成绩	

《Java 程序设计I》课程设计报告

基于图模型探索分酒问题（三容器）的一般解

学　　院：_____

班　　级：_____

学　　号：_____

姓　　名：_____

完成时间：_____

目录

1 问题描述

泊松分酒问题：

有 3 个无刻度的容器，容量分别为 A 升、B 升、C 升，A>B>C。其中 A 中装满酒，另外两个空着。要求你只用这三个容器操作，最后使得某个容器中正好有 D 升酒。

Task 1：对于 A=12，B=8，C=5，D=6，打印出最简单的一种方案。

Task 2: 对于一般情况，判断是否有解，如果有解，给出最简单的一种方案。

2 问题分析

2.1 初步分析

由于泊松分酒问题是针对三个容器的分酒问题，很自然的想到用三维直角坐标系的三个维度分别表示 ABC 三个容器的盛酒量。

以 Task1 为例，经过尝试后比较容易想到一种解：

【0】初始状态（12,0,0） 【1】A 向 B 倒入 8 升（4,8,0） 【2】B 向 C 倒入 5 升（4,3,5）

【3】C 向 A 倒入 5 升（9,3,0） 【4】B 向 C 倒入 3 升（9,0,3） 【5】A 向 B 倒入 8 升（1,8,3）

【6】B 向 C 倒入 2 升（1,6,5）

表 1：

步骤	操作	三维坐标		
		x(A)	y(B)	z(C)
0	初始状态	12	0	0
1	A 向 B 倒入 8 升	4	8	0
2	B 向 C 倒入 5 升	4	3	5
3	C 向 A 倒入 5 升	9	3	0
4	B 向 C 倒入 3 升	9	0	3
5	A 向 B 倒入 8 升	1	8	3
6	B 向 C 倒入 2 升	1	6	5

由表 1 可建立三维坐标系及坐标之间的移动关系

故而，泊松分油问题就是按照分酒的规则在三维直角坐标系中寻找从初始状态的坐标（12,0,0）到坐标（a,b,c）（a=D or b=D or c=D）的路径，表 1 的操作表示了一种路径的走法，即为一种分酒方法。

在三维直角坐标系中用 x、y、z 表示 ABC 三个容器的盛酒量。

对于 Task1 而言，x、y、z 受到以下限制：

（1）x+y+z=12

（2）0≤x≤12

（3）0≤y≤8

（4）0≤z≤5

（5）x,y,z∈Z^+

因为 x+y+z=12，所以可以只考虑 x,y，z 隐藏在 z=12-x-y 中，可将三维直角坐标系转化为二维直角坐标系。注意在转化过程中并非选择三维直角坐标系中（12,0，0）（0,12,0）（0,0,12）所在平面，而是将坐标垂直投影到 xOy 平面（即忽略 C 容器）。

在二维直角坐标系中，x、y 受到以下限制

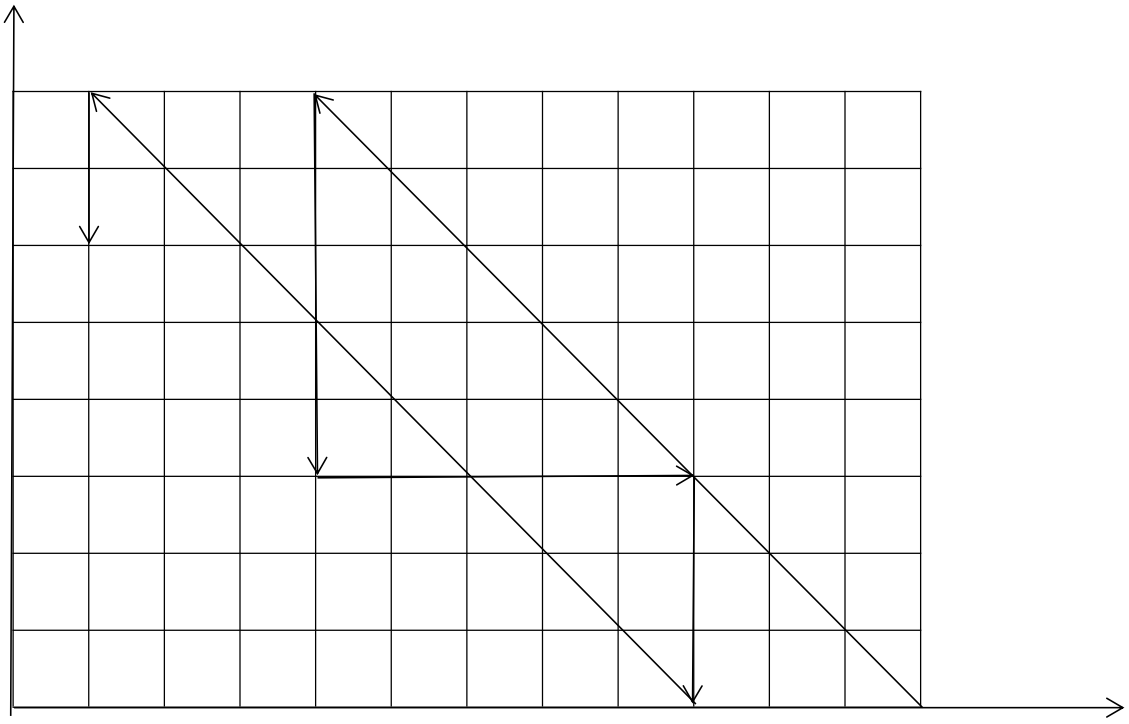
（1）0≤x≤12

（2）0≤y≤8

（3）7≤x+y≤12

（4）x,y∈Z^+

表 1 所示的分酒步骤可表示为下图：



2.2 建立图模型

在以上限制中的每一个坐标都代表了分酒过程的一种状态，共有 53 种状态。

x \ y	0	1	2	3	4	5	6	7	8
0								V0,7,5	V0,8,4
1							V1,6,5	V1,7,4	V1,8,3
2						V2,5,5	V2,6,4	V2,7,3	V2,8,2
3					V3,4,5	V3,5,4	V3,6,3	V3,7,2	V3,8,1
4				V4,3,5	V4,4,4	V4,5,3	V4,6,2	V4,7,1	V4,8,0
5			V5,2,5	V5,3,4	V5,4,3	V5,5,2	V5,6,1	V5,7,0	
6		V6,1,5	V6,2,4	V6,3,3	V6,4,2	V6,5,1	V6,6,0		
7	V7,0,5	V7,1,4	V7,2,3	V7,3,2	V7,4,1	V7,5,0			
8	V8,0,4	V8,1,3	V8,2,2	V8,3,1	V8,4,0				
9	V9,0,3	V9,1,2	V9,2,1	V9,3,0					
10	V10,0,2	V10,1,1	V10,2,0						
11	V11,0,1	V11,1,0							
12	V12,0,0								

所有路径都在这些状态的转移中产生，研究的重点就在于这些状态之间的相互转移。

状态转移常常使用图建模。

每一坐标都代表分酒的一种状态，用一个顶点（结点）表示，若两个状态之间可以相互转换，则用一条边将这两个顶点连起来，赋权值为 1。由于所有边权值均为 1，可视该图为无权的。最后删去孤立点。

借助原本的二维直角坐标系的点和线，可以构建比较完整且美观的图模型。

示意图如下：（由于设定起点，所以可视为有向的，对于一般状态图是无向的）

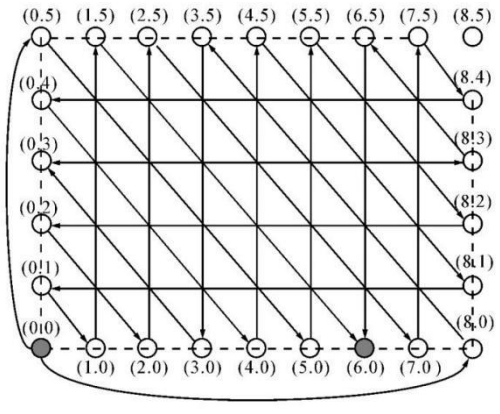


图 2　原始的几何坐标状态图

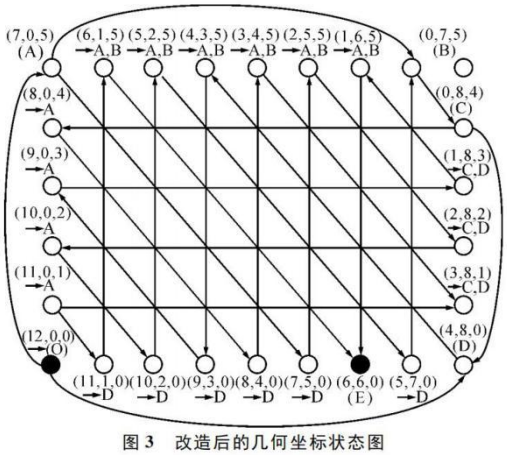


图 3　改造后的几何坐标状态图

该图有以下特点：

①无向性：因为不存在将酒倒出三个容器的行为，酒总量不变，所以相邻顶点一定可以相互转换。对称的有向图可以用无向图表示。

②没有多重边：相邻顶点间有且仅有一种转移方式。

③存在回路：相邻状态间可相互转换，存在可相互转换的回路，如 V12,0,0 --V4,8,0 --V0,8,4--V0,7,5--V7,0,5--V12,0,0 的回路。

综上所述，泊松分酒问题的图模型是一个存在简单回路的连通无向图。

解决分酒问题的方法就是从初始状态的结点搜索一条到目标结点的通路，显然该通路中不能存在重复的结点，否则可以无限循环分酒。

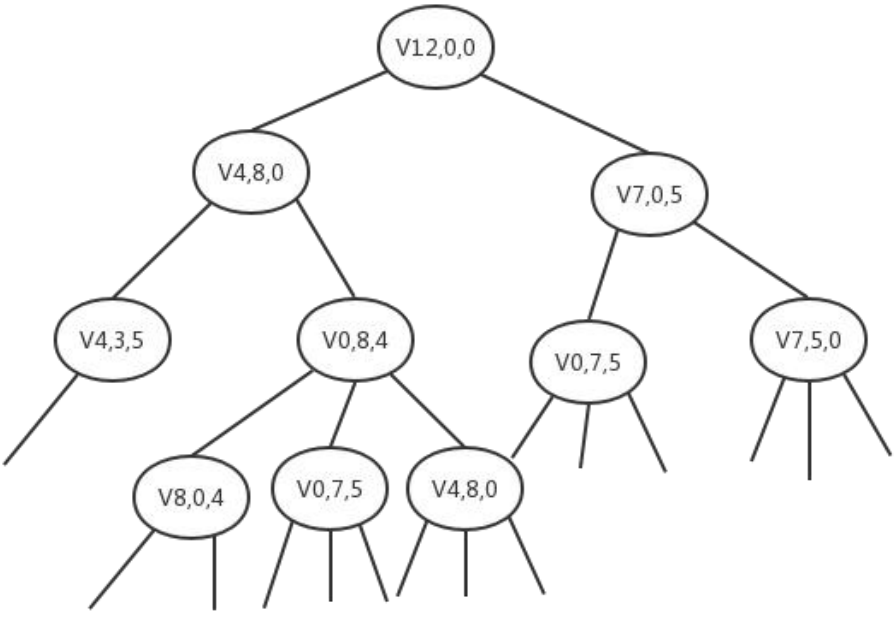
3 总体设计

显然，从问题出发，每次搜索有一个初始状态，泊松分酒问题就成为确定起点的图搜索问题。有两种比较容易想到的方式，对应生成两种树。

3.1 记忆化搜索

记忆化搜索的本质是暴力搜索，是将之前搜索过的路径和结点记录下来，这对分酒问题是重要的，因为如果可以重复搜索之前出现的状态，分酒将一直进行下去。

将除该结点的双亲外结点在图中的所有邻居都作为该结点的孩子，很显然这可以得到所有路径。一个状态结点可能会在该树中重复出现。对于这种图，可以理解为：某结点的孩子是该结点下一步可以达到的所有状态。如果需要搜索最短路径，则必须搜索完所有路径，然后比较得到最短路径。



DP 记忆化搜索的算法大体如下：

```
dfs(problem a){
    if(a has been solved)
        then: consult the record.
    else
        divide the problem a into several sub-problems(a1,a2,...,ak)
        get the solution of problem a by dfs(a1),dfs(a2),...,dfs(ak).
    finally write the optimal solution into record.
 }
```

对于分酒问题而言：

```
dfs(problem 分酒){
    if(得到目标状态)
        then: 存储路径
    else
        求下一可能的状态 state1,state2...stateN
       递归调用 dfs(state1),dfs(state2),...,dfs(stateN).
     }
```

输出最优解

3.2 广度优先搜索

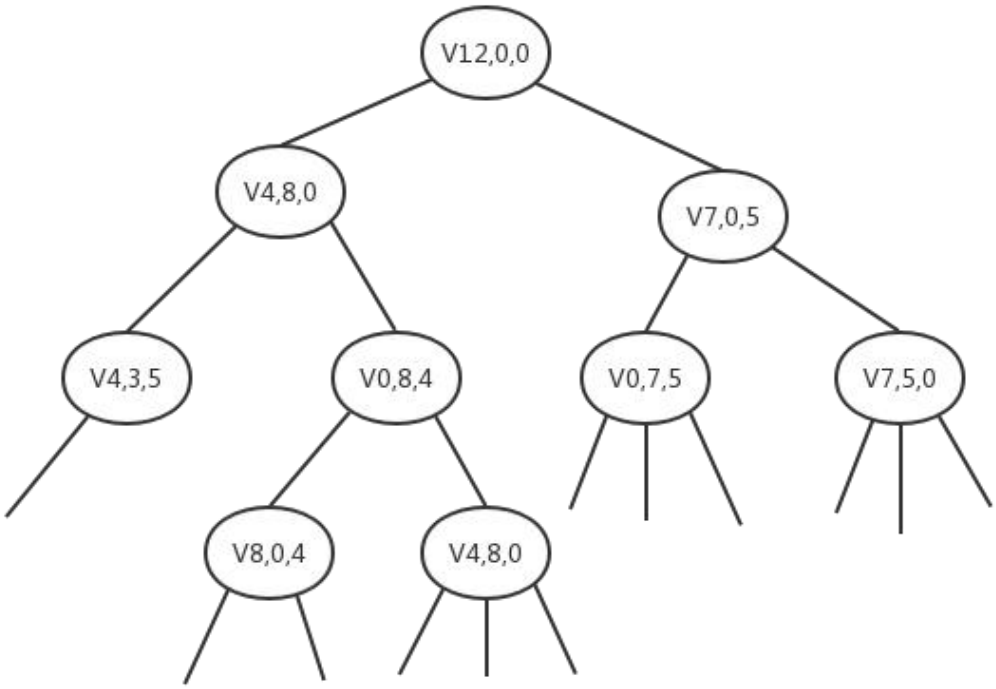
对无向图的遍历采用广度优先搜索，得到的广度优先生成树，每个结点在树中只出现一次，且保证树根到该结点路径最短。在搜索到目标结点后即可停止搜索，是比较理想的算法。

简单的说，BFS 是从根节点开始，沿着树的广度遍历树的节点，如果发现目标，则演算终止。BFS 属于一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止。

之所以称之为宽度优先算法，是因为算法自始至终一直通过已找到和未找到顶点之间的边界向外扩展，就是说，算法首先搜索和 s 距离为 k 的所有顶点，然后再去搜索和 S 距离为 k+1 的其他顶点。

为了保持搜索的轨迹，宽度优先搜索为每个顶点着色：白色、灰色或黑色。算法开始前所有顶点都是白色，随着搜索的进行，各顶点会逐渐变成灰色，然后成为黑色。在搜索中第一次碰到一顶点时，我们说该顶点被发现，此时该顶点变为非白色顶点。因此，灰色和黑色顶点都已被发现，但是，宽度优先搜索算法对它们加以区分以保证搜索以宽度优先的方式执行。若(u,v)∈E 且顶点 u 为黑色，那么顶点 v 要么是灰色，要么是黑色，就是说，所有和黑色顶点邻接的顶点都已被发现。灰色顶点可以与一些白色顶点相邻接，它们代表着已找到和未找到顶点之间的边界。

在广度优先搜索过程中建立了一棵广度优先树，起始时只包含根节点，即源顶点 s.在扫描已发现顶点 u 的邻接表的过程中每发现一个白色顶点 v，该顶点 v 及边(u,v)就被添加到树中。在广度优先树中，称结点 u 是结点 v 的先辈或父母结点。因为一个结点至多只能被发现一次，因此它最多只能有--个父母结点。



4 详细设计

4.1 记忆搜索

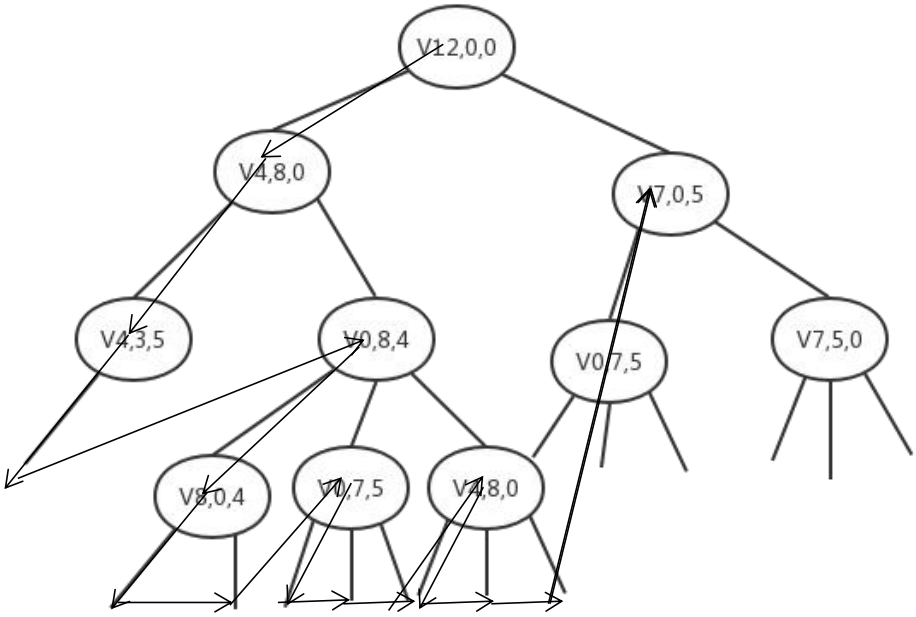
4.1.1 具体实现思想

核心思想：**递归、栈、剪枝**

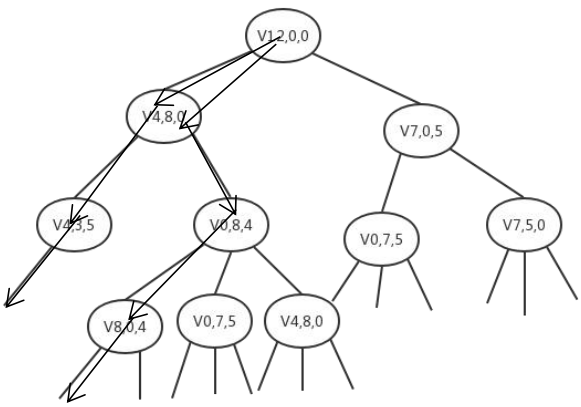
递归产生新状态（子结点），用栈记录路径，一旦产生目标状态，则将路径保存进容器。在每次递归中对潜在的子结点进行剪枝，剪去其中不能产生的结点。

若某一叶子节点不是目标结点，则转向双亲结点的另一子结点进行递归，直至求得全部路径。

示意图：

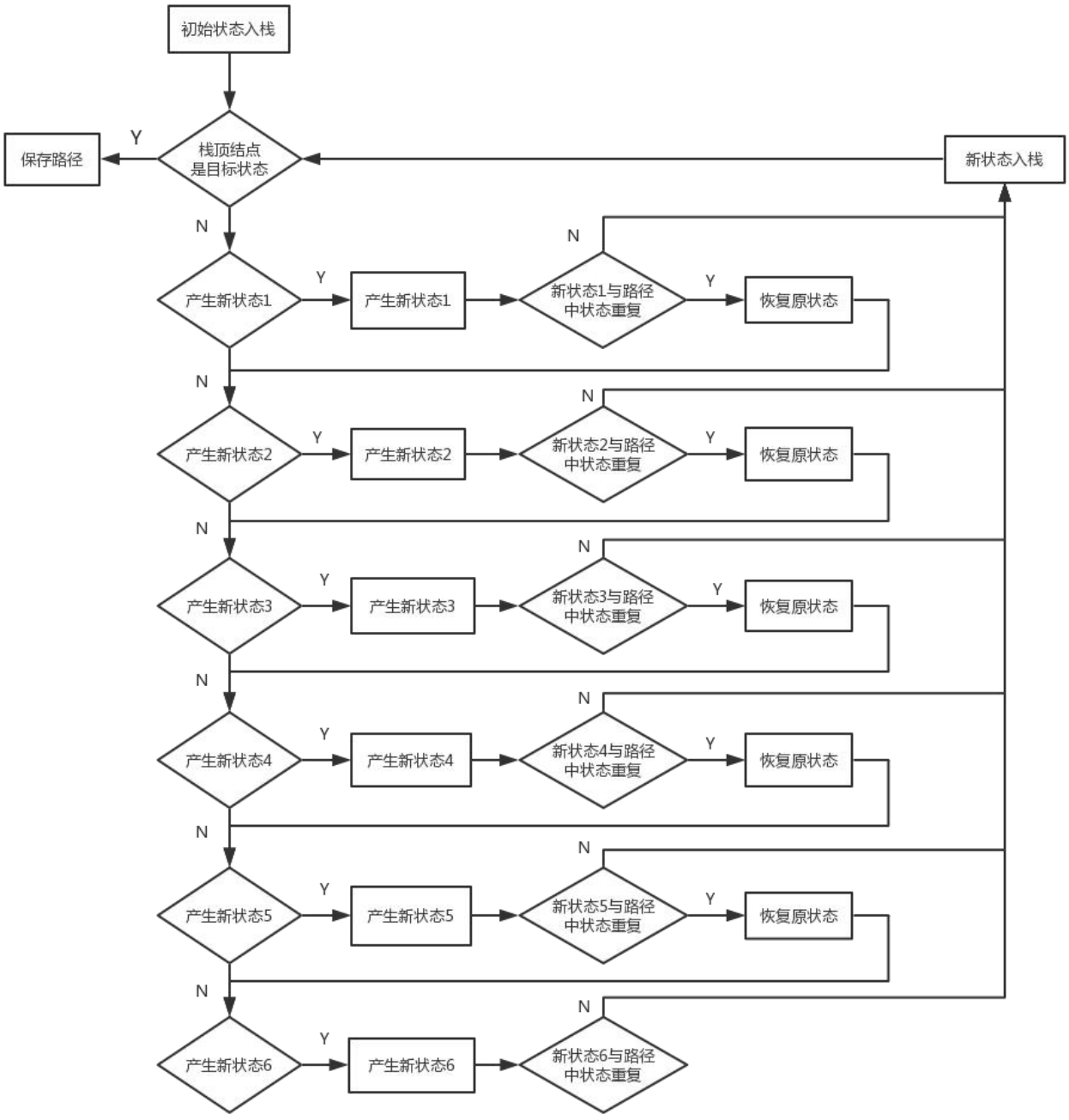


每次递归都以找到目标状态或达到叶子结点为止，否则将继续递归。故而暴力搜索的本质是将所有路径认为是潜在的目标路径，依次判断某条路径是否可行。记忆搜索只是存储了上一次搜索的路径，使得不必从根结点开始搜索，减少了运算时间。



如左图中，前两次搜索路径的本质是两条从根结点开始的路径，记忆搜索使得第二次搜索不必进行 V12,0,0->V4,8,0 这一步骤。

4.1.2 程序流程图


初始状态入栈
栈顶结点
是目标状态
Y
保存路径
新状态入栈
N
产生新状态1
Y
产生新状态1
新状态1与路径
中状态重复
N
Y
恢复原状态
产生新状态2
产生新状态2
新状态2与路径
中状态重复
产生新状态3
产生新状态3
新状态3与路径
中状态重复
产生新状态4
产生新状态4
新状态4与路径
中状态重复
产生新状态5
产生新状态5
新状态5与路径
中状态重复
产生新状态6
产生新状态6
新状态6与路径
中状态重复

4.1.3 程序说明

4.1.3.1 面向过程

由于只是面向过程设计的方法，所以在一个类 class solve 中解决所有问题。

4.1.3.2 变量说明

①状态数组 private int[] state=new int[3]

用于记录当前三容器的状态

②栈 private List<String> history=new ArrayList<>();

private List<Integer> course=new ArrayList<>();

这两个栈本质是相同的，两个栈中元素也是同步的，history 用 String 记录每一状态，如“480”为一状态；course 是辅助 history 判断该状态是否有某一容器达到目标状态。每次产生新状态时，就将新状态加入栈中，当遍历完以该状态为树根的树后，将该状态弹出栈。

③存储容器 private List<String> allResult=new ArrayList<>();

allResult 存储所有符合题目条件的路径

④容器数组 private int[] size=new int[3];

记录三个容器的最大容量

⑤目标数量 private int key;

需要得到的量

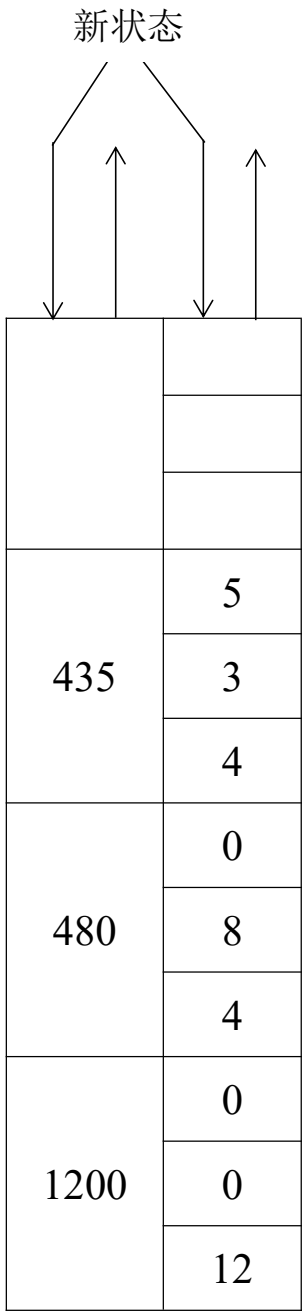
4.1.3.3 函数说明

```
private void addHistory(){
    String his= state[0]+","+ state[1]+","+ state[2]+",";
    history.add(his);
}
private void deleteHistory(){
    history.remove(history.size()-1);
}
private void addCourse(){
    for(int i = 0; i< state.length; i++){
        course.add(state[i]);
    }
}
private void deleteCourse(){
    int length=course.size();
    for(int i=length-1;i>length-4;i--){
```

```
            course.remove(i);
        }
    }
```

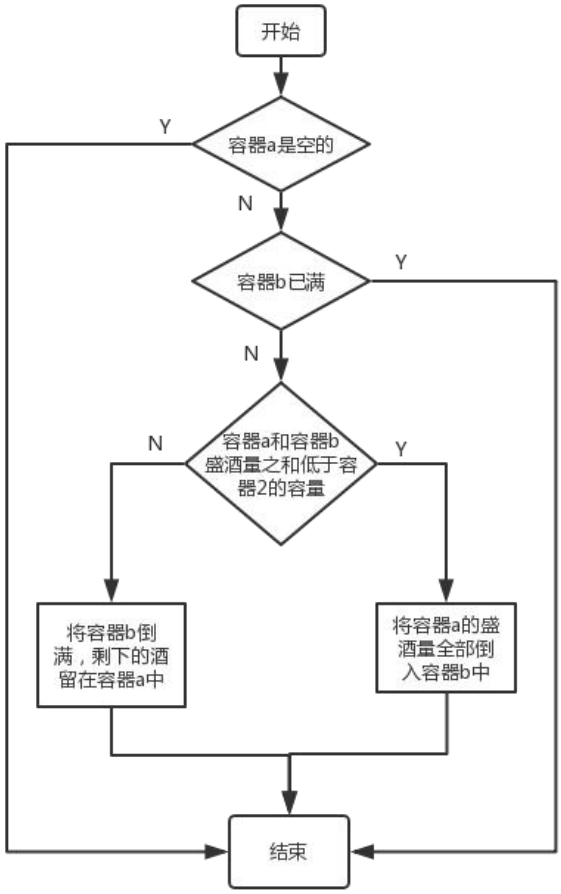
以上四个函数是对栈的操作，分别是两个入栈和两个出栈。鉴于 history 和 course 栈是完全同步的，所以两个栈可以看做一个栈，四个函数也只做了两种操作，一个入栈操作一个出栈操作。

course 是辅助 history 判断该状态是否有某一容器达到目标状态。每次产生新状态时，就将新状态加入栈中，当遍历完以该状态为树根的树后，将该状态弹出栈。



```
private boolean move(int a,int b){
    //a 酒杯是空酒杯
    if(state[a]==0) return false;
    //b 酒杯是满酒杯
    if(state[b]==size[b]) return false;

    //b 酒杯的剩余空间大于等于 a 酒杯的
    if(size[b]- state[b]>= state[a]){
        state[b]+= state[a];
        state[a]=0;
    }
    //a 酒杯减去 b 酒杯的剩余容量
    else{
        state[a]-=(size[b]- state[b]);
        //b 酒杯装满
        state[b]=size[b];
    }
    return true;
}
```



move 函数是倒酒的过程，返回布尔值代表该此倒酒是否成功。

设倒出酒的容器为容器 a，接受酒的容器为容器 b。

如果容器 a 已空或容器 b 已满，倒酒失败，退出函数。

如果容器 a 和容器 b 盛酒量之和低于容器 2 的容量，则可以将容器 a 的盛酒量全部倒入容器 b 中。

如果容器 a 和容器 b 盛酒量之和高于容器 a 的容量，则将容器 b 倒满，剩下的酒留在容器 a 中。

```
private boolean getKey(){
    for(int i = 0; i< state.length; i++){
        if(state[i]== key){return true;}
    }
    return false;
}
```

getKey 函数检测状态是否包含所需 key。

```
private boolean isInHistory(){
    String his= state[0]+","+ state[1]+","+ state[2]+",";
    return history.contains(his);
}
```

isInHistory 函数检测状态是否在 history 栈中，也就是该状态是否在之前的路径中已出现。已访问的状态不应该再被访问，否则将产生环，分酒会无限进行。所以，检测到重复就返回 false，以便排除该状态；不重复返回 true，继而入栈。

```
private void storeCourse(){
    allResult.add(String.join("",history));
}
```

storeCourse 函数用于在得到目标结点后将整个路径保存入 allResult 中。

```
private void output(){
    int min=0;
    for(int i=1;i<allResult.size();i++){
        if(allResult.get(i).length()<allResult.get(min).length()){
            min=i;
        }
    }
    //按格式输出
    System.out.println("步骤最少的移动方案：");
    String[] strings=allResult.get(min).split(",");
    for(int i=0;i<strings.length;i++){
        if(i%3==2){
            System.out.print(strings[i]+"\n");
        }
        else {
            System.out.print(strings[i] + ",");
        }
    }
}
```

output 是最后的输出函数，输出所有路径中的最短路径。由两部分组成，一是在 allResult 中找出最短路径，二是将该路径按照格式输出。

```
private void process(){
    if(getKey()){
        storeCourse();
        return;
    }
    int one,two,three;
    one= state[0];
    two= state[1];
    three= state[2];
    int[] temp={1,2,0,2,0,1};
    for(int i=0;i<6;i++){
        if(!move(i/2,temp[i])){
            continue;
        }
        if(isInHistory()){
state[0]=one;

            state[1]=two;
            state[2]=three;
            continue;
        }
        else{
            addHistory();
            addCourse();
        }
        process();
        state[0]=one;
        state[1]=two;
        state[2]=three;
        deleteCourse();
        deleteHistory();
    }
}
```

以 Task1 为例说明 process 函数：

前提：在构造函数中将初始状态（12,0,0）入栈

如果栈顶元素 getKey，就将路径存储，函数结束

不是目标状态就将该状态暂存入 one,two,three 变量以便产生子节点后能找回原状态

temp 数组用了小技巧，避免了两层循环

作为三个容器的分酒问题，每次分酒都存在六种潜在的子状态，分别由 A->B,A->C,B->A,B->C,C->A,C-B 产生。如果不能产生子状态则跳过，尝试产生另一子状态。如果是在这里跳过，则没有改变状态数组，不需要恢复原状态。

如果在路径中已有产生的新状态，就恢复原状态并返回，尝试产生另一子状态。

如果是新的状态，就入栈。

递归调用 process

注意这里要恢复原状态，因为该状态产生的所有情况已回溯完毕，需要恢复原状态，产生同级别的另一新状态

并且将该状态出栈

4.1.4 小结

对于状态的搜索，暴力搜索是最容易想到的方法。暴力搜索是一个非常一般的解决问题的技术，包括系统地枚举解决方案的所有可能的候选项，以及检查每个候选项是否符合问题的描述。例如八皇后问题的暴力算法会检查所有在 8*8 棋盘上八个“皇后”可能的摆放方法，并且对每一种摆放方法，检查其每一个“皇后”是否能攻击到其它人。

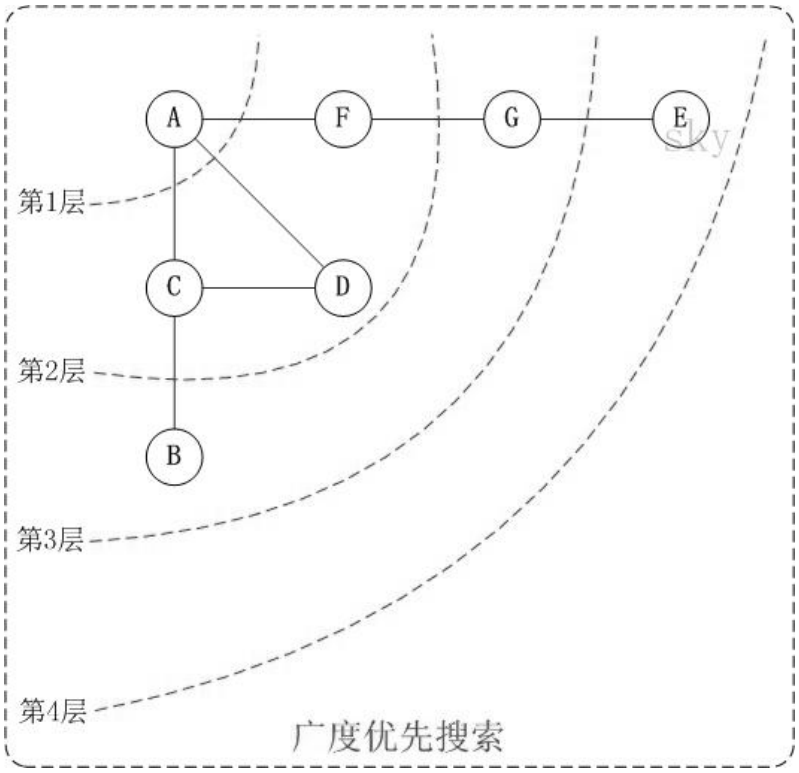
虽然暴力搜索很容易实现，并且如果解决方案存在它就一定能够找到，但是它的代价是和候选方案的数量成比例的，由于这一点，在很多实际问题中，消耗的代价会随着问题规模的增加而快速地增长。因此，当问题规模有限，或当存在可用于将候选解决方案的集合减少到可管理大小的针对特定问题的启发式算法时，通常使用暴力搜索。另外，当实现方法的简单度比速度更重要的时候，也会用到这种方法。对于三容器的泊松分酒问题，问题规模有限，总状态不多，每个结点产生的新结点也不多，递归深度也不深，所以暴力搜索在短时间内一般都可以得到最优解或判断出无解。

4.2 广度优先搜索 BFS

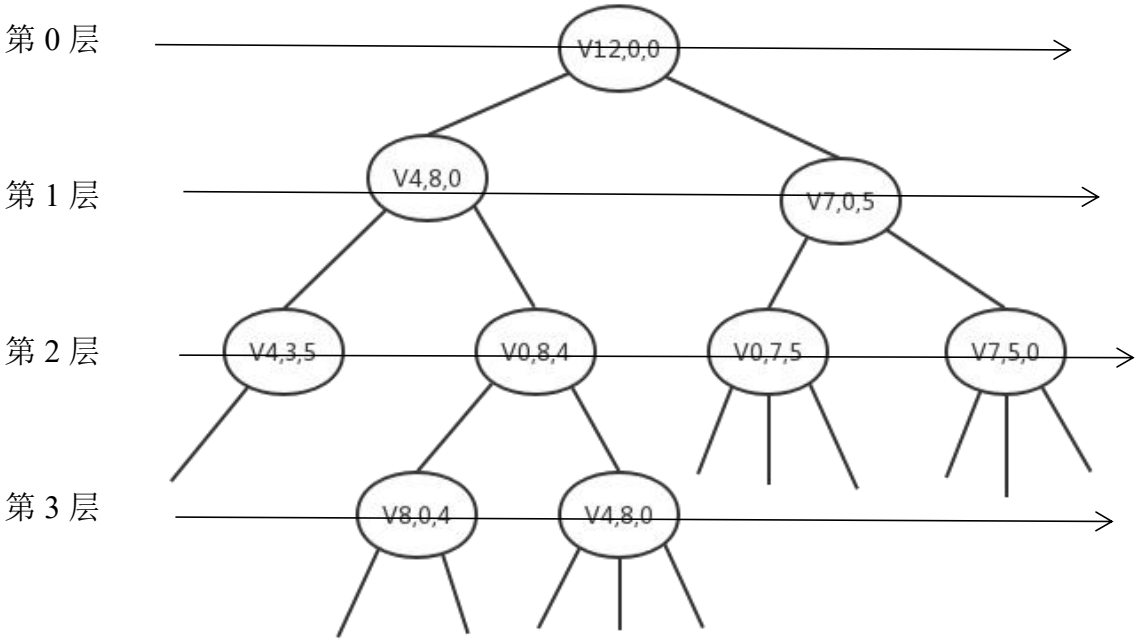
4.2.1 具体实现思想

BFS 的思想是：从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。换句话说，广度优先搜索遍历图的过程是以 v 为起点，由近至远，依次访问和 v 有路径相通且路径长度为 1、2.....的顶点。且任一顶点到第 1 层顶点 v 的路径最短。

分酒问题求最短路径时，广度优先搜索显然比暴力搜索更简便。



分酒问题的搜索方向如下：



对于分酒问题，核心思想为队列、循环、剪枝、路径记录

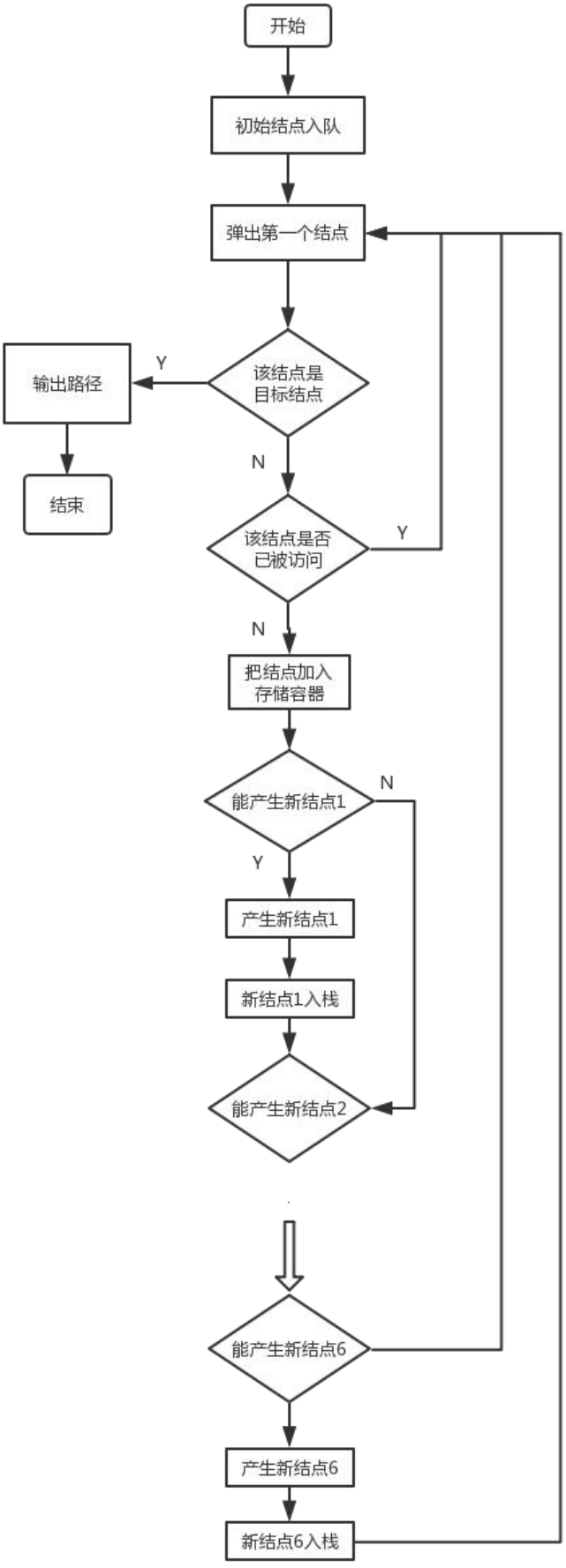
①队列：用于暂存第 n 层所有结点，一遍逐层查找未搜索结点。

②循环体现在：初始结点入队后，重复进行队列第一个元素出队列（假设该元素为第 n 层），将与该出队元素相邻的未被访问过的元素作为第 n+1 层入队，直至查找到目标结点。

③剪枝：从在对出队结点的邻居进行查找时，需要剪去不能产生的结点和已经访问过的结点。

④广度优先搜索的一个比较大的问题是路径记录问题，因为逐层查找并不能直接得到某一条路径到达所求结点。在本次解决分酒问题时，我采用的是子结点记录父结点的方式，将路径用父子关系链接起来，在输出时只需要从目标结点递归得查找父结点，而后回溯依次输出即可。

4.2.2 程序流程图


开始
初始结点入队
弹出第一个结点
该结点是
目标结点
Y
输出路径
结束
N
该结点是否
已被访问
Y
N
把结点加入
存储容器
能产生新结点1
N
Y
产生新结点1
新结点1入栈
能产生新结点2
能产生新结点6
产生新结点6
新结点6入栈

4.2.3 程序分析

4.2.3.1 面向对象

用 class Node 记录三容器的状态，另一目的是在类中可以记录某一状态的父状态，可将 Node 类近似看做 A,B,C,parent 结构体，也便于增添其他操作。如果单纯为了记录父状态，可以尝试使用 Hash 表。采用 map 容器进行存储,其可通过<key,value>的方式进行存储数据,用 key 表示一个具体的状态字符串,value 表示上一具体状态迁移变化成 key 状态过程的字符串。从而可以方便的回溯寻找最小路径。

在解决问题的 WineQuestion 中，采用面向过程的设计思想。

4.2.3.2 变量说明

1.class Node 中

```
private int A;
private int B;
private int C;
private Node parent;
```

A,B,C 分别代表某一状态容器 A,B,C 的盛酒量

parent 记录该状态的父状态

2.class WineQuestion 中：

①private List<Node> workingList = new ArrayList<>();

workingList 是工作队列，是广度优搜索 open-close 中的 open 容器（队列），用于暂存还未搜索其子结点的结点。

②private List<Node> visitedNode = new ArrayList<>();

是广度优搜索 open-close 中的 close 容器，用于存储已访问过的状态

③private int[] contains=new int[]{12,8,5};

存储三容器的容量

④private Node head = new Node(12, 0, 0);

初始状态对应的结点

⑤private int key=6;

最终状态

4.2.3.3 函数说明

①构造函数

对于所有情况，只需要一种构造函数即可，即为给 contains、head、key 赋值。

因为建议每个类都有一个默认的构造函数，选择 Task1 的参数作为默认。

```
public WineQuestion(){ }
    //构造（自定义情况）
public WineQuestion(int contains1,int contains2,int contains3,int A,int B,int C,int key){
    contains[0]=contains1;
    contains[1]=contains2;
    contains[2]=contains3;
    head.setA(A);
    head.setB(B);
    head.setC(C);
    this.key=key;
}
```

②solve 函数

所有处理都在 solve 中完成。因为是类似递归产生结点，所以代码比较简短。

```
public void solve() {
    workingList.add(head);                  //最先开始在队列中加入头结点

    //循环‘‘出-入’’直至找到目标或队列元素全部取出

    while (true) {
        if (workingList.isEmpty()) {            //队列空则退出，表示未找到解
            System.out.println("无解");
            return;
        }

        Node node = workingList.remove(0);     //弹出并接受工作队列的第一个元素

        if (node==null) {                        //  如果节点为空,就输出错误信息
            System.out.println("FAIL");
            return;
        }
        if (node.contain(key)) {              //如果得到目标，就输出序列
            printRoute(node);
            break;
        }
        if (visitedNode.contains(node)) {      //如果这个节点已经访问过，就跳过此点，直接下一次循环
            continue;
        }

        visitedNode.add(node);                   //未在访问序列出现过的状态加入到已访问节点序列

        //产生新的状态
        //A->B
        if(node.getA()>0&&node.getB()<contains[1]){
            if(node.getA()+node.getB()<=contains[1])
```

```
                    workingList.add(new Node(0,node.getA()+node.getB(),node.getC(),node));
                if(node.getA()+node.getB()>contains[1])
                    workingList.add(new
Node((node.getA()-(contains[1]-node.getB())),contains[1],node.getC(),node));
            }
            //A->C
           ......（省略）
            //B-C
            ......（省略）
            //B-A
            ......（省略）
            //C-A
            ......（省略）
            //C-B
            if(node.getC()>0&&node.getB()<contains[1]){
                if(node.getC()+node.getB()<=contains[1])
                    workingList.add(new Node(node.getA(),node.getB()+node.getC(),0,node));
                if(node.getC()+node.getA()>contains[0])
                    workingList.add(new
Node(node.getA(),contains[2],node.getC()-(contains[1]-node.getB()),node));
            }
        }
}
```

说明前做以下规定：

Node（a,b,c,parent）代表 Node 状态容器 A 中盛酒量为 a 升，B 中为 b 升，C 中为 c 升，parent 为其父状态 Node（a’,b’,c’,parent’）。

以 Task1 中参数为例说明 solve 函数的具体运行：

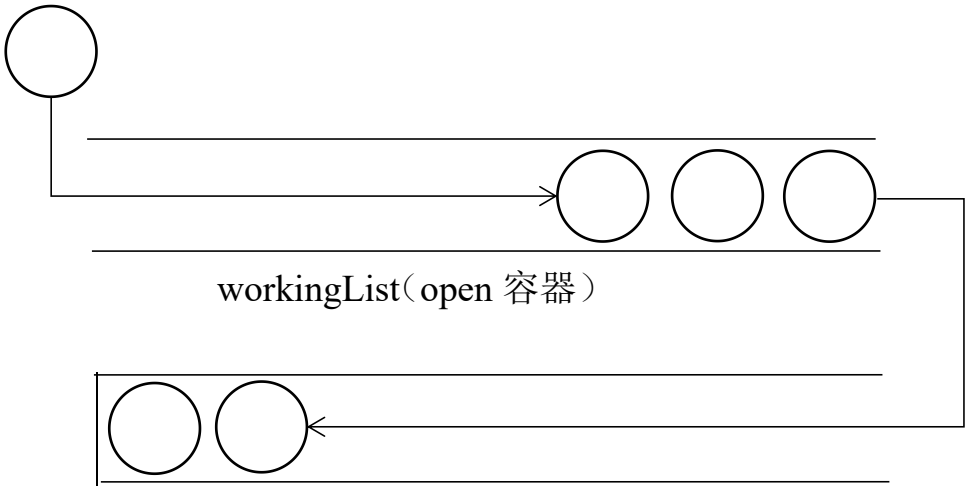
前提为初始状态 Node（12,0,0,null）入队。

首先判断工作队列是否为空，非空，队首元素 Node（12,0,0，null）出队，用变量 node 接收这个元素。

判断这个元素是否包含目标 key，如果包含代表搜索完成，则输出路径，结束函数。如果不包含 key，判断该结点是否已被访问过，如果未被访问，则加入 close 容器，如果已被访问，该结点将在下一次元素出队时被丢弃。

而后搜索该出队元素 Node 所有可能的结点，包括已被访问的结点，只要结点在图中与该结点相邻，就全部入队。

而后循环进行以上“出队-入队”操作直至 open 容器（工作队列）为空，此时 close 容器中保存了在目标结点前搜索的所有结点，且在 close 容器中出现且仅出现一次，按照层次有顺序的存放。



对该流程很容易发现一种优化方案。

现程序 version1.0 中在弹出队列首元素时判断是否为目标元素，那么对于搜索最短路径而言，弹出元素结点后的入队元素都是多余的操作。如果在产生子结点时就判断该子结点是否是目标结点，将会节约 n 层结点数量至 n+1 层结点数量的运算。相应的，修改后代码会比较长，显得不是那么简洁。因为 BFS 搜索子结点不能循环产生，必须一一判断产生，故而每次产生后都要判断产生的子结点是否为目标结点，如果是，则输出路径。

4.2.4 小结

对于求最短分酒步骤的泊松分酒问题，显然广度优先搜索比暴力搜索更好用。对于搜索最短路径，暴力搜索所有路径的时间复杂度必然为 o（V+E），加上寻找最短路径的时间复杂度 o（路径数），远远大于 BFS 时间复杂度 o（V+E）。BFS 虽是一种盲目搜索，但只有在最坏的情况下需要搜索所有边和顶点。且相较于暴力搜索需要存储所有路径，广度优先搜索在存储空间上明显更少。

5 项目总结

本文基于图模型探索泊松分酒问题的解决方法。

对于三容器的一般解，可以直接通过文中程序作解。对于四容器、五容器以及更为一般的情况，思路与三容器完全相同，只在子状态产生是需要考虑更多的可能性。故而可以认为基本解决了分酒问题的一般解问题。

对于图模型的最短路径搜索问题，考虑到分酒问题的图无权等具体情况，此次课程设计中只选择了两种较为简单的思路，即暴力搜索和广度优先搜索。读者如果有兴趣，也可尝试采用如 Prim 算法、迪杰斯特拉算法等其他算法对最短路径更为有效的算法进行搜索。

关于暴力搜索和广度优先搜索，前者搜索所有路径，对于非求最短路径的其他问题一般也可以做出解答，BFS 则对最短路径问题具有比较好的针对性。

此外，在图解法的基础上可以考虑完整状态转移图法，基于普通的图解法，比图解法功能更强大，可以求解三桶分油问题的全部解及各种特殊要求的解；在形状上，它比后者更直观，更易于找出分油问题的最优解。完整状态转移图法使三桶分油问题得到了拓展，能够对解提出各种特殊要求，如最优解、最长解、经过某一油量状态的最优解或全部解、经过某一倒油操作的最优解或全部解、符合指定倒油次数的全部解等。可参考彭世康《完整状态转移图法求三桶分油问题全部解》一文。

6 参考文献

[1]张晨.基于图论和广度优先搜索算法的分酒问题一般解的研究[N].数字技术与应用.第 36 卷第 04 期.

[2]赵翌,韦健.关于分油问题的数学模型[N].桂林师范高等专科学校学报.第 18 卷第 4 期(总第 60 期).

[3]郭俊杰,毕双艳,雷冠丽.分油间题的网络最优化解法[N].长春邮电学院学报.1989 年第 7 卷第 1 期.

[4]严蔚敏,吴伟民.数据结构(C 语言版)[M].北京:清华大学出版社,2009.