第8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。然后介绍皇后问题、旅行商问题、子集和问题、最大团问题、0/1背包问题、图着色问题等问题的回溯算法。

- 回溯方法的基本思想
- 子集树和排列树的搜索
- 皇后问题的回溯算法
- 旅行商问题的回溯算法
- 子集和问题的回溯算法
- 最大团问题的回溯算法*
- 0/1背包问题的回溯算法
- 图着色问题的回溯算法*

8.1 回溯方法的基本思想

8.1.1 解空间

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。应用回溯方法解问题时，首先应该明确问题的解空间。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。

【注】为了区别图的顶点和解空间树的顶点，将解空间树的顶点称为结点。

8.1.2 回溯方法的工作流程

设 $x_0x_1...x_{t-1}$ 是解空间树中由根到一个结点的路径，用 $T(x_0,x_1,\ldots,x_{t-1})$ 表示所有使得 $x_0x_1...x_{t-1}x_t$ 成为解空间树中路径的 x_t，$B(x_0,x_1,\ldots,x_t)$ 表示约束函数。其中，$B(x_0,x_1,\ldots,x_t)=\text{true}$ 表示路径 $x_0x_1...x_t$ 没有违背约束条件，否则，$B(x_0,x_1,\ldots,x_t)=\text{false}$。

1. 递归方法

```
function BackTrack(t) // 假定X[0 To t - 1]已经赋值, t < n
{   if(is_answer(X[0 to t - 1])) Print X[0 to t - 1]; // 输出一个答案
    else if(t < n)
    {   for(X[t] in T(X[0 to t - 1]))
            if(B(X[0 to t])) // 激活状态结点X[0 To t]
                BackTrack(t + 1); // 深度优先
    }
}
```

2. 迭代方法

```
function BackTrack()
{   for(t = 0; t >= 0)
    {   if('T(X[0 to t - 1])中值已取尽') --t; // 回溯
        else
        {   X[t] = 'T(X[0 to t - 1])中一未选值';
            if(B(X[0 to t])) // 激活状态结点X[0 To t]
            {   if(is_answer(X[0 to t])) Print X[0 to t]; // 输出答案
                else if(t < n – 1) ++t; // 深度优先
            }
        }
    }
}
```

8.1.3 举例说明

1. 皇后问题

以4个皇后为例说明。在4×4棋盘上放置4个皇后，使得每两个之间都不能互相攻击，即任意两个皇后都不能放在同一行、同一列及同一对角线上。通过将4个皇后放在4×4棋盘的不同行、不同列上构成一个决策序列，该决策序列实际上对应于{0, 1, 2, 3}的一个排列，如图8-1右侧所示。这时，解空间由4!个决策序列构成，解空间树是一个4级排列树，如图8-1左侧所示。

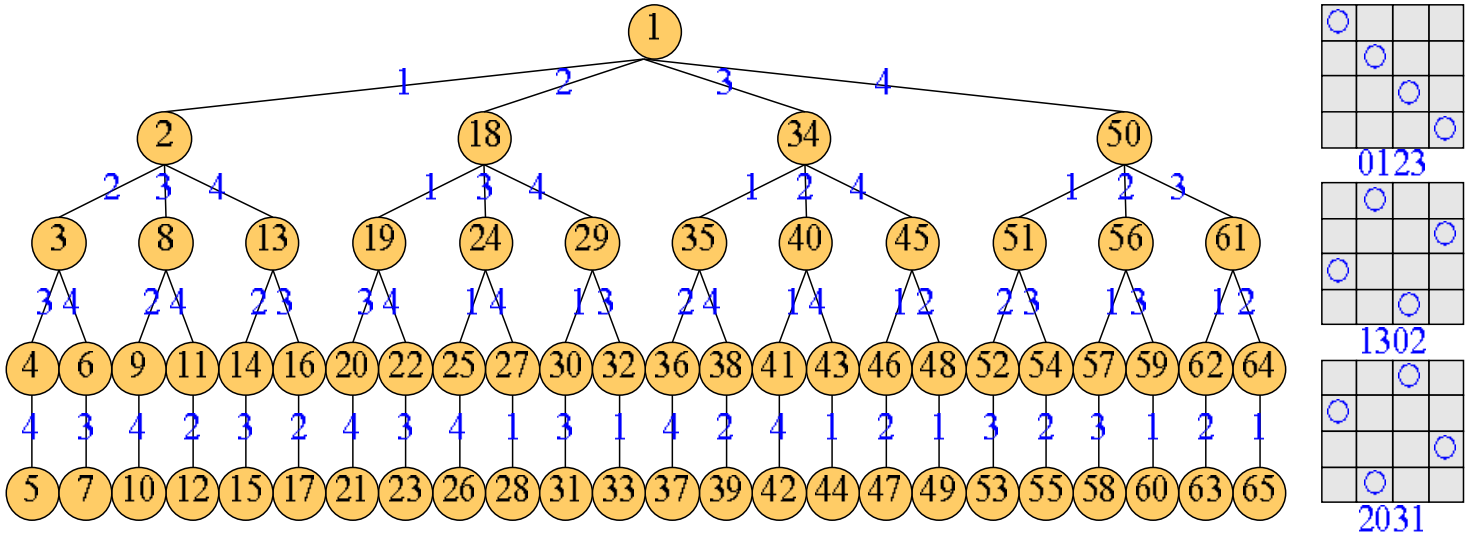


图8-1 皇后问题解空间

2. 旅行商问题

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，经过每个城市一遍，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8-2所示的实例，假定城市0是驻地。

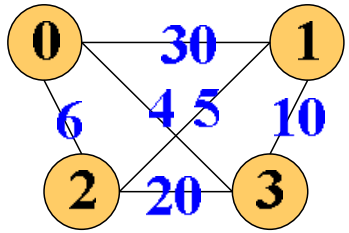


图8-2　一个TSP实例

从0出发，第一站有3种选择。第一站选定后，第二站有两种选择，如第一站选1，则第二站只能选2或3。第一、二站都选定后，第三站只有一种选择，比如，第一、二站选了1和2，则第三站只能选择3。最后由城市3返回驻地0。解空间树是一棵4级排列树（实际只有3级，因为出发站是固定的）。如图8-3所示（图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）。

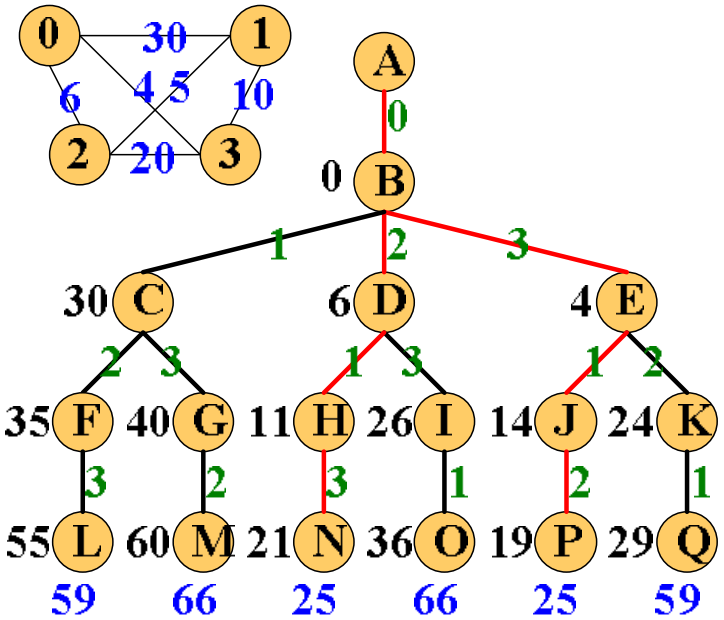


图8-3　TSP解空间

3. 子集和问题

已知正数 M 和一个正数集合 $A=\{w_1,w_2,\ldots,w_n\}$。试求 A 的所有子集 S，使得 S 中的数之和等于 M。这个问题的解可以表示成0/1数组 $(x_1,x_2,\ldots,x_n)$，依据 w_i 是否属于 S，x_i 分别取值1或0。故解空间中共有 2^n 个元素。它的树结构是一棵完全二叉树，如图8-4所示（结点中的标号是按深度优先搜索的序号，从根到叶的每条路径构成一个决策序列）。

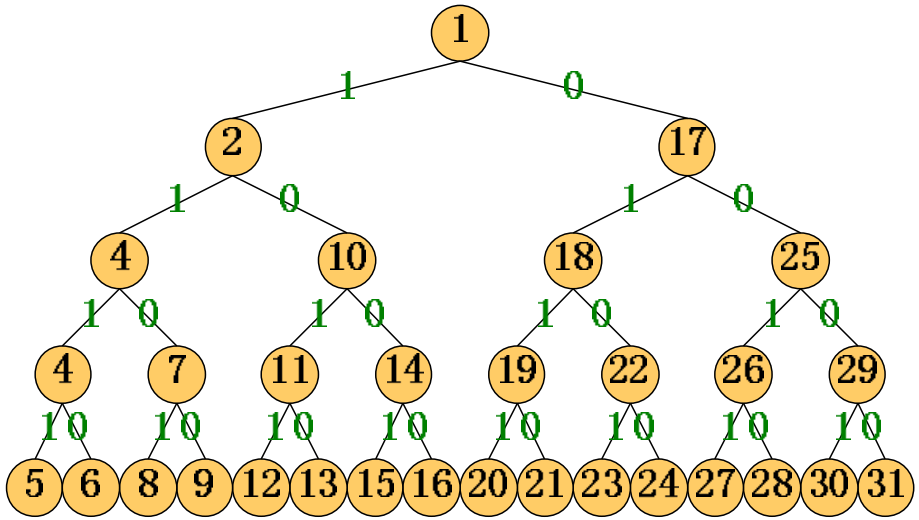


图8-4　子集和问题解空间

4. 说明

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，同时也成为当前的扩展结点。在当前的扩展结点处，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，并且成为当前的扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当前的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已无活结点时为止。

事实上，当我们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、子集和问题是存储已知的 $n+1$ 个数、皇后问题用整数对 (i,j) 表示棋盘上各个位置），不必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，对于实际问题我们不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在扩展结点处剪去不满足约束的子树；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。

8.1.4 用回溯方法解题的步骤

通常，运用回溯方法解题的步骤为

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 以深度优先的方式搜索解空间，并且在搜索过程中用剪枝函数避免无效的搜索。

8.2 子集树和排列树的搜索

为了构造子集树搜索类问题和排列树搜索类的公式化描述，这里引入了虚的合法性检查函数和界限检查函数。

为了方便，在算法描述中假定X(0 To n - 1)为解向量，Legal检查合法性，Bound检查界限。

首先介绍几个基本的支持程序。

8.2.1 支持程序

1. 子集树

为子集树搜索类问题或m-叉树搜索类问题的解向量选取一个新分量值。要求解向量中起始分量的初始值为-1。

```
function NextValue(X, t, m)
{   // 选取一个新分量值(子集树或m-叉树), 要求起始分量的初始值为-1
    (++X[t]) %= (m + 1); // 尝试为t分量选取值
    X[t + 1] = -1; // 初始化下一分量
    return X[t] < m; // X[t] < m表示有可用值
}
```

2. 排列树

为排列树搜索类问题的解向量选取一个新分量值。要求解向量中起始分量的初始值为-1。

```
function NextValue(X, t)
{   // 选取一个新分量值(排列树), 要求起始分量的初始值为-1
    do { (++X[t]) %= (n + 1); } // 尝试为t分量选取值
    while(X[t] < n and Repeated(X, t));
    X[t + 1] = -1; // 初始化下一分量
    return X[t] < n; // X[t] < n表示有可用值
}
```

```
function Repeated(X, t) // 检查重复
{   for(i = 0; i < t; ++i)
        if(X[t] == X[i]) return true;
    return false;
}
```

3. C++描述*

需要多处使用，保存在文件Backtrack.h中。

```
// Backtrack.h
#pragma once
#include <vector>
using namespace std;
```

```
bool NextValue(vector<int> &X, int t, int m)
{   // 选取一个新分量值(m-叉树), 要求起始分量的初始值为-1
    (++X[t]) %= (m + 1); // 尝试为t分量选取值
    X[t + 1] = -1; // 初始化下一分量
    return X[t] < m; // X[t] < m表示有可用值
}
```

```
bool Repeated(vector<int> &X, int t) // 检查重复
{   for(int i = 0; i < t; ++i)
        if(X[t] == X[i]) return true;
    return false;
}
```

```
bool NextValue(vector<int> &X, int t)
{   // 选取一个新分量值(排列树), 要求起始分量的初始值为-1
    int n = X.size();
    do { (++X[t]) %= (n + 1); } // 尝试为t分量选取值
    while(X[t] < n && Repeated(X, t));
    X[t + 1] = -1; // 初始化下一分量
    return X[t] < n; // X[t] < n表示有可用值
}
```

8.2.2 子集树的搜索

1. 递归方法1

```
function SubSet(t)
{   if(t >= n) Print X; // 答案
    else
    {   for(X[t] in {0, 1})
        {   if(legal(t) and bound(t)) // 可行和界限
                SubSet(t + 1);
        }
    }
}
```

2. 递归方法2

```
function SubSet(t)
{   if(t >= n) Print X; // 答案
    else
    {   if(legal(t)) // 检查可行, 生成左儿子
            X[t] = 1, SubSet(t + 1);
        if(bound(t)) // 检查界限, 生成右儿子
            X[t] = 0, SubSet(t + 1);
    }
}
```

3. 迭代方法

```
function SubSet(n) // 迭代方法
{   X[0] = -1; // 初始化
    for(t = 0; t >= 0;)
    {   if(!NextValue(X, t, 2)) --t; // 回溯
        else if(legal(t)) // 可行
        {   if(t >= n - 1) Print X; // 答案
            else if(bound(t)) ++t; // 界限
        }
    }
}
```

4. 复杂性分析

遍历子集树需 $O(n \times 2^n)$ 计算时间。

8.2.3 子集树搜索方法的C++描述*

1. 预处理

```
#include "Backtrack.h"
#include "algorithm.h"
```

```
bool legal(int t) { return true; } // 检查合法性(虚的剪枝函数)
bool bound(int t) { return true; } // 检查界限(虚的剪枝函数)
```

2. 递归方法1

```
void SubSet1(int n) // 递归方法1
{   vector<bool> X(n); // 解向量
    function<void(int)> SubSet = [&](int t)
    {   if(t >= n) cout << X << endl; // 答案
        else
        {   for(auto i : {0, 1})
            {   X[t] = i;
                if(legal(t) && bound(t)) // 可行和界限
                    SubSet(t + 1);
            }
        }
    };
    SubSet(0); // 从0号元素开始
}
```

3. 递归方法2

```
void SubSet2(int n) // 递归方法2
{   vector<bool> X(n); // 解向量
    function<void(int)> SubSet = [&](int t)
    {   if(t >= n) cout << X << endl; // 答案
        else
        {   if(legal(t)) // 检查可行, 生成左儿子
                X[t] = 1, SubSet(t + 1);
            if(bound(t)) // 检查界限, 生成右儿子
                X[t] = 0, SubSet(t + 1); // 界限
        }
    };
    SubSet(0); // 从0号元素开始
}
```

4. 迭代方法

```
void SubSet(int n) // 迭代方法
{   vector<int> X(n);
    X[0] = -1; // 初始化
    for(int t = 0; t >= 0;)
    {   if(!NextValue(X, t, 2)) --t; // 回溯
        else if(legal(t)) // 可行
        {   if(t >= n - 1) cout << X << endl; // 答案
            else if(bound(t)) ++t; // 界限
        }
    }
}
```

5. 测试程序

```
int main()
{   int n = 3;
    cout << "递归1\n", SubSet1(n);
    cout << "递归2\n", SubSet2(n);
    cout << "迭代\n", SubSet(n);
}
```

递归1	递归2	迭代
0 0 0	1 1 1	0 0 0
0 0 1	1 1 0	0 0 1
0 1 0	1 0 1	0 1 0
0 1 1	1 0 0	0 1 1
1 0 0	0 1 1	1 0 0
1 0 1	0 1 0	1 0 1
1 1 0	0 0 1	1 1 0
1 1 1	0 0 0	1 1 1

8.2.4 排列树的搜索

1. 递归方法1

```
// 生成分量的方法,X的起始分量必须初始化为-1
function Perm(t)
{   if(t >= n) Print X; // 答案
    else
    {   while(NextValue(X, t))
        {   if(legal(t) and bound(t)) // 可行和界限
                Perm(t + 1);
        }
    }
}
```

2. 递归方法2

```
// 对自然排列重新排列的方法, X必须初始化为自然排列
function Perm(t)
{   if(t >= n) Print X; // 答案
    else
    {   for(i = t; i < n; ++i)
        {   swap(X[t], X[i]);
            if(legal(t) and bound(t)) // 可行和界限
                Perm(t + 1);
            swap(X[t], X[i]);
        }
    }
}
```

3. 迭代方法

```
// 用迭代方法生成各个分量, X的起始分量必须初始化为-1
function Perm(n)
{   X[0] = -1; // 初始化
    for(t = 0; t >= 0;)
    {   if(!NextValue(t)) --t; // 回溯
        else if(legal(t)) // 可行
        {   if(t >= n - 1) Print X; // 答案
            else if(bound(t)) ++t; // 界限
        }
    }
}
```

4. 复杂性分析

遍历排列树需 $O(n \times n!)$ 计算时间。

8.2.5 排列树搜索方法的C++描述*

1. 预处理

```
#include "Backtrack.h"
#include "algorithm.h"
```

```
bool legal(int t) { return true; } // 检查合法性(虚的剪枝函数)
bool bound(int t) { return true; } // 检查界限(虚的剪枝函数)
```

2. 递归方法1

```
void Perm1(int n) // 生成分量的方法
{   vector<int> X(n); // 解向量
    function<void(int)> Perm = [&](int t)
    {   if(t >= n) cout << X << endl; // 答案
        else
        {   while(NextValue(X, t))
            {   if(legal(t) && bound(t)) // 可行和界限
                    Perm(t + 1);
            }
        }
    };
    X[0] = -1; // 初始化
    Perm(0); // 从0号元素开始
}
```

3. 递归方法2

```
void Perm2(int n) // 对自然排列重新排列的方法
{   vector<int> X(n); // 解向量
    iota(begin(X), end(X), 0); // X初始化为自然排列
    function<void(int)> Perm = [&](int t)
    {   if(t >= n) cout << X << endl; // 答案
        else
        {   for(int i = t; i < n; ++i)
            {   swap(X[t], X[i]);
                if(legal(t) && bound(t)) // 可行和界限
                    Perm(t + 1);
                swap(X[t], X[i]);
            }
        }
    };
    Perm(0); // 从0号元素开始
}
```

4. 迭代方法

```
void Perm(int n) // 用迭代方法生成各个分量
{   vector<int> X(n); // 解向量
    X[0] = -1; // 初始化
    for(int t = 0; t >= 0;)
    {   if(!NextValue(X, t)) --t; // 回溯
        else if(legal(t)) // 可行
        {   if(t >= n - 1) cout << X << endl; // 答案
            else if(bound(t)) ++t; // 界限
        }
    }
}
```

5. 测试程序

```
int main()
{   int n = 3;
    cout << "递归1\n", Perm1(n);
    cout << "递归2\n", Perm2(n);
    cout << "迭代\n", Perm(n);
}
```

递归1	递归2	迭代
0 1 2	0 1 2	0 1 2
0 2 1	0 2 1	0 2 1
1 0 2	1 0 2	1 0 2
1 2 0	1 2 0	1 2 0
2 0 1	2 1 0	2 0 1
2 1 0	2 0 1	2 1 0

8.3 皇后问题

8.3.1 问题分析

约束条件是任何两个皇后都不能位于同一对角线上。用 (i,j) 表示棋盘上第 i 行与第 j 列交叉的位置。如图8-5所示，在同一条平行于主对角线的对角线上的两点 (i,j) 和 (k,l) 满足 $i-j=k-l$；在同一条平行于副对角线的对角线上的两点 (i,j) 和 (k,l) 满足 $i+j=k+l$。这两个关系式可以统一写成 $|i-k|=|j-l|$。反之，如果棋盘上的两点 (i,j) 和 (k,l) 满足 $|i-k|=|j-l|$，则它们一定位于同一条对角线上。所以，如果 $(x_0,x_1,\ldots,x_{n-1})$ 是 $n-$ 皇后问题的一个可行解，则对任何 $i\neq k$，等式 $|i-k|=|x_i-x_k|$ 均不成立。

		(0,2)	
	(1,1)		(1,3)
(2,0)			
	(3,1)		(3,3)

	(0,1)		(0,3)
(1,0)			
	(2,1)		(2,3)
		(3,2)	

图8-5 皇后问题可行性

假定皇后 $0,1,\ldots,k-1$ 的位置已经排好，现在要安排皇后 k 的位置，必须使得对任何 $i=0,1,\ldots,k-1$ 都有 $x_i \neq x_k$，且 $|i-k| = |x_i - x_k|$ 不成立。第一个条件表明可以用排列树来求解该问题，第二个条件的验证可以用一个bool函数legal来完成。

```
function legal(X, t)
{   // 解向量, 当前皇后(当前行)
    for(i = 0; i < t; ++i)
    {   if(abs(X[i] - X[t]) == abs(i - t)) return false;
        // |X[i] - X[t]| == |i - t|表示2个皇后在同一条斜角线上
    }
    return true;
}
```

8.3.2 算法描述

1. 递归方法1

```
// 生成分量的方法,X的起始分量必须初始化为-1
function Queen(t)
{   if(t >= n) Print X; // 答案
    else
    {   while(NextValue(X, t))
        {   if(legal(X, t)) // 可行
                Queen(t + 1); // 转向下一行
        }
    }
}
```

2. 递归方法2

```
// 对自然排列重新排列的方法, X必须初始化为然排列
function Queen(t)
{   if(t >= n) Print X; // 答案
    else
    {   for(int i = t; i < n; ++i)
        {   swap(X[t], X[i]);
            if(legal(X, t)) // 可行
                Queen(t + 1); // 转向下一行
            swap(X[t], X[i]);
        }
    }
}
```

3. 迭代方法

```
function Queen(n) // 用迭代方法生成各个分量
{   // X的起始分量必须初始化为-1, X[t]是当前列
    X[0] = -1; // 初始化
    for(t = 0; t >= 0;) // t是当前行
    {   if(!NextValue(t)) --t; // 回溯
        else if(legal(t)) // 可行
        {   if(t >= n - 1) Print X; // 答案
            else ++t; // 转向下一行
        }
    }
}
```

4. 复杂性分析

在最坏情况下，每检查一个元组需要 $O(n)$ 时间，共需要检查 $O(n!)$ 个元组，所以整个算法的时间复杂度为 $O(n \times n!)$ 。

8.3.3 算法的C++描述*

1. 可行性函数

```
#include "Backtrack.h"
#include "algorithm.h"
bool legal(vector<int> &X, int t)
{   // 解向量, 当前皇后(当前行)
    for(int i = 0; i < t; ++i)
    {   if(abs(X[i] - X[t]) == abs(i - t)) return false;
        // |X[i] - X[t]| == |i - t|表示2个皇后在同一条斜角线上
    }
    return true;
}
```

2. 递归方法1

```
void Queen1(int n) // 生成分量的方法
{   vector<int> X(n); // 解向量
    function<void(int)> Queen = [&](int t)
    {   if(t >= n) cout << X << endl; // 答案
        else
        {   while(NextValue(X, t))
            {   if(legal(X, t)) // 可行
                    Queen(t + 1); // 转向下一行
            }
        }
    };
    X[0] = -1; // 初始化
    Queen(0); // 从第0行开始
}
```

3. 递归方法2

```
void Queen2(int n) // 对自然排列重新排列的方法
{   vector<int> X(n); // 解向量
    iota(begin(X), end(X), 0); // X初始化为然排列, 必须
    function<void(int)> Queen = [&](int t)
    {   if(t >= n) cout << X << endl; // 答案
        else
        {   for(int i = t; i < n; ++i)
            {   swap(X[t], X[i]);
                if(legal(X, t)) // 可行
                    Queen(t + 1); // 转向下一行
                swap(X[t], X[i]);
            }
        }
    };
    Queen(0); // 从第0行开始
}
```

4. 迭代方法

```
void Queen(int n) // 用迭代方法生成各个分量
{   vector<int> X(n); // 解向量
    X[0] = -1; // 初始化
    for(int t = 0; t >= 0;) // t是当前行
    {   if(!NextValue(X, t)) --t; // 回溯
        else if(legal(X, t)) // 可行
        {   if(t >= n - 1) cout << X << endl; // 答案
            else ++t; // 转向下一行
        }
    }
}
```

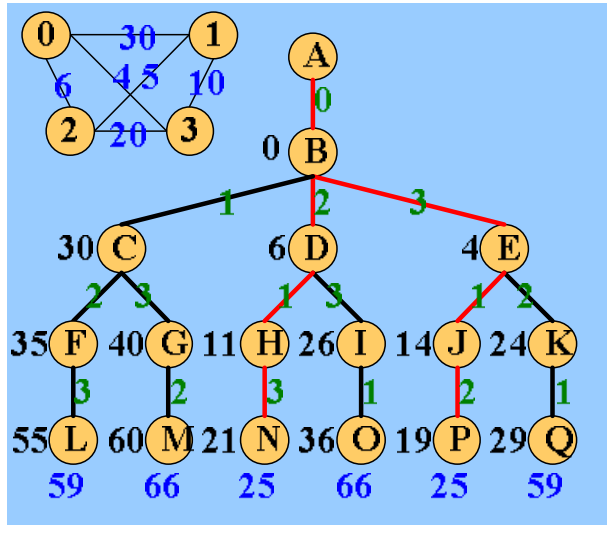
5. 测试程序

```
int main()
{   int n = 4;
    cout << "递归1\n", Queen1(n);
    cout << "递归2\n", Queen2(n);
    cout << "迭代\n", Queen(n);
}
```

递归1	递归2	迭代
1 3 0 2 2 0 3 1	1 3 0 2 2 0 3 1	1 3 0 2 2 0 3 1

8.4 旅行商问题

8.4.1 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。图中各边的权即是城市间的路程（或旅费），当然是非负的。图中任何一个经过每个顶点正好一次的回路称作一个旅行（一条周游路线、一条旅行商回路）。这样，旅行商问题可以描述为给定一个边权值非负的加权图，从中找出一个最小耗费的旅行，即权值之和最小的Hamilton圈。

在如图8-2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)是最小耗费旅行。

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，树中叶的数目为 $(n-1)!$。

回溯方法用深度优先方式从根节点开始，通过搜索解空间树发现一个最小耗费的旅行。对图8-2中的图，利用解空间树，一个可能的搜索为 $ABCFL$ 。在 L 点，旅行(0, 1, 2, 3, 0)作为当前最好的旅行被记录下来。它的耗费是59。从 L 点回溯到活节点 F 。由于 F 没有未被检查的孩子，所以它成为死节点，回溯到 C 点。C 变为扩展节点，向前移动到 G ，然后是 M 。这样构造出了旅行(0, 1, 3, 2, 0)，它的耗费是66。既然它不比当前的最佳旅行好，抛弃它并回溯到 G ，然后是 C 和 B 。从 B 点，搜索向前移动到 D ，然后是 H 和 N 。这个旅行(0, 2, 1, 3, 0)的耗费是25，比当前的最佳旅行好，把它作为当前的最好旅行。从 N 点，搜索回溯到 H ，然后是 D 。在 D 点，再次向前移动，到达 O 点。如此继续下去，可搜索完整个树，得出(0, 2, 1, 3, 0)是最少耗费的旅行。

8.4.2 算法描述

1. 递归方法1

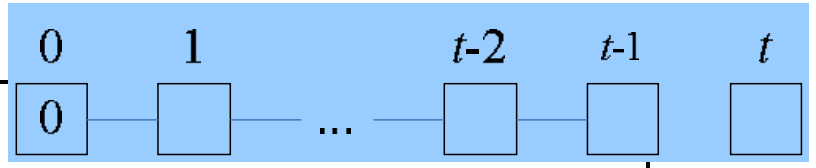
```
// 生成分量的方法
// 用X和CC记录到当前结点的路径和耗费,X的起始分量必须初始化为-1,
// 用BX和BC记录最优路径和最优耗费,BC初始值为无穷大
function TSP1(t, CC)
{   LC = CC + G(X[n - 1], 0); // 当前回路
    if(t >= n && LC < BC) BC = LC, BX = X; // 更优答案, 更新
    else if(t < n)
    {   while(NextValue(X, t))
        {   CC += G(X[t - 1], X[t]); // 当前路径
            if(CC < BC) TSP(t + 1, CC); // 可行(当前路径可能更优)
        }
    }
}
```

2. 递归方法2

```
// 对自然排列重新排列的方法
// 用X和CC记录到当前结点的路径和耗费, X初始化为自然排列,
// 用BX和BC记录最优路径和最优耗费, BC初始值为无穷大
function TSP2(t, CC)
{   double LC = CC + G(X[n - 1], 0); // 当前回路
    if(t >= n and LC < BC) BC = LC, BX = X; // 更优答案, 更新
    else if(t < n)
    {   for(i = t; i < n; ++i)
        {   swap(X[t], X[i]);
            CC += G(X[t - 1], X[t]); // 当前路径
            if(CC < BC) TSP(t + 1, CC); // 可行(当前路径可能更优)
            swap(X[t], X[i]);
        }
    }
}
```

3. 迭代方法

```
function TSP(G) // 用迭代方法生成各个分量
{   X[0] = 0, X[1] = -1; // 初始化, 起始分量必须初始化为-1
    C = 0, BC = inf; // 当前耗费, 最优耗费
    for(t = 1; t >= 1;) // 搜索排列树
    {   if(NextValue(X, t)) // 有可用顶点
        {   CC = C + G(X[t - 1], X[t]); // 当前路径
            LC = CC + G(X[n - 1], 0); // 当前回路
            if(CC < BC) // 可行(当前路径可能更优)
            {   if(t >= n - 1 and LC < BC) BC = LC, BX = X; // 更优答案, 更新
                else if(t < n - 1) C = CC, ++t;
            }
        }
```




```
        else // 无可用顶点，回溯
        {   --t;
            if(t < 1) C = 0;
            else C -= G(X[t - 1], X[t]);
        }
    }
    return BX;
}
```

4. 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是常数时间。

8.4.3 算法的C++描述*

1. 递归方法1

```
#include "Backtrack.h"
#include "algorithm.h"
```

```
auto TSP1(Matrix<double> &G)
{   int n = G.Rows(); // 顶点个数
    vector<int> X(n), BX; // 到当前结点的路径, 最优路径
    double BC = inf; // 最优耗费
```

```
    function<void(int, double)>TSP = [&](int t, double CC)
    {   double LC = CC + G(X[n - 1], 0); // 当前回路
        if(t >= n && LC < BC) BC = LC, BX = X; // 更优答案, 更新
        else if(t < n)
        {   while(NextValue(X, t))
            {   CC += G(X[t - 1], X[t]); // 当前路径
                if(CC < BC) TSP(t + 1, CC); // 可行(当前路径可能更优)
            }
        }
    };
```

```
    X[0] = 0, X[1] = -1; // 初始化
    TSP(1, 0); // 从第1站开始考虑, 当前耗费为0
    return BX;
}
```

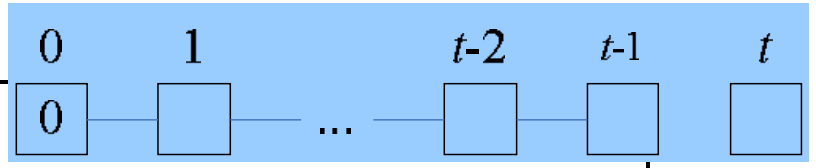
2. 递归方法2

```
auto TSP2(Matrix<double> &G)
{   int n = G.Rows(); // 顶点个数
    vector<int> X(n), BX; // 到当前结点的路径, 最优路径
    iota(begin(X), end(X), 0); // X初始化为自然排列
    double BC = inf; // 最优耗费
    function<void(int, double)>TSP = [&](int t, double CC)
    {   double LC = CC + G(X[n - 1], 0); // 当前回路
        if(t >= n && LC < BC) BC = LC, BX = X; // 更优答案, 更新
        else if(t < n)
        {   for(int i = t; i < n; ++i)
            {   swap(X[t], X[i]);
                CC += G(X[t - 1], X[t]); // 当前路径
                if(CC < BC) TSP(t + 1, CC); // 可行(当前路径可能更优)
                swap(X[t], X[i]);
            }
        }
    };
```

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

3. 迭代方法

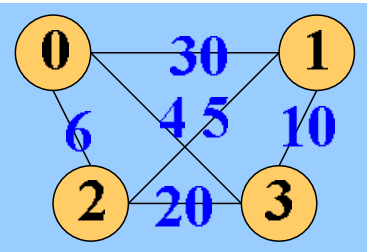
```
auto TSP(Matrix<double> &G)
{   int n = G.Rows(); // 顶点个数
    vector<int> X(n), BX; // 当前路径, 最优路径
    X[0] = 0, X[1] = -1; // 初始化
    double C = 0, BC = inf; // 当前耗费, 最优耗费
    for(int t = 1; t >= 1;) // 搜索排列树
    {   if(NextValue(X, t)) // 有可用顶点
        {   double CC = C + G(X[t - 1], X[t]); // 当前路径
            double LC = CC + G(X[n - 1], 0); // 当前回路
            if(CC < BC) // 可行(当前路径可能更优)
            {   if(t >= n - 1 && LC < BC) BC = LC, BX = X; // 更优答案, 更新
                else if(t < n - 1) C = CC, ++t;
            }
        }
```




```
        else // 无可用顶点，回溯
        {   --t;
            if(t < 1) C = 0;
            else C -= G(X[t - 1], X[t]);
        }
    }
    return BX;
}
```

4. 测试

使用如图8-2所示的图。



最优耗费: 25; 最优路径: 0 2 1 3

```
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}
    };
    double BC; // 最优值BC
    vector<int> BX; // 最优解BX
    cout << "递归1: " << TSP1(G) << endl;
    cout << "递归2: " << TSP2(G) << endl;
    cout << "迭代: " << TSP(G) << endl;
}
```

递归1: 0 2 1 3	递归2: 0 2 1 3	迭代: 0 2 1 3

8.5 子集和问题

8.5.1 问题分析

已知 M 和 $w=(w_0,w_1,...,w_{n-1})$，确定所有满足 $\sum_{i=0}^{n-1} w_i x_i = M$ 的向量 $x=(x_0,x_1,...,x_{n-1})$，其中 $x_i \in \{0,1\}, i=0,1,...,n-1$。

用回溯方法求解的过程也就是生成解空间树的一棵子树的过程，因为期间将剪掉不能产生可行解的子树（即不再对这样的子树进行搜索）。按照前面关于回溯方法的基本思想的说明，搜索采用深度优先的路线，算法只记录当前路径。假设当前路径上的 $x_0,x_1,...,x_t$ 已经确定，算法要决定下一步搜索目标。此时，有 $\sum_{i=0}^{t} w_i x_i = M$ 和 $\sum_{i=0}^{t} w_i x_i < M$ 这两种情况。

若 $\sum_{i=0}^{t} w_i x_i = M$，则当前路径已经是一个可行解，当前扩展结点是一个答案。此时应停止对该路径继续搜索，返回到最近的活结点。

若 $\sum_{i=0}^{t} w_i x_i < M$，则需要判断是否需要继续向前搜索。显然，只有当 $\sum_{i=0}^{t} w_i x_i + \sum_{i=t+1}^{n-1} w_i \geq M$ 时才有必要继续搜索。

8.5.2 算法描述

1. 递归方法1

```
// 和s = sum{W[i] * X[i] | i = 0 to t}, 余额r = sum(W[t + 1 to n - 1])
// 数组X表示每个整数是否选取
function SetSum1(t, s, r)
{   r -= W[t]; // 余额减少
    if(s == M) Print X[0 to t – 1]; // 答案
    else if(t < n)
    {   for(X[t] in {0, 1})
        {   if(X[t] == 1) s += W[t];
            if(s <= M and s + r >= M) Sum(t + 1, s, r); // 可行和界限
        }
    }
}
```

2. 递归方法2

```
// 和s = sum{W[i] * X[i] | i = 0 to t}, 余额r = sum(W[t + 1 to n - 1])
// 数组X表示每个整数是否选取
function SetSum2(t, s, r)
{   r -= W[t]; // 余额减少
    if(s == M) Print X[0 to t – 1]; // 答案
    else if(t < n)
    {   if(s + W[t] <= M) // 检查可行, 生成左儿子
            X[t] = 1, Sum(t + 1, s + W[t], r); // 可行
        if(s + r >= M) // 检查界限, 生成右儿子
            X[t] = 0, Sum(t + 1, s, r);
    }
}
```

3. 迭代方法

```
function SetSum(W, n, M) // 迭代方法
{   // 数组X表示每个整数是否选取
    s = 0, r = sum(W); // 余额r初始为所有元素之和
    r -= W[0]; // 0号数, 余额减少
    X[0] = -1; // 初始化
    for(t = 0; t >= 0;)
    {   if(!NextValue(X, t, 2))
            s -= W[t], r += W[t], --t; // 放回W[t]
        else
        {   if(X[t] == 1) s += W[t];
            if(s <= M) // 可行
            {   if(s == M) Print X[0 to t]; // 答案
                else if(t < n - 1 and s + r >= M) // 界限
                    ++t, r -= W[t]; // t+1号数, 余额减少
            }
        }
    }
}
```

4. 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。

8.5.3 算法的C++描述*

1. 递归方法1

```
#include "Backtrack.h"
#include "algorithm.h"
// 和s = sum{W[i] * X[i] | i = 0 to t}, 余额r = sum(W[t + 1 to n - 1])
void SetSum1(int W[], int n, int M)
{   vector<bool> X(n); // 数组X表示每个整数是否选取
    function<void(int, int, int)> Sum = [&](int t, int s, int r)
    {   r -= W[t]; // 余额减少
        if(s == M) cout << to_string(X, 0, t - 1) << endl; // 答案
        else if(t < n)
        {   for(auto i : {0, 1})
            {   X[t] = i;
                if(X[t] == 1) s += W[t] * X[t];
                if(s <= M && s + r >= M) Sum(t + 1, s, r); // 可行和界限
            }
        }
    };
    int s = 0, r = accumulate(W, W + n, 0); // r初始为所有元素之和
    Sum(0, s, r); // 从0号数开始
}
```

2. 递归方法2

```
void SetSum2(int W[], int n, int M)
{   vector<bool> X(n);
    function<void(int, int, int)> Sum = [&](int t, int s, int r)
    {   r -= W[t]; // 余额减少
        if(s == M) cout << to_string(X, 0, t - 1) << endl; // 答案
        else if(t < n)
        {   if(s + W[t] <= M) // 检查可行, 生成左儿子
                X[t] = 1, Sum(t + 1, s + W[t], r); // 可行
            if(s + r >= M) // 检查界限, 生成右儿子
                X[t] = 0, Sum(t + 1, s, r);
        }
    };
    int s = 0, r = accumulate(W, W + n, 0); // r初始为所有元素之和
    Sum(0, s, r); // 从0号数开始
}
```

3. 迭代方法

```
void SetSum(int W[], int n, int M) // 迭代方法
{   vector<int> X(n);
    int s = 0, r = accumulate(W, W + n, 0); // r初始为所有元素之和
    r -= W[0]; // 0号数，余额减少
    X[0] = -1; // 初始化
    for(int t = 0; t >= 0;)
    {   if(!NextValue(X, t, 2))
            s -= W[t], r += W[t], --t; // 放回W[t]
        else
        {   if(X[t] == 1) s += W[t];
            if(s <= M) // 可行
            {   if(s == M) cout << to_string(X, 0, t) << endl; // 答案
                else if(t < n - 1 && s + r >= M) // 界限
                    ++t, r -= W[t]; // t+1号数，余额减少
            }
        }
    }
}
```

4. 测试程序

```
int main()
{   int M = 30, n = 6;
    int W[n] = {5, 10, 15, 13, 12, 18};
    cout << "原数组: " << to_string(W, n) << endl;
    cout << "递归1\n", SetSum1(W, n, M);
    cout << "递归2\n", SetSum2(W, n, M);
    cout << "迭代\n", SetSum(W, n, M);
}
```

```
原数组: 5 10 15 13 12 18
```

递归1	递归2	迭代
0 0 0 0 1 1	1 1 1 0 0 0	0 0 0 0 1 1
1 0 0 1 1 0	1 0 0 1 1 0	1 0 0 1 1 0
1 1 1 0 0 0	0 0 0 0 1 1	1 1 1 0 0 0

8.6 最大团问题*

输出一个无向图的所有最大团。显然，最大团问题的解空间树是子集树。

8.6.1 剪枝函数

1. 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。

```
function Connected(G, X, t)
{   // 邻接矩阵, 解向量, 当前顶点
    for(u = 0; u < t; ++u) // 检查顶点t与当前团的连接性, legal
        if(X[u] == 1 && G[t][u] == 0) return false;
    return true;
}
```

2. 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。

用变量 cn 表示该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）；用 $\mathrm{cn}+n-t$ 作为顶点数上界 un 的值。

8.6.2 算法描述

1. 递归方法1

```
// 用X和cn记录当前团的顶点和顶点数
// 用BX和fn记录最大团的顶点和顶点数, fn初始值为-1
function Clique1(t, cn)
{   if(t >= n and cn > fn) fn = cn, BX = X; // 答案
    else if(t < n)
    {   for(X[t] in {0, 1})
        {   if(X[t] == 1) ++cn;
            if((X[t] == 0 or Connected(G, X, t)) and cn + n - t > fn) // 可行和界限
                Clique(t + 1, cn);
        }
    }
}
```

2. 递归方法2

```
// 用X和cn记录当前团的顶点和顶点数,
// 用BX和fn记录最大团的顶点和顶点数, fn初始值为-1
function Clique2(t, cn)
{   if(t >= n and cn > fn) fn = cn, BX = X; // 答案
    {   if(Connected(G, X, t)) // 检查可行, 生成左儿子
            X[t] = 1, Clique(t + 1, cn + 1);
        if(cn + n - t > fn) // 检查界限, 生成右儿子
            X[t] = 0, Clique(t + 1, cn);
    }
}
```

3. 迭代方法

```
function Clique(G)
{   // 用X和BX记录当前团的顶点和最大团的顶点
    cn = 0, fn = -1; // 当前团顶点数, 最优顶点数
    X[0] = -1; // 初始化
    for(t = 0; t >= 0;) // 当前处理顶点t
    {   if(!NextValue(X, t, 2)) --cn, --t; // 取出顶点t, 回溯
        else
        {   if(X[t] == 1) ++cn;
            if(X[t] == 0 or Connected(G, X, t)) // 可行
            {   if(t >= n - 1 and cn > fn) fn = cn, BX = X; // 答案
                else if(t < n - 1 and cn + n - t > fn) ++t; // 界限
            }
        }
    }
    return vector<bool> { begin(BX), end(BX) };
}
```

4. 复杂度分析

因为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 $O(n\times n\times 2^n)=O(n^2\times 2^n)$。

8.6.3 算法的C++描述*

1. 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 <vector>
#include "Matrix.hpp"
using namespace std;
```

```
template<class T>
bool Connected(const Matrix<bool> &G, const vector<T> &X, int t)
{   // 邻接矩阵, 解向量, 当前顶点
    for(int u = 0; u < t; ++u) // 检查顶点t与当前团的连接性, legal
        if(X[u] == 1 && G[t][u] == 0) return false;
    return true;
}
```

2. 递归方法1

```
#include "Clique.h"
#include "Backtrack.h"
#include "algorithm.h"
```

```
auto Clique1(Matrix<bool> &G)
{   int n = G.Rows(); // 图顶点数
    vector<bool> X(n), BX; // 当前团, 最大团
    int fn = -1; // 最优顶点数
```

```
    function<void(int, int)> Clique = [&](int t, int cn)
    {   if(t >= n && cn > fn) fn = cn, BX = X; // 答案
        else if(t < n)
        {   for(auto i : {0, 1})
            {   X[t] = i;
                if(X[t] == 1) ++cn;
                if((X[t] == 0 || Connected(G, X, t)) && cn + n - t > fn)
                    Clique(t + 1, cn); // 可行和界限
            }
        }
    };
```

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

3. 递归方法2

```
auto Clique2(Matrix<bool> &G)
{   int n = G.Rows(); // 图顶点数
    vector<bool> X(n), BX; // 当前团, 最大团
    int fn = -1; // 最优顶点数
    function<void(int, int)> Clique = [&](int t, int cn)
    {   if(t >= n && cn > fn) fn = cn, BX = X; // 答案
        else if(t < n)
        {   if(Connected(G, X, t)) // 检查可行, 生成左儿子
                X[t] = 1, Clique(t + 1, cn + 1);
            if(cn + n - t > fn) // 检查界限, 生成右儿子
                X[t] = 0, Clique(t + 1, cn);
        }
    };
    Clique(0, 0); // 从顶点0开始, 当前团顶点数为0
    return BX;
}
```

4. 迭代方法

```
auto Clique(Matrix<bool> &G)
{   int n = G.Rows(); // 图顶点数
    vector<int> X(n), BX; // 当前团, 最大团
    int cn = 0, fn = -1; // 当前团顶点数, 最优顶点数
    X[0] = -1; // 初始化
    for(int t = 0; t >= 0;)
    {   if(!NextValue(X, t, 2)) --cn, --t; // 取出顶点t, 回溯
        else
        {   if(X[t] == 1) ++cn;
            if(X[t] == 0 || Connected(G, X, t)) // 可行
            {   if(t >= n - 1 && cn > fn) fn = cn, BX = X; // 答案
                else if(t < n - 1 && cn + n - t > fn) ++t; // 界限
            }
        }
    }
    return BX;
}
```

5. 测试程序

使用如图8-6所示的无向图。

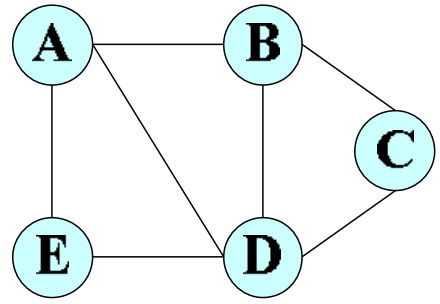


图8-6 最大团问题

```
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}
    };
    cout << "递归1: " << Clique1(G) << endl; // 递归1: 0 1 1 1 0
    cout << "递归2: " << Clique2(G) << endl; // 递归2: 1 1 0 1 0
    cout << "迭代: " << Clique(G) << endl; // 迭代: 0 1 1 1 0
}
```

8.7 0/1背包问题

8.7.1 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。

8.7.2 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。

```
double RestVal(double V[], double W[], int n, double rc, int t)
{   // 从物品t开始可能获得的最大效益, 用于回溯和分支限界
    // V是效益数组, W是重量数组, 按单价降序排列
    // rc是剩余容量
    double rv = 0;
    for(; t < n && 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;
}
```

8.7.3 算法描述

1. 递归方法1

```
// V和W分别是效益数组和重量数组, 已经按单价降序排列
// n是物品数量是, M是背包容量
// X, cv和cw分别是当前解, 当前效益和当前重量
// BX和fv分别是最优解和最后效益, fv初始值为-1
// Rest(t, rc)是Rest(V, W, n, rc, t)的简写
function BKNAP1(t, cv, cw)
{   if(t >= n and cv > fv) fv = cv, BX = X; // 答案(更优)
    else if(t < n)
    {   for(X[t] in {0, 1})
        {   if(X[t] == 1) cv += V[t], cw += W[t];
            if(cw <= M and cv + Rest(t + 1, M - cw) > fv) // 可行和界限
                KNAP(t + 1, cv, cw);
        }
    }
}
```

2. 递归方法2

```
// V和W分别是效益数组和重量数组, 已经按单价降序排列
// n是物品数量是, M是背包容量
// X, cv和cw分别是当前解, 当前效益和当前重量
// BX和fv分别是最优解和最后效益, fv初始值为-1
// Rest(t, rc)是Rest(V, W, n, rc, t)的简写
function BKNAP2(t, cv, cw)
{   if(t >= n) fv = cv, BX = X; // 答案
    else
    {   if(cw + W[t] <= M) // 检查可行, 生成左儿子
            X[t] = 1, KNAP(t + 1, cv + V[t], cw + W[t]);
        if(cv + Rest(t + 1, M - cw) > fv) // 检查界限, 生成右儿子
            X[t] = 0, KNAP(t + 1, cv, cw);
    }
}
```

3. 迭代方法

```
function BKNAP(V, W, n, M)
{   // 效益数组V, 重量数组W, 背包容量M, 共n件物品, 按单价降序排列
    // X和BX分别是当前解和最优解, Rest(t, rc)是Rest(V, W, n, rc, t)的简写
    cv = 0, cw = 0, fv = -1; // 当前效益, 当前重量, 最后效益
    X[0] = -1; // 初始化
    for(t = 0; t >= 0;) // 当前处理物品t
    {   if(!NextValue(X, t, 2)) cv -= V[t], cw -= W[t], --t; // 取出物品t, 回溯
        else
        {   if(X[t] == 1) cv += V[t], cw += W[t];
            if(cw <= M) // 可行
            {   if(t >= n - 1 and cv > fv) fv = cv, BX = X; // 答案(更优)
                else if(t < n - 1 and cv + Rest(t + 1, M - cw) > fv) // 界限
                    ++t;
            }
        }
    }
    return BX;
}
```

4. 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 $O(n)$ 时间，所以该算法的计算时间为 $O(n \times n \times 2^n) = O(n^2 \times 2^n)$ 。

8.7.4 算法的C++描述*

1. 递归方法1

```
#include "KNAP.h"
#include "Backtrack.h"
#include "algorithm.h"
```

```
auto BKNAP1(double V[], double W[], int n, double M)
{   // 效益数组, 重量数组, 物品数量, 背包容量
    auto Rest = [&](int t, double rc) { return RestVal(V, W, n, rc, t); };
    vector<bool> X(n), BX; // 当前解, 最优解
    Sort(V, W, n); // 物品数组按单价降序排列
    double fv = -1; // 最后效益
```

```
    function<void(int, double, double)>
    KNAP = [&](int t, double cv, double cw)
    {   if(t >= n && cv > fv) fv = cv, BX = X; // 答案(更优)
        else if(t < n)
        {   for(auto i : {0, 1})
            {   X[t] = i;
                if(X[t] == 1) cv += V[t], cw += W[t];
                if(cw <= M && cv + Rest(t + 1, M - cw) > fv) // 可行和界限
                    KNAP(t + 1, cv, cw);
            }
        }
    };
    int t = 0; // 从物品0开始
    double cv = 0, cw = 0; // 当前效益和重量
    KNAP(t, cv, cw);
    return BX;
}
```

2. 递归方法2

```
auto BKNAP2(double V[], double W[], int n, double M)
{   // 效益数组, 重量数组, 物品数量, 背包容量
    auto Rest = [&](int t, double rc) { return RestVal(V, W, n, rc, t); };
    vector<bool> X(n), BX; // 当前解, 最优解
    Sort(V, W, n); // 物品数组按单价降序排列
    double fv = -1; // 最后效益
    function<void(int, double, double)>
    KNAP = [&](int t, double cv, double cw)
    {   if(t >= n) fv = cv, BX = X; // 答案
        else
        {   if(cw + W[t] <= M) // 检查可行, 生成左儿子
                X[t] = 1, KNAP(t + 1, cv + V[t], cw + W[t]);
            if(cv + Rest(t + 1, M - cw) > fv) // 检查界限, 生成右儿子
                X[t] = 0, KNAP(t + 1, cv, cw);
        }
    };
```

```
    int t = 0; // 从物品0开始
    double cv = 0, cw = 0; // 当前效益和重量
    KNAP(t, cv, cw);
    return BX;
}
```

3. 迭代方法

```
auto BKNAP(double V[], double W[], int n, double M)
{   // 效益数组, 重量数组, 物品数量, 背包容量
    auto Rest = [&](int t, double rc) { return RestVal(V, W, n, rc, t); };
    Sort(V, W, n); // 物品数组按单价降序排列
    double cv = 0, cw = 0, fv = -1; // 当前效益和重量, 最后效益
    vector<int> X(n), BX; //当前解, 最优解
    X[0] = -1; // 初始化
    for(int t = 0; t >= 0;)
    {   if(!NextValue(X, t, 2)) cv -= V[t], cw -= W[t], --t; // 取出物品t, 回溯
        else
        {   if(X[t] == 1) cv += V[t], cw += W[t];
            if(cw <= M) // 可行
            {   if(t >= n - 1 && cv > fv) fv = cv, BX = X; // 答案(更优)
                else if(t < n - 1 && cv + Rest(t + 1, M - cw) > fv) // 界限
                    ++t;
            }
        }
    }
```

```
    return vector<bool> { begin(BX), end(BX) };
}
```

4. 测试程序

```
int main()
{   int n = 3; // 物品数量
    double V[] = {120, 60, 100}, W[] = {30, 10, 20}; // 效益数组, 重量数组
    double M = 50; // 背包容量
    auto X = BKNAP1(V, W, n, M);
    cout << "// 递归1\n", PrintSolution(V, W, X, n);
    X = BKNAP2(V, W, n, M);
    cout << "// 递归2\n", PrintSolution(V, W, X, n);
    X = BKNAP(V, W, n, M);
    cout << "// 迭代1\n", PrintSolution(V, W, X, n);
}
```

// 递归1	// 递归2	// 迭代1
效益: 60 100 120	效益: 60 100 120	效益: 60 100 120
重量: 10 20 30	重量: 10 20 30	重量: 10 20 30
答案: 0 1 1	答案: 0 1 1	答案: 0 1 1

8.8 图的着色问题*

8.8.1 问题描述

已知一个无向图 G 和 m 种颜色，在只准使用这 m 种颜色对图 G 的顶点进行着色的情况下，是否有一种着色方法，使图中任何两个相邻的顶点都具有不同的颜色？如果存在这样的着色方案，则说图 G 是 m - 可着色的，这样的着色称为图 G 的一种 m - 着色。使得 G 是 m - 可着色的最小数 m 称为图 G 的色数。

图的着色问题是给定无向图 G 和 m 种颜色，求出 G 的所有 m - 着色。

8.8.2 解空间与剪枝函数

1. 解空间

用邻接矩阵 W 表示图 G，$W[i,j]=1$ 表示顶点 i 与 j 相邻（有边相连），否则 $W[i,j]=0$。m 种颜色分别用 $0,1,\ldots,m-1$ 表示。易知，解空间是一个完全 $m-$ 叉树，共 $n+1$ 级，如图8-6所示（这里只画出了可行的着色）。

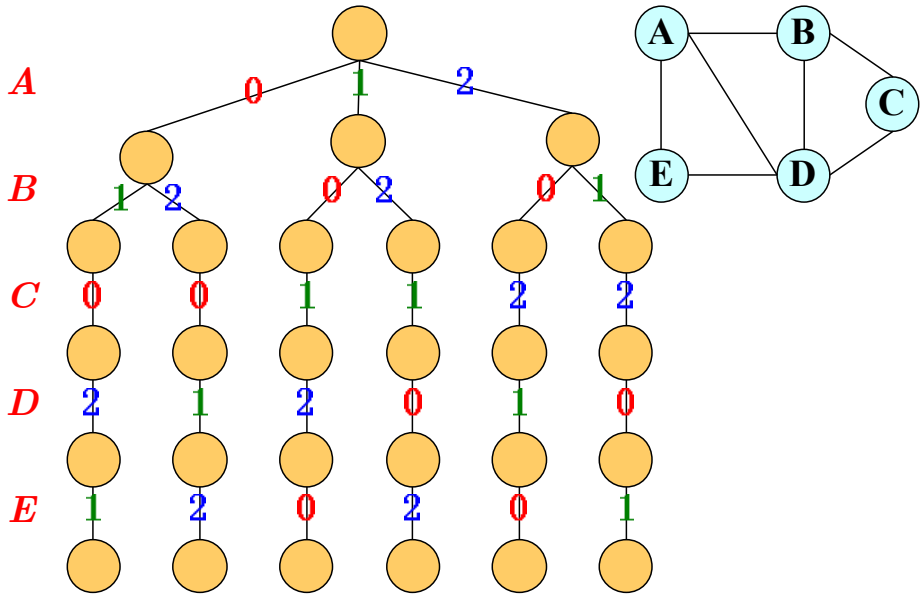


图8-6 着色问题的解空间

2. 剪枝函数

图 G 的一种 m - 着色用一个数组 X 表示，$X[i]=k$ 表示顶点 i 被着上颜色 k。X 是可行解，当且仅当

$$\text{若 } W[i,j]=1\text{，则 } X[i]\neq X[j]$$

这就是约束条件。

假如前 t 个顶点已经着色，即 $X[0],\ldots,X[t-1]$ 已经确定，则确定顶点 t 的颜色 $X[t]$ 时，根据约束条件，应该验证

$$\text{若 } W[i,t]=1\text{，则 } X[i]\neq X[t],\ i=0,\ldots,t-1$$

是否满足。这可以由下列程序完成。保存在文件GraphColor.h中。

```cpp
#pragma once
#include <vector>
#include "Matrix.hpp"
using namespace std;
```

```cpp
bool legal(Matrix<bool> &G, vector<int> &X, int t)
{   for(int i = 0; i < t; ++i) // 验证约束条件
        if(G[i][t] == 1 && X[i] == X[t]) return false;
    return true;
}
```

8.8.3 算法描述

1. 递归算法

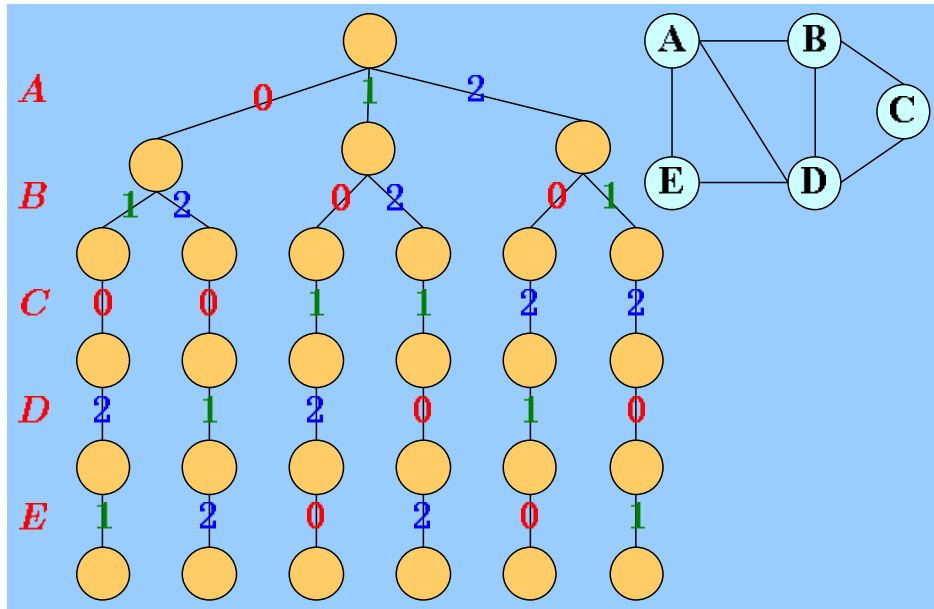
```
// G是图的邻接矩阵, 共有m种颜色(0, 1, ..., m - 1)
// 用数组X记录每个顶点的颜色
function GraphColor1(t)
{   if(t >= n) Print X; // 答案
    else if(t < n)
    {   for(X[t] = 0; X[t] < m; ++X[t])
        {   if(legal(G, X, t)) // 颜色X[t]可用, 考虑下一顶点
                GraphColor(t + 1);
        }
    }
}
```

2. 迭代算法

```
function GraphColor2(G, m) // 迭代
{   // 邻接矩阵, 颜色(0, 1, ..., m - 1)
    // 用数组X记录每个顶点的颜色
    X[0] = -1; // 起始分量初始化为-1
    for(t = 0; t >= 0;) // t是待着色顶点
    {   if(!NextValue(X, t, m)) --t; // 顶点t无可用的颜色, 回溯
        else if(legal(G, X, t)) // 颜色X[t]可用
        {   if(t >= n - 1) Print X; // 答案
            else ++t; // 下一顶点
        }
    }
}
```

3. 复杂性分析

着色问题解空间树的外部结点有 $O(m^n)$ 个（注意，不是可行解，图8-6只画出了可行解）。在最坏情况下，确定每一个外部结点均需耗时 $O(n)$。因此，该算法总耗时 $O(nm^n)$。



8.8.4 算法的C++描述*

1. 递归算法

```
#include "GraphColor.h"
#include "Backtrack.h"
#include "algorithm.h"
```

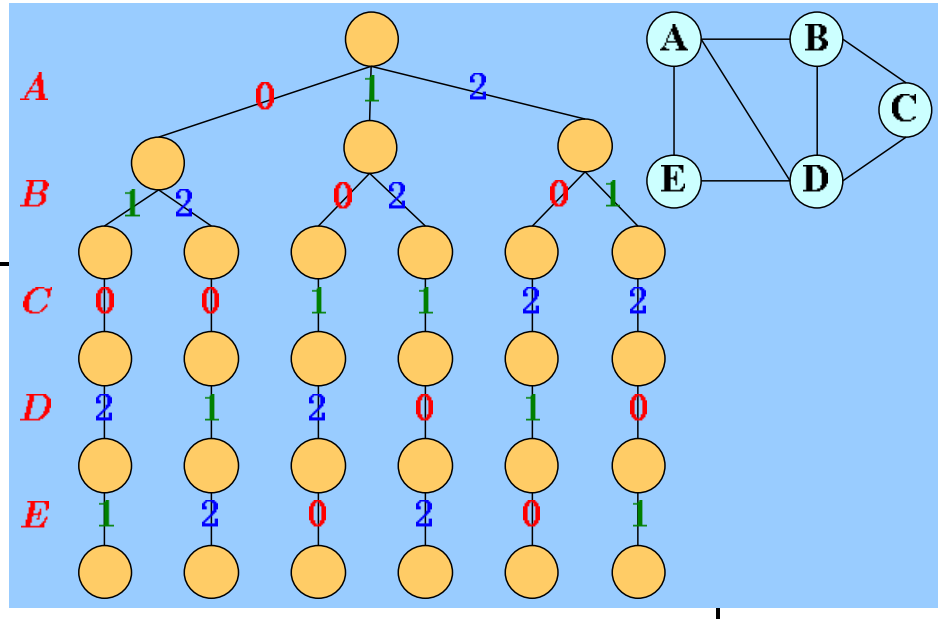
```
void GraphColor1(Matrix<bool> &G, int m)
{   // 邻接矩阵, 颜色(0, 1, ..., m - 1)
    int n = G.Rows(); // 顶点数
    vector<int> X(n); // 当前着色方案
    function<void(int)> GraphColor = [&](int t)
    {   if(t >= n) cout << X << endl; // 答案
        else if(t < n)
        {   for(X[t] = 0; X[t] < m; ++X[t])
            {   if(legal(G, X, t)) // 颜色X[t]可用, 考虑下一顶点
                    GraphColor(t + 1);
            }
        }
    };
    GraphColor(0); // 从顶点0开始
}
```

2. 迭代算法

```
void GraphColor2(Matrix<bool> &G, int m) // 迭代
{   // 邻接矩阵, 颜色(0, 1, ..., m - 1)
    int n = G.Rows(); // 顶点数
    vector<int> X(n); // 当前着色方案
    X[0] = -1; // 初始化
    for(int t = 0; t >= 0;) // t是待着色顶点
    {   if(!NextValue(X, t, m)) --t; // 顶点t无可用的颜色, 回溯
        else if(legal(G, X, t)) // 颜色X[t]可用
        {   if(t >= n - 1) cout << X << endl; // 答案
            else ++t; // 下一顶点
        }
    }
}
```

3. 举例说明

$n=5, m=3$，无向图 G 如图8-6所示。




```
int main()
{   int m = 3; // 颜色数
    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}
    };
    GraphColor1(G, m), cout << endl;
    GraphColor2(G, m), cout << endl;
}
```

0 1 0 2 1 0 2 0 1 2 1 0 1 2 0	1 2 1 0 2 2 0 2 1 0 2 1 2 0 1	0 1 0 2 1 0 2 0 1 2 1 0 1 2 0	1 2 1 0 2 2 0 2 1 0 2 1 2 0 1

8.9 练习题

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或伪代码描述用回溯方法求解下列问题的算法并分析时间复杂性。

皇后问题，Hamilton回路问题（输出所有Hamilton回路），旅行商问题，子集和问题，最大团问题，0/1背包问题

4、将用回溯方法求解子集和问题的递归算法修改成非递归形式，使得只要找到一个满足条件的子集即结束程序。

5、将用回溯方法求解最大团问题的递归算法修改成非递归形式，只要输出一个最大团即可。

6、使用一种程序设计语言或伪代码描述求解下列子集和问题的回溯算法，并分析其时间复杂性。

设 S 是 n 个正整数构成的集合，t 是一个正整数，寻找 S 的满足 $\sum_{x\in S'} x \leq t$

且使得$\sum_{x\in S'}x$最大的子集S'。