


第 10 章 深度优先搜索

10-1 深度优先搜索

10.1.1 深度优先搜索的设计思想

 假设从顶点 u 出发，**深度优先搜索** (depth-first search) 的基本思想是：访问顶点 u ，然后从 u 的未被访问的邻接点中选取一个顶点 v ，再从 v 出发进行深度优先搜索，直至图中所有和 u 有路径相通的顶点都被访问到。

算法：DFS

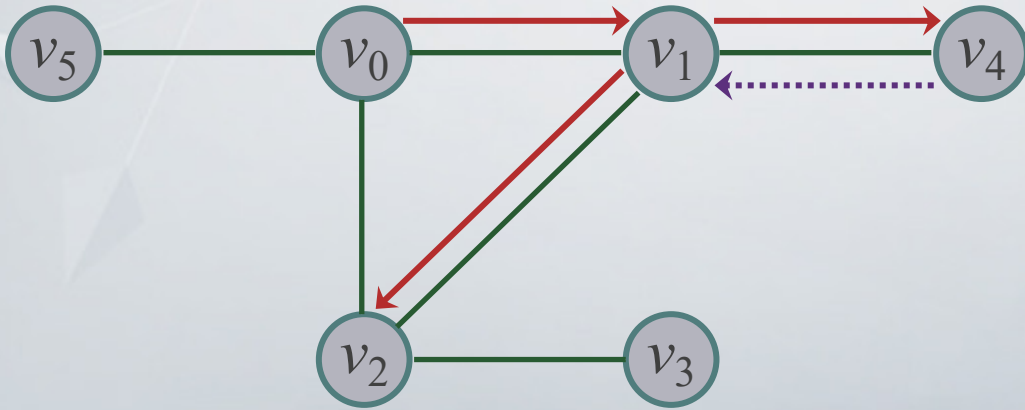
输入：起始顶点 u

输出：搜索过程中访问的顶点序列

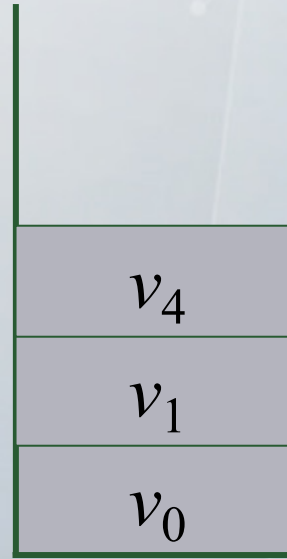
1. 访问顶点 u ；标记顶点 u 已被访问；
2. $v =$ 顶点 u 的邻接点；
3. while (v 存在)
 - 3.1 如果顶点 v 未被访问，则递归执行DFS(v)；
 - 3.2 否则 $v =$ 顶点 u 的下一个邻接点；

10.1.1 深度优先搜索的设计思想

运行实例——深入理解操作过程

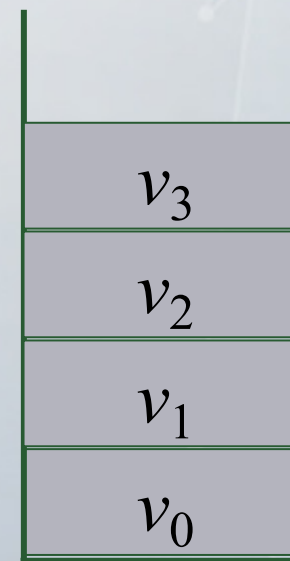
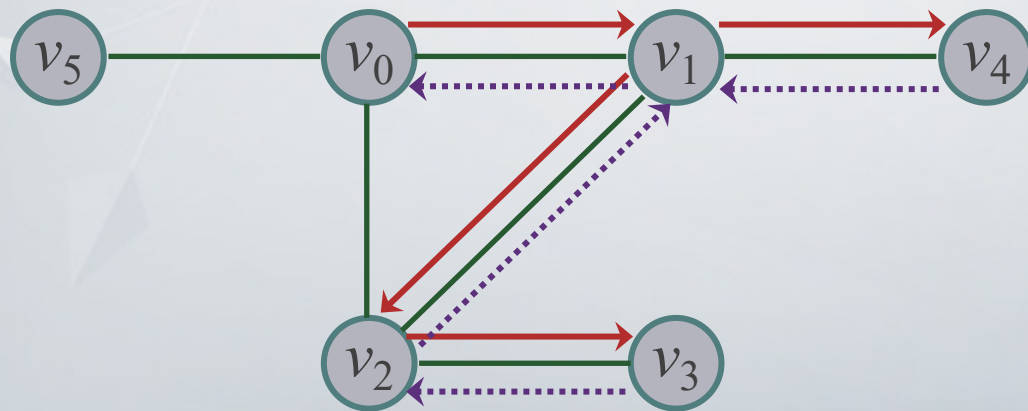


深度优先遍历序列: $v_0 \quad v_1 \quad v_4$




10.1.1 深度优先搜索的设计思想

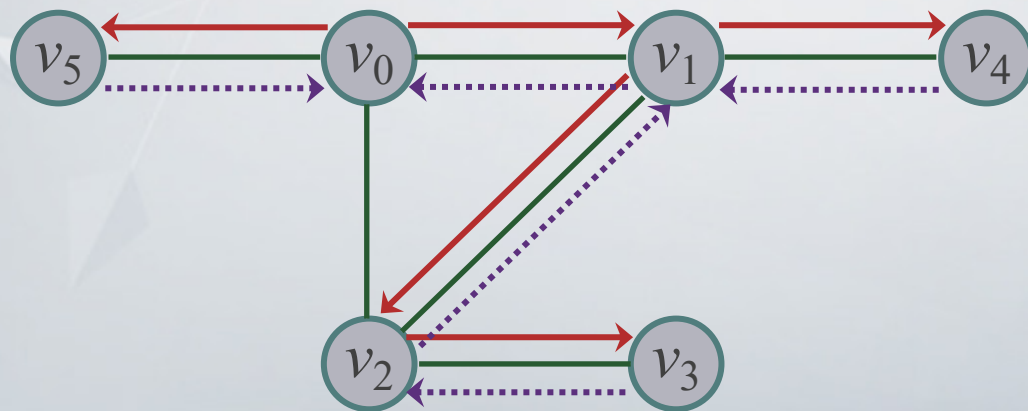
 运行实例——深入理解操作过程



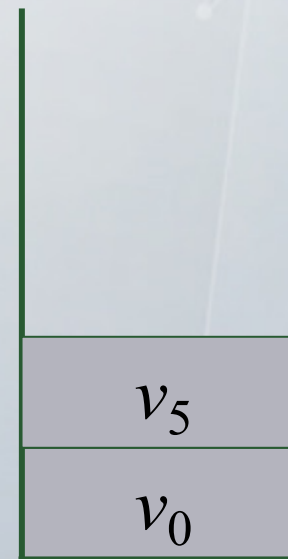
深度优先遍历序列: $v_0 \ v_1 \ v_4 \ v_2 \ v_3$

10.1.1 深度优先搜索的设计思想

 运行实例——深入理解操作过程



深度优先遍历序列: v_0 v_1 v_4 v_2 v_3 v_5



10.1.2 山洞寻宝图

【问题】 在一座山上有 n 个山洞，其中有一个山洞藏有寻宝图，有个猎人知道山上有寻宝图但不知道藏在哪个山洞里，只要猎人到达寻宝图所在的山洞就一定能够得到藏宝图。假设猎人熟悉山路，但是有些山洞之间没有山路相通。给定 n ($3 \leq n \leq 100$) 个山洞之间的连通关系、寻宝图所在山洞、以及猎人寻找的起始山洞，请问猎人是否能够得到寻宝图？

【想法】 将山洞看成顶点，山洞之间的连通关系看成边，从而将山洞及其相通关系抽象为一个无向图。假设寻宝图在山洞 v 中，猎人从山洞 u 出发，可以对无向图从顶点 u 出发进行深度优先搜索，如果可以访问顶点 v ，则猎人可以得到寻宝图，否则顶点 u 和 v 之间没有路径相通，猎人无法得到寻宝图。

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

10.1.2 山洞寻宝图

【算法实现】简单起见，将邻接矩阵`edge[n][n]`、标志数组`visited[n]`、是否找到寻宝图所在山洞标志`flag`均设为全局变量，程序如下：

```
int Dfs(int u, int v)
{
    int j;
    visited[u] = 1;
    if (u == v) return 1;
    for (j = 0; j < n; j++)
    {
        if ((edge[u][j] == 1) && (visited[j] == 0)) {
            Dfs(j, v);
            if (flag == 1) break;
        }
    }
    return flag;
}
```

10.1.3 城堡问题

【问题】 某城堡被分割成 $m \times n$ ($m \leq 50, n \leq 50$) 个方块，每个方块的四面可能有墙，“#”代表有墙，没有墙分割的方块连在一起组成一个房间，城堡外围一圈都是墙。如果 1、2、4 和 8 分别对应左墙、上墙、右墙和下墙，则可以用方块周围每个墙对应的数字之和来描述该方块四面墙的情况，请计算城堡一共有多少个房间，最大的房间有多少个方块。

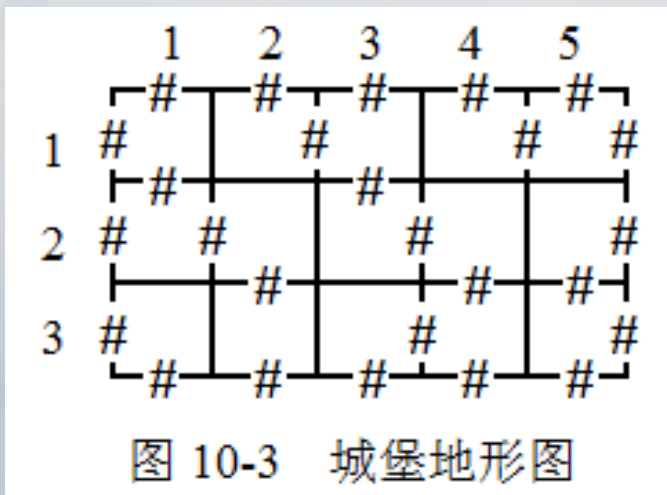


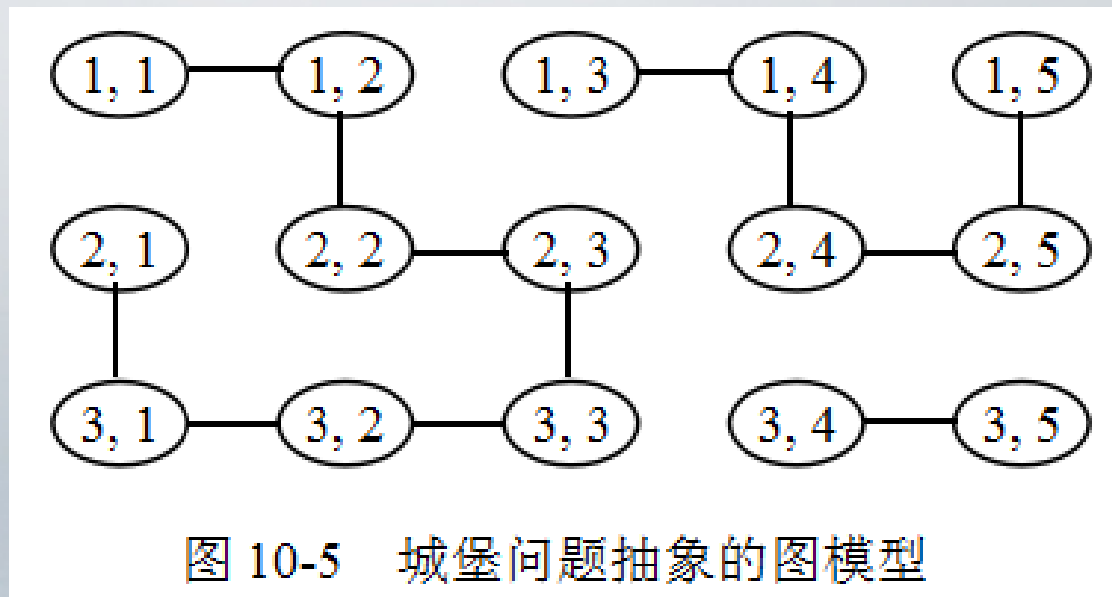
图 10-3 城堡地形图

| | 1 | 2 | 3 | 4 | 5 |
|---|----|----|----|----|----|
| 1 | 11 | 6 | 11 | 6 | 7 |
| 2 | 7 | 9 | 6 | 9 | 12 |
| 3 | 9 | 10 | 12 | 11 | 14 |

图 10-4 城堡地形矩阵

10.1.3 城堡问题

【想法】可以把方块看成顶点，相邻的方块之间如果没有墙，则在方块对应顶点之间连一条边，从而将城堡问题抽象为一个无向图。求城堡的房间个数，实际上就是求图中有多少个连通分量，求城堡的最大房间数，就是求最大连通分量包含的顶点数。



10.1.3 城堡问题

【算法】 设 $\text{Dfs}(i, j)$ 实现从顶点 (i, j) 出发对城堡对应的无向图进行深度优先搜索，反复调用 Dfs 算法直至城堡中所有方块均被访问，则调用算法 Dfs 的次数就是图中连通分量的个数。同时在算法 Dfs 搜索的过程中设置计数器，统计每次调用访问顶点的个数。



如何判断顶点之间的邻接关系？

将方块的数值分别与 1、2、4、8 执行按位与操作，判断对应的二进制位是否为0。

【算法实现】 设数组 $\text{room}[m][n]$ 表示城堡， $\text{visited}[m][n]$ 表示方格是否被访问，变量 roomNum 表示房间个数， maxRoom 表示最大房间的方块数， roomArea 表示当前房间的方块数，函数 $\text{Dfs}(i, j)$ 从方块 (i, j) 出发能够处理与 (i, j) 连通的所有方块。简单起见，将 $\text{room}[m][n]$ 、 $\text{visited}[m][n]$ 、 roomArea 、 maxRoom 设为全局变量，程序如下：

10.1.3 城堡问题

```
void Dfs(int i, int j)
{
    if (visited[i][j] == 1) return;
    roomArea++; visited[i][j] = 1;
    if ((room[i][j] & 1) == 0) Dfs(i, j-1);
    if ((room[i][j] & 2) == 0) Dfs(i-1, j);
    if ((room[i][j] & 4) == 0) Dfs(i, j+1);
    if ((room[i][j] & 8) == 0) Dfs(i+1, j);
}
```


```
int Castle(int m, int n)
{
    int i, j, roomNum = 0;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            if (visited[i][j] == 0) {
                roomNum++; roomArea = 0; Dfs(i, j);
                if (roomArea > maxRoom)
                    maxRoom = roomArea;
            }
    return roomNum;
}
```

【算法分析】 函数Castle的基本语句是Dfs(i, j), 时间复杂度是 $O(m \times n)$ 。

第 10 章 深度优先搜索

10-2 回溯法

10.2.1 问题的解空间树

 **解空间 (solution space)**：所有可能的解向量构成了问题的解空间。将问题的可能解表示为满足某个约束条件的等长向量 $X=(x_1, x_2, \dots, x_n)$ ，其中分量 x_i ($1 \leq i \leq n$) 的取值范围是某个有限集合 $S=\{a_1, a_2, \dots, a_k\}$ 。

例如，对于 n 个物品的 0/1 背包问题，可能解由等长向量 $\{x_1, x_2, \dots, x_n\}$ 组成，其中 $x_i=1$ ($1 \leq i \leq n$) 表示物品 i 装入背包， $x_i=0$ 表示物品 i 没有装入背包，则解空间由长度为 n 的 0/1 向量组成。当 $n=3$ 时，解空间是：

$\{(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$

10.2.1 问题的解空间树

问题的解空间一般用解空间树 (solution space tree, 也称状态空间树) 的方式组织, 树的根结点位于第 1 层, 表示搜索的初始状态, 第 2 层的结点表示对解向量的第一个分量做出选择后到达的状态, 第 1 层到第 2 层的边上标出对第一个分量选择的结果, 依此类推, 从树的根结点到叶子结点的路径就构成了解空间的一个可能解。

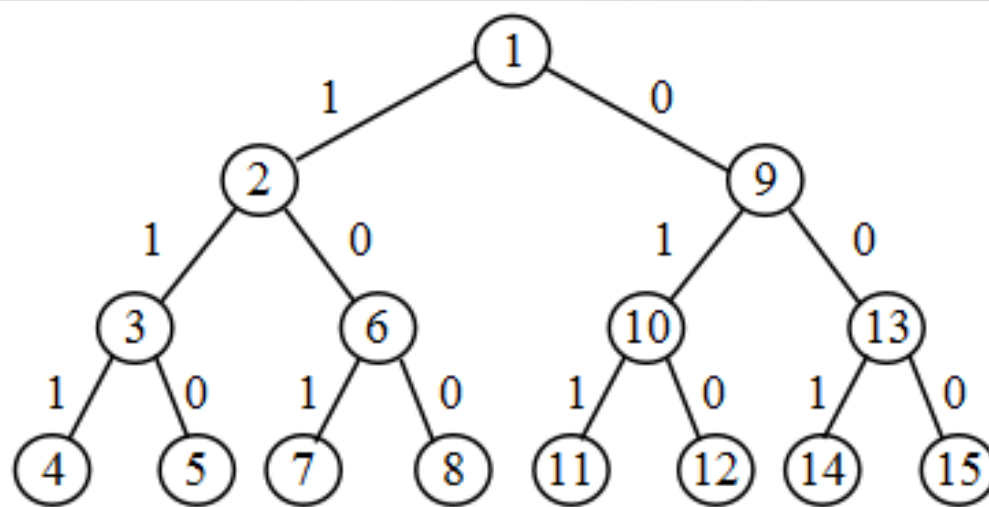


图 10-6 0/1 背包问题的解空间树

10.2.2 回溯法的设计思想

- 📎 **回溯法** (back track method) 从根结点出发, 按照深度优先搜索解空间树, 对于解空间树的某个结点, 如果该结点满足问题的约束条件, 则进入该子树继续进行搜索, 否则跳过以该结点为根的子树, 也就是**剪枝** (pruning)。
- 📎 与蛮力搜索相比, 回溯法的“**聪明**”之处在于能适时回头, 如果再往下走不可能得到解, 就及时**回溯**, 退一步另找路径, 从而避免无效搜索。

问题的解空间树是虚拟的, 并不需要在搜索过程中构建一棵真正的树结构

10.2.2 回溯法的设计思想

由于解向量 $X=(x_1, x_2, \dots, x_n)$ 中每个分量 x_i ($1 \leq i \leq n$) 的值都取自集合 $S=\{a_1, a_2, \dots, a_k\}$, 因此, 可以依次试探集合 S 的元素, 以确定当前分量 x_i 的值。一般情况下, 如果 $X=(x_1, x_2, \dots, x_{i-1})$ 是问题的部分解, 试探集合 S 的元素 a_j 作为分量 x_i 的值, 有下面三种情况:

- (1) 如果 $X=(x_1, x_2, \dots, x_i)$ 是问题的最终解, 则输出这个解
- (2) 如果 $X=(x_1, x_2, \dots, x_i)$ 是问题的部分解, 则扩展下一个分量 x_{i+1}
- (3) 如果 $X=(x_1, x_2, \dots, x_i)$ 既不是问题的部分解也不是问题的最终解, 则存在下面两种情况:

① 如果 a_j 不是集合 S 的最后一个元素, 则令 $x_i = a_{j+1}$, 即取集合 S 的下一个元素作为分量 x_i 的值;

② 如果 a_j 是集合 S 的最后一个元素, 则回溯到 $X=(x_1, x_2, \dots, x_{i-1})$, 假设分量 x_{i-1} 的当前值为 $a_{j'}$, 则令 $x_{i-1} = a_{j'+1}$, 继续进行试探。

10.2.2 回溯法的设计思想

【算法】 假设待求解问题只需得到一个解，如果需要求得所有解或最优解，请修改步骤 3.5，回溯算法的一般框架如下：

算法：回溯算法的一般框架

输入：集合 $S=\{a_1, a_2, \dots, a_k\}$

输出：解向量 $X=(x_1, x_2, \dots, x_n)$

1. 初始化解向量 x_i ($1 \leq i \leq n$) ;
2. $i = 1$ ，表示搜索从根结点开始;
3. 当($k \geq 1$)时执行下述操作：
 - 3.1 令 x_i = 当前值在集合 S 的下一个值;
 - 3.2 如果 $X=(x_1, x_2, \dots, x_i)$ 不是问题的解，转步骤 3.1 继续试探;
 - 3.3 如果试探了集合 S 的所有元素，则 $i = i - 1$ ，转步骤 3.1 进行回溯;
 - 3.4 如果 $X=(x_1, x_2, \dots, x_i)$ 是问题的部分解， $i = i + 1$ ，转步骤 3.1 继续扩展;
 - 3.5 如果 $X=(x_1, x_2, \dots, x_i)$ 是问题的最终解，输出解向量 X ，结束算法;
4. 退出循环，说明问题无解;

10.2.3 回溯法的时间性能

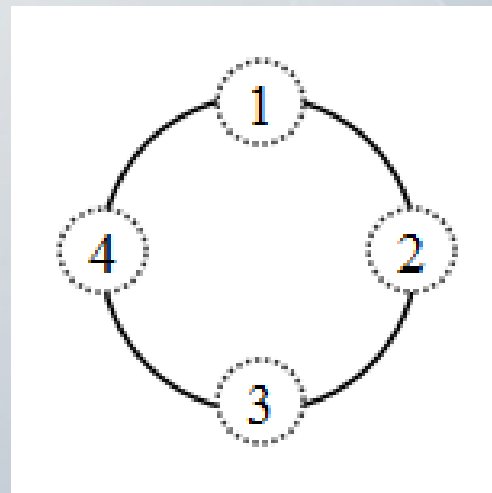
- 一般情况下，在解向量 $X=(x_1, x_2, \dots, x_n)$ 中，分量 $x_i (1 \leq i \leq n)$ 的取值范围为某个有限集合 $S=\{a_1, a_2, \dots, a_k\}$ ，因此，根结点有 $|S|$ 棵子树，则第 2 层有 $|S|$ 个结点，第 2 层的每个结点有 $|S|$ 棵子树，则第 3 层有 $|S| \times |S|$ 个结点，依此类推，第 $n+1$ 层有 $|S|^n$ 个结点，第 $n+1$ 层都是叶子结点，代表问题的所有可能解。
- 回溯法本质上属于蛮力穷举，搜索具有指数阶个结点的解空间树，在最坏情况下，时间代价肯定为指数阶。
- 回溯法的有效性体现在当问题规模 n 很大时，在搜索过程中对问题的解空间树实行大量剪枝。但是，对于具体的问题实例，很难预测回溯法的搜索行为，特别是很难估计在搜索过程中产生的结点数，这是分析回溯法时间性能的主要困难。

10.2.4 素数环问题

【问题】 把整数 $\{1, 2, \dots, 20\}$ 填写到一个环中，要求每个整数只填写一次，并且相邻的两个整数之和是素数。

【想法】 这个素数环有 20 个位置，每个位置可以填写的整数有 1~20 共 20 种可能，可以对每个位置从 1 开始进行试探，约束条件是正在试探的数满足如下条件：

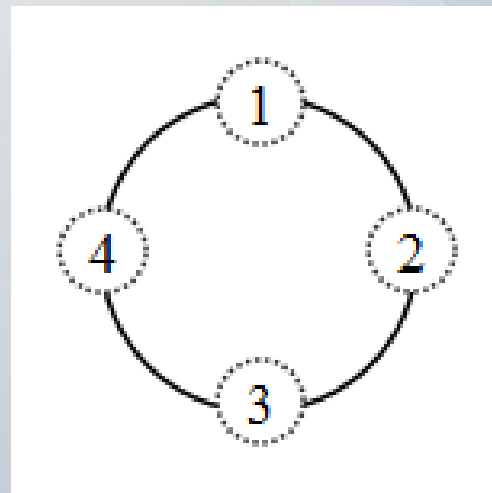
- (1) 与已经填写到素数环中的整数不重复；
- (2) 与前面相邻的整数之和是素数；
- (3) 最后一个填写到素数环中的整数与第一个填写的整数之和是素数。



10.2.4 素数环问题

【算法】 在填写第 i 个位置时，如果满足上述约束条件，则继续填写第 $i+1$ 个位置；如果 $1\sim 20$ 个数都无法填写到第 i 个位置，则取消对第 i 个位置的填写，回溯到第 $i-1$ 个位置。

【算法实现】 设函数Check判断位置 i 的填写是否满足约束条件，数组 $x[n]$ 表示素数环问题的解向量，为了和数组下标一致，素数环的位置为 $0\sim n-1$ ，程序如下：



10.2.4 素数环问题

```
void PrimeCircle(int x[ ], int n)
{
    int i, j;
    for (i = 0; i < n; i++)
        x[i] = 0;
    x[0] = 1;
    for (i = 1; i <= n; )
    {
        x[i] = x[i] + 1;
        while (x[i] <= n)
            if (Check(x, n, i) == 1) break;
        else x[i] = x[i] + 1;
        if (x[i] > n) x[i--] = 0;
        else if (i < n - 1) i = i + 1;
```

```
    else { //求解完毕，输出解
        for (j = 0; j < n; j++)
            cout<<x[j]<<" ";
        return;
    }
    cout<<"问题无解";
}
```

//x[i]填数满足约束条件
//试探下一个数
//回溯
//填写下一个位置

10.2.4 素数环问题

```
int Check(int x[ ], int n, int i)
{
    int j, flag = 0;
    for (j = 0; j < i; j++)
        if (x[i] == x[j]) return 0;
    flag = Prime(x[i] + x[i - 1]);
    if (flag == 1 && i == n - 1)
        flag = Prime(x[i] + x[0]);
    return flag;
}

int Prime(int x)
{
    int i, n;
    n = (int)sqrt(x);
    for (i = 2; i <= n; i++)
        if (x % i == 0) return 0;
    return 1;
}
```

//判断是否重复

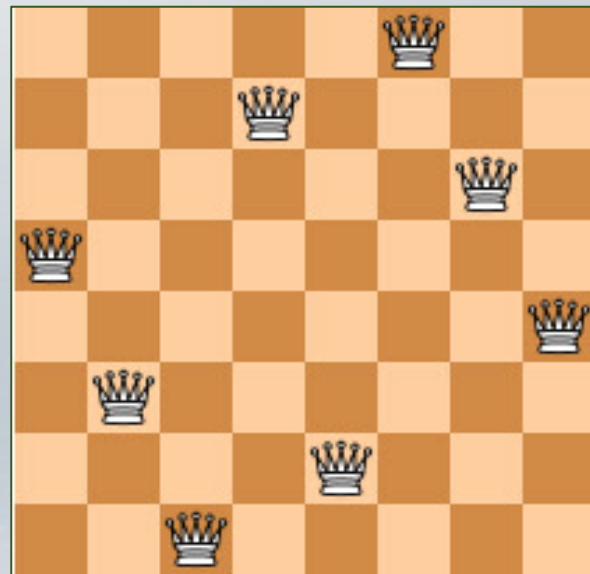
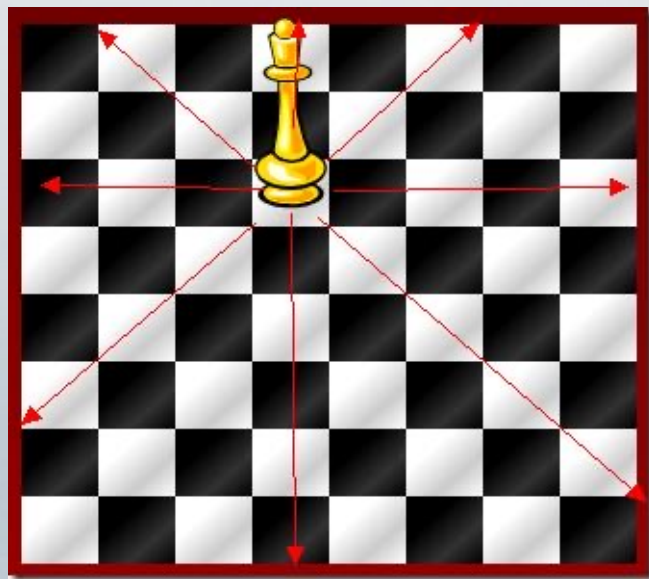
//判断相邻数之和是否素数

//判断第一个和最后一个是否素数

//判断整数x是否素数

10.2.5 八皇后问题

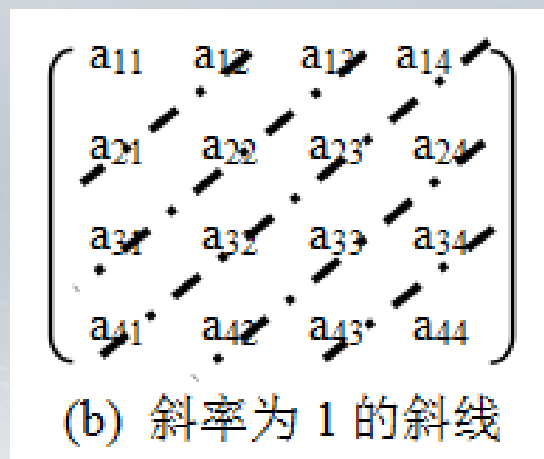
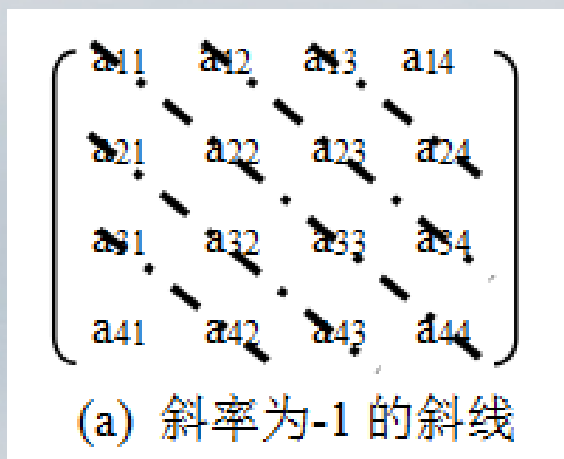
【问题】八皇后问题 (eight queen problem)：在 8×8 的棋盘上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。可以把八皇后问题扩展到 n 皇后问题，即在 $n \times n$ 的棋盘上摆放 n 个皇后，使任意两个皇后都不能处于同一行、同一列或同一斜线上。



10.2.5 八皇后问题

【想法】棋盘的每一行必须摆放一个皇后， n 皇后问题的可能解用向量 (x_1, x_2, \dots, x_n) 表示，即第 i 个皇后摆放在第 i 行第 x_i 列的位置 ($1 \leq i \leq n$ 且 $1 \leq x_i \leq n$)。由于两个皇后不能位于同一列，所以， n 皇后问题的解向量必须满足约束条件 $x_i \neq x_j$ 。

可以将 n 皇后问题的 $n \times n$ 棋盘看成是矩阵，设皇后 i 和 j 的摆放位置分别是 (i, x_i) 和 (j, x_j) ，则在棋盘上斜率为 -1 的同一条斜线上，满足条件 $i - x_i = j - x_j$ ；在棋盘上斜率为 1 的同一条斜线上，满足条件 $i + x_i = j + x_j$ 。综合上述两种情况， n 皇后问题的解必须满足约束条件： $|i - j| \neq |x_i - x_j|$ 。



10.2.5 八皇后问题

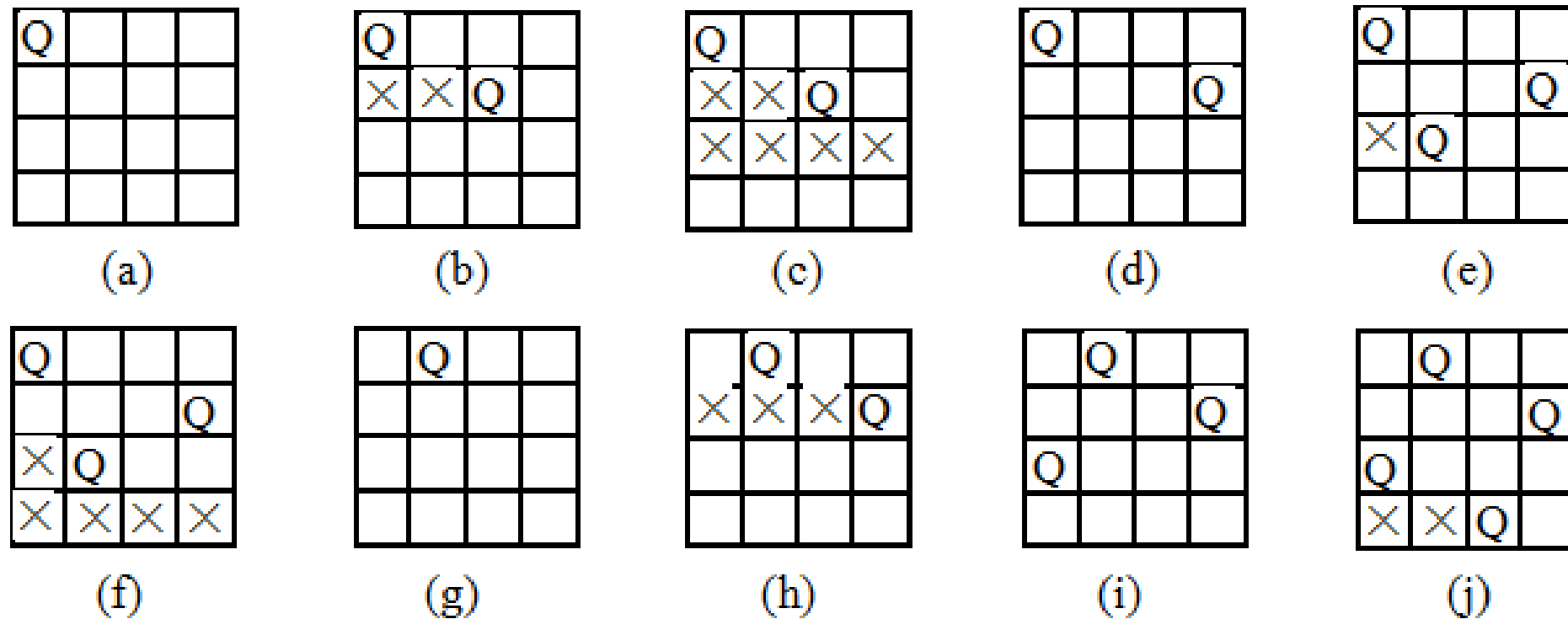


图 10-9 回溯法求解四皇后问题的搜索过程(×表示失败的尝试; Q 表示放置皇后)

10.2.5 八皇后问题

【算法】 设 n 皇后问题的可能解用向量 (x_1, x_2, \dots, x_n) 表示, 其中, 皇后 i 摆放在第 i 行第 x_i 列, 算法如下:

算法: 回溯法求解 n 皇后问题 Queue

输入: 皇后的个数 n

输出: 解向量 (x_1, x_2, \dots, x_n)

1. 初始化解向量 $x_i = 0$ ($1 \leq i \leq n$);
2. $i = 1$;
3. 当 $i \geq 1$ 时摆放皇后 i :
 - 3.1 把皇后 i 摆放在下一列的位置, 即 $x_i = x_i + 1$;
 - 3.2 如果皇后 i 摆放在 x_i 列不发生冲突, 转步骤 3.3;
否则转步骤 3.1 试探下一列;
 - 3.3 若 x_i 出界, 则回溯, $x_i = 0$, $i = i - 1$, 转步骤 3.1 重新摆放皇后 i ;
 - 3.4 若尚有皇后没摆放, 则 $i = i + 1$, 转步骤 3.1 摆放下一个皇后;
 - 3.5 若 n 个皇后已全部摆放, 则输出解向量, 算法结束;
4. 退出循环, 说明 n 皇后问题无解;

10.2.5 八皇后问题

【算法实现】 设函数Place考察皇后 i 的放置是否发生冲突，数组 $x[n]$ 表示 n 皇后问题的解，由于数组下标从0开始，皇后序号也从0开始，首先将数组 $x[n]$ 初始化为-1，然后依次试探摆放皇后 i 。程序如下：

```
void Queue(int x[ ], int n)
{
    int i, j;
    for (i = 0; i < n; i++)
        x[i] = -1;
    for (i = 0; i < n; i++)
    {
        //摆放皇后i
        //在下一列摆放皇后i
        //发生冲突
        while (x[i] < n && Place(x, i) == 1)
            x[i]++;
        if (x[i] == n) x[i--] = -1;
        //重置x[i], 回溯
        //尚有皇后未摆放
        //准备摆放下一个皇后
        else if (i < n - 1)
            i = i + 1;
    }
}
```

10.2.5 八皇后问题

```
    else {  
        for (j = 0; j < n; j++)  
            cout<<x[j]+1<<" ";  
        cout<<endl;  
        return;  
    }  
}  
cout<<"无解"<<endl;  
}  
int Place(int x[ ], int i)  
{  
    int j;  
    for (j = 0; j < i; j++)  
        if (x[i] == x[j] || abs(i - j) == abs( x[i] - x[j]))  
            return 1;  
    return 0;  
}
```

//得到一个解，输出

//打印列号从1开始

//只求出一个解

//冲突，返回1

//不冲突，返回0

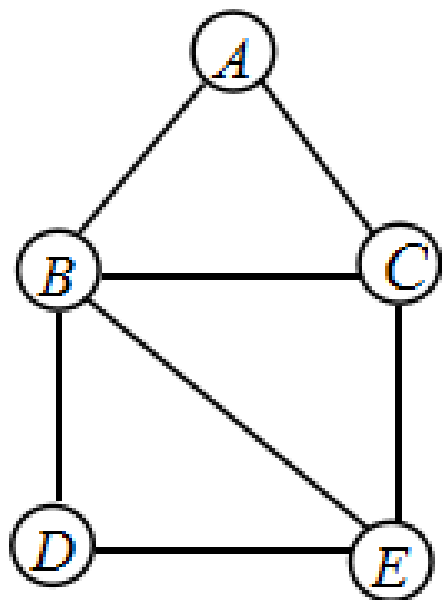
10.2.6 图着色问题

【问题】 给定无向连通图 $G=(V, E)$, **图着色问题** (graph coloring problem) 求最小的整数 m , 用 m 种颜色对 G 的顶点着色, 使得任意两个相邻顶点着色不同。

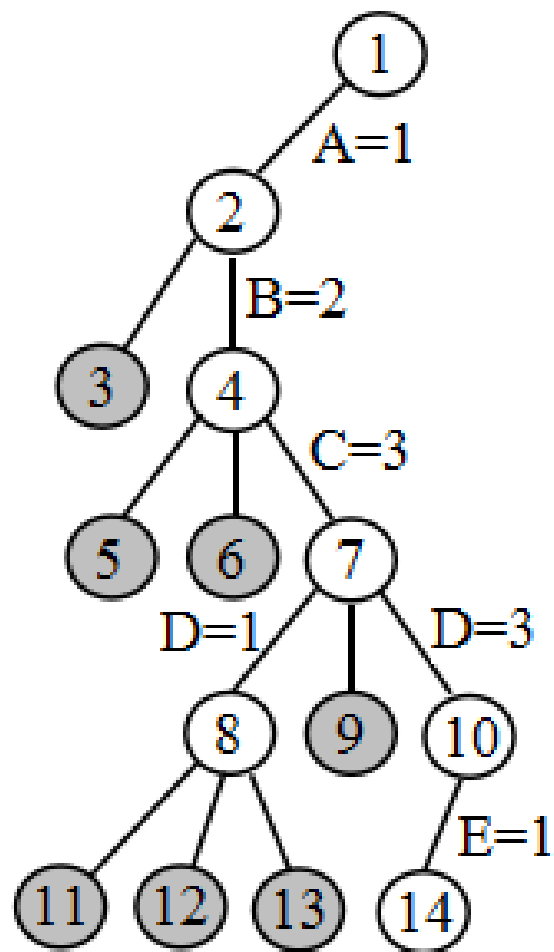
【想法】 用 m 种颜色为无向图 $G=(V, E)$ 着色, 其中, V 的顶点个数为 n , 可以用向量 $C=(c_1, c_2, \dots, c_n)$ 描述图的一种可能着色, 其中, $c_i \in \{1, 2, \dots, m\} (1 \leq i \leq n)$ 表示顶点 i 的着色。

回溯法求解图着色问题, 首先把所有顶点的颜色初始化为 0, 然后依次为每个顶点着色。如果当前顶点着色没有冲突, 则继续为下一个顶点着色, 否则, 为当前顶点着下一个颜色, 如果所有 m 种颜色都试探过并且都发生冲突, 则回溯到当前顶点的前一个顶点, 依此类推。

10.2.6 图着色问题



(a) 一个无向图



(b) 回溯法搜索空间

10.2.6 图着色问题

【算法】 对于图 $G=(V, E)$ ，设数组 $\text{color}[n]$ 表示顶点的着色情况，算法如下：

算法：回溯法求解图着色问题 GraphColor

输入：图 $G=(V, E)$ ， m 种颜色

输出： n 个顶点的涂色情况 $\text{color}[n]$

1. 将数组 $\text{color}[n]$ 初始化为 0;
2. $i = 0$;
3. 当 $i \geq 0$ 为顶点 i 着色：
 - 3.1 依次考察每一种颜色，若顶点 i 的着色与其他顶点的着色不发生冲突，则转步骤 3.2；否则，搜索下一个颜色；
 - 3.2 如果 $\text{color}[i]$ 大于 m ，重置顶点 i 的着色情况， $i=i-1$ ，转步骤 3 回溯；
 - 3.3 若顶点 i 是一个合法着色且顶点尚未全部着色，则 $i=i+1$ ，转步骤 3 处理下一个顶点；
 - 3.4 若顶点已全部着色，则输出数组 $\text{color}[n]$ ，算法结束；

10.2.6 图着色问题

【算法实现】 设无向图采用邻接矩阵 $\text{arc}[n][n]$ 存储, $\text{color}[n]$ 存储 n 个顶点的着色情况, 函数 Ok 判断顶点 i 的着色是否发生冲突, 为避免在函数间传递参数, 将数组 arc 和 color 设为全局变量, 程序如下:

```
void GraphColor(int m)
{
    int i, j;
    for (i = 0; i < n; i++)
        color[i] = 0;
    for (i = 0; i < n; i++) //为顶点i着色
    {
        color[i] = color[i] + 1; //取下一种颜色
        while (color[i] <= m && Ok(i) == 1)
            color[i] = color[i] + 1; //搜索下一个颜色
        if (color[i] > m) color[i--] = 0; //回溯
        else if (i < n - 1) i = i + 1; //处理下一个顶点
    }
}
```


10.2.6 图着色问题

【算法实现】 设无向图采用邻接矩阵 $\text{arc}[n][n]$ 存储, $\text{color}[n]$ 存储 n 个顶点的着色情况, 函数 Ok 判断顶点 i 的着色是否发生冲突, 为避免在函数间传递参数, 将数组 arc 和 color 设为全局变量, 程序如下:

```
else {                                     //求解完毕, 输出解
    for (j = 0; j < n; j++)
        cout<<color[j]<<" ";
    return;
}
}
}
int Ok(int i)
{
    for (int j = 0; j < i; j++)
        if (arc[i][j] == 1 && color[i] == color[j])
            return 1;                     //着色发生冲突返回1
    return 0;
}
```

第 10 章 深度优先搜索

10-3 拓展与演练

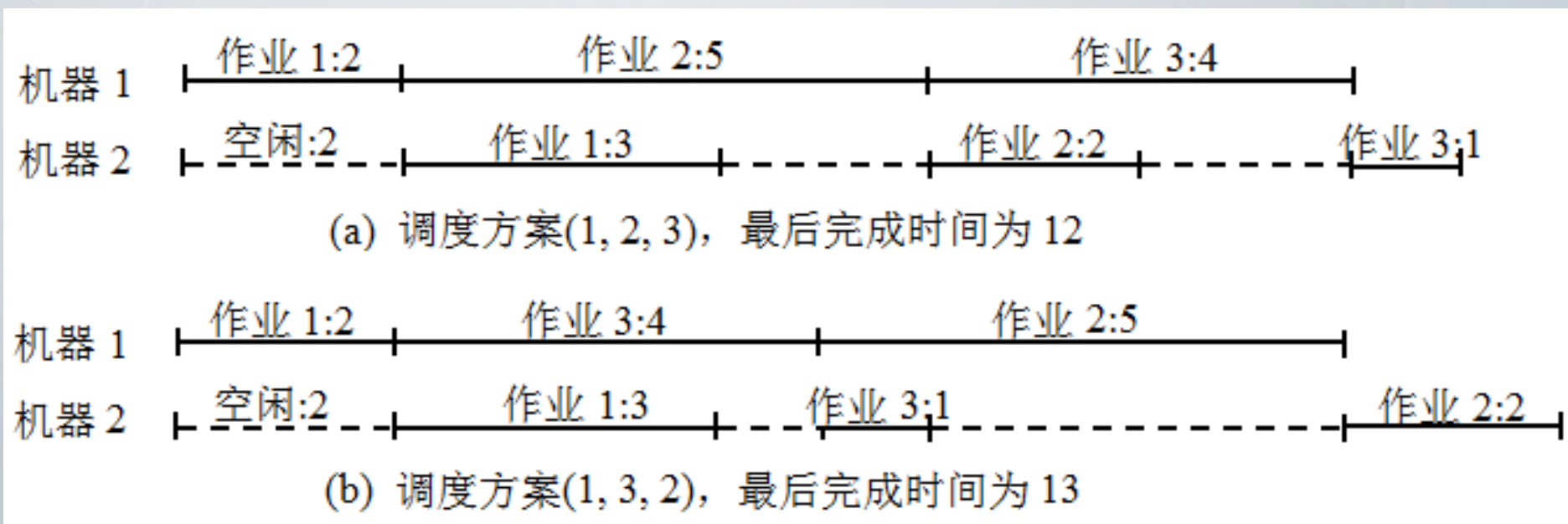
10.3.1 批处理作业调度

【问题】 n 个作业 $\{1, 2, \dots, n\}$ 要在两台机器上处理，每个作业必须先由机器 1 处理，再由机器 2 处理，机器 1 处理作业 i 所需时间为 a_i ，机器 2 处理作业 i 所需时间为 b_i ($1 \leq i \leq n$)，**批处理作业调度问题** (batch-job scheduling problem) 要求确定这 n 个作业的最优处理顺序，使得从第 1 个作业在机器 1 上处理开始，到最后一个作业在机器 2 上处理结束所需时间最少。

【想法】 显然，批处理作业的一个最优调度应使机器 1 没有空闲时间，且机器 2 的空闲时间最小。可以证明，存在一个最优作业调度使得在机器 1 和机器 2 上作业以相同次序完成。

10.3.1 批处理作业调度

例如，有三个作业 $\{1, 2, 3\}$ ，在机器 1 上所需的处理时间为 $(2, 5, 4)$ ，在机器 2 上所需的处理时间为 $(3, 2, 1)$ ，则存在 6 种可能的调度方案： $\{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$ ，相应的完成时间为 $\{12, 13, 12, 14, 13, 16\}$ ，最佳调度方案是 $(1, 2, 3)$ 和 $(2, 1, 3)$ ，最短完成时间为 12。



10.3.1 批处理作业调度

【算法】 设数组 $a[n]$ 存储 n 个作业在机器 1 上的处理时间, $b[n]$ 存储 n 个作业在机器 2 上的处理时间。设数组 $x[n]$ 表示 n 个作业批处理的一种调度方案, 其中 $x[i]$ 表示第 i 个处理的作业编号, $sum1[n]$ 和 $sum2[n]$ 保存在调度过程中机器 1 和机器 2 的当前完成时间, 其中 $sum1[i]$ 表示在安排第 i 个作业后机器 1 的当前完成时间, $sum2[i]$ 表示在安排第 i 个作业后机器 2 的当前完成时间, 且根据下式进行更新:

$$sum1[i] = sum1[i-1] + a[x[i]]$$

$$sum2[i] = \max(sum1[i], sum2[i-1] + b[x[i]])$$

10.3.1 批处理作业调度

算法：回溯法求解批处理调度BatchJob

输入： n 个作业在机器 1 上的处理时间 $a[n]$ ，在机器 2 上的处理时间 $b[n]$

输出：最短完成时间bestTime，最优调度序列 $x[n]$

1. 初始化解向量 $x[n] = \{-1\}$ ；最短完成时间bestTime = MAX;
2. 初始化调度方案中机器 1 和机器 2 的完成时间：
 $sum1[n] = sum2[n] = \{0\}; i = 0;$
3. 当 $i \geq 0$ 时调度第 i 个作业：
 - 3.1 依次考察每一个作业，如果作业 $x[i]$ 尚未处理，则转步骤 3.2，
否则尝试下一个作业，即 $x[i]++$;
 - 3.2 处理作业 $x[i]$:
 - 3.2.1 $sum1[i] = sum1[i-1] + a[x[i]]$;
 - 3.2.2 $sum2[i] = \max\{sum1[i], sum2[i-1]\} + b[x[i]]$;
 - 3.2.3 若 $sum2[i] < bestTime$ ，则转步骤 3.4，否则转步骤 3.3 实施剪枝；
 - 3.3 回溯， $x[i] = -1$ ， $i--$ ，转步骤 3.1 重新处理第 i 个作业；
 - 3.4 若尚有作业没被处理，则 $i++$ ，转步骤 3.1 处理下一个作业；
 - 3.5 若 n 个作业已全部处理，则输出一个解，算法结束；

10.3.2 哈密顿回路

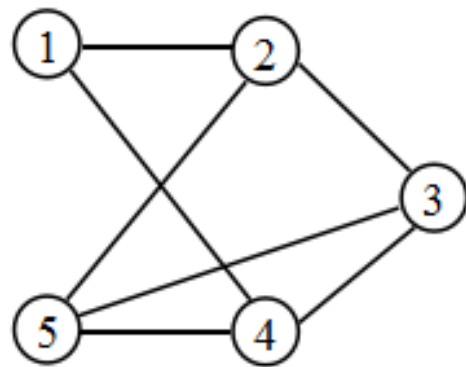
【问题】爱尔兰数学家哈密顿 (William Hamilton, 1805—1865) 提出了著名的周游世界问题。**哈密顿回路问题** (Hamilton cycle problem) 要求从一个城市出发, 经过每个城市恰好一次, 然后回到出发城市。

【想法】图 $G=(V, E)$ 的顶点集为 $V=\{1, 2, \dots, n\}$, 哈密顿回路的可能解表示为 n 元组 $X=(x_1, x_2, \dots, x_n)$, 其中, $x_i \in \{1, 2, \dots, n\}$ 。有如下约束条件:

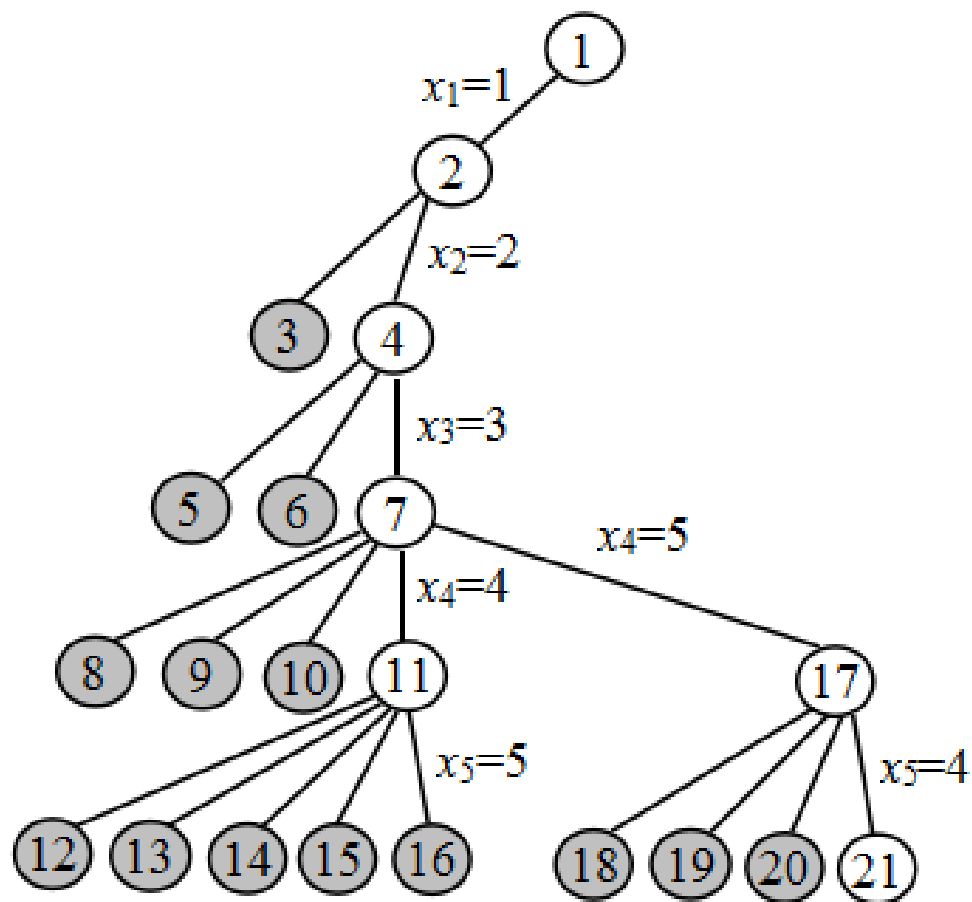
$$\begin{cases} (x_i, x_{i+1}) \in E \quad (1 \leq i \leq n-1) \\ (x_n, x_1) \in E \\ x_i \neq x_j \quad (1 \leq i, j \leq n, \quad i \neq j) \end{cases}$$

回溯法求解哈密顿回路问题, 在解空间树中从**根结点**开始搜索, 如果从根结点到当前结点对应一个部分解, 则在当前结点处选择**第一棵子树**继续搜索, 否则, 对当前子树的**兄弟结点**进行搜索, 如果当前结点的所有子树都已试探并且发生冲突, 则**回溯**到当前结点的父结点。

10.3.2 哈密顿回路



(a) 一个无向图



(b) 哈密顿回路的搜索空间

10.3.2 哈密顿回路

【算法】 设数组 $x[n]$ 存储哈密顿回路上的顶点，数组 $visited[n]$ 存储顶点的访问标志， $visited[i]=1$ 表示哈密顿回路经过顶点 i ，算法如下：

算法：回溯法求解哈密顿回路问题 Hamilton

输入：无向图 $G=(V, E)$

输出：哈密顿回路

1. 将顶点数组 $x[n]$ 初始化为 -1，标志数组 $visited[n]$ 初始化为 0；
2. 从顶点 0 出发构造哈密顿回路： $visited[0] = 1; x[0] = 0; i = 1;$
3. 当 $i \geq 1$ 时扩展哈密顿回路的第 i 个顶点：
 - 3.1 $x[i] = x[i] + 1$ ，搜索下一个顶点；
 - 3.2 如果顶点 $x[i]$ 不在哈密顿回路上并且 $(x[i-1], x[i]) \in E$ ，转步骤 3.4；
否则转步骤 3.1 试探下一个顶点；
 - 3.3 如果试探了 n 个顶点， $visited[i] = 0$ ， $i = i - 1$ ，转步骤 3 进行回溯；
 - 3.4 若数组 $x[n]$ 构成哈密顿路径的部分解，则 $i = i + 1$ ，转步骤 3；
 - 3.5 若数组 $x[n]$ 已形成哈密顿路径，则输出数组 $x[n]$ ，算法结束；