|  |  |
| --- | --- |
| **数据结构与算法 实验报告** | |
| 第四次 | |
| 图片包含 标牌  已生成极高可信度的说明 | |
|  | |
|  | |
|  | |
| **姓名** | 代珉玥 |
| **班级** | 软件001班 |
| **学号** | 2205223077 |
| **电话** | 18585038226 |
| **Email** | 2040257842@qq.com |
| **日期** | 2020-12-02 |

目录

[实验1 1](#_Toc176)

[1. 题目 1](#_Toc32692)

[2. 数据结构设计 3](#_Toc27535)

[3. 算法设计 4](#_Toc21433)

[4. 主干代码说明 5](#_Toc9400)

[5. 运行结果展示 7](#_Toc12083)

[6. 总结和收获 8](#_Toc2445)

[7. 参考文献 8](#_Toc30350)

# 实验1

## 题目

### 背景

“Six Degrees of separation”是六度分割理论，也被称为小世界理论，其描述为:”你和任何一个陌生人之间所间隔的人不会超过五个，也就是说，最多通过五个人你就能够认识任何一个陌生人。”根据这个理论，你和世界上的任何一个人之间只隔着五个人，不管对方在哪个国家，属哪类人种，是哪种肤色。

基于上面的理论，有一款以好莱坞影星Kevin Bacon为主线的游戏，其名称为“Six Degrees of Kevin Bacon”。本次综合训练的主要任务就是实现这个游戏。

这款游戏以Kevin Bacon为中心，对于任何一个给定的演员，给出该演员和Kevin Bacon之间进行联系的“Bacon Number“。Bacon Number的定义规则为：

1. Kevin Bacon的 Bacon Number值为 0；
2. 和Kevin Bacon在一个电影里出现的所有演员的Bacon Number值为 1；
3. 任何演员的Bacon Number值为与该演员在同一个电影里的Bacon Number值最 小的那个演员的Bacon Number值加 1。

实践证明，几乎好莱坞的每一个演员都拥有一个小于6的Bacon Number。对于好莱坞明星来说，Bacon Number值一般最多到3，也就是最多通过三部电影就能找到和Kevin Bacon的联系。

关于该游戏的更详细的介绍，可以参看维基百科：

http://en.wikipedia.org/wiki/Six\_Degrees\_of\_Kevin\_Bacon 。

你也可以尝试通过在线方式玩该游戏，网址为：http://oracleofbacon.org/

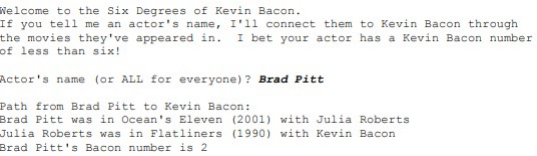
### 任务 1

建立为实现该游戏的图的抽象描述结构，包括图中顶点的意义以及存储的信息、边的意义以及存储的信息。并给出该图的逻辑示意图。

### **任务 2**

在任务1的基础上，并结合教材中图的抽象数据类型的定义，设计并实现一个为该游戏而使用的具体的Graph Class。

### 任务 3

通过给定的数据文件simple.txt构建图。Simple.txt中的格式为：每一行代表一个电影，每行中的信息都用’/’进行分割，其中第一个信息为电影名称，其后所有的信息都是出现在该电影的演员名。利用图和相应的算法，你可以根据用户输入的演员名，给出该演员的Bacon Number，并且列出该数计算的依据，也就是通过哪些电影建立了和 Kevin Bacon 的联系。运行的样例模式如下：

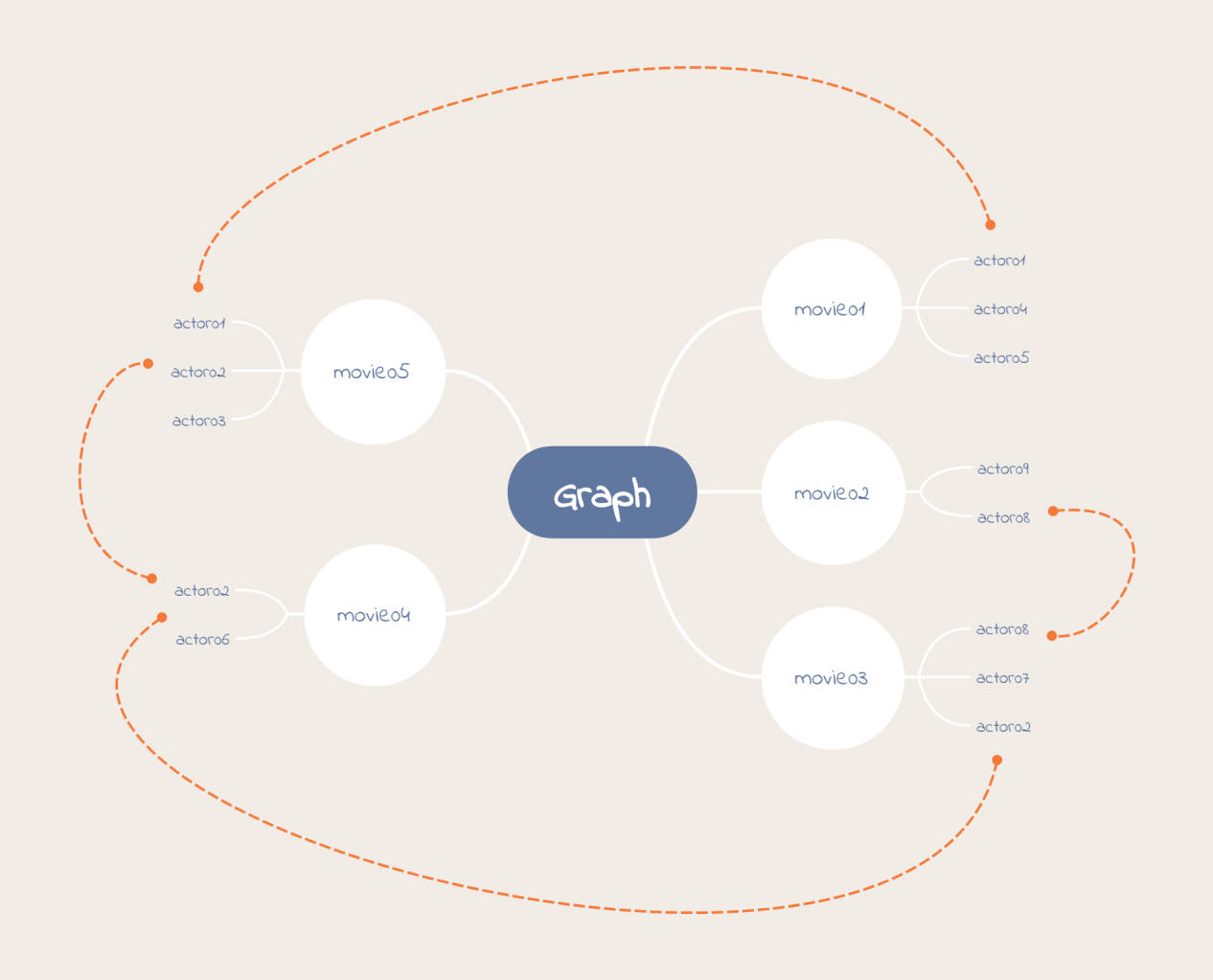
### **任务 4（选做）**

Complex.txt 是一个数据规模远超Simple.txt的文件，两者的格式是一样的。尝试使用任务 3 的解决方案执行Complex.txt。如果运行效果不理想，请分析问题的原因，并尝试进行改变。

## 数据结构设计

Vertex： movies

Edge: actors



public class Graph implements GraphInterface{  
 public Actor KB = new Actor("Kevin Bacon");   
 public Movie[] movies = new Movie[100000]; //An array that stores all movies  
 public MyHash allActors = new MyHash(); //A hash table that stores all actors

public int moviesNum; //The number of movies  
 public MyQueue01 queue = new MyQueue01(); //Queue  
 public boolean[] isout = new boolean[100000]; //If movies[i] is processed, isout[i] is true

}

public class Actor {  
 public Movie inMovie; //In the final path, which movie the actor was in.  
 public Actor relatedTo; //In the final path, who the actor was with.  
 public String name; //The actor’s name

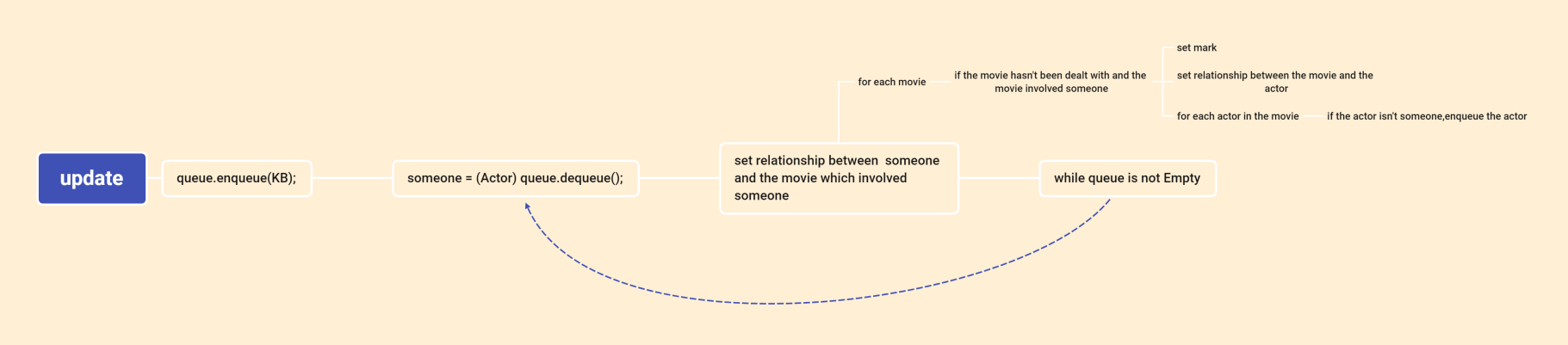
}

public class Movie {  
 public Actor[] actors = new Actor[1000]; //Actors who were in the movie  
 public int sumOfActor; //the number of actors in the movie  
 public String movieName; //the name of the movie

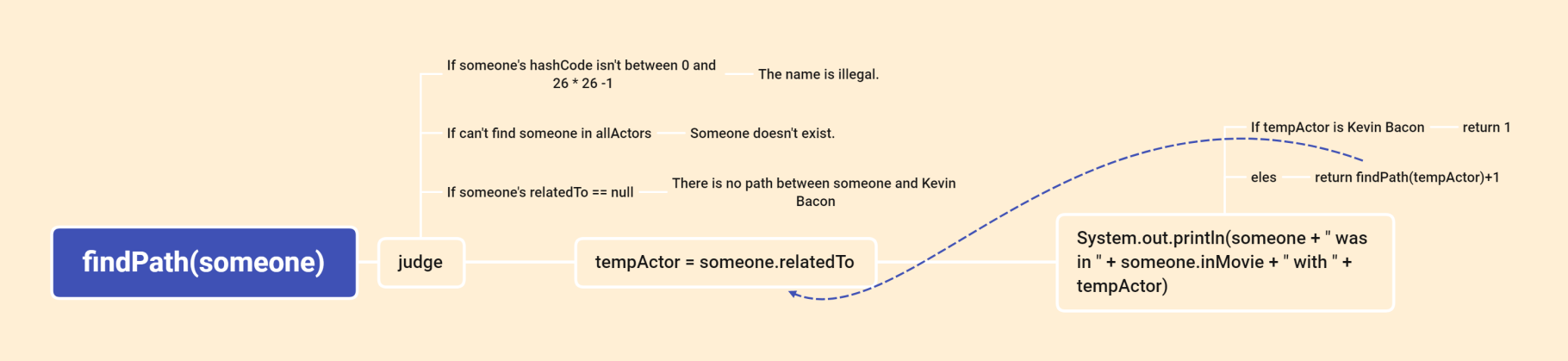
}

## 算法设计

### Update method



### Find path method



## 主干代码说明

### Update method

public void update(){  
 queue.enqueue(KB);  
 Actor someone = (Actor) queue.dequeue();  
 for (int i = 0; i < moviesNum; i++) {  
 if (!isout[i]&&movies[i].hasSomeone(someone)){  
 movies[i].setRelation(someone); //set relationship between the movie and the actor  
 isout[i] = true; //set mark  
 for (int j = 0; j < movies[i].sumOfActor; j++) {  
 if (!movies[i].actors[j].equals(someone)){  
 queue.enqueue(movies[i].actors[j]);  
 }  
 }  
 }  
 }  
 while (!queue.isEmpty()){  
 update01();  
 }  
}  
  
public void update01(){  
 Actor someone = (Actor) queue.dequeue();  
 for (int i = 0; i < moviesNum; i++) {  
 if (!isout[i]&&movies[i].hasSomeone(someone)){  
 movies[i].setRelation(someone); //set relationship between the movie and the actor  
 isout[i] = true; //set mark  
 for (int j = 0; j < movies[i].sumOfActor; j++) {  
 if (!movies[i].actors[j].equals(someone)){  
 queue.enqueue(movies[i].actors[j]);  
 }  
 }  
 }  
 }  
}

public void setRelation(Actor someone) {  
 for (int i = 0; i < sumOfActor; i++) {  
 if (!actors[i].equals(someone)){  
 actors[i].setRalationIn(this,someone); //set relationship between someone and everyone in the movie  
 }  
 }  
}

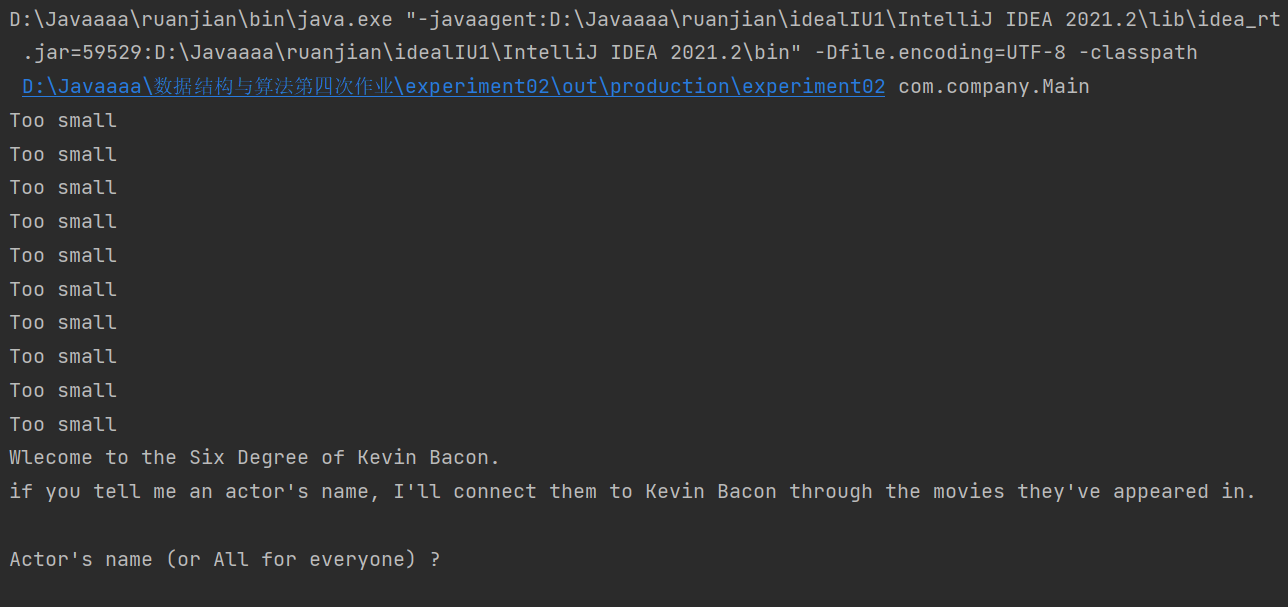
public void setRalationIn(Movie movie, Actor someone) {  
 inMovie = movie;  
 relatedTo = someone;  
}

### Find path method

public void findPath(String someone){  
 Actor temp = new Actor(someone);  
 if (temp.hashCode()<0||temp.hashCode()>26\*26-1) {  
 System.out.println("The name is illegal.");  
 }  
 if (temp.equals(allActors.find(temp))){ //someone exists   
 temp = allActors.find(temp);  
 if (temp.relatedTo!=null) {  
 System.out.println(someone + "'s Bacon number is "+findPath(temp));  
 }else {  
 System.out.println("No path");  
 }  
 } else {  
 System.out.println(someone+" doesn't exist.");  
 }  
}  
  
//return the Bacon Number  
public int findPath(Actor someone) {  
 Actor tempActor = someone.relatedTo;  
 if (KB.equals(tempActor)) {  
 System.out.println(someone + " was in " + someone.inMovie + " with " + tempActor);  
 return 1;  
 }else {  
 System.out.println(someone + " was in " + someone.inMovie + " with " + tempActor);  
 return findPath(tempActor)+1;  
 }  
}

## 运行结果展示

### 过程性结果



刚开始的哈希表用数组实现，在处理了相同的哈希值的情况后，依然容量不够，改用链表实现哈希表之后问题得以解决。

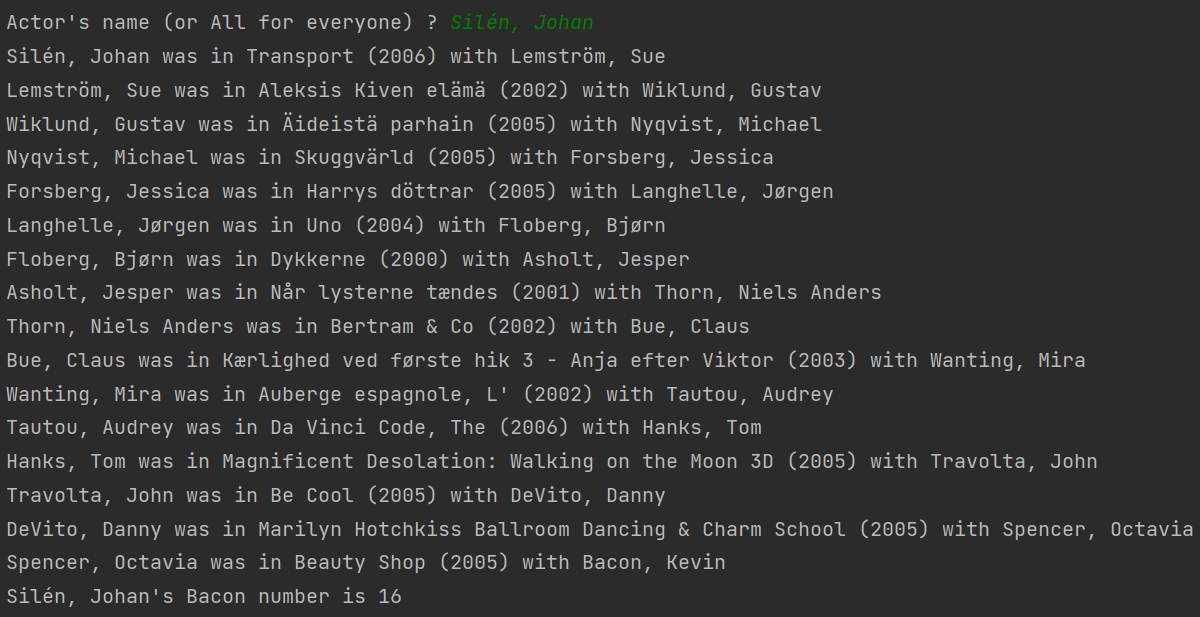
### 最终结果

Simple:



Complex:





## 总结和收获

通过此次实验，对于图的编写更加熟悉。

之前几次实验一开始就写结果写了很久之后发现有问题，于是这次先构思了很久才开始写。但是只想不写也缺乏感觉，下次可以以写接口的方式来构思，不具体实现方法，而是以抽象的方式进行架构。

在实验中，之前用到了用数组实现的循环队列，发现掌握有一点问题，太久没有写有些生疏，在运用不太熟练的类时，因先充分测试。

## 参考文献

无。