湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课 程 名 称	算法设计与分析
章节、专题	回溯方法
教学目标及基 本 要 求	熟悉回溯方法的基本思想和工作流程，能够使用回溯方法求解各种子集类问题和排列类问题。
教 学 重 点	子集树和排列树的搜索，旅行商问题和最大团问题的回溯算法
教 学 难 点	旅行商问题、子集和问题和0/1背包问题的回溯算法
教学内容与时 间 分 配	（1）解空间与回溯方法概述（0.8课时） （2）子集树和排列树的搜索（0.5课时） （3）旅行商问题的回溯算法（0.4课时） （4）最大团问题的回溯算法（0.5课时） （5）子集和问题的回溯算法（0.4课时） （6）0/1背包问题的回溯算法（0.5课时） （7）着色问题的回溯算法（0.4课时） 共计4课时。
习　　题	第7.8.1节（基础知识题），第7.8.2节（实验题）。

第7章　回溯方法

本章首先介绍回溯方法的基本思想，包括相关概念、回溯方法的工作流程、用回溯方法解题的步骤以及子集树和排列树的搜索等，并构建子集类问题和排列类问题回溯算法的公式化描述。然后介绍旅行商问题、子集和问题、0/1背包问题和着色问题等问题的回溯算法。

- 解空间与回溯方法概述
- 子集树和排列树的搜索
- 旅行商问题的回溯算法
- 最大团问题的回溯算法
- 子集和问题的回溯算法
- 0/1背包问题的回溯算法
- 着色问题的回溯算法

7.1 解空间与回溯方法概述

回溯方法有“通用解题方法”之称。使用回溯方法设计的算法称为回溯算法。应用回溯方法解题时，首先应该明确问题的解空间。

7.1.1 解空间

一个复杂问题的解决往往由多部分构成，即一个大的解决方案可以看作由若干个小决策组成，很多时候它们构成一个决策序列。解决一个问题的所有可能的决策序列构成该问题的解空间，解空间中满足约束条件的决策序列称为可行解，使目标达到最优的可行解称为最优解。

在定义问题的解空间以后，还应该将解空间很好地组织起来，使得使用回溯方法能够很方便地搜索整个解空间。通常将解空间组织成树的形式，称为解空间树。

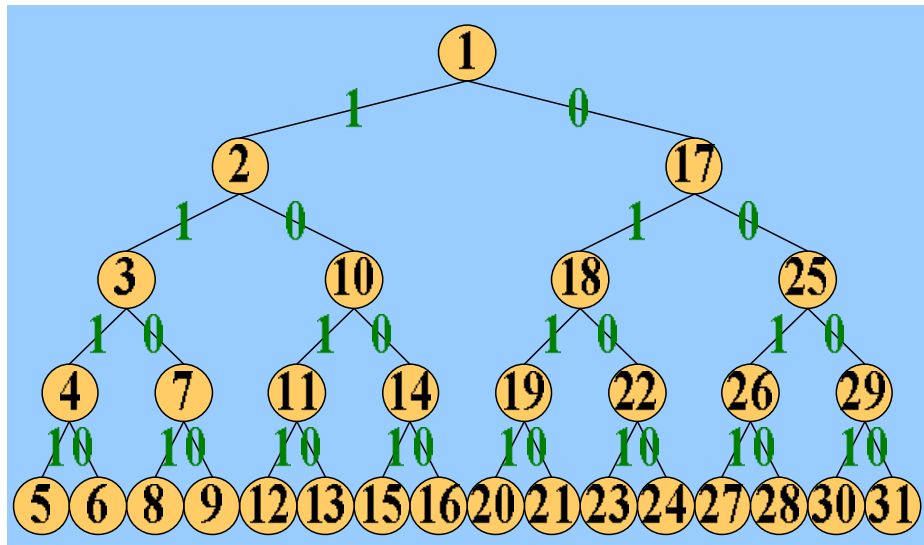
【注】为了区分图中的顶点和解空间树中的顶点，通常将解空间树中的顶点称为结点。

7.1.2 三个典型问题的解空间

1. 子集和问题

【问题描述】设 $W=\{w_0,w_1,\ldots,w_{n-1}\}$ 是一个由 n 个正整数构成的集合，M 是一个正整数，请判断 W 是否存在满足 $\sum S=M$（S 中的正整数之和等于 M）的子集 S。

【解空间】子集和问题的每一个解都可以表示成一个0/1数组 $(x_0,x_1,\ldots,x_{n-1})$，依据 w_i 是否属于 S，x_i 分别取值1或0。解空间中共有 2^n 个元素，结构是一棵完全二叉树（如图7-1所示），从根到叶的每条路径都构成一个决策序列。



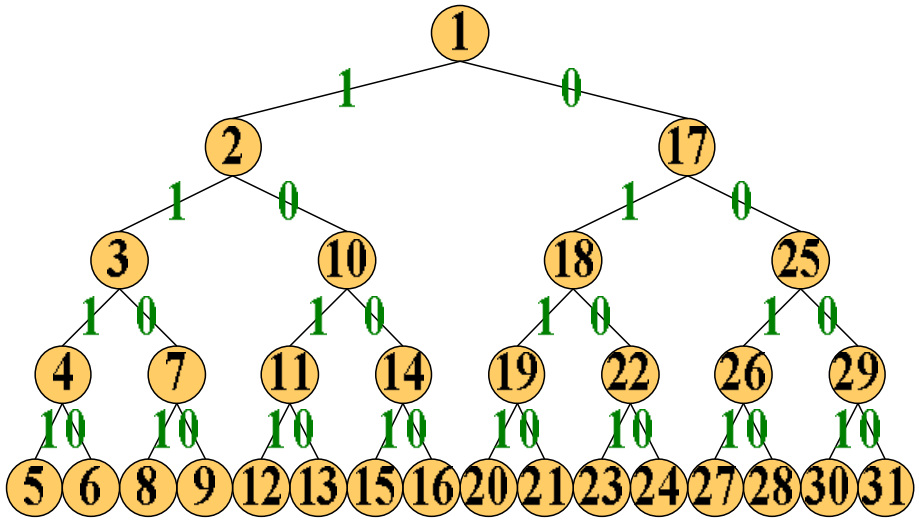


图7-1　子集和问题解空间

2. 旅行商问题

【问题描述】某售货员要到若干个城市去推销商品。已知各个城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市正好一次，最后回到驻地的路线，使得总路程（或总旅费）最小。这就是旅行商问题，简称为TSP。例如，考虑图7-2所示的实例，假定城市0是驻地。

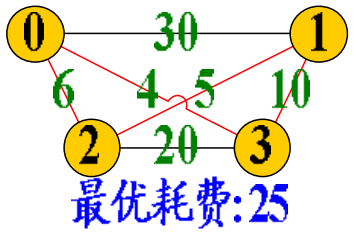
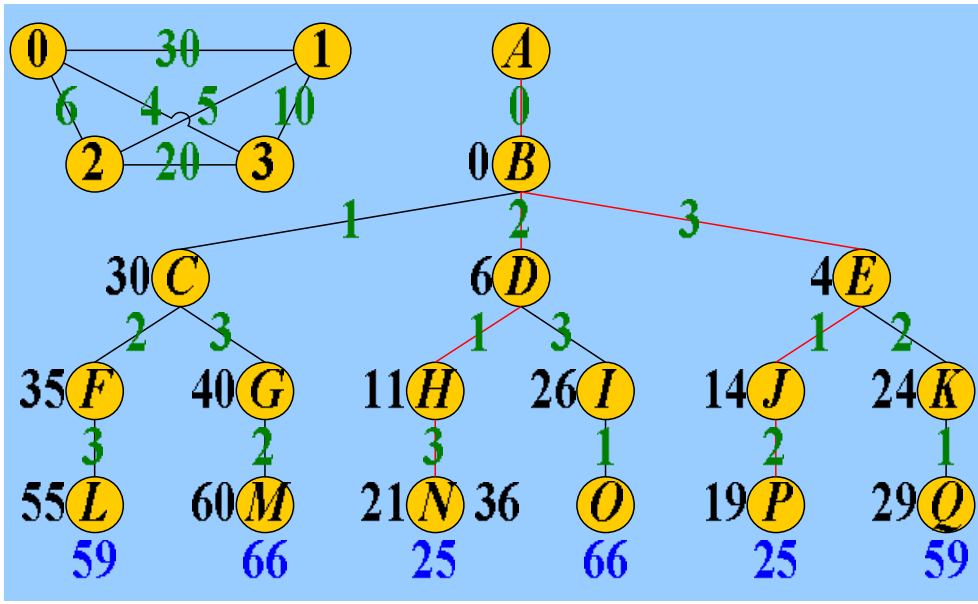


图7-2　一个TSP实例

【解空间】如果将问题中的所有城市从0开始统一编号，则问题的每一个解都可以一个表示成整数数组 $(x_0, x_1, \ldots, x_{n-1})$，其中，$x_i$ 表示路线中第 i 站选取的城市，x_0 表示驻地，最后由城市 x_{n-1} 返回驻地 x_0。因为 $(x_0, x_1, \ldots, x_{n-1})$ 正好是 $(0,1,\ldots,n-1)$ 的一个排列，所以解空间树是一棵排列树。在该排列树中，从根到叶的每条路径都构成一个决策序列，即 $(0,1,\ldots,n-1)$ 的一个排列。



对于图7-2所示的TSP实例，如果假定城市0是驻地，则解空间如图7-3所示。因为城市0是驻地，不参与排列，所以该解空间树实际上是一棵3级排列树。

【注】在图7-3中，结点左侧的数字表示该结点的费用，叶结点下侧的数字表示相应回路的费用，连线上的数字表示加权图中的顶点。

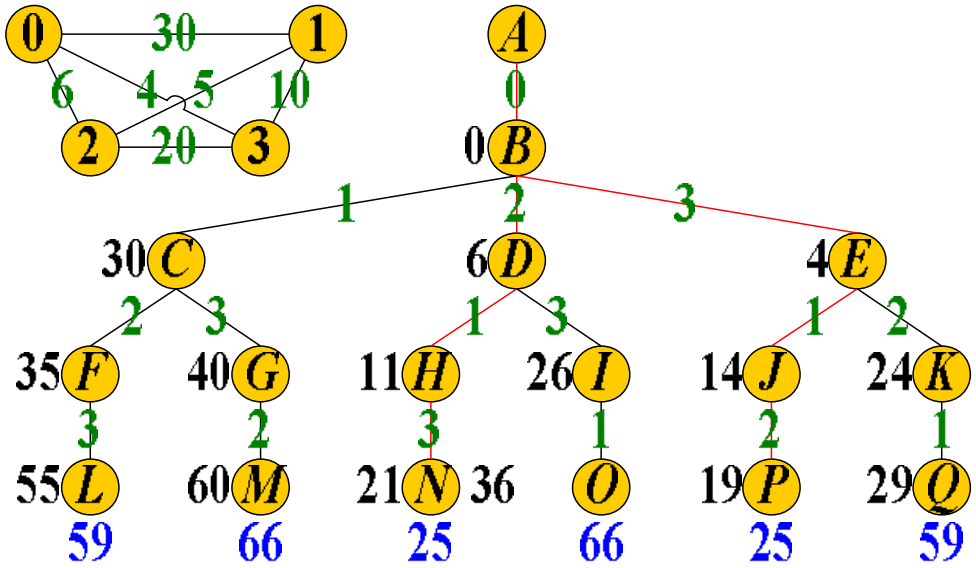

0
30
1
6
4
5
10
2
20
3
A
0
0 B
1
2
3
30 C
6 D
4 E
2
3
1
3
1
2
35 F
40 G
11 H
26 I
14 J
24 K
3
2
3
1
2
1
55 L
60 M
21 N
36 O
19 P
29 Q
59
66
25
66
25
59

图7-3　TSP解空间

3. 着色问题

【问题描述】判断一个无向图是否可m着色。

【解空间】如果 m 种颜色分别用 $1,2,\ldots,m$ 表示，则着色问题的每一个解都可以表示成一个整数数组 $(x_0,x_1,\ldots,x_{n-1})$，其中 x_i 的值代表顶点 i 选用的颜色。易知，着色问题的解空间中共有 m^n 个元素，结构是一棵完全m-叉树，共有 m^n 个外部结点，如图7-4所示（这里只给出了部分结点和3个可行解）。

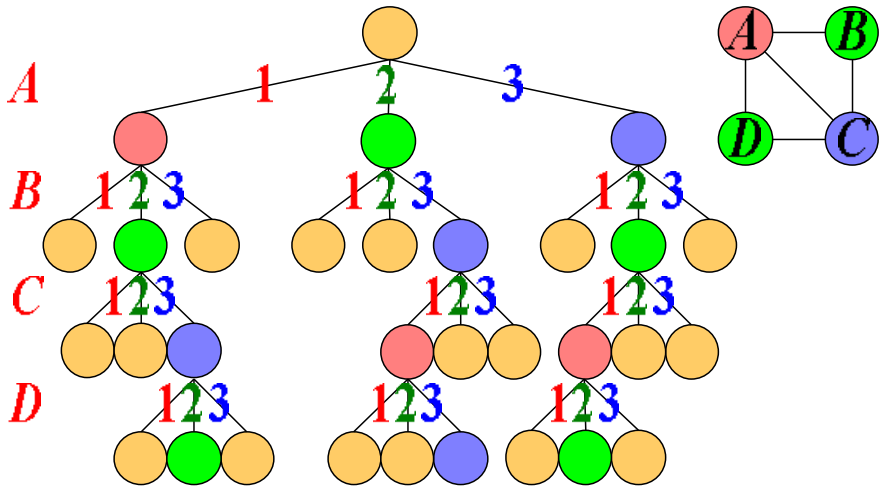


图7-4　着色问题的解空间

7.1.3 回溯方法

1. 回溯方法的基本思想

确定了解空间的组织结构后，回溯方法从开始结点（解空间树的根结点）出发，以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点，同时也成为当前扩展结点。在当前扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点，并且成为当前扩展结点。如果在当前扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点。此时应往回移动（回溯）至最近一个活结点处，并使这个活结点成为当前扩展结点。回溯方法以这种工作方式递归地在解空间中搜索，直至找到要求的解或解空间中无活结点时为止。

2. 剪枝函数

事实上，在将问题的有关数据以一定的方式存储好以后（例如，旅行商问题存储加权图的邻接矩阵，子集和问题存储已知的$n+1$个数），不必首先建立一个解空间树，然后再搜索该树寻找所需要的解。回溯方法实际上是在搜索的同时逐步生成解空间树（实际只需要一部分）。也就是说，在使用回溯方法求解问题时，没有必要搜索整个解空间。

为了使搜索更加有效，常常在搜索过程中加一些判断以决定搜索是否应该终止或改变路线。通常采用两种策略来避免无效的搜索，提高回溯方法的搜索效率。其一是使用约束函数（可行性函数）在扩展结点处剪去不满足约束条件（可行性条件）的子树；其二是用限界函数剪去不能得到最优解的子树。这两种函数统称为剪枝函数。

3. 回溯方法的工作流程

这里的核心任务是在确定解向量X的前t个分量$X[0..(t-1)]$的值以后，如何确定分量$X[t]$的值。为了方便描述，引入下列两个函数。

- $T(X[0..(t-1)])$。表示在确定$X[0..(t-1)]$以后，$X[t]$所有可能取值的集合。
- $B(X[0..t])$。表示$X[0..t]$是否违背约束条件，其中，$B(X[0..t])$ = true表示$X[0..t]$没有违背约束条件，$B(X[0..t])$ = false表示$X[0..t]$违背了约束条件。

```
function BackTrack(t) // 假定X[0 .. (t-1)]已经确定, t ≤ n
{   if(IsAnswer(X)) Print X; // 答案
    else if(t < n)
        for(X[t] in T(X[0 .. (t - 1)]))
            if(B(X[0 .. t])) // X[0 .. t]满足约束条件, 成为当前扩展结点
                BackTrack(t + 1); // 深度优先
}
```

【注】这里必须根据具体问题指定解向量各分量的初始值。例如，对于排列类问题，解向量通常需要初始化为自然排列。

4. 用回溯方法解题的步骤

通常，运用回溯方法解题的基本步骤如下。

（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的搜索。

7.2 子集树和排列树的搜索

为了构造子集树（包括m-叉树）搜索类问题和排列树搜索类问题回溯算法的公式化描述，这里引入了虚的可行性检查函数和界限检查函数（保存在文件dummy.h中），这两个函数的返回值直接规定为true。并且，为了方便，在算法描述中假定$X[0..n)$为解向量。

```
#include "algorithm.h"
bool legal(int t) { return true; } // 检查可行性(虚的剪枝函数)
bool bound(int t) { return true; } // 检查界限(虚的剪枝函数)
```

7.2.1 *m*-叉树的搜索

可以将子集树的搜索拓展为*m*-叉树的搜索（子集树就是2-叉树），核心任务是如何为解向量的每个分量选取一个新值。

1. 算法描述

（1）变量声明及初始化。

```
#include "dummy.h" // 虚的剪枝函数
void MSet(int n, int m)
{   vector<int> X(n); // 解向量
```

（2）深度优先遍历解空间树。使用依次选用候选值的方法为解向量的每个分量选取一个新值。

```
function<void(int)> MSet = [&](int t)
{   if(t >= n) cout << X << endl; // 答案
    else
        for(X[t] = 1; X[t] <= m; ++X[t]) // X[t] in [1 .. m]
        {   if(legal(t) and bound(t)) // 可行和界限
                MSet(t + 1);
        }
};
```

（3）深度优先遍历的调用。

```
    MSet(0); // 从0号元素开始
}
```

2. 复杂性分析

解空间树的外部结点有 $O(m^n)$ 个。在最坏情况下，确定每一个外部结点均需经历 $O(n)$ 步，每一步耗时 $O(1)$。因此，该算法总耗时 $O(n \times m^n)$。

7.2.2 子集树的搜索

对于子集树的搜索，如果解向量确实代表某一集合的一个子集，则解向量的每个分量选用0时，不需要检查可行，选用1时，不需要检查界限。

1. 算法描述

（1）变量声明及初始化。

```
#include "dummy.h" // 虚的剪枝函数
void SubSet(int n)
{   vector<bool> X(n); // 解向量
```

（2）深度优先遍历解空间树。注意，这里最好不要交换$X[t]=1$和$X[t]=0$的顺序。

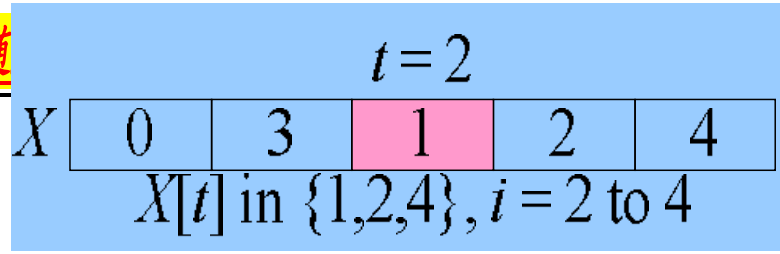
```
function<void(int)> Set = [&](int t)
{   if(t >= n) cout << X << endl; // 答案
    else
    {   X[t] = 1;
        if(legal(t)) Set(t + 1); // 可行
        X[t] = 0;
        if(bound(t)) Set(t + 1); // 界限
    }
};
```

（3）深度优先遍历的调用。

```
    Set(0); // 从0号元素开始
}
```

2. 复杂性分析

解空间树的外部结点有 $O(2^n)$ 个。在最坏情况下，确定每一个外部结点均需经历 $O(n)$ 步，每一步耗时 $O(1)$。因此，该算法总耗时 $O(n\times 2^n)$。



7.2.3 排列树的搜索

1. 算法描述

通过对自然排列重新排列的方法为解向量选取一个新分量值，方法是将当前分量依次与后续分量一一交换（回溯时需依次恢复）。

（1）变量声明及初始化。

```
#include "dummy.h" // 虚的剪枝函数
void B_Perm(int n)
{   vector<int> X(n); // X是解向量, 初始值必须为自然排列
    iota(begin(X), end(X), 0); // 解向量需初始化为自然排列
```

（2）深度优先遍历解空间树。

```
function<void(int)> Perm = [&](int t)
{   if(t >= n) // 答案
        cout << X << endl;
    else
        for(int i = t; i < n; ++i)
        {   swap(X[t], X[i]); // 将当前分量依次与后续分量一一交换
            if(legal(t) and bound(t)) // 可行和界限
                Perm(t + 1);
            swap(X[t], X[i]); // 回溯时需依次恢复
        }
};
```

（3）深度优先遍历的调用。

```
    Perm(0); // 从0号元素开始
}
```

$t=2$

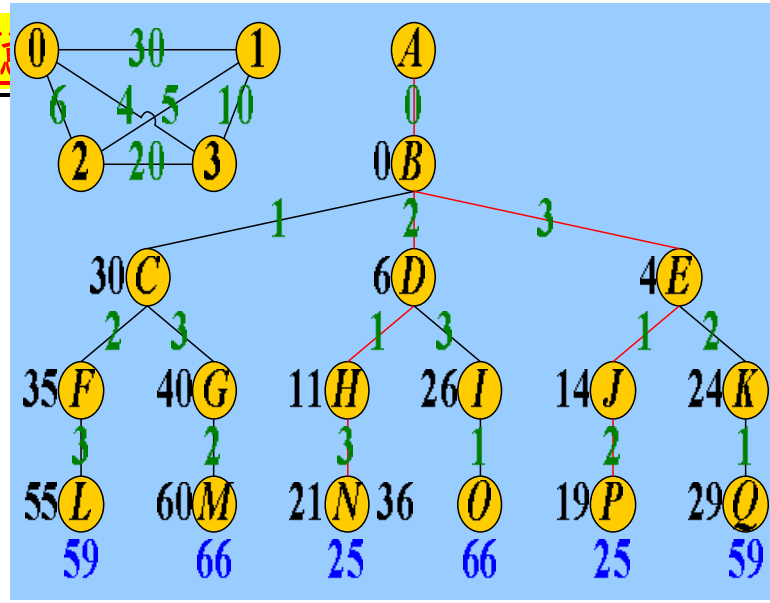
X	0	3	1	2	4

$X[t]$ in $\{1,2,4\}$, $i=2$ to 4

通过对自然排列重新排列的方法为解向量选取一个新分量值，方法是将当前分量依次与后续分量一一交换（回溯时需依次恢复）。

2. 复杂性分析

解空间树的外部结点有 $O(n!)$ 个。在最坏情况下，确定每一个外部结点均需经历 $O(n)$ 步，每一步耗时 $O(1)$。因此，该算法总耗时 $O(n\times n!)$。



7.3 旅行商问题

7.3.1 问题分析

可以用一个无向加权图来表示旅行商实例，顶点代表城市，边表示城市之间的道路。图中各边的权值就是城市间的路程（或旅费），当然是非负的。这样，旅行商问题可以描述为从一个边权值非负的加权图中找出一条最优旅行，即权值之和最小的Hamilton圈。

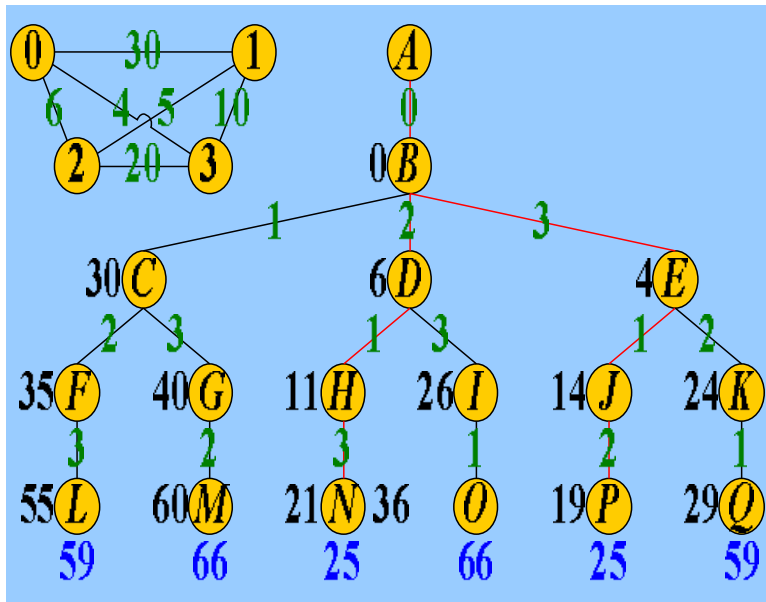
在如图7-2所示的图中，旅行(0, 1, 3, 2, 0)的耗费为66，(0, 2, 1, 3, 0)的耗费为25，(0, 3, 2, 1, 0) 的耗费为59，其中，(0, 2, 1, 3, 0)是最优旅行。

可能的旅行可用一棵树描述，每一条从根到叶的路径定义一个旅行。从根到叶的路径中各边的标号表示一个旅行(附加0作为终点)。例如，到节点L的路径表示旅行(0, 1, 2, 3, 0)，而到节点O的路径表示旅行(0, 2, 3, 1, 0)。图中的每一个旅行都由树中的一条从根到叶的路径表示。因此，树中叶子的数目为$(n-1)!$。

7.3.2 答案与剪枝

1. 答案

若当前扩展结点是解空间树的叶结点，则找到了一条TSP回路。例如，对于图7-2所示的实例，若当前扩展结点是L，因为L是解空间树的叶结点，所以找到了一条TSP回路(0, 1, 2, 3, 0)，费用为59。若该回路的费用比当前最优值小，则将答案更改为该回路并修改当前最优值。例如，对于图7-2所示的实例，若当前最优值是59，则由扩展结点N得到的回路费用比当前最优值小。

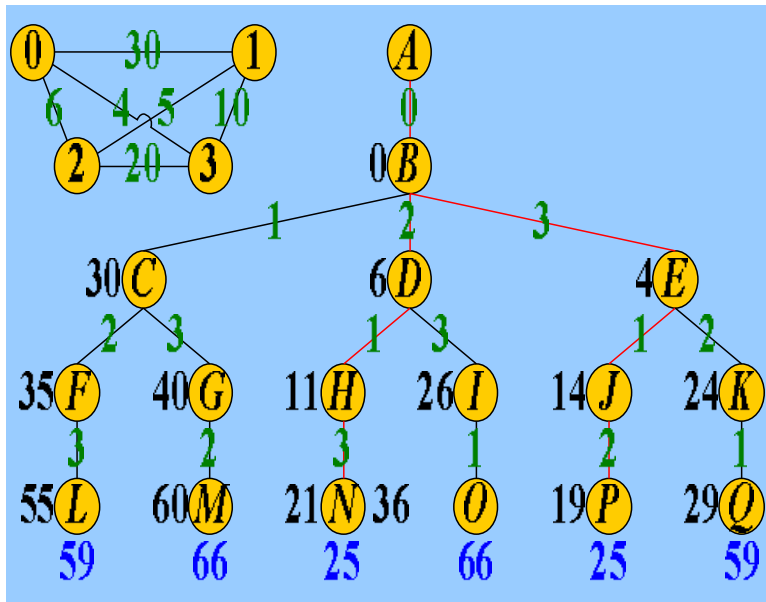


2. 剪枝函数

（1）可行性函数。若当前边的费用已经超过当前最优值，则没有必要扩展该节点，以该节点为根的子树被剪掉。例如，对于图7-2所示的实例，若当前最优值是25，则边(0, 1)费用已经超过当前最优值，没有必要扩展结点C。

（2）界限函数。若从根结点到当前扩展结点的费用（当前费用）已经超过当前最优值，则没有必要扩展该节点，以该节点为根的子树被剪掉。例如，对于图7-2所示的实例，若当前最优值是25，则从根结点到结点I的费用已经超过当前最优值，没有必要扩展结点I。

显然，满足界限条件可以保证满足可行性条件，没有必要专门检查可行性。



7.3.3 回溯算法

1. 算法描述

（1）变量声明及初始化。

```
#include "algorithm.h"
auto TSP(const Matrix<double> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<int> X(n); // 到当前结点的路径(当前路径，解向量)
    iota(begin(X), end(X), 0); // 解向量需初始化为自然排列
    vector<int> BX; // 最优回路
    double BC = inf; // 最优耗费，初始值为无穷大
```

（2）深度优先遍历解空间树。

```
function<void(int, double)>
TSP = [&](int t, double C) // C是当前路径的耗费(当前耗费)
{   if(t >= n and C + G(X[n - 1], 0) < BC) // 答案(当前回路更优)
        BC = C + G(X[n - 1], 0), BX = X;
    else if(t < n)
        for(int i = t; i < n; ++i)
        {   swap(X[t], X[i]);
            if(C + G(X[t - 1], X[t]) < BC) // 可行(当前路径更优)
                TSP(t + 1, C + G(X[t - 1], X[t]));
            swap(X[t], X[i]);
        }
};
```

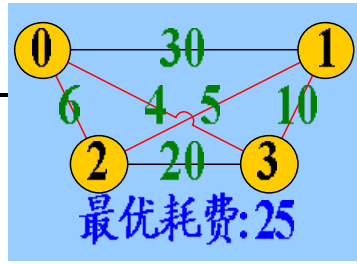
（3）深度优先遍历的调用。

```
TSP(1, 0); // 从第1站开始考虑, 当前耗费为0
return BX;
}
```

2. 复杂性分析

$O((n-1)\times(n-1)!)=O(n!)$，因为只有 $n-1$ 个顶点参与排列，且排列树中每个结点的构造和检查都耗时 $O(1)$。

```
{   if(t >= n and C + G(X[n - 1], 0) < BC)
        BC = C + G(X[n - 1], 0), BX = X;
    else if(t < n)
        for(int i = t; i < n; ++i)
        {   swap(X[t], X[i]);
            if(C + G(X[t - 1], X[t]) < BC)
                TSP(t + 1, C + G(X[t - 1], X[t]));
            swap(X[t], X[i]);
        }
};
```



3. 测试

使用如图7-2所示的图。

```
int main()
{   Matrix<double> G = //  邻接矩阵
    {   {0, 30, 6, 4},
        {30, 0, 5, 10},
        {6, 5, 0, 20},
        {4, 10, 20, 0}
    };
    cout << TSP(G) << endl; // 0 2 1 3
}
```

7.4 最大团问题

输出指定无向图的一个最大团。显然，最大团问题的解空间树是子集树。

7.4.1 剪枝函数

1. 上界函数

上界函数规定为有足够多的可选择顶点使得算法有可能找到更大的团。

如果用变量cn表示结点对应的团的大小，t表示结点在解空间树中的层次，即当前正在处理的顶点（编号从0开始），则可用cn+n-(t+1)作为顶点数上界值。

2. 可行性函数

可行性函数规定为顶点t到每一个已选顶点都有边相连，可由下列Connected()实现，保存在文件Clique.h中。

```
// Clique.h
#pragma once
#include "algorithm.h"
bool Connected(const Matrix<bool> &G, const vector<bool> &X, int t)
{   // 邻接矩阵, 解向量, 当前顶点
    int n = G.rows(); // 顶点数
    for(int u = 0; u < n; ++u) // 顶点t与团X是否相连
    {   if(u == t) continue; // 忽略顶点t
        if(X[u] == 1 and G(t, u) == 0) // 顶点t与团中的顶点u不相连
            return false;
    }
    return true;
}
```

7.4.2 回溯算法

1. 算法描述

（1）变量声明及初始化。

```
#include "Clique.h"
auto Clique(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<bool> X(n, 0), BX; // 当前团(初始为空集), 最大团
    int fn = -1; // 最大团顶点数, 初始值为-1
```

（2）深度优先遍历解空间树。

```
function<void(int, int)>
Clique = [&](int t, int cn) // 当前顶点，当前团顶点数
{   if(t >= n and cn > fn) // 答案(当前团更优)
        fn = cn, BX = X;
    else if(t < n)
    {   X[t] = 1;
        if(Connected(G, X, t)) // 可行(当前成团)
            Clique(t + 1, cn + 1);
        X[t] = 0;
        if(cn + n - (t + 1) > fn) // 界限(可能有更大的团)
            Clique(t + 1, cn);
    }
};
```

（3）深度优先遍历的调用。

```
    Clique(0, 0); // 从顶点0开始，当前团顶点数为0
    return BX;
}
```

2. 复杂性分析

因为在最坏情况下，每选取一个顶点需要进行 $O(n)$ 次比较，所以该算法的计算时间为 $O(n \times n \times 2^n) = O(n^2 \times 2^n)$ 。

```
function<void(int, int)>
Clique = [&](int t, int cn) // 当前顶点，当前团顶点数
{   if(t >= n and cn > fn) // 答案(当前团更优)
        fn = cn, BX = X;
    else if(t < n)
    {   X[t] = 1;
        if(Connected(G, X, t)) // 可行(当前成团)
            Clique(t + 1, cn + 1);
        X[t] = 0;
        if(cn + n - (t + 1) > fn) // 界限(可能有更大的团)
            Clique(t + 1, cn);
    }
};
```

3. 测试☼

使用如图7-5所示的无向图。

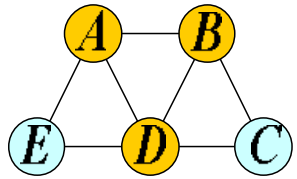


图7-5 最大团

```
int main()
{   Matrix<bool> G = // 邻接矩阵(5个顶点)
    {   {0, 1, 0, 1, 1},
        {1, 0, 1, 1, 0},
        {0, 1, 0, 1, 0},
        {1, 1, 1, 0, 1},
        {1, 0, 0, 1, 0}
    };
    cout << Clique(G) << endl; // 1 1 0 1 0
}
```

7.5 子集和问题※

7.5.1 问题描述

设 $W=\{w_0,w_1,\dots,w_{n-1}\}$ 是一个由 n 个正整数构成的集合，M 是一个正整数，请判断 W 是否存在满足 $\sum W'=M$ 的子集 W'，即是否存在满足 $\sum\{w_ix_i\}_{i=0}^{n-1}=M$ 的0/1向量 $X=(x_0,x_1,\dots,x_{n-1})$。

7.5.2 答案与剪枝

假设 $x_0, x_1, \ldots, x_{t-1}$ 已经确定，算法需要决定下一步搜索目标。此时，有 $\sum\{w_i x_i\}_{i=0}^{t-1} = M$ 和 $\sum\{w_i x_i\}_{i=0}^{t-1} < M$ 这两种情况。

若 $\sum\{w_i x_i\}_{i=0}^{t-1} = M$，则当前扩展结点是一个答案，应该停止在解空间树中继续搜索该路径，返回到最近的活结点。

若 $\sum\{w_i x_i\}_{i=0}^{t-1} < M$，则需要判断是否有必要继续向前搜索。显然，只有当 $\sum\{w_i x_i\}_{i=0}^{t} \le M$（可行）且 $\sum\{w_i x_i\}_{i=0}^{t} + \sum\{w_i\}_{i=t+1}^{n-1} \ge M$（界限）时才有必要继续搜索。

【注】根据继续向前搜索需要满足的条件，不可能有 $\sum\{w_i x_i\}_{i=0}^{t-1} > M$。

7.5.3 回溯算法

1. 算法描述

（1）变量声明及初始化。

```
#include "algorithm.h"
auto SetSum(const vector<int> &W, int M) // 正整数集合，和数
{   int n = W.size(); // 正整数个数
    vector<bool> X(n, 0); // 每个整数是否选取，初始时未选
    vector<bool> Y; // 答案数组，空数组表示未找到答案
```

（2）深度优先遍历解空间树。

```
function<void(int, int, int)>
SetSum = [&](int t, int s, int r)
{   // s为已选整数的和(和), r为待处理的所有整数的和(余额)
    if(not Y.empty()) return; // 已经记录答案, 不再继续寻找
    if(s == M) Y = X; // 若已经找到答案, 则记录答案
    else if(t < n)
    {   X[t] = 1;
        if(s + W[t] <= M) // 可行, t+1号数, 和增加, 余额减少
            SetSum(t + 1, s + W[t], r - W[t]);
        X[t] = 0;
        if(s + (r - W[t]) >= M) // 界限, t+1号数, 和不变, 余额减少
            SetSum(t + 1, s, r - W[t]);
    } // 不可交换X[t]=1和X[t]=0的顺序
};
```

（3）深度优先遍历的调用。

```
    int s = 0, r = accumulate(begin(W), end(W), 0); // 初始化和s与余额r
    SetSum(0, s, r); // 从0号数开始
    return Y; // 返回答案
}
```

2. 复杂性分析

$O(n \times 2^n)$，因为每个结点的构造和检查都是常数时间。

3. 测试

```
int main()
{   int M = 30;
    vector<int> W = {5, 10, 15, 13, 12, 18};
    cout << W << endl; // 5 10 15 13 12 18
    cout << SetSum(W, M) << endl; // 1 1 1 0 0 0
}
```

```
{   if(not Y.empty()) return;
    if(s == M) Y = X;
    else if(t < n)
    {   X[t] = 1;
        if(s + W[t] <= M)
            SetSum(t + 1, s + W[t], r - W[t]);
        X[t] = 0;
        if(s + (r - W[t]) >= M)
            SetSum(t + 1, s, r - W[t]);
    } // 不可交换X[t]=1和X[t]=0的顺序
};
```

7.6 0/1背包问题※

7.6.1 问题描述

有 n 件物品，第 i 件的重量和价值分别是 w_i 和 v_i。要将这 n 件物品的一部分装入容量为 c 的背包中，要求每件物品或整个装入或不装入。0/1背包问题就是要给出装包算法，使得装入背包的物品的总价值最大。本节采用回溯方法求解，选择出满足目标函数极大化和重量要求的一个子集。

7.6.2 限界函数

解空间由各分量 x_i 分别取0或1值的 2^n 个不同的 n 元向量组成。如果在结点 Z 处已经确定了 x_i 的值（ $0 \leq i \leq t$ ），则 Z 的上界可由下述方法获得。

（1）对 $t+1 \leq i < n$，将 $x_i \in \{0,1\}$ 的要求放宽为 $x_i \in [0,1]$。

（2）用贪心方法求解这个放宽要求的问题（使用下述RestVal过程，需要多处使用，保存在文件Knap.h中）。

（3）当前已获得的效益与贪心方法所得结果的和作为 Z 的上界。

```
// 从物品t开始可能获得的最大效益, 用于回溯和分枝限界
auto RestVal(const vector<double> &V, const vector<double> &W,
             double rc, int t)
{   // 效益数组, 重量数组, 剩余容量, 当前物品
    // 物品已经按单价降序排列
    int n = min(V.size(), W.size()); // 物品数
    double rv = 0; // 剩余部分最优效益
    for(; t < n and W[t] <= rc; ++t)
        rv += V[t], rc -= W[t]; // 可完整装包的物品完整装包
    if(t < n) rv += rc * V[t] / W[t]; // 不能完整装包的物品部分装包
    return rv;
}
```

7.6.3 回溯算法

1. 算法描述

（1）变量声明及初始化。

```
#include "Knap.h"
auto Knap(vector<double> &V, vector<double> &W, double M)
{   // 效益数组，重量数组，背包容量
    Sort(V, W); // 物品数组按单价降序排列
    int n = min(V.size(), W.size()); // 物品数
    vector<bool> X(n), BX; // 当前装包方案，最优装包方案
    double fv = -1; // 最优效益，初始为-1
```

（2）深度优先遍历解空间树。

```
function<void(int, double, double)>
Knap = [&](int t, double cv, double cw)
{   // 当前物品, 当前效益, 当前重量
    if(t >= n and cv > fv) // 答案(更优)
        fv = cv, BX = X;
    else if(t < n)
    {   X[t] = 1;
        if(cw + W[t] <= M) // 可行(当前不超重)
            Knap(t + 1, cv + V[t], cw + W[t]);
        X[t] = 0;
        auto up = cv + RestVal(V, W, M - cw, t + 1); // 上限值
        if(up > fv) // 界限(可能有更优的装包方案)
            Knap(t + 1, cv, cw);
    }
};
```

```
function RestVal(V, W, rc, t)
{   rv = 0; // 剩余部分最优效益
    for(; t < n and W[t] <= rc; ++t)
        rv += V[t], rc -= W[t];
    if(t < n) rv += rc * V[t] / W[t];
    return rv;
}
```

（3）深度优先遍历的调用。

```
    Knap(0, 0, 0); // 从物品0开始, 当前效益和当前重量均为0
    return BX;
}
```

2. 复杂性分析

因为每考虑一个物品都有可能调用限界函数，限界函数需要耗时 $O(n)$，所以该算法的计算时间为 $O(n \times n \times 2^n) = O(n^2 \times 2^n)$。

3. 测试

```
int main()
{   vector<double> V = {120, 60, 100}; // 效益数组
    vector<double> W = {30, 10, 20}; // 重量数组
    double M = 50; // 背包容量
    auto X = Knap(V, W, M);
    PrintSolution(V, W, X);
}
```

```
效益: 60 100 120
重量: 10 20 30
答案: 0 1 1
```

```
{   if(t >= n and cv > fv)
        fv = cv, BX = X;
    else if(t < n)
    {   X[t] = 1;
        if(cw + W[t] <= M)
            Knap(t + 1, cv + V[t], cw + W[t]);
        X[t] = 0;
        if(cv + Rest(t + 1, M - cw) > fv)
            Knap(t + 1, cv, cw);
    }
};
```

7.7 着色问题

本节讨论判定形式的着色问题。

7.7.1 问题分析

1. 问题描述

判断一个无向图是否可m着色。若是，则记录一种着色方案。

2. 剪枝函数

图用邻接矩阵表示，m种颜色分别用$1,2,\ldots,m$表示，并用-1表示未着色，着色方案用一个数组X表示，$X[i]=k$表示顶点i着色k。

X是可行解，当且仅当对任何已着色顶点i,j，都有“若$G[i,j]=1$，则$X[i]\neq X[j]$”，这就是约束条件。

假设有一部分顶点已经着色，则确定当前顶点t的颜色$X[t]$时，根据约束条件，应该验证是否有“对任何已着色顶点i，若$G[t,i]=1$，则$X[t]\neq X[i]$”。这可以由下列程序完成。保存在文件Chromatic.h中。

```
// Chromatic.h
#pragma once
#include "algorithm.h"
bool legal(const Matrix<bool> &G, vector<int> &X, int t)
{   // 邻接矩阵, 当前方案(负数表示未着色), 当前顶点
    int n = G.rows(); // 顶点数
    for(int i = 0; i < n; ++i) // 验证约束条件
    {   if(X[i] < 0) continue; // 忽略未着色的顶点
        if(G(i, t) == 1 and X[i] == X[t]) return false;
    }
    return true;
} // 耗时O(n)
```

7.7.2 回溯算法

1. 算法描述

（1）变量声明及初始化。

```
#include "Chromatic.h"
auto Chromatic(const Matrix<bool> &G, int m) // 邻接矩阵, 颜色数
{   int n = G.rows(); // 顶点数
    vector<int> X(n, -1); // 各顶点颜色(1, 2, ..., m), 负数表示未着色
    vector<int> W; // 答案数组, 空数组表示未找到答案
```

（2）深度优先遍历解空间树。

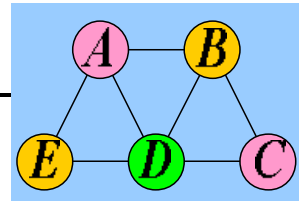
```
function<void(int)> Chromatic = [&](int t)
{   if(not W.empty()) return; // 已经记录答案，不再继续寻找
    if(t >= n) W = X; // 已经找到答案，记录答案
    else
        for(X[t] = 1; X[t] <= m; ++X[t])
            if(legal(G, X, t)) // 颜色X[t]可用
                Chromatic(t + 1); // 下一顶点
};
```

（3）深度优先遍历的调用。

```
    Chromatic(0); // 从顶点0开始
    return W; // 返回答案
}
```

2. 复杂性分析

解空间树的外部结点有 $O(m^n)$ 个。在最坏情况下，确定每一个外部结点均需经历 $O(n)$ 步，每一步耗时 $O(n)$。因此，该算法总耗时 $O(n^2 \times m^n)$。



3. 测试☼

$n=5, m=3$，无向图 G 如图7-5所示。

```
void test(const Matrix<bool> &G, int m) // 邻接矩阵, 颜色数
{   auto X = Chromatic(G, m);
    if(X.empty())
        cout << "不可" << m << "着色" << endl;
    else
        cout << X << endl;
}
```

```
int main()
{   Matrix<bool> G = // 邻接矩阵
    {   {0, 1, 0, 1, 1},
        {1, 0, 1, 1, 0},
        {0, 1, 0, 1, 0},
        {1, 1, 1, 0, 1},
        {1, 0, 0, 1, 0}
    };
    test(G, 3); // 1 2 1 3 2
    test(G, 2); // 不可2着色
}
```

7.8 练习题

7.8.1 基础知识题

1. 编写一个使用递归方法生成含 n 个分量的所有排列的程序。

2. 编写一个使用递归方法生成 n 个元素的所有子集的程序。

3. 使用一种程序设计语言描述求解旅行商问题（输出最优解）的回溯算法，并分析时间复杂性。

4. 使用回溯方法求解子集和问题，要求只输出一个满足条件的子集，或报告不存在满足条件的子集。

5. 使用回溯方法判断一个无向图是否可 m 着色。若是，则记录一种着色方案。

7.8.2 实验题

1. 使用一种程序设计语言描述求解Hamilton回路问题（输出所有Hamilton回路）的回溯算法，并分析时间复杂性。

2. 使用回溯方法求解Hamilton回路问题，要求只输出一条Hamilton回路，或报告不存在Hamilton回路。

3. 使用一种程序设计语言描述求解最大团问题（输出指定无向图的一个最大团）的回溯算法，并分析时间复杂性。

4. 使用一种程序设计语言描述求解0/1背包问题（输出最优解）的回溯算法，并分析时间复杂性。

5. 使用一种程序设计语言描述求解子集和问题（输出所有解）的回溯算法，并分析时间复杂性。

6. 着色问题的回溯算法。请使用回溯方法为一个需要着色的无向图选取一种颜色数目最少的着色方案。

【提示】可参考“分枝限界方法”一章。

7.8.3 进阶训练题

1. 设S是一个正整数集合，t是一个正整数，寻找S的满足子集和不大于t且使得子集和最小的子集。为该问题设计回溯算法。

2. 设S是一个正整数集合，t是一个正整数，寻找S的满足子集和不小于t且使得子集和最小的子集。为该问题设计回溯算法。

3. 顶点覆盖问题的回溯算法。为最小顶点覆盖问题（输出一个最小顶点覆盖）设计回溯算法。

4. 集合覆盖问题的回溯算法。已知一个有限集X和X的一组子集F，如果F中所有子集的并等于X，则称F覆盖X。请使用回溯方法从F中选取一组覆盖X且子集数目最少的子集。

5. 精确覆盖问题的回溯算法。有限集X的集合覆盖F中覆盖X的互不相交的子集族T称为$<X, F>$的精确覆盖。为精确覆盖问题（$<X, F>$是否有精确覆盖）设计回溯算法。

6. 子集和问题的回溯算法。使用回溯方法从一个正整数集W中选取一个和最接近正整数t的子集。

7. 均衡划分的回溯算法。为均衡划分问题（将一个正整数集分为和最接近的两

个子集）设计回溯算法。

8. 多机调度问题的回溯算法。为多机调度问题（输出一种最优调度）设计回溯算法。

9. CNF-SAT的回溯算法。为CNF-SAT设计回溯算法，用于计算给定合取范式的最大可满足子句数，并输出一种最优赋值。相关概念参阅问题的复杂性一章（11.2.3和11.4.3小节）。

10. 考查课选题问题的回溯算法。某一门考查课的任课老师准备了n道考核候选题，每位考生选取m道题完成，要求每两个考生的选题方案之间至少有k道题是不相同的，请问这些候选题能够保证多少考生能够顺利按照要求完成选题。为该问题设计回溯算法。

11. 考查课出题问题的回溯算法。某一门考查课的考核方法是由任课老师准备一些考核候选题，每位考生选取m道题完成，要求每两个考生的选题方案之间至少有k道题是不相同的，请问任课老师至少需要准备多少道候选题才能保证n个考生能够顺利按照要求完成选题。为该问题设计回溯算法。