作业4

202208010512 计科2205 刘志垚

- 算法分析题 5-3 回溯法重写0-1背包
  - 1.题目描述
  - 2.思路
  - 3.代码
  - 4.结果
  - 5.分析
- 算法分析题 5-5 旅行商问题(剪枝)
  - 1.题目描述
  - 2.解答
  - 3.代码
  - 4.结果
  - 5.分析
- 算法实现题 5-2 最小长度电路板排列问题
  - 1.题目描述
  - 2.思路
  - 3.代码
  - 4.结果
  - 5.分析
- 算法实现题 5-7 n色方柱问题
  - 1.题目描述
  - 2.思路
  - 3.代码
  - 4.结果
  - 5.分析

算法分析题 5-3 回溯法重写0-1背包

1.题目描述

5-3　重写 0-1 背包问题的回溯法，使算法能输出最优解。

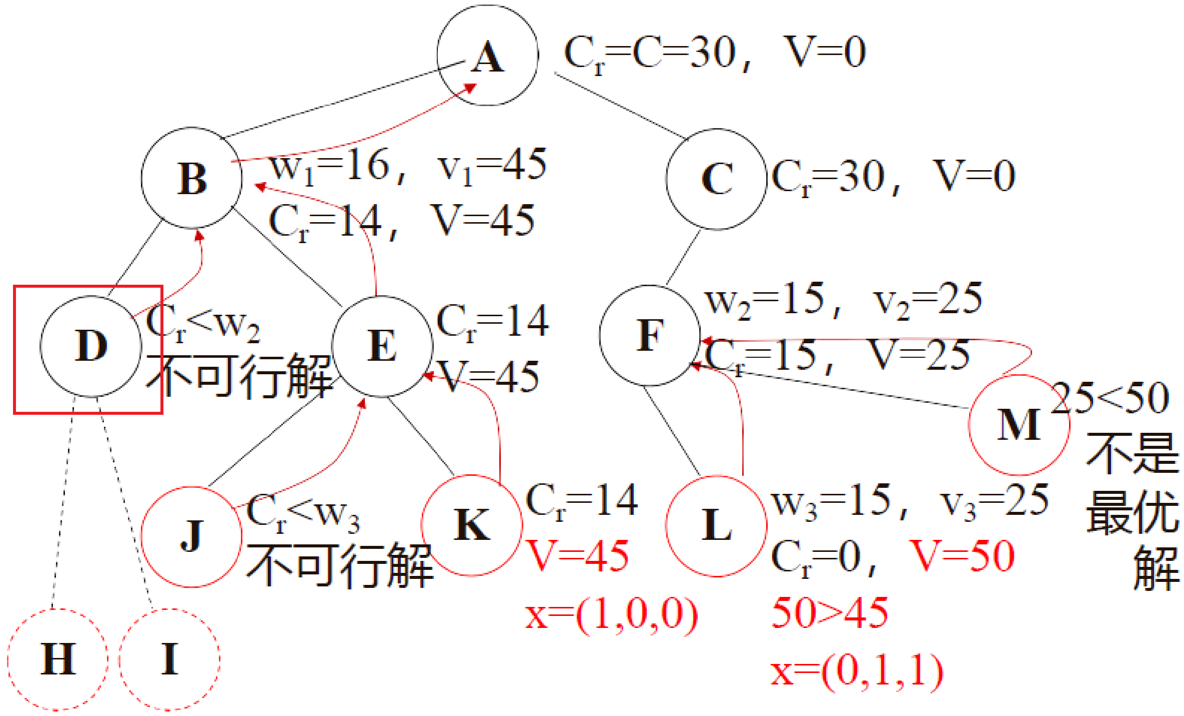
背包容量为 c，物体 i 的重量和价值分别为 $w[i]$ 和 $v[i]$，求最大价值。

2.思路

构造子集树: 每一个物品有两种状态：放和不放，使用一个数组 $x[]$ 来记录。解空间是一个满二叉树。

确定约束函数: $\sum_{i=1}^{n} x_i w_i \leq c$。

确定限界函数:



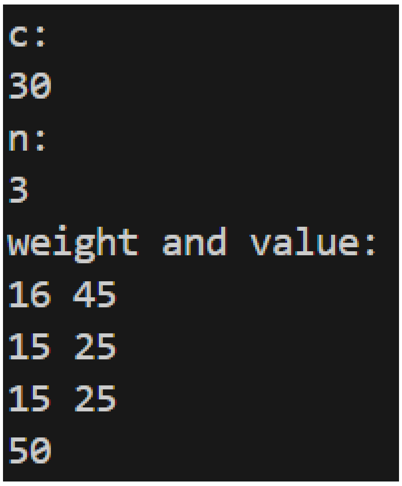
如果某一节点的重量已经大于背包容量 c，便不再扩展此节点。例如上图的节点D。

3.代码

```
#include <iostream>
#define endl '\n'
using namespace std;
const int N = 1005;
int n, c;
int w[N], v[N];
int x[N]; // 标记是否选择物品i
int ans = 0;
// 定义界限函数
bool bound(int dep, int sum_weight) {
    return sum_weight <= c;
}
void backtrack(int dep, int sum_weight, int sum_value) {
    if(sum_weight > c) return; // 剪枝
    if(dep >= n) { // 深度超过n，终止
        if(sum_value > ans) ans = sum_value;
    } else {
        // for(int i = 0; i <= 1; ++i) {
        //      x[dep] = i;
        //      // 判断是否满足界限条件
        //      if(bound(dep)) { // 满足界限，向下递归
        //          backtrack(dep + 1);
        //      }
        // }
        // 满足界限条件，向下递归
        if(bound(dep, sum_weight)) {
            // 选择物品
            backtrack(dep + 1, sum_weight + w[dep], sum_value + v[dep]);
            // 不选择物品
            backtrack(dep + 1, sum_weight, sum_value);
        }
```

```cpp
    }
    return;
}
int main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    cin >> c;

    cin >> n;

    for(int i = 0; i < n; i++) cin >> w[i] >> v[i];
    backtrack(0, 0, 0);
    cout << ans;
    return 0;
}
```

4.结果



```
c:
30
n:
3
weight and value:
16 45
15 25
15 25
50
```

5.分析

深度：回溯树的深度是物品的数量 `n`。

分支数：每个物品的选择有两个分支：选择该物品或不选择该物品，所以每个节点的分支数是 2。

因此，回溯树的总节点数是 `2^n`，即存在 `2^n` 种选择方案。时间复杂度为:

$$O(2^n)$$

算法分析题 5-5 旅行商问题(剪枝)

1.题目描述

5-5　设 G 是一个有 n 个顶点的有向图，从顶点 i 发出的边的最大费用记为 $\max(i)$。

（1）证明旅行售货员回路的费用不超过 $\sum_{i=1}^{n}\max(i)+1$。

（2）在旅行售货员问题的回溯法中，用上面的界作为 bestc 的初始值，重写该算法，并尽可能地简化代码。

2.解答

（1）旅行销售员问题的路径总费用，就是所有被选择的边的费用之和。如果最坏情况下每个顶点 i 都选择了从 i 出发的最大费用边 $max(i)$，那么路径的总费用就是 $\sum_{i=1}^{n} \max(i)$。由于路径的实际费用可能小于这个最大费用和，因此加上 1 作为常数项，用来确保这个上限覆盖所有可能的情况。

因此：旅行销售员回路的费用不超过 $\sum_{i=1}^{n} \max(i) + 1$。

（2）**代码设计思路**:

易得解空间树是一个**排列树**。

在回溯法中，我们通常使用一个变量 bestc 来存储当前最小的路径费用，初始时 bestc 可以设置为 $\sum_{i=1}^{n} \max(i) + 1$，这是一个上界值。这样可以加速搜索过程，避免搜索不必要的部分。

如果 $currCost + cost[curr][i] < bestc$，剪枝。

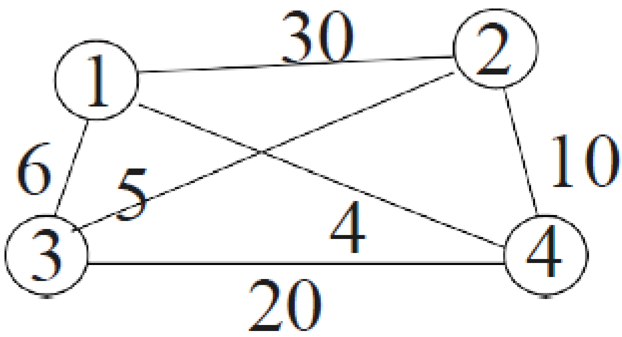
3.代码

```
#include <iostream>
#include <algorithm>
#define endl '\n'
using namespace std;
const int N = 1005;
int n;
int bestc;
int cost[N][N];
int vis[N];
int cal_bestc() {
    int maxSum = 0;
    for(int i = 0; i < n; i++) {
        maxSum += *max_element(cost[i], cost[i] + n);
    }
    return maxSum + 1;
}
void dfs(int s, int curr, int cnt, int currCost) {
    // 如果已经遍历完所有城市
    if(cnt > n) {
        currCost += cost[curr][s];
        bestc = min(bestc, currCost);
        return;
    }
    for(int i = 1; i <= n; i++) {
        if(!vis[i] && currCost + cost[curr][i] < bestc) { // 剪枝
            vis[i] = 1;
            dfs(s, i, cnt + 1, currCost + cost[curr][i]);
            vis[i] = 0;
        }
    }
}
int main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    cin >> n;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) {
            cin >> cost[i][j];
        }
```

```
    }
    bestc = cal_bestc(); // 初始上界

    dfs(1, 1, 1, 0); // 从起点开始

    cout << bestc;
    return 0;

}
```

4.结果

给定这样一个图：



最佳路径为 (1,3,2,4,1)，费用为25。

```
4
0 30 6 4
30 0 5 10
6 5 0 20
4 10 20 0
25
```

5.分析

在最坏情况下，回溯法需要遍历所有可能的路径。对于 n 个城市，可能的路径数量为 $(n-1)!$，因为第一个城市固定为起点，剩下的 $n-1$个城市可以任意排列。

因此，回溯法求解TSP问题的时间复杂度为:

$$O((n-1)!)$$

剪枝一定程度上加快了搜索速率。

算法实现题 5-2 最小长度电路板排列问题

1.题目描述

5-2 最小长度电路板排列问题。

问题描述： 最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设 $B=\{1, 2, \cdots, n\}$ 是 n 块电路板的集合。集合 $L=\{N_1, N_2, \cdots, N_m\}$ 是 n 块电路板的 m 个连接块。其中每个连接块 N_i 是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。在最小长度电路板排列问题中，连接块的长度是指该连接块中第 1 块电路板到最后 1 块电路板之间的距离。

试设计一个回溯法，找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

算法设计： 对于给定的电路板连接块，设计一个算法，找出所给 n 个电路板的最佳排列，使得 m 个连接块中最大长度达到最小。

数据输入： 由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 m（$1\leqslant m, n\leqslant 20$）。接下来的 n 行中，每行有 m 个数。第 k 行的第 j 个数为 0 表示电路板 k 不在连接块 j 中，为 1 表示电路板 k 在连接块 j 中。

结果输出： 将计算的电路板排列最小长度及其最佳排列输出到文件 output.txt。文件的第一行是最小长度；接下来的 1 行是最佳排列。

输入文件示例	输出文件示例
input.txt	output.txt
8 5	4
11111	54316287
01010	
01110	
1011010100	
11010	
00001	
01001	

2.思路

考虑利用回溯法系统地搜索电路板的排列，同时通过连接块的状态来判断插槽之间的连线密度。

1. **问题建模**:
   - 使用数组 `B` 表示电路板与连接块的关系，其中 `B[i][j]` 的值为1表示电路板 `i` 在连接块 `N_j` 中。
   - 定义 `total[j]` 为连接块 `N_j` 中的电路板数量。
2. **部分排列与状态定义**:
   - 对于电路板的部分排列 `x[1:i]`，定义 `now[j]` 为在 `x[1:i]` 中包含的连接块 `N_j` 的电路板数量。
   - 通过 `now[j]` 可以判断插槽之间的连线情况。
3. **连线跨越条件**:
   - 插槽 `i` 和 `i+1` 之间的连线密度的条件是：
     - `now[j] > 0`: 表示插槽 `i` 中包含连接块 `N_j` 的电路板。
     - `now[j] != total[j]`: 表示在前 `i` 个插槽中，并未用尽连接块 `N_j` 中的所有电路板，意味着在后续插槽中仍可能存在该连接块的电路板。
4. **插槽跨越限制**:
   - 每个插槽最多被一条导线跨越一次，这意味着每根导线只能在某个相邻的插槽之间进行跨越。
   - 通过这种方式，可以计算出连接块的密度，进而评估不同排列的优劣。

5. **回溯法应用**:
    - 利用回溯法遍历所有可能的电路板排列，动态更新 `now[j]` 的值，并在每次插入新电路板时检查插槽之间的连线条件。
    - 通过这种方式，逐步逼近最优解。

关键点

- **状态管理**: 通过 `now` 数组有效跟踪当前排列中连接块的状态，确保在回溯过程中能够准确判断插槽之间的连线情况。
- **条件判断**: 理解 `now[j] > 0` 和 `now[j] != total[j]` 的意义是解决问题的关键，这直接影响到插槽之间的连线密度计算。

3.代码

```
// 电路板排列问题
#include<bits/stdc++.h>
using namespace std;
class Board
{
    friend int Arrangement(int **, int, int, int*);
    private:
        void Backtrack(int i, int cd);
        int n,      //电路板数
            m,      //连接块数
            *x,     //当前解
            *bestx, //当前最优解
            bestd,  //当前最优密度
            *total, //total[j]为连接块j的电路板数
            *now,   //now[j]为当前解中所含连接块j的电路板数
            **B;    //连接块数组
};
void Board::Backtrack(int i, int cd)        //回溯搜索排列树,cd表示已经确定的x[1:i]个插槽中相邻两个插槽被跨越数最大的(就是密度)
{
    static int k = 1;
    if(i == n)  //由于算法仅完成那些比当前最优解更好的排列,故cd肯定优于bestd,直接更新
    {
        cout<<"当前已经确定下来最后一个插槽,我们选择"<<x[n]<<endl;
        cout<<"第"<<k++<<"个方案为: ";
        for(int j=1; j<=n; j++)
        {
            bestx[j] = x[j];
            cout<<x[j]<<" ";
        }
        bestd = cd;
        cout<<"获得的密度为: "<<bestd<<endl<<"到达最后一层,回溯一层到达第"<<n-1<<"层"
<<endl;
    }
    else
    {
        for(int j=i; j<=n; j++)      //选择x[j]为下一块电路板
        {
            int ld = 0;      //新的排列部分密度
```

```
            for(int k=1; k<=m; k++)     //遍历连接块1~m,并且计算得到跨越插槽i和i+1的导线
数ld
            {
                now[k] += B[x[j]][k];
                if(now[k]>0 && total[k]!=now[k])  //判断是否发生了跨越(左边有,右边也
有)
                    ld++;
            }
            for(int j=1; j<=m; j++) cout<<now[j]<<" ";
            cout<<endl;
            if(cd > ld)
            {
                ld = cd;
                cout<<"ld<cd, ld已经被更新为"<<cd<<endl;
            }
            if(ld < bestd)  //满足剪枝函数,搜索子树
            {
                swap(x[i], x[j]);
                cout<<"满足剪枝函数,递归深入一层,将到达第"<<i+1<<"层"<<endl;
                Backtrack(i+1, ld);
                cout<<"当前第"<<i+1<<"层,递归回退一层,将到达第"<<i<<"层"<<endl;
                swap(x[i], x[j]);
                for(int k=1; k<=m; k++)     //恢复状态
                    now[k] -= B[x[j]][k];
                cout<<"第"<<i<<"层撤销选择电路板"<<x[j]<<",恢复now[]数组为(now[j]表示当
前解所含连接块j的电路板数): ";
                for(int j=1; j<=m; j++) cout<<now[j]<<" ";
                cout<<endl;

            }
            else cout<<"目前获得的密度已经大于最优值,故直接剪枝."<<endl;
            if(j==n) cout<<"当前层所有情况遍历完,回溯"<<endl;
        }
    }
}
int Arrangement(int **B, int n, int m, int *bestx)
{
    Board X;
    //初始化X
    X.x = new int[n+1];
    X.total = new int[m+1];
    X.now = new int[m+1];
    X.B = B;
    X.n = n;
    X.m = m;
    X.bestx = bestx;
    X.bestd = m+1;
    //初始化total和now
    for(int i=1; i<=m; i++)
    {
        X.total[i] = 0;
        X.now[i] = 0;
    }
    //初始化x为单位排列并计算total
    for(int i=1; i<=n; i++)
    {
```

```
        X.x[i] = i;
        for(int j=1; j<=m; j++)
            X.total[j] += B[i][j];
    }
    cout<<"total数组为: ";
    for(int i=1; i<=m; i++) cout<<X.total[i]<<" ";
    cout<<endl;
    X.Backtrack(1, 0);
    delete[] X.x;
    delete[] X.total;
    delete[] X.now;
    return X.bestd;
}
int main()
{
    cout<<"请输入电路板个数和连接块个数:";
    int n, m;
    while(cin>>n>>m && n && m)
    {
        cout<<"输入连接块矩阵"<<endl;
        int **B = new int*[n+1];
        for(int i=0; i<=n; i++) B[i] = new int[m+1];
        for(int i=1; i<=n; i++)
            for(int j=1; j<=m; j++)
                cin>>B[i][j];
        int *bestx = new int[n+1];
        for(int i=1; i<=n; i++) bestx[i] = 0;
        int ans = Arrangement(B, n, m, bestx);
        cout<<"得到的最小密度为:"<<ans<<endl;
        for(int i=0; i<=n; i++) delete[] B[i];
        delete[] B;
        delete[] bestx;
        cout<<"请输入电路板个数和连接块个数";
    }
    system("pause");
    return 0;
}

//使用的B数组
//0 0 1 0 0
//0 1 0 0 0
//0 1 1 1 0
//1 0 0 0 0
//1 0 0 0 0
//1 0 0 1 0
//0 0 0 0 1
//0 0 0 0 1
```

4.结果

5.分析

电路板排列问题涉及到所有电路板的排列，因此状态空间的大小主要由电路板的数量 `n` 决定。所有电路板的全排列数量是 $O(n!)$。

在回溯法中，每次选择一个电路板的位置，然后递归搜索剩余电路板的排列。

1. 每次递归调用中，我们需要判断当前插槽之间的连线情况，这涉及到对 `now` 数组的维护和更新。
2. 假设在每个插槽中，最坏的情况下都需要查看所有连接块，这样的操作可能是 $O(m)$（连接块的数量）。

结合上述分析，回溯法的时间复杂度可以表示为：

$$O(n! \cdot m)$$

算法实现题 5-7 n色方柱问题

1.题目描述

5-7 n 色方柱问题。

问题描述： 设有 n 个立方体，每个立方体的每面用红、黄、蓝、绿等 n 种颜色之一染色。要把这 n 个立方体叠成一个方形柱体，使得柱体的 4 个侧面的每侧均有 n 种不同的颜色。试设计一个回溯算法，计算出 n 个立方体的一种满足要求的叠置方案。

算法设计： 对于给定的 n 个立方体以及每个立方体各面的颜色，计算出 n 个立方体的一种叠置方案，使得柱体的 4 个侧面的每一侧均有 n 种不同的颜色。

数据输入： 由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n（$0<n<27$），表示给定的立方体个数和颜色数均为 n。第 2 行是 n 个大写英文字母组成的字符串。该字符串的第 k（$0 \leq k<n$）个字符代表第 k 种颜色。接下来的 n 行中，每行有 6 个数，表示立方体各面的颜色。立方体各面的编号如图 5-12 所示。

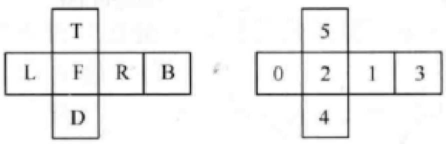


图 5-12　立方体各面的编号

图 5-12 中 F 表示前面，B 表示背面，L 表示左面，R 表示右面，T 表示顶面，D 表示底面。相应地，2 表示前面，3 表示背面，0 表示左面，1 表示右面，5 表示顶面，4 表示底面。

例如，在示例输出文件中，第 3 行的 6 个数 0、2、1、3、0、0 分别表示第 1 个立方体的左面的颜色为 R，右面的颜色为 B，前面的颜色为 G，背面的颜色为 Y，底面的颜色为 R，顶面的颜色为 R。

结果输出： 将计算的 n 个立方体的一种可行的叠置方案输出到文件 output.txt。每行 6 个字符，表示立方体各面的颜色。如果不存在所要求的叠置方案，输出“No Solution!”。

输入文件示例	输出文件示例
input.txt	output.txt
4	RBGYRR
RGBY	YRBGRG
0 2 1 3 0 0	BGRBGY
3 0 2 1 0 1	GYYRBB
2 1 0 2 1 3	
1 3 3 0 2 2	

2.思路

使用两个子图来表示每个立方体的边。对于每个立方体，选择连接相邻面的边并验证选边的合法性。检查当前选择的边是否构成合法的颜色配置，保证顶点度不超过限制。使用回溯技巧，通过当前边的选择递归地尝试下去。如果遇到冲突，则回溯。

关键步骤:

- `out()` 函数负责输出当前的颜色排列，依据当前选择的边。
- `search()` 函数中有主要的逻辑，用于试探性地构建颜色排列，通过不断尝试可连接的彩面。

3.代码

```
// 5-7 n色方柱问题
#include<bits/stdc++.h>
#define endl '\n'
using namespace std;
const int MAX=150;
int board[MAX][6];  //存储n个立方体各面的颜色
int solu[MAX][6];   //存储解
int n;  //立方体个数、颜色种数
int ans=0;  //解的个数
int used[MAX];
char color[MAX];

//找到一个解后，输出
void out(int edge[])
{
    int i, j, k, a, b, c, d;
    for(i=0; i<2; i++) //2个子图
    {
        for(j=0; j<n; j++)
            used[j] = 0;
        do{
            j = 0;
            d = c = -1;
            while(j<n && used[j]>0) //找下一条未用的边
                j++;
            if(j < n)
                do{
                    a = board[j][edge[i*n+j]*2];
                    b = board[j][edge[i*n+j]*2+1];
                    if(b == d)  //如果上一条边的终点与b相同，说明b为始点，交换，保证a为始
点
                        swap(a, b);  //保证有向边的始点对应于前面和左面，终点对应于背面和
右面
                    solu[j][i*2] = a;
                    solu[j][i*2+1] = b;
                    used[j] = 1;
                    if(c<0)  //开始顶点
                        c = a;
                    d = b;
                    for(k=0; k<n; k++)  //找下一个立方体
                        if(used[k]==0 && (board[k][edge[i*n+k]*2]==b || board[k]
[edge[i*n+k]*2+1]==b))
```

```
                        j = k;
                }while(b != c);  //找了一圈，回到起点
        }while(j<n);  //所有立方体都找遍
    }
    for(j=0; j<n; j++) //立方体的顶面和底面的颜色
    {
        k = 3 - edge[j] - edge[j+n];
        a = board[j][k*2];
        b = board[j][k*2+1];
        solu[j][4] = a;
        solu[j][5] = b;
    }
    for(i=0; i<n; i++)
    {
        for(j=0; j<6; j++)
            cout << color[solu[i][j]];
        cout << endl;
    }
}

void search()
{
    int i, t, cube;
    bool ok, newg, over;
    int *vert = new int[n];  //记录子图中每个顶点的度，应均为2
    int *edge = new int[n*2];  //记录每个立方体中边被选用的条数，每个立方体只有3条边，有两
个子图要选用
    for(i=0; i<n; i++)
        vert[i] = 0;
    t = -1;
    newg = true;
    while(t > -2)
    {
        t++;
        cube = t % n;  //每个立方体找2次，得到真实的立方体编号，也是子图中边的编号
        if(newg)  //如果没有边被选入子图
            edge[t] = -1;
        over = false;  //是否结束，即两个子图构建完成
        ok = false;    //标记边是否已用过，两个子图不应有公共边
        while(!ok && !over)
        {
            edge[t]++;  //边被选用加入子图，使用次数增加
            if(edge[t]>2)  //在立方体每对相对面的顶点连一条边，每个立方体只有3条边
                over = true;
            else
                ok = (t<n || edge[t]!=edge[cube]);  //是否已用过
        }
        if(!over)
        {           //检测边的两个顶点的度
            if(++vert[board[cube][edge[t]*2]] > 2+t/2*2) //如果是第一个子图，顶点度不
能超过2
                ok = false;             //如果是第二个子图，加上第一个子图，顶点度不能超
过4
            if(++vert[board[cube][edge[t]*2+1]] > 2+t/2*2)
                ok = false;
            if(t%n == n-1 && ok)  //如果一个或两个子图已构建完成
```

```
            for(i=0; i<n; i++)
                if(vert[i] > 2+t/n*2)
                    ok = false;
        if(ok)
        {
            if(t == n*2-1) //找到解
            {
                ans++;
                out(edge);
                return;
            }
            else
                newg = true;
        }
        else //取下一条边
        {
            --vert[board[cube][edge[t]*2]];  //边的两个顶点
            --vert[board[cube][edge[t]*2+1]];
            t--;
            newg = false;
        }
    }
    else //回溯
    {
        t--;
        if(t > -1)
        {
            cube = t % n;
            --vert[board[cube][edge[t]*2]];
            --vert[board[cube][edge[t]*2]];
        }
        t--;
        newg = false;
    }
  }
}

int main()
{
    cin >> n;
    for(int i=0; i<n; i++)
    {
        cin >> color[i];
    }
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<6; j++)
        {
            cin >> board[i][j];
        }
    }
    search();
    if(ans == 0)
        cout << "No Solution! \n";
    cout << endl;
    return 0;
```

```
}
```

4.结果

```
4
RGBY
0 2 1 3 0 0
3 0 2 1 0 1
2 1 0 2 1 3
1 3 3 0 2 2
RBGYRR
YRBGRG
BGRBGY
GYYRBB
```

5.分析

每个立方体有三个边可以选择，且每个立方体的边选择是独立的。在最坏情况下，假设每个立方体都有可能选择的边数是常数（这里是3），我们可以认为每个立方体的选择是独立的。对于 `n` 个立方体，选择边的组合会导致可能的状态数为 $O(3^n)$。

由于每次选择边后，还需要进行合法性检查，这个检查的复杂度是常数时间（因为每次检查的顶点度数是固定的）。因此，整体的时间复杂度可以表示为：

$$O(3^n)$$

在实际运行中，由于有很多剪枝和有效性检查，可能不会遍历所有的组合，因此在某些情况下可能会比 $O(3^n)$ 更快，但最坏情况下的复杂度确实是 $O(3^n)$。