实验二

一、实验目的

1. 理解动态规划的基本思想。

2. 运用动态规划算法解决最短路径问题。

3. 理解回溯法的基本原理，掌握使用回溯法求解实际问题。

4. 运用回溯法解决皇后问题及图着色问题。

二、实验仪器及设备

1. 硬件环境：PC 机一台（1G 以上内存）

2. 软件环境：Windows 环境、C 语言（Visual Studio 2015）

三、实验内容

1. 编写实验程序，实现利用动态规划方法求解图 2.1 中从顶点 0 到顶点 6 的

最短路径问题。

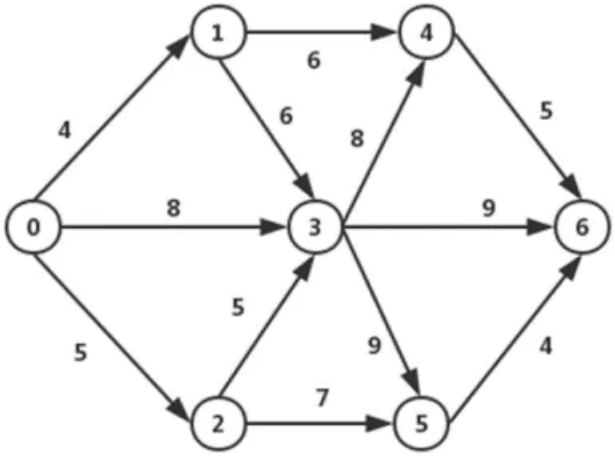


图 2.1

这是一张有向图的示意图，图中有7个节点（0到6）和多个带权边。每个节点用一个圆圈表示，并标有数字。每条边有一个箭头指示方向，并在边上标注了权重。

具体来说：

- 节点0连接到节点1、2和3。
  - 边(0, 1)的权重为4。
  - 边(0, 2)的权重为5。
  - 边(0, 3)的权重为8。
- 节点1连接到节点3和4。
  - 边(1, 3)的权重为6。
  - 边(1, 4)的权重为6。
- 节点2连接到节点3和5。
  - 边(2, 3)的权重为5。
  - 边(2, 5)的权重为7。

- 节点3连接到节点4、5和6。
  - 边(3, 4)的权重为8。
  - 边(3, 5)的权重为9。
  - 边(3, 6)的权重为9。
- 节点4没有出边。
- 节点5连接到节点6。
  - 边(5, 6)的权重为4。
- 节点6没有出边。

2. 编写程序实现 4 皇后问题的求解；

3. 编写程序实现用 3 种颜色为图 2.2 着色问题。

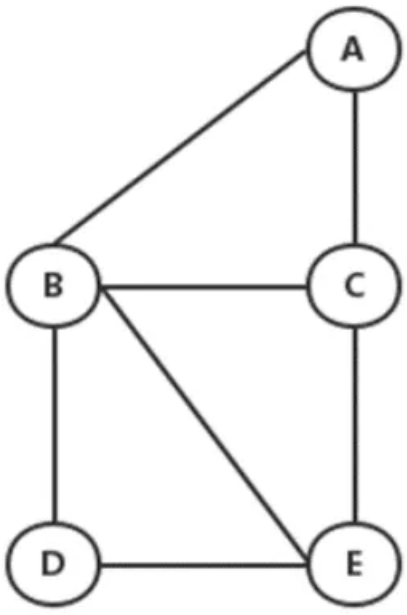


图 2.2

四.实验原理和分析

1. 动态规划问题基本原理：

   （1）经分解得到的各个子问题往往不是相互独立的。

   比如:A1A2A3 与 A2A3A4 有共同的子问题 A2A3

   （2）在求解过程中，将已解决的子问题的最优值进行保存，在需要时可以轻松找出。

   （3）通常采用表的形式记录子问题的最优值，即在实际求解过程中，一旦某个子问题被计算过，不管该问题以后是否用得到，都将其计算结果填入该表，需要的时候就从表中找出该子问题的最优值。

   （4）根据最优值，记录最优决策，构造最优解。

2. 最短路径问题算法：

```
核心代码如下：
struct NODE { // 邻接表结点的数据结构
    int v_num; // 邻接顶点的编号
    Type len; // 邻接顶点与该顶点的费用
    struct NODE *next; // 下一个邻接顶点
};
struct NODE node[n]; // 多段图邻接表头结点
Type cost[n];
int route[n];
int path[n];
Type fgraph(struct NODE node[ ], int route[ ], int n)
    9
{ int i;
  struct NODE *pnode;
  int *path = new int[ n ];
  Type min_cost, *cost = new Type[ n ];
  for(i=0; i<n; i++) {
      cost[ i ] = MAX_VALUE_TYPE;
      path[ i ] = -1; rouet[ i ] = 0;
  }
  cost[ n-1 ] = ZERO_VALUE_OF_TYPE;
  for (i=n-2; i>=0; i--) {
      pnode = node[ i ]->next;
      while (pnode != NULL) {
          if (pnode->len + cost[pnode->v_num] < cost[ i ]) {
              cost[ i ] = pnode->len + cost[pnode->v_num];
              path[ i ] = pnode->v_num;
          }
          pnode = pnode->next;
      }
  }
  i = 0;
  while ((route[ i ] != n-1) && (path[ i ] != -1)) {
      i++;
      route[ i ] = path[ route[ i-1 ] ];
  }
  min_cost = cost[ 0 ];
  delete path; delete cost;
  return min_cost;
 }
```

3. 回溯算法基本原理：回溯法是在仅给出初始结点、目标结点及产生子结点的条件（一般由问题题意隐含给出）的情况下，构造一个图（隐式图），然后按照

深度优先搜索的思想，在有关条件的约束下扩展到目标结点，从而找出问题的解。回溯法是一种“能进则进，进不了则换，换不了则退”的基本搜索方法。

4. 皇后问题的算法：

核心代码如下：

```c
void n_queens(int n,int x[])
{
int k=1;
x[1]=0;
while (k>0){
x[k]=x[k] +1;//在当前列加 1 的位置开始搜索
while ((x[k]<=n)&&(!place(x,k)))//当前列位置是否满足条件
x[k] = x[k] +1;//不满足条件，继续搜索下一列位置
if (x[k]<=n) { //存在满足条件的列
 if (k==n) break; //是最后一个皇后，完成搜索
 else{
 K=K+L; x[k]=0; //不是，则处理下一个行皇后
 }
 }
else{ // 已判断完 n 列，均没有满足条件
 x[k]=0; k=k-1;// 第 k 行复位为 0,回溯到前一行.
 }
 }
}
```

5. 图着色问题的算法：

核心代码如下：

（1） 数据结构

```c
  int n; // 顶点个数
  int m; // 最大颜色数
  int k; // 顶点号码，或搜索深度
  int x[n]; // 顶点的着色
BOOL c[n][n]; // 布尔值表示的图的邻接矩阵
函数 ok：判断顶点着色的有效的性有效返回 TRUE，无效返回 FALSE
 BOOL ok(int x[ ], int k, BOOL c[ ][ ], int n)
11
 {
 int i;
 for (i=0; i<k; i++) {
 if (c[ k ][ i ] && (x[ k ] == x[ i ])
 return FALSE;
 return TRUE;
 }
void m_coloring(int n, int m, int x[ ], BOOL c[ ][ ])
{ int i,k;
 for (i=0;i<n;i++) x[i] = 0;
 k = 0;
 while (k >= 0) {
 x[ k ] = x[ k ] + 1;
 while ((x[k] <= m) && (!ok(x, k, c, n))) x[k] = x[k] + 1;
 if (x[k] <= m) { if (k == n-1) break;
 else k = k + 1; }
 else { x[k] = 0; k = k - 1; }
 }
if (k==n-1) return TRUE;
 else FALSE;
}
```

五 问题一分析

```c
int CreateGraph()
{
    int point, edge;
    cout << "请输入顶点和边的个数：" << endl;
    cin >> point >> edge;
    for (int i = 0; i < point; i++) { // 初始化边的权值
        for (int j = 0; j < point; j++) {
            node[i][j] = INF;
        }
    }
    int weight;
    for (int k = 0; k < edge; k++) {
        int i, j;
        cout << "请输入第" << k + 1 << "条边的两个顶点和权值：" << endl;
        cin >> i >> j >> weight;
        node[i][j] = weight;
    }
    return point;
}
// 求 n个顶点的多段图的最短路径

int Path(int n)
{
    int i, j;
    int cost[100], path[100]; // 存储路径长度和路径
    for (i = 1; i < n; i++) {
        cost[i] = INF; // 初始化
        path[i] = -1;
    }
    cost[0] = 0;
    path[0] = -1;
    for (j = 1; j < n; j++) { // 前驱节点
        for (i = j - 1; i >= 0; i--) {
            if (cost[i] + node[i][j] < cost[j]) {
                cost[j] = cost[i] + node[i][j]; // 更新值
                path[j] = i; // 将i的值


```

您提到的动态规划的基本原理非常准确，下面我将结合您的具体问题——求解多段图的最短路径问题，来详细解释如何应用这些原理。

动态规划应用于多段图最短路径问题

1. 子问题重叠

在多段图的最短路径问题中，从起点到终点的最短路径可能经过多个中间顶点。例如，如果我们要求从顶点 `0` 到顶点 `n-1` 的最短路径，那么从 `0` 到 `1`、`0` 到 `2`、...、`0` 到 `n-2` 都可以被视为子问题。这些子问题之间存在重叠，因为从 `0` 到 `n-1` 的最短路径可能包含了从 `0` 到 `n-2` 的最短路径。

2. 记录子问题的最优值

为了避免重复计算相同的子问题，我们可以使用一个数组 `cost[]` 来记录从起点到每个顶点的最短距离。当计算出从起点到某个顶点的最短距离后，我们将其保存在 `cost[]` 中，以便后续计算其他顶点的最短距离时直接使用。

3. 保存子问题的最优解

除了记录最短距离，我们还需要记录最短路径。为此，可以使用另一个数组 `path[]` 来记录每个顶点的前驱顶点。通过 `path[]`，我们可以回溯从终点到起点的完整最短路径。

4. 构造最优解

在计算完所有顶点的最短路径后，我们可以通过 `path[]` 数组回溯从终点到起点的最短路径。具体步骤如下：

- 从终点开始，找到其前驱顶点。
- 继续找到前驱顶点的前驱顶点，直到回到起点。
- 将这些顶点按顺序输出，即为最短路径。

具体实现

下面是结合上述原理的 `Path` 函数的具体实现：

```cpp
#include <iostream>
#include <vector>
#include <climits> // For INT_MAX

using namespace std;

const int INF = INT_MAX;
vector<vector<int>> node; // 使用 vector 动态管理图

int Path(int n) {
    vector<int> cost(n, INF), path(n, -1); // 使用 vector 动态管理数组
    cost[0] = 0; // 起点的成本设为0

    // 动态规划计算最短路径
    for (int j = 1; j < n; j++) { // 对于每一个顶点
        for (int i = 0; i < j; i++) { // 检查所有之前的顶点
            if (node[i][j] != INF && cost[i] + node[i][j] < cost[j]) {
                cost[j] = cost[i] + node[i][j];
                path[j] = i;
            }
        }
    }

    // 输出最短路径和成本
    for (int i = 0; i < n; i++) {
        cout << "到顶点 " << i << " 的最短路径为 " << cost[i] << "，路径为：";
        int current = i;
        vector<int> pathTrace;
        while (current != -1) {
            pathTrace.push_back(current);
            current = path[current];
        }
        reverse(pathTrace.begin(), pathTrace.end());
        for (auto p : pathTrace) {
            cout << p << " ";
        }
        cout << endl;
    }

    return 0; // 或者返回任何其他有意义的值
}

int main() {
    int point, edge;
    cout << "请输入顶点和边的个数：" << endl;
    cin >> point >> edge;
    node.resize(point, vector<int>(point, INF));
```

```
48
49      for (int k = 0; k < edge; k++) {
50          int i, j, weight;
51          cout << "请输入第" << k + 1 << "条边的两个顶点和权值: " << endl;
52          cin >> i >> j >> weight;
53          node[i][j] = weight;
54      }
55
56      Path(point);
57
58      return 0;
59  }
```

代码解释

1. **初始化**:
   - `node` 是一个二维向量，用于存储图的邻接矩阵。
   - `cost` 和 `path` 是一维向量，分别用于存储从起点到每个顶点的最短距离和前驱顶点。
2. **动态规划计算**:
   - 外层循环遍历每个顶点 `j`。
   - 内层循环遍历所有比 `j` 小的顶点 `i`，检查是否存在一条从 `i` 到 `j` 的边。
   - 如果存在这样的边，并且通过这条边到达 `j` 的距离更短，则更新 `cost[j]` 和 `path[j]`。
3. **输出结果**:
   - 遍历每个顶点，输出从起点到该顶点的最短距离和路径。
   - 使用 `pathTrace` 向量来存储从终点到起点的路径，并在输出前反转该向量。

通过这种方式，我们不仅避免了重复计算，还有效地记录了最短路径，从而实现了动态规划的核心思想。

六 问题二分析

> 4. 皇后问题的算法:
>
> 核心代码如下:
>
> void n_queens(int n,int x[])
>
> {

```
int k=1;
x[1]=0;
while (k>0){
x[k]=x[k] +1;//在当前列加 1 的位置开始搜索
while ((x[k]<=n)&&(!place(x,k)))//当前列位置是否满足条件
x[k] = x[k] +1;//不满足条件，继续搜索下一列位置
if (x[k]<=n) { //存在满足条件的列
if (k==n) break; //是最后一个皇后，完成搜索
else{
K=K+L; x[k]=0; //不是，则处理下一个行皇后
}
}
else{ // 已判断完 n 列，均没有满足条件
x[k]=0; k=k–1;// 第 k 行复位为 0,回溯到前一行.
}
}
}
```

1. 问题定义

N皇后问题的目标是在一个N×N的棋盘上放置N个皇后，使得任意两个皇后不能在同一行、同一列或同一条对角线上。

2. 数据结构

- `private static final int N = 4;`：定义棋盘的大小为4×4。
- `private static int[] board = new int[N + 1];`：使用一个数组来表示棋盘的状态。数组的索引表示行号（从1开始），值表示该行的皇后所在的列号。数组长度为`N + 1`是为了方便从1开始索引。

3. 核心算法

3.1 检查安全性

- `isSafe(int row, int col)`：这个方法用于检查在第 `row` 行第 `col` 列放置皇后是否安全。
  - **检查列冲突**：遍历所有已经放置了皇后的行（从1到 `row-1`），如果某一行的皇后也在第 `col` 列，则当前位置不安全。
  - **检查对角线冲突**：如果某一行的皇后所在列与当前列的差的绝对值等于行号差的绝对值，则说明它们在同一条对角线上，当前位置也不安全。

3.2 递归求解

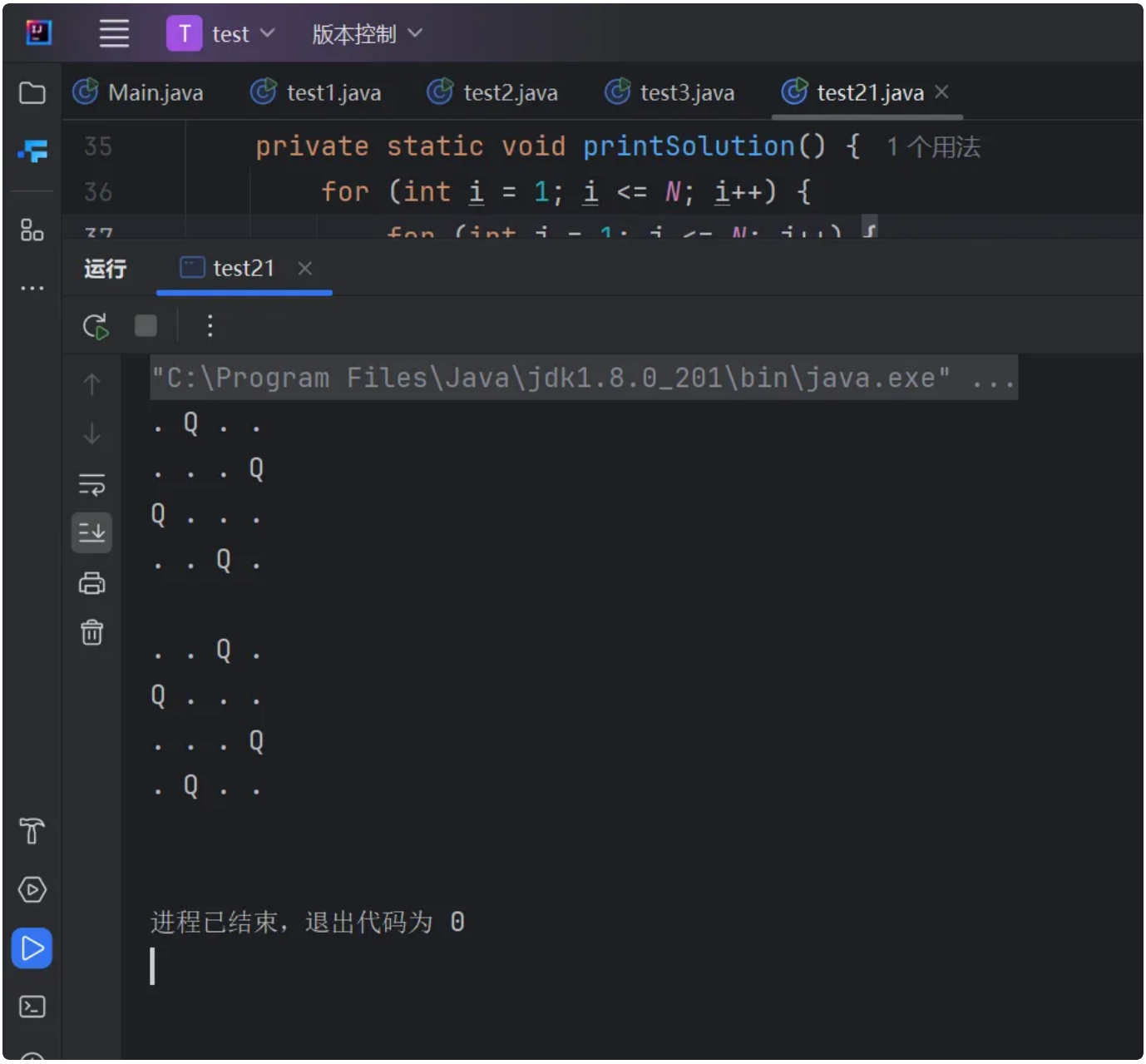
- `solveNQueens(int row)`：这是核心的递归方法，用于尝试在第 `row` 行放置皇后。
  - **遍历所有可能的列**：对于第 `row` 行的每一列（从1到 `N`），检查是否可以放置皇后。
  - **放置皇后**：如果当前位置安全，就在该位置放置皇后（即 `board[row] = col`）。
  - **递归处理下一行**：如果当前行是最后一行（即 `row == N`），则已经找到了一个解，调用 `printSolution` 方法打印解；否则，继续递归调用 `solveNQueens(row + 1)` 处理下一行。
  - **回溯**：如果在下一行无法找到合适的位置放置皇后，递归会返回，当前行的皇后位置会被重新选择。

4. 打印解

- `printSolution()`：当找到一个解时，调用此方法打印当前的棋盘状态。
  - **遍历棋盘**：双层循环遍历整个棋盘，如果当前位置有皇后（即 `board[i] == j`），则打印 `Q`；否则打印 `.`。
  - **换行**：每打印完一行后，输出一个换行符，以便下一行的输出。

5. 主方法

- `main(String[] args)`：程序的入口点，调用 `solveNQueens(1)` 开始从第1行放置皇后。


test
版本控制
Main.java
test1.java
test2.java
test3.java
test21.java
35 private static void printSolution() { 1 个用法
36 for (int i = 1; i <= N; i++) {
运行
test21
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
. Q . .
. . . Q
Q . . .
. . Q .
. . Q .
Q . . .
. . . Q
. Q . .
进程已结束，退出代码为 0

```
public class test21 {

    private static final int N = 4; // 棋盘大小
    private static int[] board = new int[N + 1]; // 存储每行皇后的列位置，索引从1开始

    public static void main(String[] args) {
        solveNQueens(1); // 从第1行开始放置皇后
    }

    // 检查当前位置是否可以放置皇后
    private static boolean isSafe(int row, int col) {
        for (int i = 1; i < row; i++) {
            if (board[i] == col || Math.abs(board[i] - col) == Math.abs(i - row)) {
                return false;
            }
        }
        return true;
    }

    // 解决N皇后问题的核心函数
    private static void solveNQueens(int row) {
        for (int col = 1; col <= N; col++) {
            if (isSafe(row, col)) {
                board[row] = col; // 放置皇后
                if (row == N) {
                    printSolution(); // 如果已经放置了所有皇后，打印解
                } else {
                    solveNQueens(row + 1); // 继续放置下一行的皇后
                }
            }
        }
    }

    // 打印当前解
    private static void printSolution() {
        for (int i = 1; i <= N; i++) {
            for (int j = 1; j <= N; j++) {
                if (board[i] == j) {
                    System.out.print("Q ");
                } else {
                    System.out.print(". ");
                }
            }
```

```
                System.out.println();
            }
            System.out.println();
        }
    }
```

七 问题三分析

5. 图着色问题的算法：

核心代码如下：

（1） 数据结构

int n;

// 顶点个数

int m;

// 最大颜色数

int k;

// 顶点号码，或搜索深度

int x[n];

// 顶点的着色

BOOL c[n][n]; // 布尔值表示的图的邻接矩阵

函数 ok：判断顶点着色的有效的性有效返回 TRUE，无效返回 FALSE

BOOL ok(int x[ ], int k, BOOL c[ ][ ], int n)11

{

int i;

for (i=0; i<k; i++) {

if (c[k][i] && (x[k] == x[i])

return FALSE;

return TRUE;

}

```
void m_coloring(int n, int m, int x[ ], BOOL c[ ][ ])
{ int i,k;
for (i=0;i<n;i++) x[i] = 0;
k = 0;
while (k >= 0) {
x[ k ] = x[ k ] + 1;
while ((x[k] <= m) && (!ok(x, k, c, n))) x[k] = x[k] + 1;
if (x[k] <= m) { if (k == n–1) break;
else k = k + 1; }
else { x[k] = 0; k = k — 1; }
}
if (k==n–1) return TRUE;
else FALSE;
}
```

Java

```java
import java.util.Scanner;

public class test23 {

    private static final int MAXN = 1005;
    private static int[][] G = new int[MAXN][MAXN]; // 用于存储图中的边
    private static int[] color = new int[MAXN];     // 用于存储每个节点的颜色
    private static int n, m;                        // n表示图中节点的数量，m表示可供选择的颜色数目

    public static boolean ok(int u, int c) {
        for (int i = 1; i <= n; i++) {
            if (G[u][i] == 1 && color[i] == c) {
                return false;
            }
        }
        return true;
    }

    public static boolean dfs(int u) {
        if (u > n) {
            return true;
        }
        for (int i = 1; i <= m; i++) {
            if (ok(u, i)) {
                color[u] = i;
                if (dfs(u + 1)) {
                    return true;
                }
                color[u] = 0; // 回溯
            }
        }
        return false;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("请输入顶点数和可用颜色数：");
        n = scanner.nextInt();
        m = scanner.nextInt();

        System.out.println("请输入边数：");
        int e = scanner.nextInt();

```

```
        System.out.println("请输入相连接的顶点: "); // 顶点从1开始
        for (int i = 0; i < e; i++) {
            int u = scanner.nextInt();
            int v = scanner.nextInt();
            G[u][v] = G[v][u] = 1;
        }

        if (dfs(1)) {
            for (int i = 1; i <= n; i++) {
                System.out.println("结点 " + i + " 的颜色为:  " + color[i]);
            }
        } else {
            System.out.println("No solution");
        }

        scanner.close();
    }
}
```

八、思考题

1. 用递归函数设计一个求解货郎担问题的动态规划算法，并估计其时间复杂性。

1. 货郎担问题（旅行商问题, TSP）的递归动态规划算法及其时间复杂度

问题描述:
货郎担问题是一个经典的组合优化问题，目标是在给定的一组城市以及每对城市之间的距离的情况下，找到一条从起始城市出发，经过所有其他城市恰好一次后返回起始城市的最短路径。

递归动态规划算法设计:
为了使用递归加动态规划的方法解决TSP问题，我们可以定义状态为 `dp[S][j]` ，其中 `S` 是已经访问过的城市集合， `j` 是从集合 `S` 中最后访问的城市。 `dp[S][j]` 表示从起点出发，经过集合 `S` 中的所有城市恰好一次后到达城市 `j` 的最短路径长度。

- **基础情况**: 当集合 `S` 只包含起点城市时， `dp[S][j]` 就是从起点到城市 `j` 的距离。

- **递归关系**：对于任意集合 `S` 和城市 `j`， `dp[S][j]` 可以通过考虑从集合 `S-{j}` 中的每个城市 `i` 转移到城市 `j` 的所有可能来计算，即 `dp[S][j] = min(dp[S-{j}][i] + dist[i][j])`，其中 `dist[i][j]` 是从城市 `i` 到城市 `j` 的距离。

时间复杂度分析：

- 对于每个集合 `S`，我们有 `|S|` 个选择来决定哪个城市作为最后一个访问的城市 `j`。因为总共有 `2^n - 1` 个非空子集（不包括起点），所以总的状态数大约是 `n * 2^n`。
- 每个状态的计算需要 `O(n)` 的时间，因为我们可能需要检查集合 `S` 中的每一个城市来更新 `dp[S][j]`。
- 因此，总体时间复杂度为 `O(n^2 * 2^n)`。

1. 使用递归动态规划解决货郎担问题 (TSP)

```java
import java.util.*;

public class TravelingSalesmanProblem {
    private static final int INF = Integer.MAX_VALUE;
    private static int[][] dist; // 城市之间的距离
    private static int n; // 城市数量
    private static int[][] memo; // 记忆化数组

    public static int tsp(int mask, int pos) {
        if (mask == (1 << n) - 1) return dist[pos][0]; // 如果所有城市都被访问过了，则返回从当前位置回到起点的距离
        if (memo[mask][pos] != -1) return memo[mask][pos];

        int answer = INF;
        for (int city = 0; city < n; ++city) {
            if ((mask & (1 << city)) == 0) { // 如果城市city还未被访问
                int newAnswer = dist[pos][city] + tsp(mask | (1 << city), city);
                answer = Math.min(answer, newAnswer);
            }
        }
        memo[mask][pos] = answer;
        return answer;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        n = 4; // 假设有4个城市
        dist = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = scanner.nextInt(); // 输入城市间的距离
            }
        }
        memo = new int[1 << n][n];
        for (int[] row : memo) Arrays.fill(row, -1); // 初始化记忆化数组

        System.out.println("最小路径长度: " + tsp(1, 0)); // 假设从城市0开始
    }
}
```

2. 使用回溯算法解八皇后问题时，在最坏情况下，求所生成的搜索树的结点总数。

问题描述：
八皇后问题是将八个皇后放置在一个8×8的国际象棋棋盘上，使得没有两个皇后可以互相攻击，即任何两个皇后都不能处于同一行、同一列或同一对角线上。

最坏情况下的搜索树结点总数：
在最坏的情况下，回溯算法会尝试所有可能的位置组合，直到找到一个有效的解决方案或者确定没有解决方案。由于每个皇后都可以放在棋盘上的任何一列，并且每一行都必须放置一个皇后，因此初始的搜索空间为 `8!`（即40320种不同的排列方式）。

但是，实际的搜索过程中，许多部分搜索树会被剪枝，因为一旦发现当前的部分配置会导致冲突（例如，两个皇后在同一列或对角线上），那么以这个配置开始的所有后续搜索都会被跳过。

然而，在理论上的最坏情况下，假设没有任何早期剪枝发生，每个皇后都可能被放置在当前行的任何一列中，这将导致每一层都有8个分支。因此，完整的搜索树会有 `8^8`（即16777216）个叶子节点。但是，考虑到实际上会有很多剪枝操作，实际生成的结点数通常远小于这个理论值。

总结来说，虽然理论上最坏情况下可能生成的搜索树结点总数为 `8^8`，但实际上由于有效的剪枝策略，生成的结点数会显著减少。

```java
public class EightQueens {
    private static final int N = 8;
    private static boolean[] col = new boolean[N]; // 列占用标记
    private static boolean[] diag1 = new boolean[2 * N - 1]; // 正对角线占用标记
    private static boolean[] diag2 = new boolean[2 * N - 1]; // 反对角线占用标记
    private static int count = 0;

    public static void solve(int row) {
        if (row == N) {
            count++;
            printSolution();
            return;
        }

        for (int column = 0; column < N; column++) {
            if (!col[column] && !diag1[row + column] && !diag2[row - column + N - 1]) {
                placeQueen(row, column);
                solve(row + 1);
                removeQueen(row, column);
            }
        }
    }

    private static void placeQueen(int row, int column) {
        col[column] = true;
        diag1[row + column] = true;
        diag2[row - column + N - 1] = true;
    }

    private static void removeQueen(int row, int column) {
        col[column] = false;
        diag1[row + column] = false;
        diag2[row - column + N - 1] = false;
    }

    private static void printSolution() {
        for (int row = 0; row < N; row++) {
            for (int column = 0; column < N; column++) {
                if (col[column] && (row + column == diag1[row + column] || row - column + N - 1 == diag2[row - column + N - 1])) {
                    System.out.print("Q ");
                } else {
                    System.out.print(". ");
                }
```

```
                }
                System.out.println();
            }
            System.out.println();
        }

        public static void main(String[] args) {
            solve(0);
            System.out.println("Total solutions: " + count);
        }
    }
```