年都无法解答。
- 接下来三十年间一百多位数学家也束手无策。

路线着色问题例



- 如果按右图中所示方式将 16 条边着色，那么不管你从哪里出发：
  - 如果按照“蓝⟶红⟶红、蓝⟶红⟶红、蓝⟶红⟶红”（这是道路的颜色）的方式行走，不管从哪个点出发都能到黄点；
  - 如果是“蓝⟶蓝⟶红、蓝⟶蓝⟶红、蓝⟶蓝⟶红”，则一定能到绿点。
- 解决路线着色问题要涉及到图论、群论、矩阵论、概率论、代数学、拓扑学、数值分析等多个数学分支。

路线着色定理



- 如果，G 是一个有向图
- 如果，G 的每个顶点的出度都是 k。
- 如果，G 的一个同步着色满足以下条件：
  - G 的每个顶点有且只有一条出边被染成了 1 到 k 之间的某种颜色；
  - G 的每个顶点都对应一种走法，不管你从哪里出发，按该走法走，最后都结束在该顶点。
- 那么，有向图 G 存在同步着色的必要条件是 G 是**强连通**而且是**非周期**的。

图中包含的所有环的长度没有大于1的公约数

对于每一对v_i,v_j, $v_i \neq v_j$, 从v_i到v_j和从v_j到v_i都存在路径

大器晚成的艾夫拉汉-特雷特曼

- 2007年9月，以色列数学家艾夫拉汉·特雷特曼在网上的一个数学文献库里贴出他的解题方法。
- 数学界为之震惊了，并认为他已经掌握了破解路线着色谜题的要领和诀窍。
- 世界上众多著名学术刊物编辑部得知此事后，纷纷向他约稿。
- 2008年2月，特雷特曼进一步完善了自己的解题方法。
- 出于对以色列的热爱，他把论文发表在《以色列数学杂志》上。

大器晚成的艾夫拉汉-特雷特曼

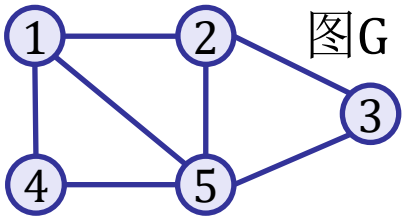
- 艾夫拉汉-特雷特曼1945年出生在俄罗斯叶卡捷琳堡的一个犹太人家庭。
- 1972年在乌拉尔州立大学获得数学博士学位，之后在乌拉尔科技大学任教。虽然他是个小有名气的数学家，犹太人身份使他在工作中受到歧视和排挤。
- 1992年移居以色列。身无分文，生活仍然十分艰苦。为养家糊口他经常去教会领救济品，后来在好心人的介绍下，他成了一名值夜班的保安员。
- 1995年，特雷特曼被聘为巴尔伊兰大学的教员。
- 当年把他招进这所大学的以色列数学家斯图尔特-马戈利斯教授回忆说，“我第一次见到他时，他穿着守夜人的制服，不修边幅，衣服很脏。”

大器晚成的艾夫拉汉-特雷特曼

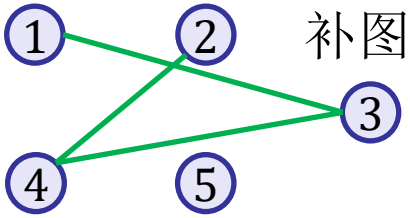
- 以色列人也为特雷特曼取得的成就感到无比的骄傲。特拉维夫电视台中断了正常的节目播放，以第一时间发布了这一重大消息，连中东其他国家的主流媒体也作了大篇幅的相关报道。
- 特雷特曼在数学上的这一成果极为令人瞩目，英国《独立报》为此事专门发表了一篇题为“身无分文的移民成了数学超级明星”的文章，给予了高度的评价。

完全图的概念

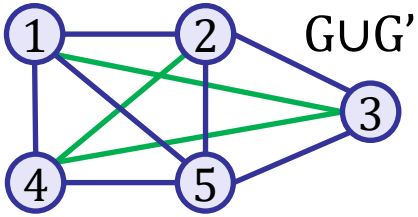
- 给定无向图 G = (V, E)，其中V是顶点集；E 是边集。
- 如右图，V={1, 2, 3, 4, 5}，E = {(1, 2), (1, 4), (1, 5), (2, 3), (2, 5), (3, 5), (4, 5)}。
- 完全图 G 就是指图 G 的每两个顶点之间都有连边。例如，GUG' 是一个完全图。



图G



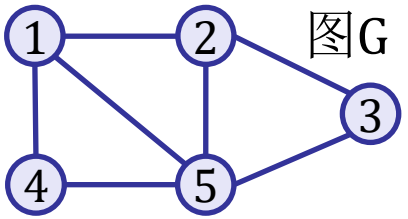
补图G'



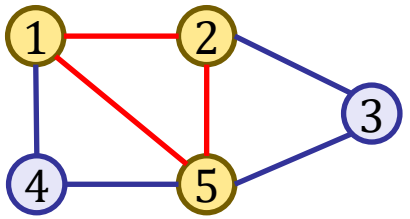
GUG'

最大团的概念

- 如果 U 是图 G 的一个顶点子集，并且对于U中的任意两个顶点 u 和 v，都有边 $(u, v) \in E$，则称 U 是 G 的一个**完全子图**。



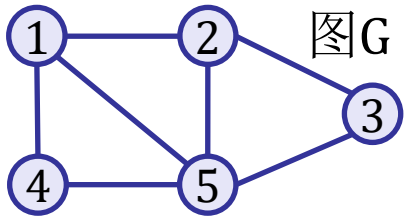
- {1, 2}、{3, 2, 5}、{1, 2, 5}、{1, 4, 5} 等都是G的完全子图。



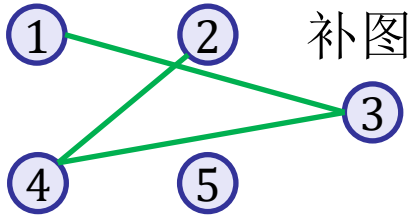
- 一个完全子图 U 的**尺寸**是指图 U 中顶点的数量。
- 如果一个完全子图 U 不被包含在 G 的一个更大的完全子图中，称它是图 G 的一个**集团**。
- {1, 2} 是图 G 的一个完全子图，但不是一个团，因为它包含于 G 的更大的完全子图 {1, 2, 5}之中。
- **最大集团**是具有最大尺寸的集团，简称**最大团**。

独立集的概念题

- 如果 U ∈ V 且对任意 u, v ∈ U 有 (u, v) 不属于 E，则称 U 是 G 的**空子图**。如 {1, 3}, {2, 4}, {3, 4}。
- 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。
- 如 {2, 4} 是 G 的一个空子图，同时也是 G 的一个最大独立集。
- 虽然 {1, 2} 也是 G' 的空子图，但它不是 G' 的独立集，因为它包含在 G' 的空子图 {1, 2, 5} 中。
- G 的**最大独立集**是 G 中所含顶点数最多的独立集，如 {1, 3}。



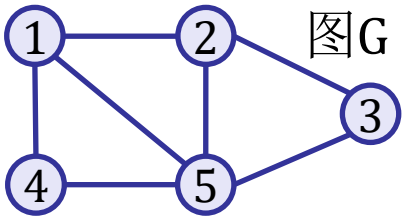
图G



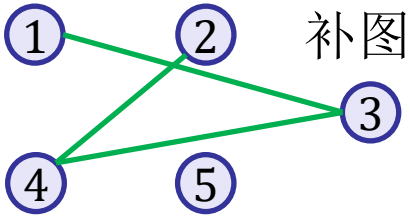
补图G'

最大团问题

- 如果 U 是 G 的完全子图，例如 {1, 2, 5}，则它也是 G’ 的空子图，反之亦然。
- G 的团与 G’ 的独立集之间存在一一对应的关系。
- 例如，{2, 3}, {1, 2, 5} 都是 G’ 的独立集，它们同时是 G 的团。
- 特殊地，U 是 G 的最大团当且仅当 U 是 G’ 的最大独立集。 {1, 2, 5} 是 G’ 的最大独立集。{1, 4, 5} 和 {2, 3, 5} 也是 G’ 的最大独立集。
- 求最大团和最大独立集两个问题都是 NP－难度问题。



图G



补图G’

最大团问题的状态空间树

- 图 G 的最大团和最大独立集问题都可以看做是图 G 的顶点集 V 的子集选取问题。因此可以用子集树来表示问题的解空间。
- 状态空间树类似 0/1 背包问题的状态空间树。
- 最大团问题的状态空间树：元组长度等于图的顶点数，分量取值 $\{0, 1\}$：$x_i = 1$ 表示顶点 i 在所考虑的集团中.
- 检查根节点到状态空间树的某一状态节点的路径上对应的图的顶点子集是否构成一个完全子图？如果不是则不再展开。
- 和 0/1 背包问题相似，使用当前集团的顶点数 cn ＋ 剩余顶点数 ≤ bestn 进行限界。

最大团问题的回溯算法

- 初始时最大团为一个空集，依次考虑每个顶点。
- 查看当前顶点加入集合之后是否仍然构成一个团(即是否该点到集合中各点均有边)：
  - 可以成团，将该顶点加入团；
  - 如果不行，直接舍弃；
- 递归判断下一顶点。
- 如果剩余未考虑的顶点数加上团中顶点数(cn+n-i)不大于当前解的顶点数(bestn)，回溯。
- 当搜索到一个叶结点时，停止搜索，更新最优解和最优值。
- 无向图的最大团和最大独立集问题都可以用回溯法在$O(n2^n)$的时间内解决。

最大团的回溯法

```
void AdjacencyGraph::maxClique(int i)
{//计算最大完备子图的回溯代码
   if (i>n){//在叶子上找到一个更大的完备子图，更新
      for (int j=1; j<=n; j++) bestx[j]=x[j];
      bestn=cn;
      return; }
   //在当前完备子图中检查顶点i是否与其他顶点相连
   int OK=1;
   for (int j=1; j<i; j++)
```

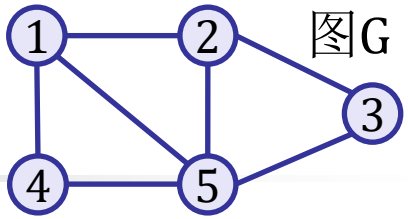
计算最大完备子图的回溯代码

```
if (x[j] && a[i][j] == NoEdge) {// i不与j相连
        OK=0;
        break; }
 if (OK) {//尝试 x[i]=1
      x[i]=1; //把i加入完备子图
      cn++;
      maxClique(i+1);
      x[i]=0;
      cn--;}
   if (cn+n-i>bestn) {//尝试 x[i]=0
      x[i]=0;
      maxClique(i+1);}
}
```

计算最大完备子图的回溯代码

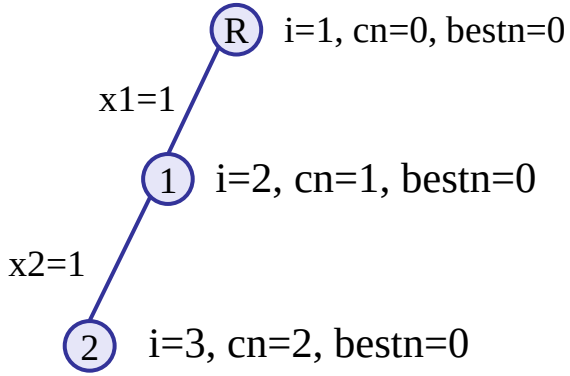
```
int AdjacencyGraph::MaxClique(int v[])
{  //返回最大完备子图的大小，//完备子图的顶点放入v[1:n]
   //初始化
    x=new int[n+1];
    cn=0;
    bestn=0;
    bestx=v;
    //寻找最大完备子图
    maxClique(1);
    delete []x;
    return bestn;
}
```

最大团问题例

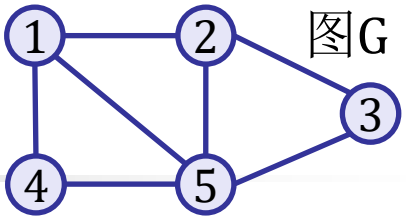


假设我们按照 1 → 2 → 3 → 4 → 5的顺序深度搜索：

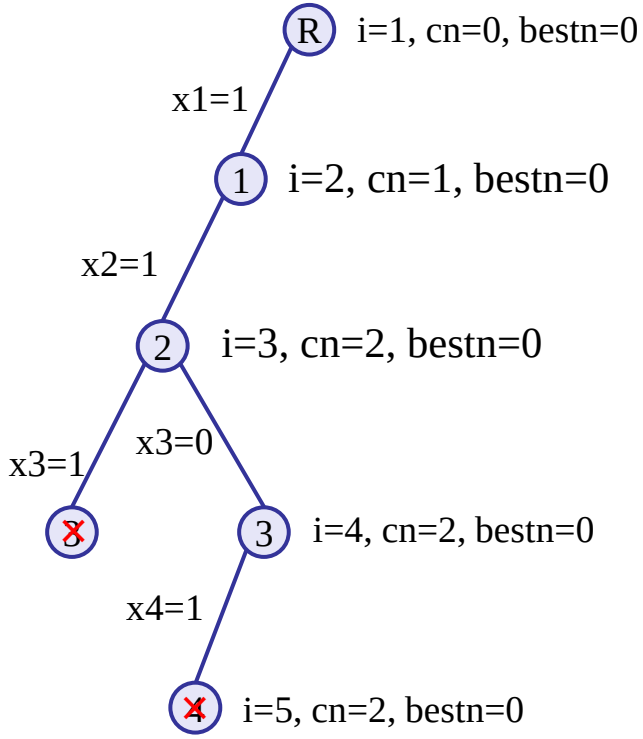
- i = 1,根结点 R 是唯一活结点，也是当前扩展结点, 当前团的顶点数 cn = 0, 此时最大团的顶点数 bestn = 0。
- i = 2，搜索至顶点 1。此时当前团的顶点数 cn = 1，最大团的顶点数 bestn = 0
- i = 3，搜索至顶点 2，此时当前团的顶点数 cn = 2，最大团的顶点数 bestn = 0.



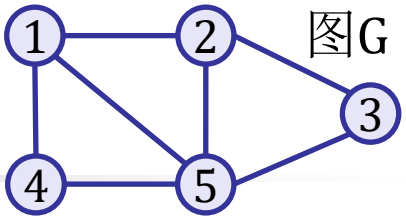
最大团问题例



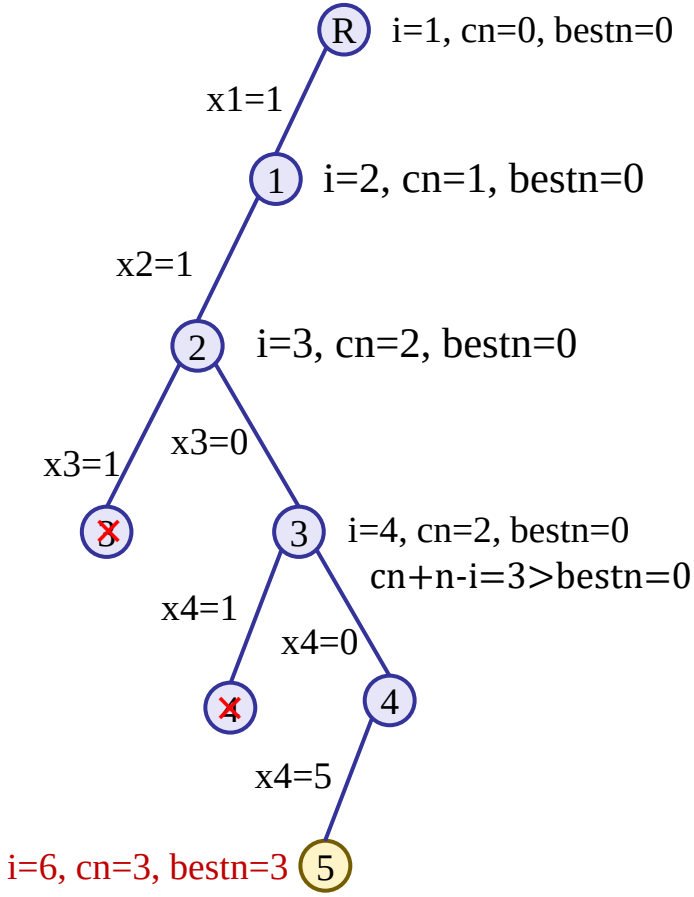
- i = 4，搜索至顶点 3，由于顶点 3 和 2 有边相连但与顶点 1 无边相连，则利用剪枝函数剪去该枝。
- 此时由于 cn + n – i = 2 + 5 – 3 = 4 > bestn = 0，则回溯到结点 2 处进入右子树，开始搜索。此时当前团的顶点数 cn = 2，最大团的顶点数 bestn = 0。
- i = 5，搜索至顶点 4，由于顶点 2 和 4 无边相连，剪去该枝。



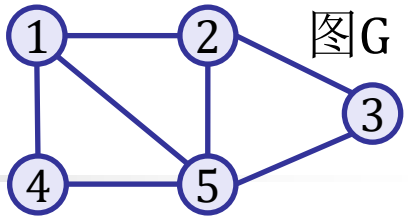
最大团问题例



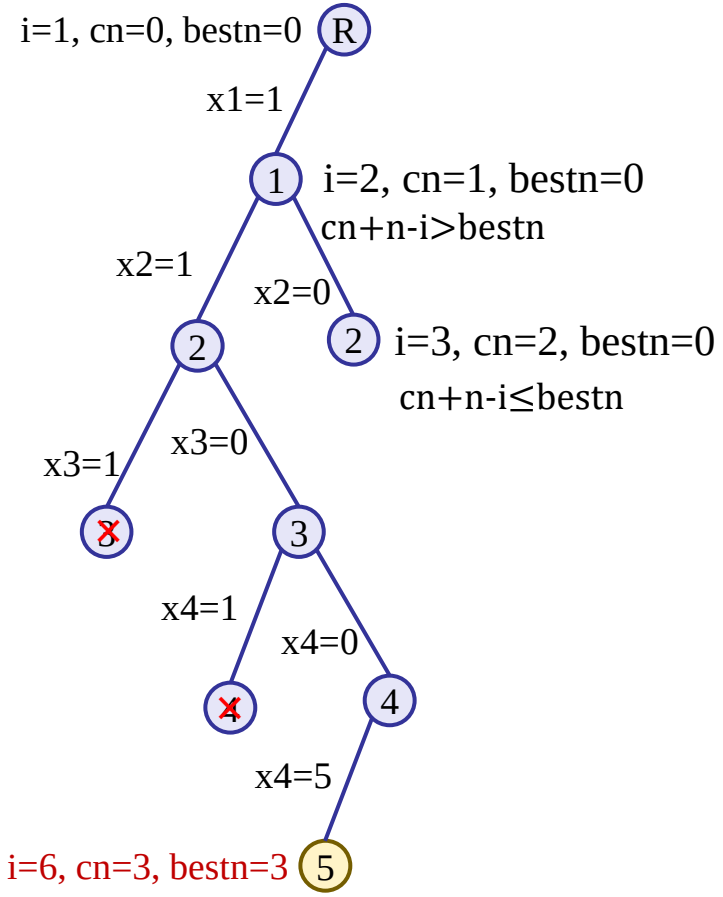
- 回溯到结点 3 处进入右子树，此时当前团的顶点数 cn = 2，最大团的顶点数 bestn = 0。
- i = 6，搜索至顶点 5，由于顶点 5 和顶点 1、2 都有边相连，进入左子树搜索。
- 由于结点 5 是一个叶结点，我们得到一个可行解。此时当前团的顶点数 cn = 3，最大团的顶点数 bestn = 3。
- v_i 的取值由顶点 1 至顶点 5 所唯一确定，即v=(1, 2, 5)。



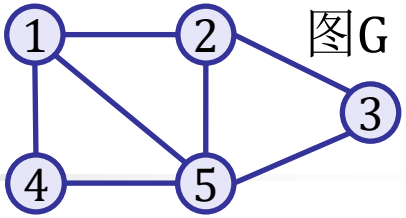
最大团问题例

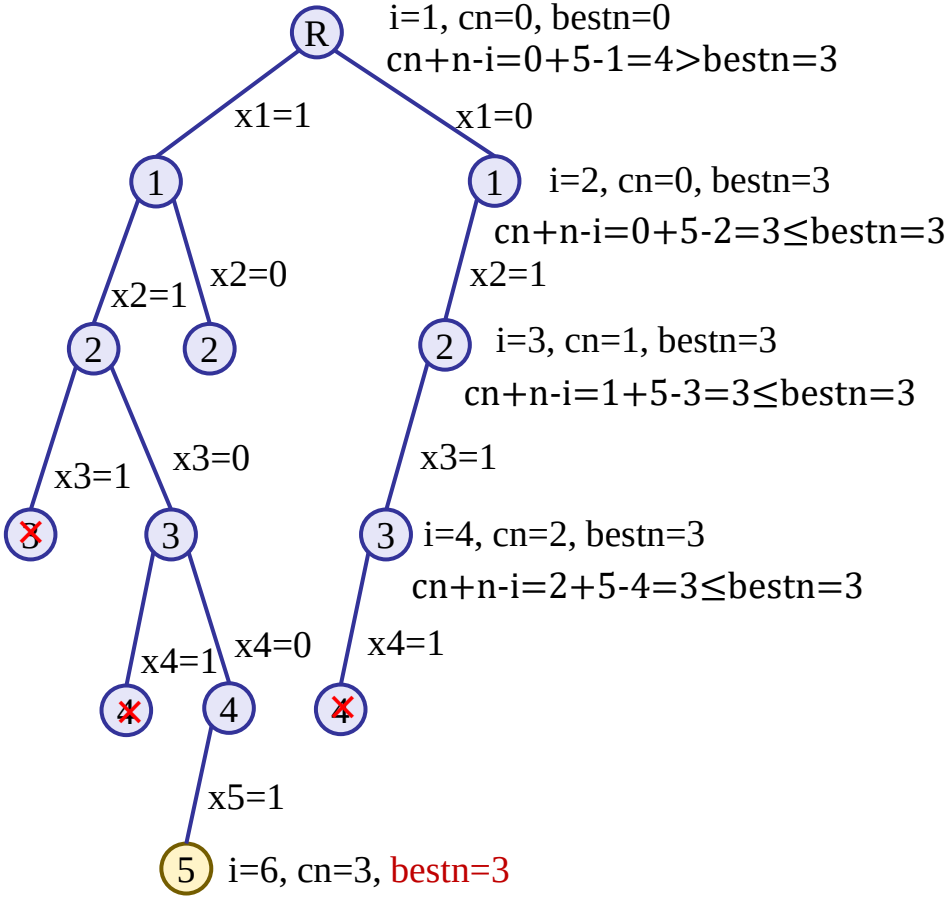


- 此时顶点 5 已不能再纵深扩展，成为死结点，我们一直回溯到结点 1 处。
- 此时 cn + n - i = 1 + 5 – 2 = 4 > bestn = 3，进入右子树。
- 由于顶点 3 和 1 不相连，剪枝。又由于 cn + n – i = 3 ≤ bestn，不展开右子树。
- 继续回溯到根结点R。沿着右子树的纵深方向移动，直至遍历整个解空间。



最大团问题例

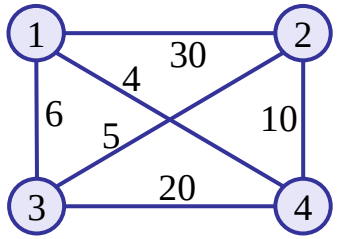


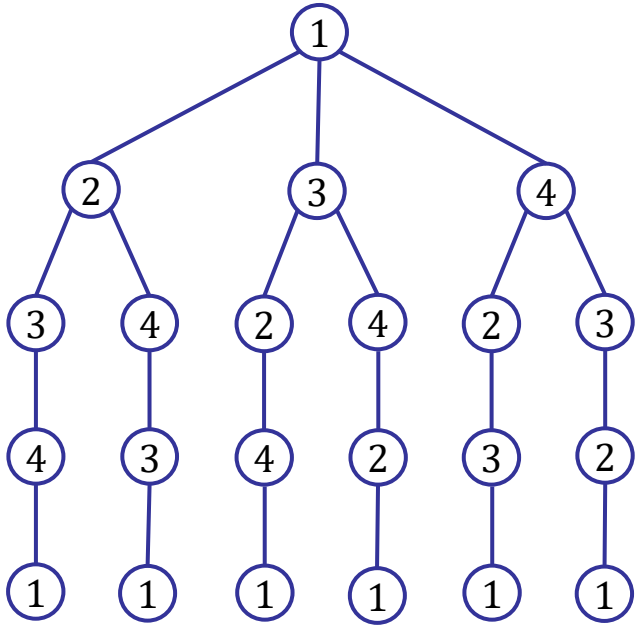


- 得到图 G 的按照1 → 2 → 3 → 4 → 5 的顺序深度搜索的最大团为 U = {1, 2, 5}。
- 最大团不唯一。
- {1, 4, 5} 和 {2, 3, 5} 也是其最大团。
- **思考**：如何得到 U = {1, 2, 5}以外的其他最大团？

旅行商问题

- 给定一个 n 节点的网络，一条包含网络中 n 个节点的环路为一条周游路线。
- **要求**：找出一条最小成本的周游路线。
- 状态空间树是一棵置换树，有 (n-1)! 个叶子节点。
- 树中每个节点上的编号代表一个城市编号。
- 算法中最难的问题之一!!



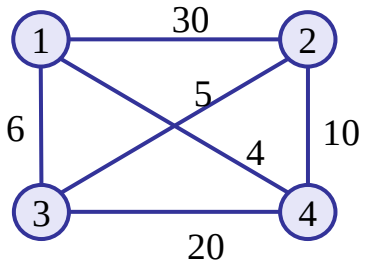


置换树：从城市1出发,经过城市2,...n,最终回到城市1

旅行商问题

- 右图的邻接矩阵为：

$$C=\begin{pmatrix} \infty & 30 & 6 & 4 \\ 30 & \infty & 5 & 10 \\ 6 & 5 & \infty & 20 \\ 4 & 10 & 20 & \infty \end{pmatrix}$$

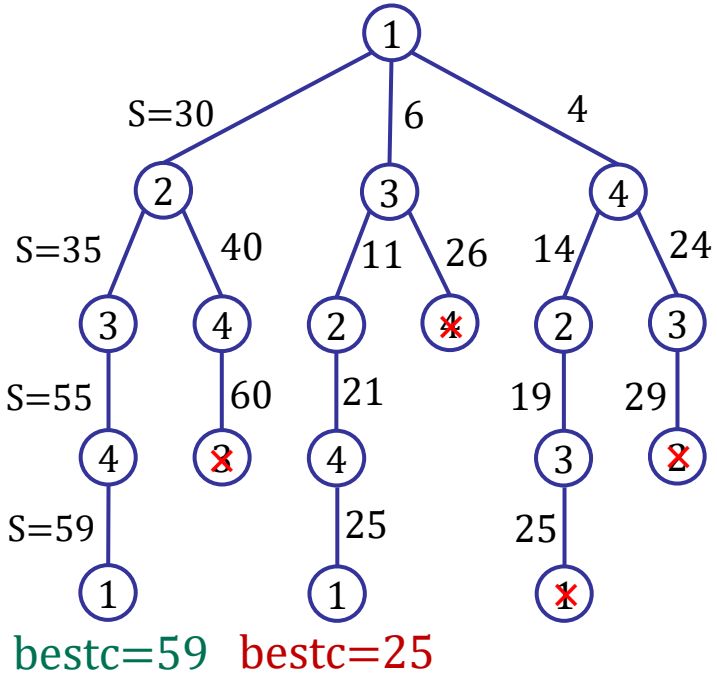


- **限界条件**：

  显示：当前节点和路径上的下一个节点之间有边。

  隐式：当前路径长度 < bestc。

- **启发式**：优先产生边长最小的子节点，有望获得更好的限界效果。



TSP的预处理程序

```
template<class T>
T AdjacencyWDigraph<T>::TSP(int V[]){
  //用回溯算法解决旅行商问题
   //返回最优旅游路径的耗费，最优路径存入v[1:n]
   //初始化
   x=new int [n+1]; //x是排列
   for (int i=1; i<=n; i++)
       x[i]=i;
   bestc=NoEdge;
   bestx=v; //使用数组v来存储最优路径
   cc=0;
   //搜索x[2:n]的各种排列
   tSP(2);
   delete [] x;
   return bestc;}
```

旅行商问题的迭代回溯算法

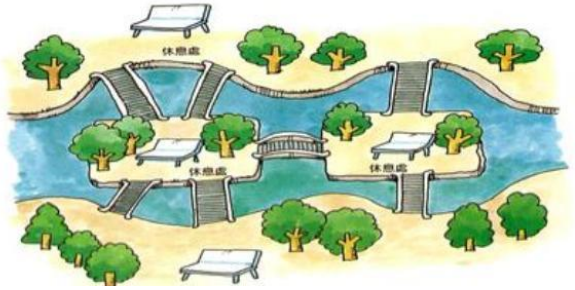
```
void AdjacencyWDigraph<T>::tSP(int i){
//旅行商问题的回溯算法
 if (i==n) {//位于一个叶子的父节点
   //通过增加两条边来完成旅行
    if (a[x[n-1]][x[n]]=NoEdge && a[x[n]][1]!=NoEdge &&
       (cc+a[x[n-1]][x[n]]+a{x[n]][1] <bestc||bestc==NoEdge))
       {//找到更优的旅行路径
          for (int j=1; j<=n; j++)
            bestx[j]=x[j];
            bestc=cc+a[x[n-1]][x[n]]+a[x[n]][1];
       }
  }
```

旅行商问题的迭代回溯算法(续)

```
        else {//尝试子树
                for (int j=i; j<=n; j++)
                //能移动到子树x[j]吗？
                if (a[x[i-1]][x[j]]!=NoEdge &&
                (cc+a[x[i-1]][x[i]]<bestc||bestc==NoEdge))
                 {//能搜索该子树
                    Swap(x[i], x[j]);
                    cc+=a[x[i-1]][x[i]];
                    tSP(i+1);
                    cc-=a[x[i-1]][x[i]];
                    Swap(x[i], x[j]); }
        }
}
```

哥尼斯堡的七座桥

- 18 世纪初普鲁士的哥尼斯堡，有一条河穿过，河上有两个小岛，有七座桥把两个岛与河岸联系起来(如右图)。



- 当地的市民有一项非常有趣的消遣活动：在星期六作一次走过所有七座桥的散步，每座桥只能经过一次而且起点与终点必须是同一地点。
- 人们对此很感兴趣，纷纷进行试验，但在相当长的时间里，始终未能解决。利用普通数学知识，如果每座桥均走一次，那这七座桥所有的走法一共有 5040 种。这么多情况，要一一试验，这将会是很大的工作量。但怎么才能找到成功走过每座桥而不重复的路线呢？因而形成了著名的“哥尼斯堡七桥问题”。

哥尼斯堡七桥问题

- 1735 年，有几名大学生写信给当时正在俄罗斯的彼得斯堡科学院任职的天才数学家欧拉，请他帮忙解决这一问题。
- 欧拉在亲自观察了哥尼斯堡七桥后，认真思考走法，但始终没能成功，于是他怀疑七桥问题是不是原本就无解呢？
- 1736 年，在经过一年的研究之后，29 岁的欧拉提交了《哥尼斯堡七座桥》的论文，提出了欧拉定理，圆满解决了这一问题，同时开创了数学新一分支 — 图论，并未后来的数学新分支 — 拓扑学的建立奠定了基础。

哥尼斯堡的七座桥

- 给定无孤立结点图 G，若存在一条路，经过图中每边一次且仅一次，该条路称为**欧拉路**；
- 若存在一条回路，经过图中每边一次且仅一次，该回路称为**欧拉回路**。
- **欧拉定理**：
  1. 凡是由偶点组成的连通图，一定可以一笔画成。画时可以把任一偶点为起点，最后一定能以这个点为终点画完此图。
  2. 凡是只有两个奇点的连通图（其余都为偶点），一定可以一笔画成。画时必须把一个奇点为起点，另一个奇点为终点。
  3. 其他情况的图都不能一笔画出。(奇点数除以二便可算出此图需几笔画成。)