

北京邮电大学软件学院

2017-2018 学年第一学期实验报告

课程名称: 算法分析与设计

项目名称: 实验二: 回溯法

项目完成人:

姓名: 刘禾子 学号: 2017526019

指导教师: 李朝晖

日 期: 2017 年 11 月 07 日

一、 实验目的

1. 深刻理解并掌握回溯法的设计思想；
2. 提高应用回溯法设计算法的技能。

二、 实验内容

1. 用回溯法解决 0/1 背包问题、迷宫问题并对所设计的算法进行时间复杂度分析；
2. 选做：最短路径求迷宫问题，用回溯法实现图着色问题。

三、 实验环境

IntelliJ IDEA Community Edition 2017.2.4

四、 实验结果

1. 回溯法解决 0/1 背包问题

```
"C:\Program Files\Java\jdk1.8.0_112\bin\j
请输入物品个数以及背包容量：
4 10
请分别输入物品的重量和价值
7 42
3 12
4 40
5 25
最佳方案为（0：放入 1：不放入）：
0 0 1 1 总价值为：65

Process finished with exit code 0
```

2. 回溯法解决迷宫问题

• 0 1 2 3 4 5 6 7 8 9

• 0 ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

• 1 ■ □ □ ■ □ □ □ ■ □ ■

• 2 ■ □ □ ■ □ □ □ ■ □ ■

• 3 ■ □ □ □ □ ■ ■ □ □ ■

• 4 ■ □ ■ ■ ■ □ □ □ □ ■

• 5 ■ □ □ □ ■ □ □ □ □ ■

• 6 ■ □ ■ □ □ □ ■ □ □ ■

• 7 ■ □ ■ ■ ■ □ ■ ■ □ ■

• 8 ■ ■ □ □ □ □ □ □ ■

• 9 ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

求解迷宫问题（选做）

迷宫图案，白色代表通道，黑色代表墙。

入口 (1,1),出口 (8,8)

要求：最短路径

(2) 队列（有Enqueue(), Dequeue(),Front()等函数）

(3) 用Solution函数解决迷宫路径问题

(4) 具有用户从键盘输入或打开文件的界面；

(5) 输出迷宫

(6) 输出路径如(6,8)-(3, 4)..等

```

"C:\Program Files\Java\jdk1.8.0_112\bin\java" ...
迷宫 (1: 墙 0: 通路 2: 已走过的点, 最外层均为1):
起点 (1, 1), 终点 (8, 8)
1 1 1 1 1 1 1 1 1 1
1 0 0 1 0 0 0 1 0 1
1 0 0 1 0 0 0 1 0 1
1 0 0 0 0 1 1 0 0 1
1 0 1 1 1 0 0 0 0 1
1 0 0 0 1 0 0 0 0 1
1 0 1 0 0 0 1 0 0 1
1 0 1 1 1 0 1 1 0 1
1 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1

输出路径:
(1,1) (2,1) (3,1) (4,1) (5,1)
(6,1) (7,1) (7,1) (6,1) (5,1) (5,2) (5,3) (6,3)
(6,4) (6,5) (7,5) (8,5) (8,6)
(8,7) (8,8)
输出迷宫:
1 1 1 1 1 1 1 1 1 1
1 2 0 1 0 0 0 1 0 1
1 2 0 1 0 0 0 1 0 1
1 2 0 0 0 1 1 0 0 1
1 2 1 1 1 0 0 0 0 1
1 2 2 2 1 0 0 0 0 1
1 2 1 2 2 2 1 0 0 1
1 2 1 1 1 2 1 1 0 1
1 1 0 0 0 2 2 2 2 1
1 1 1 1 1 1 1 1 1 1

Process finished with exit code 0

```

3. 迷宫最短路

```

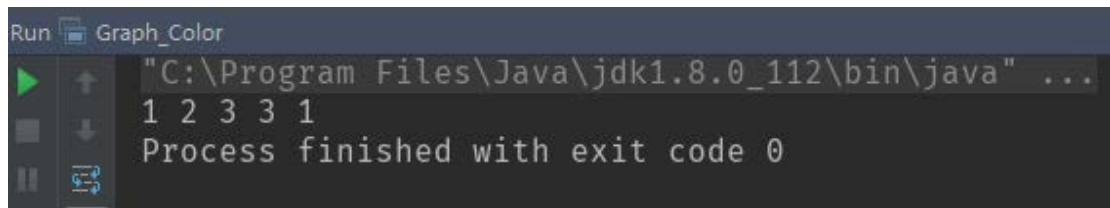
迷宫 (1: 墙 0: 通路 最外层均为1):
起点 (1, 1), 终点 (8, 8)
1 1 1 1 1 1 1 1 1 1
1 0 0 1 0 0 0 1 0 1
1 0 0 1 0 0 0 1 0 1
1 0 0 0 0 1 1 0 0 1
1 0 1 1 1 0 0 0 0 1
1 0 0 0 1 0 0 0 0 1
1 0 1 0 0 0 1 0 0 1
1 0 1 1 1 0 1 1 0 1
1 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1

输出路径:
(1,1) (2,1) (3,1) (4,1) (5,1) (5,2) (5,3)
(6,3) (6,4) (6,5) (7,5) (8,5) (8,6) (8,7)
(8,8) 总长度为: 14
输出迷宫:
1 1 1 1 1 1 1 1 1 1
1 0 1 1 7 8 9 1 0 1
1 1 2 1 6 7 8 1 0 1
1 2 3 4 5 1 1 0 0 1
1 3 1 1 1 11 12 13 0 1
1 4 5 6 1 10 11 12 13 1
1 5 1 7 8 9 1 13 0 1
1 6 1 1 1 10 1 1 0 1
1 1 0 13 12 11 12 13 14 1
1 1 1 1 1 1 1 1 1 1

Process finished with exit code 0

```

4. 图着色问题



```
Run Graph_Color
"C:\Program Files\Java\jdk1.8.0_112\bin\java" ...
1 2 3 3 1
Process finished with exit code 0
```

五、 附录

1. 回溯法求 0/1 背包问题

(1) 问题分析

给定 n 种物品和一个背包，物品 i ($1 \leq i \leq n$) 的重量是 w_i ，其价值为 v_i ，背包容量为 C ，对每种物品只有两种选择：装入背包或不装入背包。求解如何选择装入背包的物品使得装入背包中物品的总价值最大。

(2) 设计方案

用一个 $flag[n]$ 数组来存放物品放入（值为 1）或不放入（值为 0），在一次循环中分别评估假设第 i 个物品放入或不放入的情况，超出总容量则不考虑，不超出则算出放入该物品后当前的总重量 cw ，以及总价值 cp ，然后递归调用评估第 $i+1$ 个物品放入或不放入的情况，若 $i+1$ 个物品不能放入则令 cw 和 cp 分别减去第 $i+1$ 个物品的重量和价值，若能放入就一直递归下去直到评估好了所有的物品及 i 达到停止循环。

(3) 算法分析

用回溯法解决 0/1 背包问题，它在问题的解空间树中，按深度优先策略，从节点出发搜索空间树，可选物品为 n ，限制条件为 $cw + flag[i] * goods[i].weight < c$ ，需要 $O(n)$ 时间，在最坏的情况下有 $O(2^n)$ 个结点需要计算这个上界条件，回溯算法需要的计算时间为 $O(n2^n)$ 即接近于用蛮力法解决背包问题的复杂度。

(4) 源代码

```
public class BackTracking {
    public static class good{
        private int weight;
        private int value;
        good(int weight, int value){
            this.weight=weight;
            this.value=value;
        }
    }

    private static int bestvalue;
    private static int[] solu=new int[10];
    private static int[] flag=new int[10];
    public static void main(String args[]){
        int n;//物品个数
        int c;//背包容量
        int cw=0, cp=0;
```

```

        System.out.println("请输入物品个数以及背包容量：");
        Scanner sc=new Scanner(System.in);
        n=sc.nextInt();c=sc.nextInt();
        good[] goods=new good[n];
        for (int i=0;i<n;i++) {
            goods[i]= new good(0,0);
        }
        System.out.println("请分别输入物品的重量和价值");
        for (int i=0;i<n;i++) {
            goods[i].weight=sc.nextInt();
            goods[i].value=sc.nextInt();
        }
        sc.close();
        back_track(0,cw,cp,n,c,goods);
        System.out.println("最佳方案为：");
        for (int i=0;i<n;i++) {
            System.out.print(solu[i]+" ");
        }
        System.out.println("总价值为："+bestvalue);
    }
    private static void back_track(int i, int cw, int cp, int n, int c, good goods[])
    {
        if (i>n-1){
            if (bestvalue<cp){
                bestvalue=cp;
                System.arraycopy(flag, 0, solu, 0, n);
            }else {
                for (int j=1;j>=0;j--) {
                    flag[i]=j;
                    if (cw+flag[i]*goods[i].weight<c) {
                        cw+=goods[i].weight*flag[i];
                        cp+=goods[i].value*flag[i];
                        back_track(i+1,cw,cp,n,c,goods);
                        cw-=goods[i].weight*flag[i];
                        cp-=goods[i].value*flag[i];
                    }
                }
            }
        }
    }
}

```

2. 回溯法解决迷宫问题

(1) 问题分析

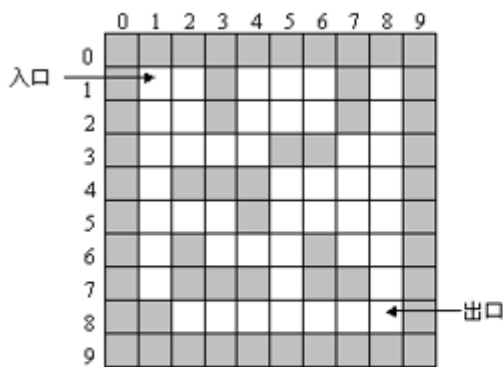


图 3.3 迷宫示意图

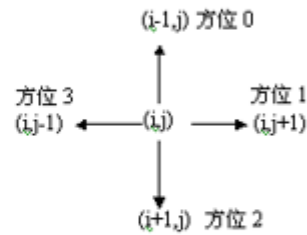


图 3.4 方位图

如上图所示，利用回溯法解决迷宫问题，运动的方向只可能是上下左右其中的一个，从入口出发顺某一方向试探，若能走通，则继续往前走，否则原路返回，换另一个方向继续试探，直至走出去为止。

(2) 设计方案

`int[][] map`: 一个 10*10 的二维数组用于存储迷宫的情况

`int[][] direction`: 4*2 的二维数组用于存储下右左上四个方向

`Stack<Point> path`: 存放点类型的栈

`Point entey`: 入口点

`Point cur`: 当前点

`boolean solution(int map[],int direction[],Stack<Point> path,Point entry)`

让当前点从入口出发顺着四个方向摸索，若能前进则将点的位置压入 `path` 栈内，若走不出，则令当前点的位置更改为栈顶点的位置，并弹出栈顶，继续摸索，每次压栈就输出当前点的坐标，显示其路径，若找到路径则打印迷宫否则返回找不到路径。

(3) 算法分析

每走一步都要对上下左右四个方向进行判断，若到达终点有 n 步在最坏情况下，若迷宫无解，就得否定所有的解，也就是说会考虑所有的可能成为解的情况，时间复杂度达到 4^n ，但是不是所有的点必须要考虑全部的可行方向，即存在剪枝操作，这样减少了搜索的次数，实际的时间复杂度应该小于 4^n 。

(4) 源代码

```
import java.awt.*;
import java.util.Stack;
public class puzzle {
    public static void main(String args[]) {
        Stack<Point> path= new Stack<>();
        int[][] map={
            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
```

```

        {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
        {1, 0, 0, 1, 0, 0, 0, 1, 0, 1},
        {1, 0, 0, 0, 0, 1, 1, 0, 0, 1},
        {1, 0, 1, 1, 1, 0, 0, 0, 0, 1},
        {1, 0, 0, 0, 1, 0, 0, 0, 0, 1},
        {1, 0, 1, 0, 0, 0, 1, 0, 0, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 0, 1},
        {1, 1, 0, 0, 0, 0, 0, 0, 0, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
    };
    int[][] direction={{1,0},{0,1},{0,-1},{-1,0}};//分别代表下右左上四个方向, 因为是起点和终点构成对角线所以优先考虑下和右
    Point pos=new Point(1,1);
    System.out.println("迷宫 (1: 墙 0: 通路 2: 已走过的点, 最外层均为 1):");
    ");
    System.out.println("起点 (1, 1), 终点 (8, 8)");
    print_map(map);
    System.out.println();
    System.out.println("输出路径: ");
    System.out.print("(" + pos.x + ", " + pos.y + ") " + " ");
    if (solution(map, direction, path, pos))
        System.out.println("输出迷宫: ");
        print_map(map);
    }
    private static boolean solution(int map[][],int direction[][],Stack<Point> path,Point entry){
        Point cur=new Point(entry.x,entry.y);
        int count=0;
        path.push(cur.getLocation());
        overloop:while (!path.isEmpty()){
            if (cur.x!=entry.x&&cur.y!=entry.y&&cur.x==8&&cur.y==8){//终点为 (8,
8)
                map[cur.x][cur.y]=2;
                System.out.println();
                return true;
            }
            map[cur.x][cur.y]=2;
            for (int i=0;i<4;i++){
                cur.x+=direction[i][0];
                cur.y+=direction[i][1];
                if (check(map, cur)){
                    path.push(cur.getLocation());
                    count++;
                    if (count%5==0)
                        System.out.println();
                    System.out.print("(" + cur.x + ", " + cur.y + ") " + " ");
                    map[cur.x][cur.y]=2;
                    continue overloop;//继续最外层的循环, 下面的代码不再执行
                }else {
                    cur.x-=direction[i][0];
                    cur.y-=direction[i][1];
                }
            }
        }
    }

```

```

        }
    }
    cur.setLocation(path.peek()); //获取栈顶的点的位置，从栈顶的点的位置
继续摸索
    path.pop(); //将走不通的点弹出栈顶
    System.out.print("(" + cur.x + ", " + cur.y + ") " + " ");
}
return false;
}
private static boolean check(int[][] map, Point next) {
    return
(next.x < 10 && next.y < 10 && next.x >= 0 && next.y >= 0 && (map[next.x][next.y] == 0));
}
private static void print_map(int[][] map) {
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            System.out.print(map[i][j] + " ");
        }
        System.out.println();
    }
}
}
}

```

3. 利用队列解决迷宫最短路径问题

(1) 问题分析

借助于栈求解迷宫问题时，并不能保证找到一条从迷宫入口到迷宫出口的最短路径。而借助队列，可以找到从迷宫入口到迷宫出口的最短路径（如果有的话），该解决方案很好地表现了广度优先搜索是如何与队列先进先出（FIFO）的思想联系起来的，通过不断取得某个状态后能够达到的所有状态并将其加入队列，并且由于队列本身的特性先加入队列的状态总是先得到处理，这样就达到了一个目的：总是先将需要转移次数更少的状态进行分析处理。

(2) 设计方案

在这个问题中，找到从起点到终点的最短路径其实就是一个建立队列的过程：

- ① 从起点开始，先将其加入队列，设置距离为 0；
- ② 从队列首端取出位置，将从这个位置能够到达的位置加入队列，并且让这些位置的距离为上一个位置的距离加上 1；
- ③ 循环②直到将终点添加到队列中，这时说明已经找到了路径；

(3) 算法分析

在上述过程中，每次处理的位置所对应的距离是严格递增的，因此一旦找到终点当时的距离就是最短距离。若迷宫规模为 $M \times N$ ，那么本算法主要的时间开销就在求路径深度图算法上。算法主要过程是逐步判断每个单元格是否与周围 4 个单元格通和不通的关系。时间复杂度为 $O(4 \times M \times N)$ 。

(4) 源代码

```

import java.awt.*;
import java.util.LinkedList;

```



```

import java.util.Queue;
public class puzzle_shorst {
    public static void main(String args[]) {
        int[][] map={
            {1,1,1,1,1,1,1,1,1,1},
            {1,0,0,1,0,0,0,1,0,1},
            {1,0,0,1,0,0,0,1,0,1},
            {1,0,0,0,0,1,1,0,0,1},
            {1,0,1,1,1,0,0,0,0,1},
            {1,0,0,0,1,0,0,0,0,1},
            {1,0,1,0,0,0,1,0,0,1},
            {1,0,1,1,1,0,1,1,0,1},
            {1,1,0,0,0,0,0,0,0,1},
            {1,1,1,1,1,1,1,1,1,1}
        };

        Queue<Point> path = new LinkedList<>();
        int[][] direction={{1,0},{0,1},{0,-1},{-1,0}};//分别代表下右左上
        Point entry=new Point(1,1);
        Point out=new Point(8,8);
        System.out.println("迷宫 (1: 墙 0: 通路 最外层均为1): ");
        System.out.println("起点 (1, 1), 终点 (8, 8)");
        print_map(map);
        System.out.println();
        System.out.println("输出路径: ");
        solution(map,direction,path,entry,out);
        System.out.println("输出迷宫: ");
        print_map(map);
    }

    private static void solution(int[][] map,int[][] direction,
        Queue<Point> path, Point entry,Point out) {
        Point cur=new Point(entry.x,entry.y);
        Point next=new Point(entry.x,entry.y);
        path.offer(next.getLocation());
        map[cur.x][cur.y]=0;
        int i; int length=0;
        while(path.size()!=0) {
            cur.setLocation(path.poll());
            for (i=0;i<4;i++) {
                next.x=cur.x+direction[i][0];
                next.y=cur.y+direction[i][1];
                if (check(map,next,entry)) {
                    path.add(next.getLocation());
                    map[next.x][next.y]=map[cur.x][cur.y]+1;
                    if (next.x==out.x&&next.y==out.y) {
                        length=map[next.x][next.y];
                        break;
                    }
                }
            }
        }
    }
}

```

```

    }
    if (i!=4) {
        Point[] out_put=new Point[length];
        out_put[length-1]=next.getLocation();
        out:for (int a=length-1;a>0;a--) {
            for (int j=0;j<4;j++) {
                next.x+=direction[j][0];
                next.y+=direction[j][1];
                if (map[next.x][next.y]==a&&next.y!=0) {
                    out_put[a-1]=next.getLocation();
                    continue out;
                } else {
                    next.x-=direction[j][0];
                    next.y-=direction[j][1];
                }
            }
        }
        System.out.print("(" +entry.x+", "+entry.y+")"+" ");
        for (int b=0;b<length;b++) {
            if ((b+1)%7==0)
                System.out.println();
        }
        System.out.print("(" +out_put[b].x+", "+out_put[b].y+")"+" ");
    }
    System.out.print("总长度为: "+length);
    System.out.println();
    break;
}
}

private static boolean check(int[][] map,Point next,Point entry) {
    return next.x >= 0 && next.x < 10 && next.y >= 0 && next.y < 10 &&
    map[next.x][next.y]==0&&(next.x!=entry.x||next.y!=entry.y);
}

private static void print_map(int[][] map) {
    for (int i=0;i<10;i++) {
        for (int j=0;j<10;j++) {
            System.out.print(map[i][j]+" ");
        }
        System.out.println();
    }
}
}
}

```

4. 回溯法解决图着色问题

(1) 问题分析

给定无向连通图 $G=(V, E)$ 和正整数 m , 求最小的整数 m , 使得用 m 种颜色对 G 中的顶点着色, 使得任意两个相邻顶点着色不同。

(2) 设计方案

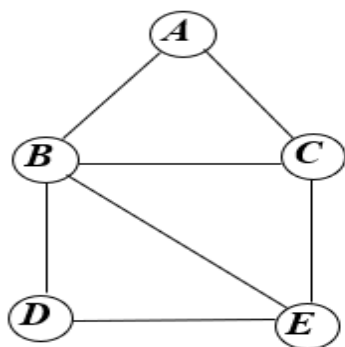
由于用 m 种颜色为无向图 $G=(V, E)$ 着色, 其中, V 的顶点个数为 n , 可以用一个 n 元组 $C=(c_1, c_2, \dots, c_n)$ 来描述图的一种可能着色, 其中, $c_i \in \{1, 2, \dots, m\}$ ($1 \leq i \leq n$) 表示赋予顶点 i 的颜色。例如, 5 元组 $(1, 2, 2, 3, 1)$ 表示对具有 5 个顶点的无向图的一种着色, 顶点 1 着颜色 1, 顶点 2 着颜色 2, 顶点 3 着颜色 2, 如此等等。如果在 n 元组 C 中, 所有相邻顶点都不会着相同颜色, 就称此 n 元组为可行解, 否则为无效解。

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。在图着色问题的解空间树中, 如果从根节点到当前节点对应一个不分解, 也就是所有的颜色指派都没有冲突, 则在当前结点处选择第一棵子树继续搜索, 也就是为下一个顶点着颜色 1, 否则, 对当前子树的兄弟子树继续搜索, 也就是为当前顶点着下一个颜色, 如果所有 m 种颜色都已尝试过并且都发生冲突, 则回溯到当前节点的父节点处, 上一个顶点的颜色被改变, 以此类推。

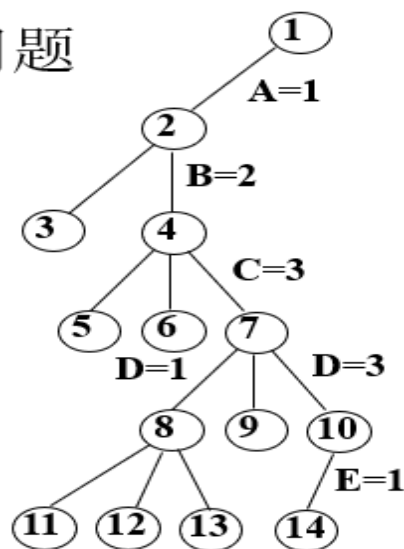
(3) 算法分析

用 m 种颜色为一个具有 n 个顶点的无向图着色, 共有 m^n 种可能的着色组合, 因此, 解空间树是一棵完全 m 叉树, 树中每一个结点都有 m 棵子树, 最后一层有 m^n 个叶子结点, 每个叶子结点代表一种可能着色, 最坏情况下的时间性能是 $O(m^n)$ 。对于本次实验样例中的无向图, 解空间树中共有 364 个结点, 而回溯法只搜索了其中的 14 个结点后就找到了问题的解。

用回溯法实现图着色问题



(a) 一个无向图



(b) 回溯法搜索空间

(4) 源代码

```
public class Graph_Color {  
    private static int[][] graph={  
        {0, 1, 1, 0, 0},  
        {1, 0, 1, 1, 1},  
        {1, 1, 0, 0, 1},  
        {0, 1, 0, 0, 1},  
        {0, 1, 1, 1, 0}};
```

```

private static int[] color={0,0,0,0,0};
private static void GraphColor(int m){
    int k;
    k=0;
    while(k>=0){
        color[k]=color[k]+1;
        while (color[k]<=m){
            if (Ok(k))
                break;
            else color[k]=color[k]+1;
        }
        if (color[k]<=m&& k==4){
            for (int i=0;i<5;i++){
                System.out.print(color[i]+" ");
            }
            return;
        }
        if (color[k]<=m&& k<4)
            k=k+1;
        else
            color[k--]=0;
    }
}

private static boolean Ok(int k) {
    for (int i=0;i<k;i++){
        if (graph[k][i]==1&& color[i]==color[k])
            return false;
    }
    return true;
}

public static void main(String args[]){
    GraphColor(3);
}
}

```

5. 调试心得

经过本次实验，对回溯法的基本思想、栈与队列的应用以及广度优先搜索的思想有了深刻的认识，在调试过程中出现了各种各样的问题，出现数组下标越界，压栈时出现两个对象共享地址单元的情况，从而导致栈中的元素值一直都是几个一样的值，在单步调试查看内存状态之后才意识到问题，针对解决问题上，对迷宫问题，到达终点的限定条件不太明确导致程序出错，总而言之吃一堑长一智，下次再碰到一些基础错误希望能过一眼就知道问题出在哪儿。