USTC

Chapter 6
Backtracking

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn/

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
- 通过应用范例学习回溯法的设计策略
  - 装载问题
  - 批处理作业调度
  - 符号三角形问题

- n后问题
- 0-1背包问题
- 最大团问题
- 图的m着色问题
- 旅行售货员问题
- 圆排列问题
- 电路板排列问题
- 连续邮资问题

回溯法

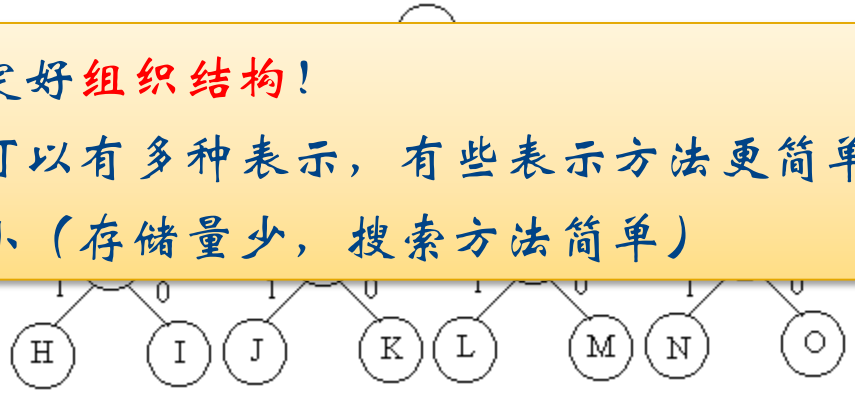
- 当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时
- 回溯法的基本做法是搜索
  - 一种组织得井井有条的，能避免不必要搜索的穷举式搜索法
  - 适用于解一些组合数相当大的问题
- 回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树
  - 算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解
  - 如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索

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 $(x_1,x_2,\ldots,x_n)$ 的形式
  - 显约束：对分量 x_i 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：要确定好**组织结构**！

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

- 节点类型
  - 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

生成问题状态的基本方法

- 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

回溯法的基本思想

- 解空间
  - 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
- 搜索
  - 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

常用剪枝函数：

1）用约束函数在扩展结点处剪去不满足约束的子树

2）用限界函数剪去得不到最优解的子树

回溯法的基本思想

- 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 $O(h(n)!)$ 的内存空间

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

```
1.    void backtrack (int t)
2.    {
3.        if (t>n) output(x);
4.        else
5.          for (int i=f(n,t);i<=g(n,t);i++) {
6.              x[t]=h(i);
7.              if (constraint(t)&&bound(t)) backtrack(t+1);
8.          }
9.    }
```

叶节点

子节点遍历

解赋值

剪枝函数

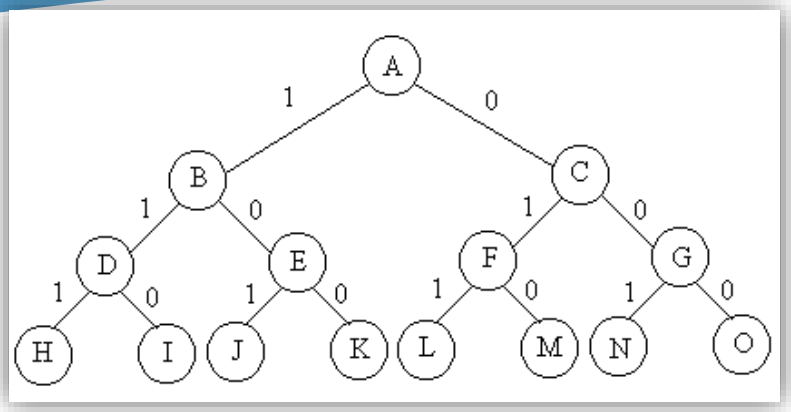
进入子节点

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

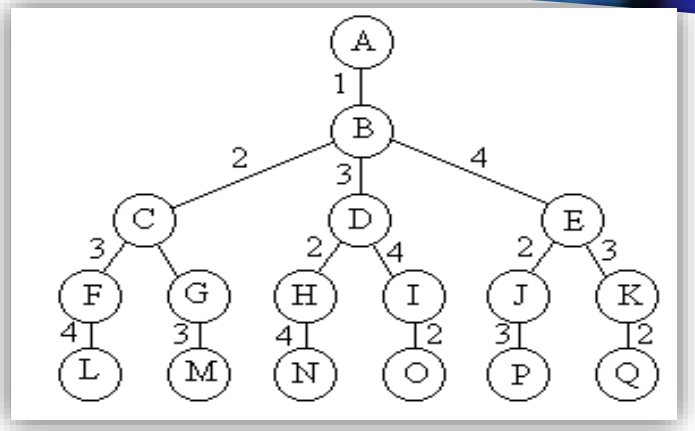
```
1.  void iterativeBacktrack ()
2.  {
3.        int t=1;
4.        while (t>0) {
5.             if (f(n,t)<=g(n,t))
6.                  for (int i=f(n,t);i<=g(n,t);i++) {
7.                       x[t]=h(i);
8.                       if (constraint(t)&&bound(t)) {
9.                            if (solution(t)) output(x);
10.                      else t++;}
11.                 }
12.       else t--;
13.       }
14. }
```

子集树与排列树



遍历子集树需$O(2^n)$计算时间

```
1.  void backtrack (int t)
2.  {
3.    if (t>n) output(x);
4.    else
5.        for (int i=0;i<=1;i++) {
6.            x[t]=i;
7.            if (legal(t)) backtrack(t+1);
8.        }
9.  }
```



遍历排列树需要$O(n!)$计算时间

```
1.  void backtrack (int t)
2.  {
3.     if (t>n) output(x);
4.     else
5.        for (int i=t;i<=n;i++) {
6.           swap(x[t], x[i]);
7.           if (legal(t)) backtrack(t+1);
8.           swap(x[t], x[i]);
9.        }
10. }
```

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

$$\sum_{i=1}^{n} w_i \le c_1 + c_2$$

- 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

装载问题

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：

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

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

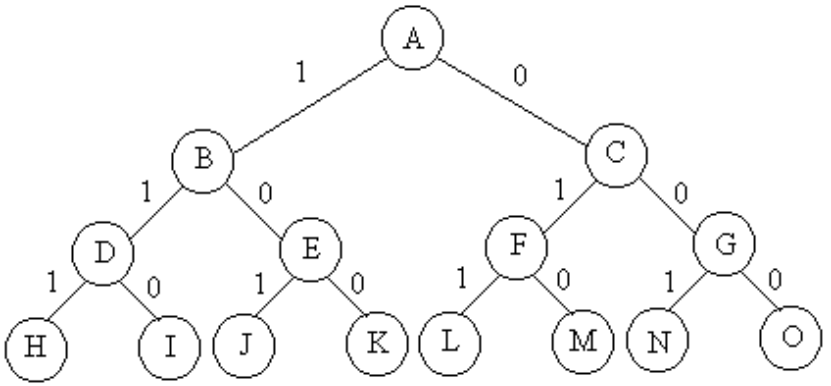
$$x_i \in \{0,1\}, 1 \le i \le n$$

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

- 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 $r \le$ 当前最优载重量 $bestw$

装载问题

```
1.   void backtrack (int i)  // 搜索第i层结点
2.   {
3.       if (i > n)  // 到达叶结点
4.           更新最优解 bestx,bestw;return;
5.       r -= w[i];
6.       if (cw + w[i] <= c) {// 搜索左子树
7.           x[i] = 1;
8.           cw += w[i];
9.           backtrack(i + 1);
10.          cw -= w[i];
11.      }
12.      if (cw + r > bestw)  { // 搜索右子树
13.          x[i] = 0;
14.          backtrack(i + 1);
15.      }
16.      r += w[i];
17.  }
```



如果不满足，则直接截断右子树

P.121

批处理作业调度

- 问题描述
  - 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 上的完成处理时间
  - 所有作业在机器 2 上完成处理时间和称为该作业调度的完成时间和

$$f = \sum_{i=1}^{n} F_{2i}$$

- 批处理作业调度问题要求对于给定的 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

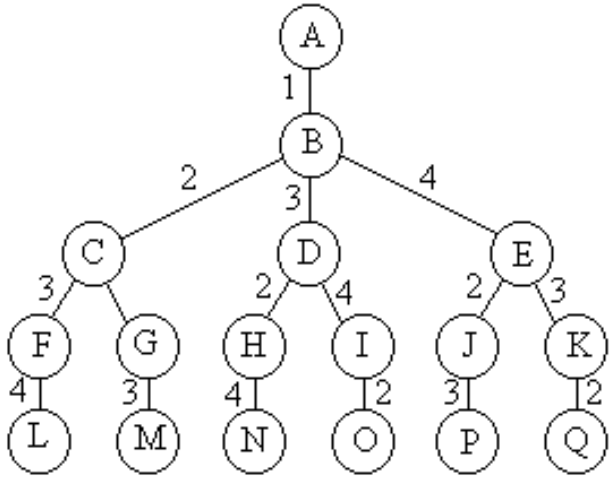
批处理作业调度

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

- 这3个作业的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
  - 它们所相应的完成时间和分别是19，18，20，21，19，19
- →最佳调度方案是1,3,2，其完成时间和为18

批处理作业调度

- 解空间：排列树




```
1.  void Flowshop::Backtrack(int i)
2.  {
3.    if (i > n) {
4.        for (int j = 1; j <= n; j++)
5.            bestx[j] = x[j];
6.        bestf = f;
7.    }
8.    else
9.        for (int j = i; j <= n; j++) {
10.           f1+=M[x[j]][1];
11.           f2[i]=((f2[i-1]>f1)?f2[i-1]:f1)+M[x[j]][2];
12.           f+=f2[i];
13.           if (f < bestf) {
14.               Swap(x[i], x[j]);
15.               Backtrack(i+1);
16.               Swap(x[i], x[j]);
17.           }
18.           f1- =M[x[j]][1];
19.           f- =f2[i];
20.       }
21. }
```

```
1.  class Flowshop {
2.    friend Flow(int**, int, int []);
3.    private:
4.      void Backtrack(int i);
5.      int  **M,          // 各作业所需的处理时间
6.           *x,           // 当前作业调度
7.           *bestx,       // 当前最优作业调度
8.           *f2,          // 机器2完成处理时间
9.            f1,          // 机器1完成处理时间
10.           f,           // 完成时间和
11.           bestf,       // 当前最优值
12.           n;           // 作业数
13. };
```

符号三角形问题

下图是由 14 个“+”和 14 个“-”组成的符号三角形，2 个同号下面都是“+”，2 个异号下面都是“-”：

- 在一般情况下，符号三角形的第一行有 n 个符号
- 符号三角形问题要求对于给定的 n，计算有多少个不同的符号三角形，使其所含的“+”和“-”的**个数相同**

```
+ + - + - + +
 + - - - - +
  - + + + -
   - + + -
    - + -
     - -
      +
```

符号三角形问题

- 解向量：用 n 元组 $x[1:n]$ 表示符号三角形的第一行
- 可行性约束函数：当前符号三角形所包含的“+”个数与“-”个数均不超过 $n*(n+1)/4$
- 无解的判断：$n*(n+1)/2$ 为奇数

```
+ + - + - + +
 + - - - - +
  - + + + -
   - + + -
    - + -
     - -
      +
```

符号三角形问题

```
1.  void Triangle::Backtrack(int t)
2.  {
3.      if ((count>half)||(t*(t-1)/2-count>half)) return;
4.      if (t>n) sum++;
5.      else
6.          for (int i=0;i<2;i++) {
7.              p[1][t]=i;
8.              count+=i;
9.              for (int j=2;j<=t;j++) {
10.                 p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
11.                 count+=p[j][t-j+1];
12.             }
13.             Backtrack(t+1);
14.             for (int j=2;j<=t;j++)
15.                 count-=p[j][t-j+1];
16.             count-=i;
17.         }
18. }
```

约束条件

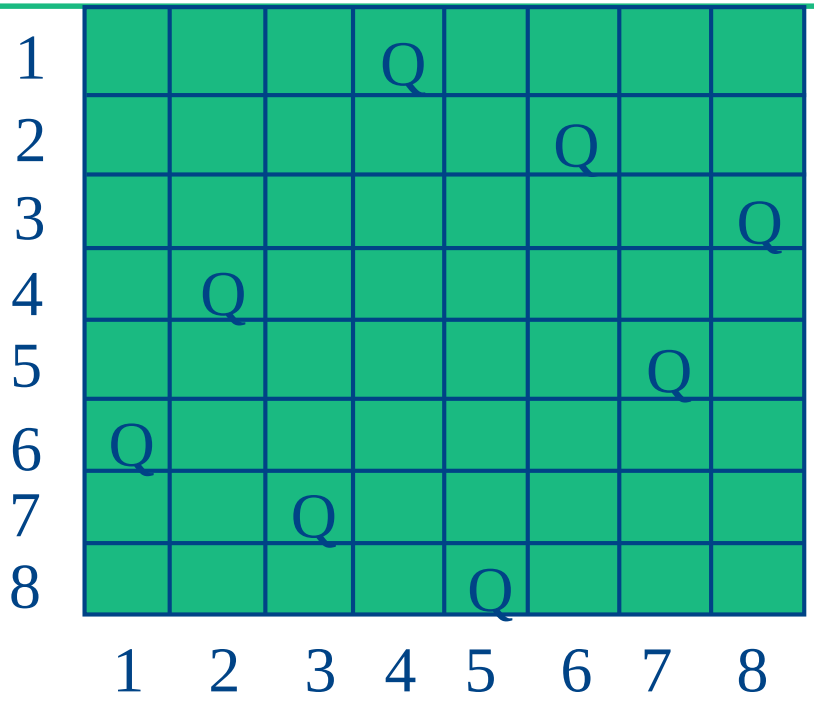
复杂度分析

计算可行性约束需要 $O(n)$ 时间，在最坏情况下有 $O(2^n)$ 个结点需要计算可行性约束，故解符号三角形问题的回溯算法所需的计算时间为 $O(n2^n)$

count ——可行性判断的关键计数器 O(n)

n 后问题

在 $n \times n$ 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 $n \times n$ 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：$(x_1, x_2, \ldots, x_n)$
  - x_i 表示皇后放在棋盘的第 i 行第 x_i 列
- 显约束：$x_i=1,2, \ldots ,n$
- 隐约束：
  - 不同列：$x_i \neq x_j$
  - 不处于同一正、反对角线：$|i-j| \neq |x_i-x_j|$

```
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])) return false;
5.      return true;
6.  }

7.  void Queen::Backtrack(int t)
8.  {
9.      if (t>n) sum++;
10.     else
11.         for (int i=1;i<=n;i++) {
12.             x[t]=i;
13.             if (Place(t)) Backtrack(t+1);
14.         }
15. }
```

子集树回溯法？
排列树回溯法？

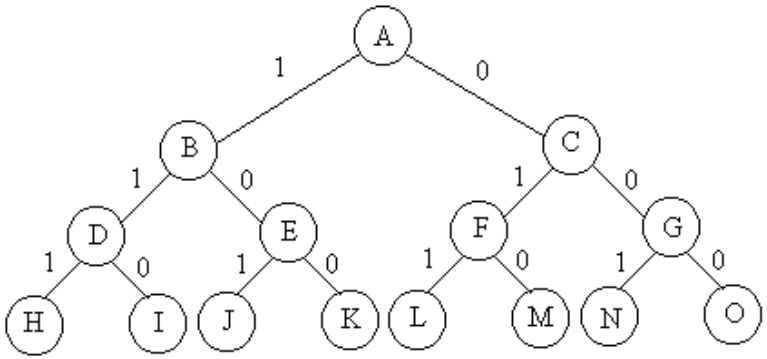
0-1背包问题

- 解空间：子集树
- 可行性约束函数：

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

- 上界函数：

$$\sum_{j=1}^{i} w_i v_i + r \le bestp$$




```
1.  template<class Typew, class Typep>
2.  Typep Knap<Typew, Typep>::Bound(int i)
3.  {   // 计算上界
4.      Typew cleft = c - cw;  // 剩余容量
5.      Typep b = cp;
6.      // 以物品单位重量价值递减序装入物品
7.      while (i <= n && w[i] <= cleft) {
8.          cleft -= w[i];
9.          b += p[i];
10.         i++;
11.     }
12.     // 装满背包
13.     if (i <= n) b += p[i]/w[i] * cleft;
14.     return b;
15. }
```

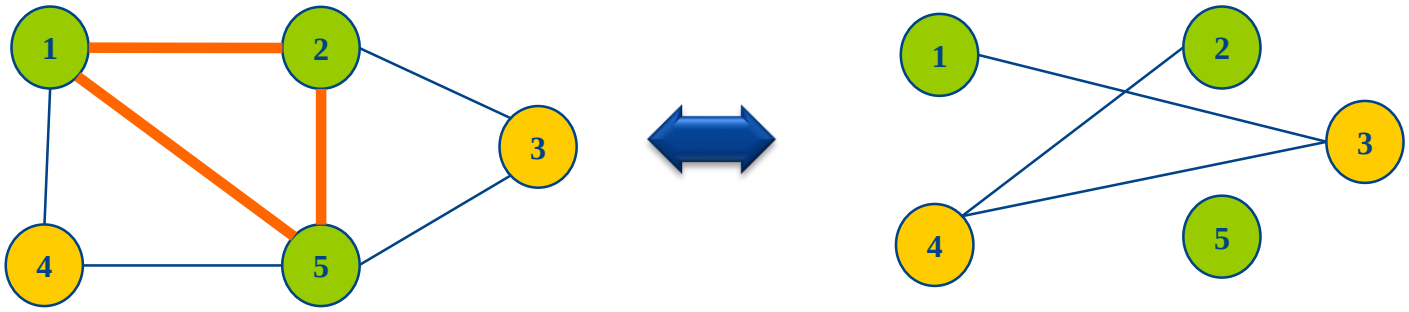
最大团问题

- 团
  - 给定无向图 $G=(V, E)$，如果 $U \subseteq V$，且对任意 $u,v \in U$ 有 $(u,v) \in E$，则称 U 是 G 的完全子图
  - 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
- 独立集
  - 如果 $U \subseteq V$ 且对任意 $u,v \in U$ 有 $(u,v) \notin E$，则称 U 是 G 的空子图
  - 空子图 U 是 G 的独立集当且仅当 U 不包含在 G 的更大空子图中
  - G的最大独立集是 G 中所含顶点数最多的独立集

最大团问题

- 对于任一无向图 $G=(V, E)$，其补图 $G'=(V_1, E_1)$ 定义为：$V_1=V$，且 $(u,v)\in E_1$ 当且仅当 $(u,v)\notin E$

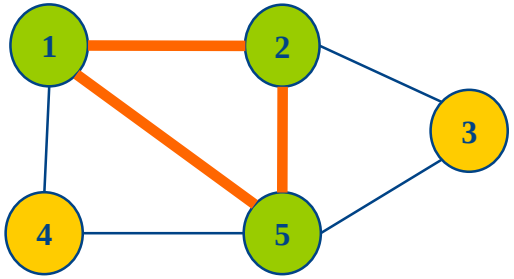
U 是 G 的最大团当且仅当 U 是 G' 的最大独立集



最大团问题

```
1.   void Clique::Backtrack(int i)
2.   {  // 计算最大团
3.      if (i > n) {// 到达叶结点
4.          for (int j = 1; j <= n; j++) bestx[j] = x[j];
5.          bestn = cn;   return;}
6.     // 检查顶点 i 与当前团的连接
7.     int OK = 1;
8.     for (int j = 1; j < i; j++)
9.         if (x[j] && a[i][j] == 0) {
10.            // i与j不相连
11.            OK = 0;  break;
12.        }
13.    if (OK) {// 进入左子树
14.        x[i] = 1;  cn++;
15.        Backtrack(i+1);
16.        x[i] = 0;  cn--;
17.    }
18.    if (cn + n - i > bestn) { // 进入右子树
19.        x[i] = 0;
20.        Backtrack(i+1);
21.    }
22.  }
```

- 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



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

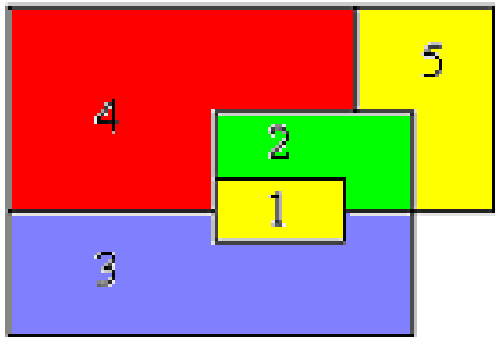
- 选择合适的搜索顺序，可以使得上界函数更有效的发挥作用
  - 例如在搜索之前可以将顶点按度从小到大排序，这在某种意义上相当于给回溯法加入了启发性

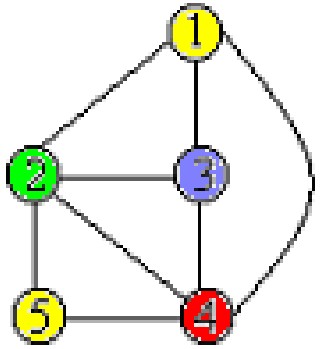
- 定义 $S_i=\{v_i, v_{i+1},...,v_n\}$，依次求出 $S_n,S_{n-1},...,S_1$ 的解，从而得到一个更精确的上界函数，若 $cn+S_i<=\max$ 则剪枝
  - 同时注意到：从 S_{i+1} 到 S_i，如果找到一个更大的团，那么 v_i 必然属于找到的团，此时有 $S_i=S_{i+1}+1$，否则 $S_i=S_{i+1}$

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

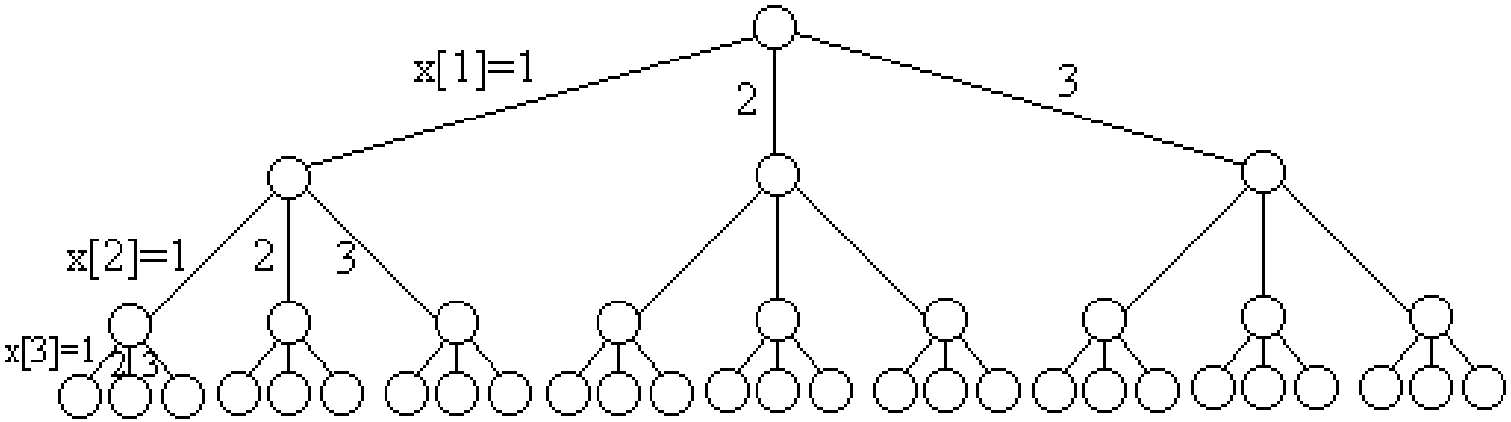
平面图
四色猜想





图的m着色问题

- 解向量：$(x_1, x_2, \ldots, x_n)$ 表示顶点 i 所着颜色 $x[i]$
- 可行性约束函数：顶点 i 与已着色的相邻顶点颜色不重复



图的 m 着色问题

```
1.  void Color::Backtrack(int t)
2.  {
3.      if (t>n) {
4.          sum++;
5.          for (int i=1; i<=n; i++)
6.              cout << x[i] << ' ';
7.          cout << endl;
8.      }
9.      else
10.         for (int i=1;i<=m;i++) {
11.             x[t]=i;
12.             if (Ok(t)) Backtrack(t+1);
13.         }
14. }

15. bool Color::Ok(int k)
16. {   // 检查颜色可用性
17.     for (int j=1;j<=n;j++)
18.         if ((a[k][j]==1)&&(x[j]==x[k])) return false;
19.     return true;
20. }
```

复杂度分析

图m可着色问题的解空间树中内结点个数是 $\sum_{i=0}^{n-1} m^i$

对于每一个内结点，在最坏情况下，用 Ok 检查当前扩展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

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

旅行售货员问题

- 某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)，他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小
- 问题建模
  - 路线是一个带权图，图中各边的费用（权）为正数
  - 图的一条周游路线是包括 V 中的每个顶点在内的一条回路，周游路线的费用是这条路线上所有边的费用之和
  - 旅行售货员问题的解空间可以组织成一棵树，从树的根结点到任一叶结点的路径定义了图的一条周游路线——旅行售货员问题要在图 G 中找出费用最小的周游路线

旅行售货员问题

- 解空间：排列树

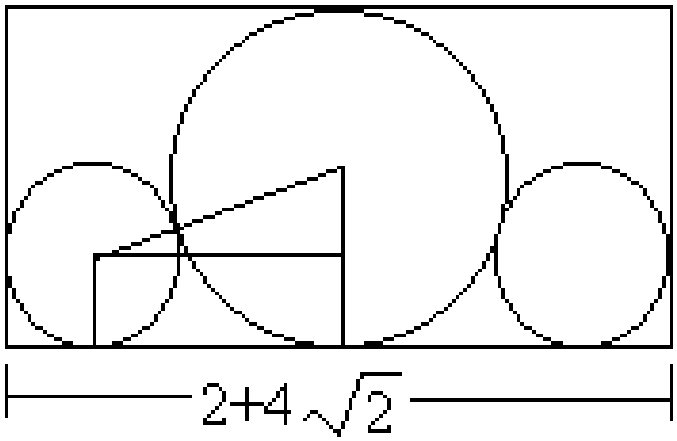
```
1.  template<class Type>
2.  void Traveling<Type>::Backtrack(int i)
3.  { // 获得一个排列
4.    if (i == n) {
5.      if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
6.        (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc == NoEdge)) { // 当前排列更优
7.        for (int j = 1; j <= n; j++)  bestx[j] = x[j];
8.        bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];}
9.      }
10.   else { // 继续递归
11.     for (int j = i; j <= n; j++)
12.       // 是否可进入x[j]子树?
13.       if (a[x[i-1]][x[j]] != NoEdge &&
14.         (cc + a[x[i-1]][x[i]] < bestc || bestc == NoEdge)) { // 可能有更优解，搜索子树
15.           Swap(x[i], x[j]);
16.           cc += a[x[i-1]][x[i]];
17.           Backtrack(i+1);
18.           cc -= a[x[i-1]][x[i]];
19.           Swap(x[i], x[j]);
20.       }
21.   }
22. }
```

复杂度分析

算法 **backtrack** 在最坏情况下可能需要更新当前最优解 $O((n-1)!)$ 次，每次更新 bestx 需计算时间 $O(n)$，从而整个算法的计算时间复杂性为 $O(n!)$

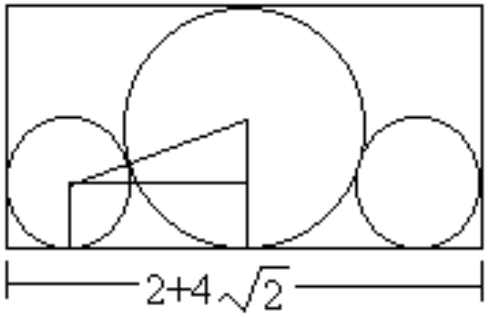
圆排列问题

- 问题描述
  - 给定 n 个大小不等的圆 $c_1,c_2,\ldots,c_n$，现要将这 n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切
  - 圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列
- 示例
  - 当 n=3，且所给的 3 个圆的半径分别为 1，1，2 时，这 3 个圆的最小长度的圆排列如图所示，其最小长度为 $2+4\sqrt{2}$



圆排列问题

```
1.  void Circle::Backtrack(int t)
2.  {
3.      if (t>n) Compute();
4.      else
5.        for (int j = t; j <= n; j++) {
6.          Swap(r[t], r[j]);
7.          float centerx=Center(t);
8.          if (centerx+r[t]+r[1]<min) { //下界约束
9.            x[t]=centerx;
10.           Backtrack(t+1);
11.         }
12.         Swap(r[t], r[j]);
13.       }
14. }
```




```
1.  float Circle::Center(int t)
2.  { // 计算当前所选择圆的圆心横坐标
3.    float temp=0;
4.    for (int j=1;j<t;j++) { // 扫描前面所有圆
5.      float valuex=x[j]+2.0*sqrt(r[t]*r[j]);
6.      if (valuex>temp) temp=valuex;
7.    }
8.    return temp;
9.  }
```

```
1.  void Circle::Compute(void)
2.  { // 计算当前圆排列的长度
3.    float low=0,  high=0;
4.    for (int i=1;i<=n;i++) {
5.      if (x[i]-r[i]<low) low=x[i]-r[i]; // 前端点
6.      if (x[i]+r[i]>high) high=x[i]+r[i];  // 后端点
7.    }
8.    if (high-low<min)
9.      min=high-low;
10. }
```

圆排列问题

复杂度分析

由于算法backtrack在最坏情况下可能需要计算 $O(n!)$ 次当前圆排列长度，每次计算需 $O(n)$ 计算时间，从而整个算法的计算时间复杂性为 $O((n+1)!)$

- 许多改进的余地
  - 象 $1,2,\ldots,n\text{-}1,n$ 和 $n,n\text{-}1,\ldots,2,1$ 这种互为镜像的排列具有相同的圆排列长度，只计算一个就够了，可减少约一半的计算量
  - 另一方面，如果所给的 n 个圆中有 k 个圆有相同的半径，则这 k 个圆产生的 $k!$ 个完全相同的圆排列，只计算一个就够了

连续邮资问题

- 假设国家发行了 n 种不同面值的邮票，并且规定每张信封上最多只允许贴 m 张邮票
- 连续邮资问题要求对于给定的 n 和 m 的值，给出**邮票面值的最佳设计**——在 1 张信封上可贴出从邮资 1 开始，增量为 1 的**最大连续邮资区间**

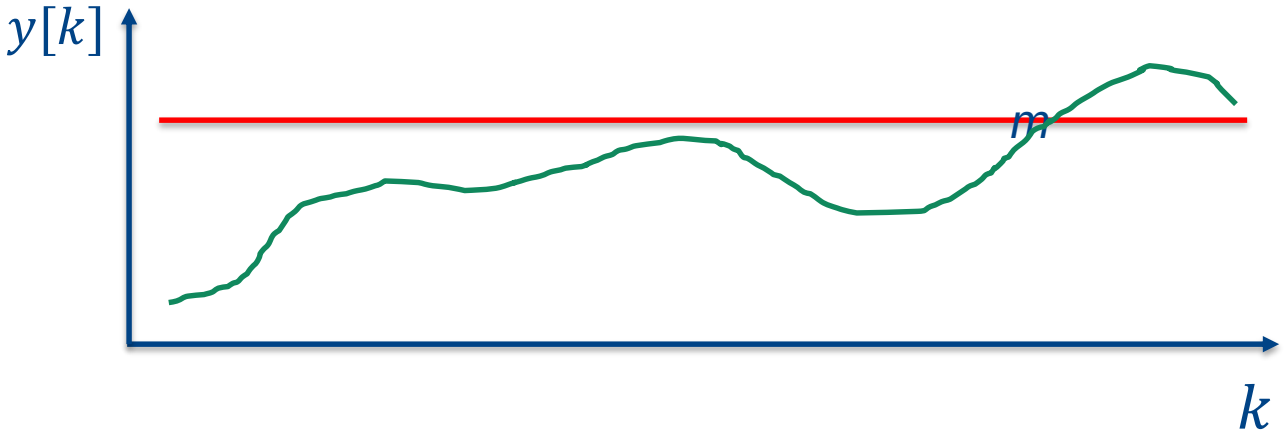
例如，当 n=5 和 m=4 时，面值为 (1,3,11,15,32) 的 5 种邮票可以贴出邮资的最大连续邮资区间是 1 到 70

连续邮资问题

- 解向量
  - 用 n 元组 $x[1:n]$ 表示 n 种不同的邮票面值
  - 约定它们从小到大排列
  - $x[1]=1$ 是唯一的选择
- 可行性约束函数
  - 已选定 $x[1:i-1]$，最大连续邮资区间是$[1:r]$
  - 接下来 $x[i]$ 的可取值范围是 $[x[i-1]+1:r+1]$
  - →连续约束下最大化 r

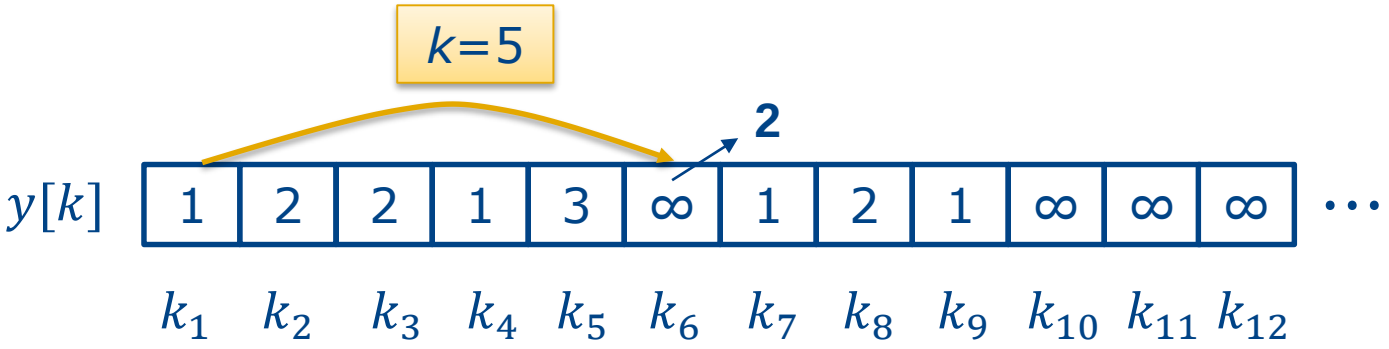
连续邮资问题

- 如何确定 r 的值？
  - 计算 $x[1:i]$ 的最大连续邮资区间在本算法中被频繁使用到，因此势必要找到一个高效的方法
  - 考虑到直接递归的求解复杂度太高，我们尝试计算用不超过 m 张面值为 $x[1:i]$ 的邮票贴出邮资 k 所需的最少邮票数 $y[k]$
  - 通过 $y[k]$ 可以很快推出 r 的值



连续邮资问题

- $y[k]$ 可以通过递推在 $O(n)$ 时间内解决：



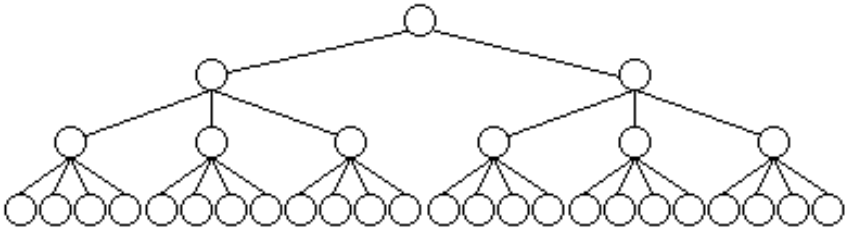

```
1.  for (int j=0; j<= x[i-2]*(m-1); j++)
2.      if (y[j]<m)
3.          for (int k=1;k<=m-y[j];k++)
4.              if (y[j]+k<y[j+x[i-1]*k]) y[j+x[i-1]*k]=y[j]+k;

5.  while (y[r]<maxint) r++;
```

回溯法效率分析

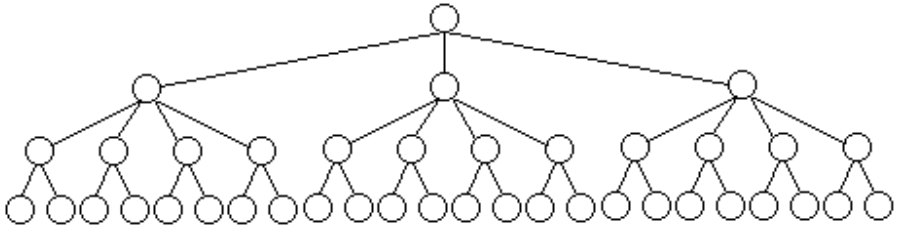
- 通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：
  - (1) 产生 $x[k]$ 的时间
  - (2) 满足显约束的 $x[k]$ 值的个数
  - (3) 计算约束函数 constraint 的时间
  - (4) 计算上界函数 bound 的时间
  - (5) 满足约束函数和上界函数约束的所有 $x[k]$ 的个数
- 好的约束函数能显著地减少所生成的结点数，但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷！

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

Next

- 分支界限法
  - Branch and bounding