用图解决问题作业报告

第一次

姓名	史佳鑫
班级	软件 2302 班
学号	2236115195
电话	17342941319
Email	3168609153@qq. com
日期	2024-12-16

目录

任务 1 2
任务 2 4
任务 3 14
附录 31

任务 1

1、 题目 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。

2、 具体设计

1. 图中的顶点（节点）

意义: 图中的每个顶点表示一个有效单词。

存储的信息:

单词本身: 节点存储单词作为标识符。

单词属性（可选）: 例如单词的长度、词频或其他辅助信息。

2. 图中的边

意义: 边表示两个单词之间可以通过改变一个字母相互转换。

存储的信息:

权重（可选）: 边可以存储权重；否则权重默认为 1。

3. 图的性质

无向图: 两个单词之间的转换是双向的，因此边是无向的。

稠密性: 对于大量的单词集合（词典），图是稀疏的，因为并不是所有单词都与其他单词直接相连。

连通性: 在合理的词典下，该图通常是部分连通的，即一些单词之间无法通过任何路径连接。

图的逻辑示意图

以下是以一个简单的单词集合为例的示意图:

单词集合: ["cat", "bat", "bet", "bot", "bog", "dog"]

顶点集合 (V):

V = {cat, bat, bet, bot, bog, dog}

边集合 (E):

cat ↔ bat （改变第一个字母）

bat ↔ bet （改变最后一个字母）

bet ↔ bot （改变中间字母）

bot ↔ bog （改变最后一个字母）

bog ↔ dog （改变第一个字母）

图的示意图:

```
cat --- bat --- bet
                |           |
                bot --- bog --- dog
```

图的实现结构：邻接表表示

cat: [bat]

bat: [cat, bet]

bet: [bat, bot]

bot: [bet, bog]

bog: [bot, dog]

dog: [bog]

任务 2

1、 题目 在任务 1 的基础上，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

2、 数据设计

Edge 类数据设计

属性：

v1 (int): 边的起点顶点编号。

v2 (int): 边的终点顶点编号。

weight (int): 边的权重，表示边的“成本”或“距离”。

方法：

构造函数：接收起点、终点和权重作为参数，初始化边的实例。

Graph 类数据设计

属性：

adjList (Map<Integer, List<Edge>>): 邻接表，用于存储图的边信息。键是顶点编号，值是与该顶点相连的边的列表。

numVertices (int): 图中顶点的数量。

numEdges (int): 图中边的数量。

marks (int[]): 顶点的标记数组，用于在图的遍历或搜索算法中标记顶点的状态（如访问过、未访问等）。

wordToVertex (Map<String, Integer>): 单词到顶点编号的映射，用于快速查找单词对应的顶点编号。

vertexToWord (List<String>): 顶点编号到单词的映射，用于快速通过顶点编号获取对应的单词。

方法:

构造函数：初始化邻接表、顶点和边的数量、标记数组、单词到顶点编号的映射和顶点编号到单词的映射。

n()：返回图中顶点的数量。

e()：返回图中边的数量。

addWord(String word)：添加一个单词作为图中的新顶点，如果单词已存在，则不添加。

getVertex(String word)：根据单词获取对应的顶点编号。

getWord(int vertex)：根据顶点编号获取对应的单词。

first(int v)：获取与顶点 v 相连的第一条边。

next(int v, Edge edge)：获取与顶点 v 相连的、紧跟在给定边 edge 之后的下一条边。

isEdge(int i, int j)：检查顶点 i 和 j 之间是否存在边。

setEdge(int i, int j, int weight)：在顶点 i 和 j 之间添加一条边，如果边已存在，则不添加。

delEdge(int i, int j)：删除顶点 i 和 j 之间的边。

weight(int i, int j)：获取顶点 i 和 j 之间边的权重。

setMark(int v, int val)：设置顶点 v 的标记值。

getMark(int v)：获取顶点 v 的标记值。

addWordEdge(String word1, String word2)：如果两个单词仅相差一个字母，则在它们对应的顶点之间添加一条边。

differByOne(String word1, String word2)：检查两个单词是否仅相差一个字母。

printGraph()：打印图的邻接表。

printVertices()：打印顶点映射。

3、 算法设计

添加单词

addWord(String word)：如果单词不在图中，则添加一个新的顶点，并更新 wordToVertex 和 vertexToWord。

添加边

addWordEdge(String word1, String word2)：检查两个单词是否仅相差一个字母，如果是，则在对应的顶点之间添加一条权重为 1 的边。

边的查询和操作

isEdge(int i, int j)：检查两个顶点之间是否存在边。

setEdge(int i, int j, int weight)：在两个顶点之间添加一条边，如果边已存在则不添加。

delEdge(int i, int j)：删除两个顶点之间的边。

weight(int i, int j)：获取两个顶点之间边的权重。

辅助功能

differByOne(String word1, String word2)：检查两个单词是否仅相差一个字母。

4、 主干代码说明

```java
import java.util.*;

16 个用法
class Edge {
    1 个用法
    int v1, v2; // 起点和终点
    3 个用法
    int weight; // 权重

    2 个用法
    public Edge(int v1, int v2, int weight) {
        this.v1 = v1;
        this.v2 = v2;
        this.weight = weight;
    }
}

2 个用法
public class Graph {
    11 个用法
    private Map<Integer, List<Edge>> adjList; // 邻接表
    3 个用法
    private int numVertices; // 顶点数量
    4 个用法
    private int numEdges; // 边的数量
    3 个用法
    private int[] marks; // 顶点的标记
    6 个用法
    private Map<String, Integer> wordToVertex; // 单词到顶点编号映射
    4 个用法
    private List<String> vertexToWord; // 顶点编号到单词的映射

    // 构造函数
    1 个用法
    public Graph() {
        this.adjList = new HashMap<>();
        this.numVertices = 0;
        this.numEdges = 0;
        this.marks = new int[1000]; // 假设最多1000个顶点
```

```
        this.wordToVertex = new HashMap<>();
        this.vertexToWord = new ArrayList<>();
    }

    // 返回顶点数量
    1 个用法
    public int n() {
        return numVertices;
    }

    // 返回边的数量
    1 个用法
    public int e() {
        return numEdges;
    }

    // 添加单词到图中作为顶点
    2 个用法
    public int addWord(String word) {
        if (!wordToVertex.containsKey(word)) {
            int vertexId = numVertices++;
            wordToVertex.put(word, vertexId);
            vertexToWord.add(word);
            adjList.put(vertexId, new ArrayList<>()); // 初始化邻接表的列表
        }
        return wordToVertex.get(word);
    }

    // 获取单词对应的顶点编号
    2 个用法
    public int getVertex(String word) {
        return wordToVertex.getOrDefault(word,  defaultValue: -1);
    }

    // 获取顶点对应的单词
    2 个用法
    public String getWord(int vertex) {
        return vertex >= 0 && vertex < vertexToWord.size() ? vertexToWord.get(vertex) : null;
    }
```

```
    // 返回顶点v的第一条边
    0 个用法
    public Edge first(int v) {
        List<Edge> edges = adjList.get(v);
        return (edges != null && !edges.isEmpty()) ? edges.get(0) : null;
    }

    // 返回顶点v的下一条边
    0 个用法
    public Edge next(int v, Edge edge) {
        List<Edge> edges = adjList.get(v);
        if (edges != null) {
            int index = edges.indexOf(edge);
            if (index != -1 && index + 1 < edges.size()) {
                return edges.get(index + 1);
            }
        }
        return null;
    }

    // 判断是否有边
    2 个用法
    public boolean isEdge(int i, int j) {
        List<Edge> edges = adjList.get(i);
        if (edges != null) {
            for (Edge edge : edges) {
                if (edge.v2 == j) {
                    return true;
                }
            }
        }
        return false;
    }

    // 添加边
    1 个用法
    public void setEdge(int i, int j, int weight) {
        if (!isEdge(i, j)) {
```

```
    1 个用法
    public void setEdge(int i, int j, int weight) {
        if (!isEdge(i, j)) {
            adjList.get(i).add(new Edge(i, j, weight));
            adjList.get(j).add(new Edge(j, i, weight)); // 无向图
            numEdges++;
        }
    }

    // 删除边
    0 个用法
    public void delEdge(int i, int j) {
        List<Edge> edgesI = adjList.get(i);
        List<Edge> edgesJ = adjList.get(j);
        if (edgesI != null) {
            edgesI.removeIf(edge -> edge.v2 == j);
        }
        if (edgesJ != null) {
            edgesJ.removeIf(edge -> edge.v2 == i);
        }
        numEdges--;
    }

    // 获取边的权重
    0 个用法
    public int weight(int i, int j) {
        List<Edge> edges = adjList.get(i);
        if (edges != null) {
            for (Edge edge : edges) {
                if (edge.v2 == j) {
                    return edge.weight;
                }
            }
        }
        return 0;
    }

    // 设置顶点标记
    0 个用法
```

```
    0 个用法
    public void setMark(int v, int val) {
        marks[v] = val;
    }

    // 获取顶点标记
    0 个用法
    public int getMark(int v) {
        return marks[v];
    }

    // 添加单词之间的边（如果它们仅相差一个字母）
    1 个用法
    public void addWordEdge(String word1, String word2) {
        int v1 = addWord(word1);
        int v2 = addWord(word2);
        if (differByOne(word1, word2)) {
            setEdge(v1, v2,  weight: 1);
        }
    }

    // 检查两个单词是否仅相差一个字母
    1 个用法
    private boolean differByOne(String word1, String word2) {
        if (word1.length() != word2.length()) return false;
        int diffCount = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diffCount++;
                if (diffCount > 1) return false;
            }
        }
        return diffCount == 1;
    }

    // 打印图的邻接表
    1 个用法
```

```
// 打印图的邻接表
1 个用法
public void printGraph() {
    System.out.println("Adjacency List:");
    for (Map.Entry<Integer, List<Edge>> entry : adjList.entrySet()) {
        System.out.print(getWord(entry.getKey()) + ": ");
        for (Edge edge : entry.getValue()) {
            System.out.print(getWord(edge.v2) + "(" + edge.weight + ") ");
        }
        System.out.println();
    }
}

// 打印顶点映射
1 个用法
public void printVertices() {
    System.out.println("Vertices:");
    for (Map.Entry<String, Integer> entry : wordToVertex.entrySet()) {
        System.out.println(entry.getKey() + " -> " + entry.getValue());
    }
}
```

测试代码

```
public class WordLadderGame {
    public static void main(String[] args) {
        String[] words = {"cat", "bat", "bet", "bot", "bog", "dog"};
        Graph graph = new Graph();

        // 添加单词和边
        for (String word1 : words) {
            for (String word2 : words) {
                graph.addWordEdge(word1, word2);
            }
        }

        // 打印邻接表和顶点映射
        graph.printVertices();
        graph.printGraph();

        // 查询图的属性
        System.out.println("Number of vertices: " + graph.n());
        System.out.println("Number of edges: " + graph.e());
        System.out.println("Is there an edge between 'cat' and 'bat'? " +
                graph.isEdge(graph.getVertex( word: "cat"), graph.getVertex( word: "bat")));
    }
}
```

5、 运行结果展示

```
Vertices:
bet -> 2
bat -> 1
bot -> 3
cat -> 0
bog -> 4
dog -> 5
Adjacency List:
cat: bat(1)
bat: cat(1) bet(1) bot(1)
bet: bat(1) bot(1)
bot: bat(1) bet(1) bog(1)
bog: bot(1) dog(1)
dog: bog(1)
Number of vertices: 6
Number of edges: 6
Is there an edge between 'cat' and 'bat'? true

进程已结束，退出代码为 0
```

6、 总结和收获

通过实现这个 Graph Class，学习了如何表示和操作图结构。了解了图的基本概念，如顶点、边、邻接表等，并实现了图的基本操作，如添加顶点、添加边、查询边等。此外，还学习了如何使用 数据结构，如 Map 和 List，来存储图的数据。对图的抽象数据类型有了更深入的理解。

任务 3

1、 题目 该任务中会提供一个所有长度为 5 的单词列表文件 words5.txt，需要针对提供的这个单词列表解 决如下问题： ①针对 words5.txt 文件中的单词列表，生成一个 noladder.txt 文件，该文件中记录的单词是无 法和其他单词形成字梯的所有单词。

② 编写一个具有交互功能的程序，给用户随机抽两个单词（注：这两个单词必须要保证能够有 字梯链），接受用户的输入，判断用户的每次输入是否是正确的，直到用户失败或者成功。（如果可能，还可以增加判断用户的成功输入是否是最短的变化链路）

子任务一

1、 题目 针对 words5.txt 文件中的单词列表，生成一个 noladder.txt 文件，该文件中记录的单词是无法和其他单词形成字梯的所有单词。

2、 数据设计

数据结构

adjList (Map<String, List<String>>):

邻接表，用于存储图的边信息。键是单词，值是与该单词相差一个字母的单词列表。

wordList (List<String>):

存储从文件中读取的所有单词。

3、 算法设计

读取单词列表

readWords(String fileName):

读取文件中的单词，并将每个单词添加到 wordList 中。

构建邻接表

buildAdjacencyList():

遍历 wordList 中的每个单词，对于每个单词，检查 wordList 中的其他单词，如果两个单词相差一个字母，则将它们添加到对方的邻接列表中。

生成 noladder.txt 文件

generateNoLadderWords(String outputFile):

打开文件 noladder.txt 准备写入。

遍历 wordList，对于每个单词，如果其在 adjList 中的邻接列表为空（即没有与之相差一个字母的单词），则将该单词写入 noladder.txt 文件中。

检查两个单词是否仅相差一个字母

differByOne(String word1, String word2):

检查两个单词的长度是否相同。

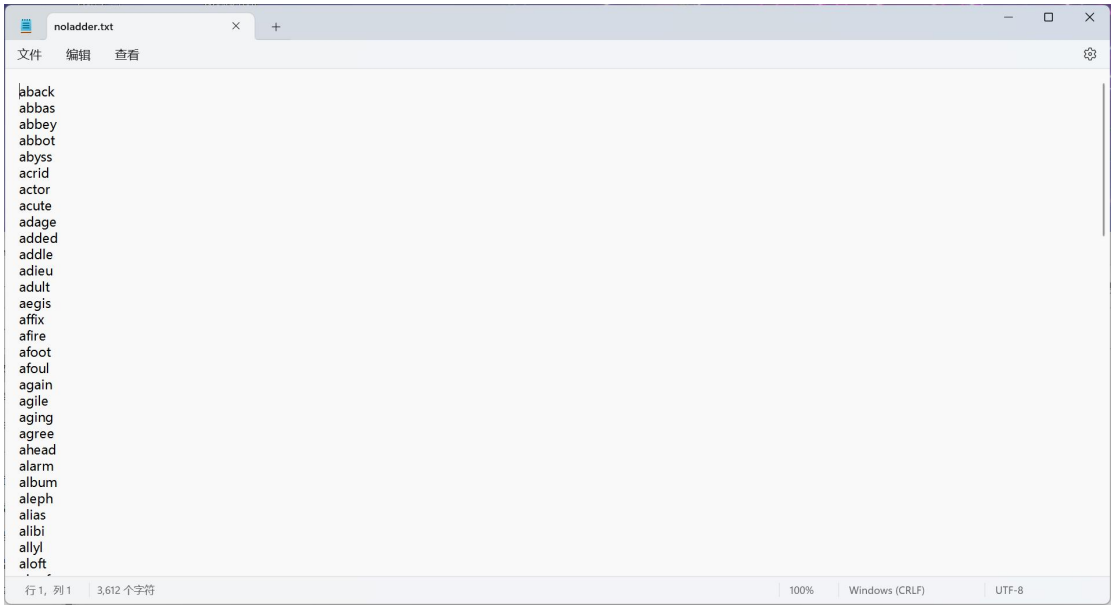
比较两个单词的每个字符，如果只有一个字符不同，则返回 true。

4、 主干代码说明

```java
import java.io.*;
import java.util.*;

public class WordLadderGenerator {

    // 图的邻接表
    3 个用法
    private static Map<String, List<String>> adjList = new HashMap<>();
    4 个用法
    private static List<String> wordList = new ArrayList<>();

    public static void main(String[] args) {
        // 给定的输入和输出文件名
        String inputFile = "words5.txt"; // 提供的单词列表文件
        String outputFile = "noladder.txt"; // 输出无字梯单词的文件

        try {
            // 1. 读取单词列表并生成邻接表
            readWords(inputFile);
            buildAdjacencyList();

            // 2. 生成无字梯单词文件
            generateNoLadderWords(outputFile);
            System.out.println("Generated noladder.txt with words that cannot form a word ladder.");

        } catch (IOException e) {
            System.out.println("An error occurred while processing the files: " + e.getMessage());
        }
    }

    // 读取单词列表
    1 个用法
    private static void readWords(String fileName) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = br.readLine()) != null) {
            wordList.add(line.trim());
        }
        br.close();
```

```
    1 个用法
    private static void buildAdjacencyList() {
        for (String word1 : wordList) {
            adjList.putIfAbsent(word1, new ArrayList<>());
            for (String word2 : wordList) {
                if (differByOne(word1, word2)) {
                    adjList.get(word1).add(word2);
                }
            }
        }
    }

    // 生成 noladder.txt 文件
    1 个用法
    private static void generateNoLadderWords(String outputFile) throws IOException {
        BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile));

        for (String word : wordList) {
            if (adjList.get(word).isEmpty()) { // 如果该单词没有邻居，则写入文件
                bw.write(word);
                bw.newLine();
            }
        }

        bw.close();
    }

    // 检查两个单词是否仅相差一个字母
    1 个用法
    private static boolean differByOne(String word1, String word2) {
        if (word1.length() != word2.length()) return false;
        int diffCount = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diffCount++;
                if (diffCount > 1) return false;
            }
        }
        return diffCount == 1;
```

5、 运行结果展示(具体结果在附录)



6、 总结和收获

我们学习了从文件中读取数据并将其存储在内存中，使用哈希表和列表来构建和操作图的邻接表。实现一个算法来检查两个单词是否仅相差一个字母，这是字梯问题的核心逻辑。识别无法形成字梯的单词，并将这些单词输出到文件中。这个练习提高对图论和算法的理解，还增强处理文件和数据结构的能力。通过实际编码，能够更好地理解如何将理论知识应用于解决实际问题。

子任务二

1、 题目 编写一个具有交互功能的程序，给用户随机抽两个单词（注：这两个单词必须要保证能够有字梯链），接受用户的输入，判断用户的每次输入是否是正确的，直到用户失败或者成功。（如果可能，还可以增加判断用户的成功输入是否是最短的变化链路）

2、 数据设计

adjList: 邻接表，用于存储单词之间的连接关系。键是单词，值是与该单词相差一个字母的所有单词列表。

wordList: 存储所有有效的单词。

方法及其作用:

readWords(String fileName): 从文件中读取单词列表，并添加到 wordList 中。

buildAdjacencyList(): 构建邻接表 adjList，通过比较 wordList 中的每个单词与其他所有单词，找出相差一个字母的单词并添加到邻接表中。

playGame(): 启动游戏，随机选择两个单词，并与用户进行交互，直到游戏结束。

differByOne(String word1, String word2): 检查两个单词是否只相差一个字母。

isPathExists(String start, String end): 使用广度优先搜索（BFS）算法判断两个单词之间是否存在路径。

findShortestPathLength(String start, String end): 使用 BFS 算法计算从 start 到 end 的最短路径长度。

getRandomWord(): 从 wordList 中随机选择一个单词。

getNextValidWord(String currentWord, String endWord): 找到与当前单词相差一个字母且离目标单词更近的有效单词。

3、 算法设计

BFS 算法：用于判断路径存在性和计算最短路径长度。通过层级遍历单词的邻接表，可以有效地找到从起点到终点的最短路径。

邻接表构建：通过双层循环比较 wordList 中的每个单词，构建出每个单词的邻接表，这是字梯游戏的核心数据结构。

4、 主干代码说明

```
import java.io.*;
import java.util.*;

public class WordLadderInteractive {

    5 个用法
    private static Map<String, List<String>> adjList = new HashMap<>();
    6 个用法
    private static List<String> wordList = new ArrayList<>();

    public static void main(String[] args) throws IOException {
        String inputFile = "words5.txt"; // 单词列表文件

        // 1. 读取单词列表并生成邻接表
        readWords(inputFile);
        buildAdjacencyList();

        // 2. 启动交互式字梯游戏
        playGame();
    }

    // 读取单词列表
    1 个用法
    private static void readWords(String fileName) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = br.readLine()) != null) {
            wordList.add(line.trim());
        }
        br.close();
    }

    // 构建邻接表
    1 个用法
    private static void buildAdjacencyList() {
        for (String word1 : wordList) {
            adjList.putIfAbsent(word1, new ArrayList<>());
            for (String word2 : wordList) {
                if (differByOne(word1, word2)) {
```

```
            adjList.putIfAbsent(word1, new ArrayList<>());
            for (String word2 : wordList) {
                if (differByOne(word1, word2)) {
                    adjList.get(word1).add(word2);
                }
            }
        }
    }

    // 启动交互式字梯游戏
    1 个用法
    private static void playGame() {
        Scanner scanner = new Scanner(System.in);

        // 随机选择两个单词
        String startWord, endWord;
        do {
            startWord = getRandomWord();
            endWord = getRandomWord();
        } while (!isPathExists(startWord, endWord)); // 确保两单词之间有路径

        System.out.println("字梯游戏，启动!");
        System.out.println("起始单词: " + startWord);
        System.out.println("结束单词: " + endWord);

        // 计算最短路径长度
        int shortestPathLength = findShortestPathLength(startWord, endWord);

        // 游戏交互
        String currentWord = startWord;
        int steps = 0;

        System.out.println("逐个改变字符以到达结束单词。");
        while (true) {
            System.out.print("你的输入: ");
            String userWord = scanner.nextLine().trim();

            // 检查用户输入是否有效
            if (!wordList.contains(userWord)) {
```

```
            System.out.println("输入无效！该单词不在字典中，请重新输入。");
            continue;
        }

        if (!differByOne(currentWord, userWord)) {
            // 给出正确的下一个单词
            String nextValidWord = getNextValidWord(currentWord, endWord);
            System.out.println("输入的单词与当前单词相差超过一个字母！下一个有效单词是：" + nextValidWord);
            continue;
        }

        currentWord = userWord;
        steps++;

        if (currentWord.equals(endWord)) {
            System.out.println("恭喜！你成功完成了字梯游戏，步数为 " + steps + " 步。");
            if (steps == shortestPathLength) {
                System.out.println("太棒了！你找到了最短路径！");
            } else {
                System.out.println("不过，最短路径是 " + shortestPathLength + " 步。");
            }
            break;
        }
    }

    scanner.close();
}

// 检查两个单词是否仅相差一个字母
2 个用法
private static boolean differByOne(String word1, String word2) {
    if (word1.length() != word2.length()) return false;
    int diffCount = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1.charAt(i) != word2.charAt(i)) {
            diffCount++;
            if (diffCount > 1) return false;
        }
    }
```

```
        return diffCount == 1;
    }

    // 判断是否存在从 start 到 end 的路径（BFS）
    1 个用法
    private static boolean isPathExists(String start, String end) {
        if (start.equals(end)) return true;

        Set<String> visited = new HashSet<>();
        Queue<String> queue = new LinkedList<>();
        queue.add(start);
        visited.add(start);

        while (!queue.isEmpty()) {
            String current = queue.poll();

            for (String neighbor : adjList.get(current)) {
                if (neighbor.equals(end)) {
                    return true;
                }
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(neighbor);
                }
            }
        }

        return false;
    }

    // 计算从 start 到 end 的最短路径长度（BFS）
    3 个用法
    private static int findShortestPathLength(String start, String end) {
        if (start.equals(end)) return 0;

        Set<String> visited = new HashSet<>();
        Queue<String> queue = new LinkedList<>();
        queue.add(start);
        visited.add(start);
```

```
        int level = 0;

        while (!queue.isEmpty()) {
            int size = queue.size();
            level++;

            for (int i = 0; i < size; i++) {
                String current = queue.poll();

                for (String neighbor : adjList.get(current)) {
                    if (neighbor.equals(end)) {
                        return level;
                    }
                    if (!visited.contains(neighbor)) {
                        visited.add(neighbor);
                        queue.add(neighbor);
                    }
                }
            }
        }

        return -1; // 无法到达
    }

    // 随机获取一个单词
    2 个用法
    private static String getRandomWord() {
        Random random = new Random();
        return wordList.get(random.nextInt(wordList.size()));
    }

    // 获取与当前单词相差一个字母并且接近目标单词的下一个有效单词
    1 个用法
    private static String getNextValidWord(String currentWord, String endWord) {
        // 遍历与当前单词相差一个字母的所有单词
        for (String neighbor : adjList.get(currentWord)) {
            // 返回与目标单词距离最短的那个单词
            if (findShortestPathLength(neighbor, endWord) < findShortestPathLength(currentWord, endWord)) {
                return neighbor;
            }
        }
        return currentWord; // 如果没有找到合适的单词，返回当前单词
    }
```

5、 运行结果展示

第一次是按照最短路径测试，最终会提示找到了最短路径极其步数；第二次是按照非最短路径测试，最终会提示成功但不是最短路径。我还设置了提示方法，如果用户输入错误则根据最短路径提示下一步的单词应该是什么。

第一次

```
Word Ladder Game!
Start Word: booty
End Word: ionic
Enter words one by one to reach the end word.
Your input: booty
输入的单词与当前单词相差超过一个字母！下一个有效单词是: booth
Your input: booth
Your input: booth
输入的单词与当前单词相差超过一个字母！下一个有效单词是: sooth
Your input: sooth
Your input: sooth
输入的单词与当前单词相差超过一个字母！下一个有效单词是: south
Your input: south
Your input: south
输入的单词与当前单词相差超过一个字母！下一个有效单词是: sough
Your input: sough
Your input: sough
输入的单词与当前单词相差超过一个字母！下一个有效单词是: rough
Your input: rough
Your input: rough
输入的单词与当前单词相差超过一个字母！下一个有效单词是: rouge
Your input: rouge
Your input: rouge
输入的单词与当前单词相差超过一个字母！下一个有效单词是: rouse
Your input: rouse
Your input: rouse
输入的单词与当前单词相差超过一个字母！下一个有效单词是: louse
Your input: louse
Your input: louse
输入的单词与当前单词相差超过一个字母！下一个有效单词是: loose
Your input: loose
Your input: loose
输入的单词与当前单词相差超过一个字母！下一个有效单词是: noose
Your input: noose
Your input: noose
输入的单词与当前单词相差超过一个字母！下一个有效单词是: noise
Your input: noise
```

```
Your input: canny
Your input: canny
输入的单词与当前单词相差超过一个字母！下一个有效单词是: canna
Your input: canna
Your input: canna
输入的单词与当前单词相差超过一个字母！下一个有效单词是: manna
Your input: manna
Your input: manna
输入的单词与当前单词相差超过一个字母！下一个有效单词是: mania
Your input: mania
Your input: mania
输入的单词与当前单词相差超过一个字母！下一个有效单词是: manic
Your input: manic
Your input: manic
输入的单词与当前单词相差超过一个字母！下一个有效单词是: monic
Your input: monic
Your input: monic
输入的单词与当前单词相差超过一个字母！下一个有效单词是: ionic
Your input: ionic
恭喜！你成功完成了字梯游戏，步数为 29 步。
太棒了！你找到了最短路径！

进程已结束，退出代码为 0
```

第二次

```
Word Ladder Game!
Start Word: coach
End Word: crazy
Enter words one by one to reach the end word.
Your input: coach
输入的单词与当前单词相差超过一个字母！下一个有效单词是: poach
Your input: poach
Your input: coach
Your input: paoch
输入无效！该单词不在字典中，请重新输入。
Your input: poach
Your input: poach
输入的单词与当前单词相差超过一个字母！下一个有效单词是: peach
Your input: peach
Your input: peach
输入的单词与当前单词相差超过一个字母！下一个有效单词是: peace
Your input: peace
Your input: peace
输入的单词与当前单词相差超过一个字母！下一个有效单词是: place
Your input: place
Your input: place
输入的单词与当前单词相差超过一个字母！下一个有效单词是: plane
Your input: plane
Your input: plane
输入的单词与当前单词相差超过一个字母！下一个有效单词是: plank
Your input: plank
Your input: plank
输入的单词与当前单词相差超过一个字母！下一个有效单词是: clank
Your input: clank
Your input: clank
输入的单词与当前单词相差超过一个字母！下一个有效单词是: crank
Your input: crank
Your input: crank
输入的单词与当前单词相差超过一个字母！下一个有效单词是: crane
Your input: crane
Your input: crane
输入的单词与当前单词相差超过一个字母！下一个有效单词是: craze
Your input:
```

```
输入的单词与当前单词相差超过一个字母！下一个有效单词是: poise
Your input: poise
Your input: poise
输入的单词与当前单词相差超过一个字母！下一个有效单词是: posse
Your input: posse
Your input: posse
输入的单词与当前单词相差超过一个字母！下一个有效单词是: passe
Your input: passe
Your input: passe
输入的单词与当前单词相差超过一个字母！下一个有效单词是: paste
Your input: paste
Your input: paste
输入的单词与当前单词相差超过一个字母！下一个有效单词是: pasty
Your input: pasty
Your input: pasty
输入的单词与当前单词相差超过一个字母！下一个有效单词是: party
Your input: party
Your input: party
输入的单词与当前单词相差超过一个字母！下一个有效单词是: parry
Your input: parry
Your input: pary
输入无效！该单词不在字典中，请重新输入。
Your input: parry
输入的单词与当前单词相差超过一个字母！下一个有效单词是: harry
Your input: harry
Your input: harry
输入的单词与当前单词相差超过一个字母！下一个有效单词是: hardy
Your input: hardy
Your input: hardy
输入的单词与当前单词相差超过一个字母！下一个有效单词是: handy
Your input: handy
Your input: handy
输入的单词与当前单词相差超过一个字母！下一个有效单词是: candy
Your input: candy
Your input: candy
输入的单词与当前单词相差超过一个字母！下一个有效单词是: canny
Your input: canny
Your input: canny
```

```
输入的单词与当前单词相差超过一个字母！下一个有效单词是: crane
Your input: crane
Your input: crane
输入的单词与当前单词相差超过一个字母！下一个有效单词是: craze
Your input: craze
Your input: crane
Your input: craze
Your input: crazy
恭喜！你成功完成了字梯游戏，步数为 15 步。
不过，最短路径是 11 步。

进程已结束，退出代码为 0
```

6、 总结和收获

通过这个项目，我们可以学习到：

如何从文件中读取数据并处理。

如何构建和使用邻接表来表示单词之间的关系。

BFS 算法在路径搜索问题中的应用。

如何设计一个交互式的程序，提高用户体验。

此外，代码中还包含了一些优化点，比如在 getNextValidWord 方法中，通过比较到达目标单词的最短路径长度来选择下一个单词，这样可以引导用户更接近目标单词，提高游戏的可玩性。

附录：每个题的源代码

任务二

```
import java.util.*;


class Edge {
    int v1, v2; //  起点和终点
    int weight; //  权重


    public Edge(int v1, int v2, int weight) {
        this.v1 = v1;
        this.v2 = v2;
        this.weight = weight;
    }
}


public class Graph {
    private Map<Integer, List<Edge>> adjList; //  邻接表
    private int numVertices; //  顶点数量
    private int numEdges; //  边的数量
    private int[] marks; //  顶点的标记
    private Map<String, Integer> wordToVertex; //  单词到顶点编号
```

映射

```
    private List<String> vertexToWord; // 顶点编号到单词的映射

    // 构造函数
    public Graph() {
        this.adjList = new HashMap<>();
        this.numVertices = 0;
        this.numEdges = 0;
        this.marks = new int[1000]; // 假设最多 1000 个顶点
        this.wordToVertex = new HashMap<>();
        this.vertexToWord = new ArrayList<>();
    }

    // 返回顶点数量
    public int n() {
        return numVertices;
    }

    // 返回边的数量
    public int e() {
        return numEdges;
    }
```

```
    // 添加单词到图中作为顶点
    public int addWord(String word) {
        if (!wordToVertex.containsKey(word)) {
            int vertexId = numVertices++;
            wordToVertex.put(word, vertexId);
            vertexToWord.add(word);
            adjList.put(vertexId, new ArrayList<>()); // 初始化邻接表的列表
        }
        return wordToVertex.get(word);
    }

    // 获取单词对应的顶点编号
    public int getVertex(String word) {
        return wordToVertex.getOrDefault(word, -1);
    }

    // 获取顶点对应的单词
    public String getWord(int vertex) {
        return vertex >= 0 && vertex < vertexToWord.size() ? vertexToWord.get(vertex) : null;
    }
```

```
    // 返回顶点 v 的第一条边
    public Edge first(int v) {
        List<Edge> edges = adjList.get(v);
        return (edges != null && !edges.isEmpty()) ? edges.get(0) :
null;
    }

    // 返回顶点 v 的下一条边
    public Edge next(int v, Edge edge) {
        List<Edge> edges = adjList.get(v);
        if (edges != null) {
            int index = edges.indexOf(edge);
            if (index != -1 && index + 1 < edges.size()) {
                return edges.get(index + 1);
            }
        }
        return null;
    }

    // 判断是否有边
    public boolean isEdge(int i, int j) {
```

```
        List<Edge> edges = adjList.get(i);
        if (edges != null) {
            for (Edge edge : edges) {
                if (edge.v2 == j) {
                    return true;
                }
            }
        }
        return false;
    }

    // 添加边
    public void setEdge(int i, int j, int weight) {
        if (!isEdge(i, j)) {
            adjList.get(i).add(new Edge(i, j, weight));
            adjList.get(j).add(new Edge(j, i, weight)); // 无向图
            numEdges++;
        }
    }

    // 删除边
    public void delEdge(int i, int j) {
```

```
        List<Edge> edgesI = adjList.get(i);
        List<Edge> edgesJ = adjList.get(j);
        if (edgesI != null) {
            edgesI.removeIf(edge -> edge.v2 == j);
        }
        if (edgesJ != null) {
            edgesJ.removeIf(edge -> edge.v2 == i);
        }
        numEdges--;
    }

    // 获取边的权重
    public int weight(int i, int j) {
        List<Edge> edges = adjList.get(i);
        if (edges != null) {
            for (Edge edge : edges) {
                if (edge.v2 == j) {
                    return edge.weight;
                }
            }
        }
        return 0;
```

```
    }

    // 设置顶点标记
    public void setMark(int v, int val) {
        marks[v] = val;
    }

    // 获取顶点标记
    public int getMark(int v) {
        return marks[v];
    }

    // 添加单词之间的边（如果它们仅相差一个字母）
    public void addWordEdge(String word1, String word2) {
        int v1 = addWord(word1);
        int v2 = addWord(word2);
        if (differByOne(word1, word2)) {
            setEdge(v1, v2, 1);
        }
    }

    // 检查两个单词是否仅相差一个字母
```

```
    private boolean differByOne(String word1, String word2) {
        if (word1.length() != word2.length()) return false;
        int diffCount = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diffCount++;
                if (diffCount > 1) return false;
            }
        }
        return diffCount == 1;
    }

    // 打印图的邻接表
    public void printGraph() {
        System.out.println("Adjacency List:");
        for (Map.Entry<Integer, List<Edge>> entry :
adjList.entrySet()) {
            System.out.print(getWord(entry.getKey()) + ": ");
            for (Edge edge : entry.getValue()) {
                System.out.print(getWord(edge.v2) + "(" +
edge.weight + ") ");
            }
```

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

        // 打印顶点映射
        public void printVertices() {
            System.out.println("Vertices:");
            for (Map.Entry<String, Integer> entry :
wordToVertex.entrySet()) {
                System.out.println(entry.getKey() + " -> " +
entry.getValue());
            }
        }
}
```

任务三子任务一

```
import java.io.*;
import java.util.*;

public class WordLadderGenerator {

    // 图的邻接表
```

```
    private static Map<String, List<String>> adjList = new
HashMap<>();
    private static List<String> wordList = new ArrayList<>();

    public static void main(String[] args) {
        // 给定的输入和输出文件名
        String inputFile = "words5.txt"; // 提供的单词列表文件
        String outputFile = "noladder.txt"; // 输出无字梯单词的文
件

        try {
            // 1. 读取单词列表并生成邻接表
            readWords(inputFile);
            buildAdjacencyList();

            // 2. 生成无字梯单词文件
            generateNoLadderWords(outputFile);
            System.out.println("Generated noladder.txt with words
that cannot form a word ladder.");

        } catch (IOException e) {
```

```
                System.out.println("An error occurred while processing
the files: " + e.getMessage());
            }
        }

        // 读取单词列表
        private static void readWords(String fileName) throws
IOException {
            BufferedReader br = new BufferedReader(new
FileReader(fileName));
            String line;
            while ((line = br.readLine()) != null) {
                wordList.add(line.trim());
            }
            br.close();
        }

        // 构建邻接表
        private static void buildAdjacencyList() {
            for (String word1 : wordList) {
                adjList.putIfAbsent(word1, new ArrayList<>());
                for (String word2 : wordList) {
```

```
                    if (differByOne(word1, word2)) {
                        adjList.get(word1).add(word2);
                    }
                }
            }
        }

        // 生成 noladder.txt 文件
        private static void generateNoLadderWords(String outputFile)
throws IOException {
            BufferedWriter bw = new BufferedWriter(new
FileWriter(outputFile));

            for (String word : wordList) {
                if (adjList.get(word).isEmpty()) { // 如果该单词没有邻居,
则写入文件
                    bw.write(word);
                    bw.newLine();
                }
            }

            bw.close();
```

```
    }

    // 检查两个单词是否仅相差一个字母
    private static boolean differByOne(String word1, String word2) {
        if (word1.length() != word2.length()) return false;
        int diffCount = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diffCount++;
                if (diffCount > 1) return false;
            }
        }
        return diffCount == 1;
    }
}
```

输出结果

```
aback
abbas
abbey
abbot
abyss
acrid
```

actor

acute

adage

added

addle

adieu

adult

aegis

affix

afire

afoot

afoul

again

agile

aging

agree

ahead

alarm

album

aleph

alias

alibi

allyl

aloft

aloof

amaze

amend

amity

angry

angst

annex

annoy

anvil

aorta

apart

aphid

apron

arena

argue

arhat

aroma

array

arrow

askew

asset

atlas

auric

avail

avoid

await

awful

axial

axiom

azure

bagel

balsa

banjo

batik

bayed

bayou

beaux

bebop

bedim

beefy

befit

befog

beige

below

beryl

bicep

bigot

biota

blimp

blitz

bongo

borax

boric

bowie

boyar

broil

brute

bugle

burro

buses

buteo

butte

butyl

buxom

buyer

bylaw

byway

cabin

cacao

cache

cacti

cagey

cairn

calla

canst

canto

cargo

caulk

cedar

chalk

chaos

chevy

chirp

churn

chute

cigar

cilia

circa

claim

clerk

cliff

cocoa

colza

comma

corps

coupe

coypu

credo

crude

cruel

csnet

cubic

culpa

cumin

cycad

cycle

datum

davit

dealt

degas

degum

delta

delve

depot

derby

devil

diary

dicta

digit

diode

dirge

disco

divan

dizzy

dogma

donor

doubt

drama

druid

ducat

dwarf

eagle

earth

easel

ebony

eclat

edify

eerie

egret

elbow

elegy

elfin

elves

emcee

empty

endow

enemy

entry

envoy

epoch

epoxy

equal

equip

erase

erode

error

erupt

estop

ethic

ethos

ethyl

evade

event

every

evoke

exert

exile

exist

extol

extra

facet

facto

faith

false

fancy

farad

fauna

fetid

fetus

filet

final

first

fjord

flora

fluid

forum

fovea

fraud

freon

froze

fugal

fugue

fungi

furze

gaffe

gamin

gamut

gassy

gator

gaudy

gauss

gawky

gecko

geese

genus

ghost

ghoul

gibby

gimpy

glory

glyph

gnash

gnome

gourd

grebe

gruff

guano

guard

gules

gutsy

gypsy

habit

haiku

halma

harem

havoc

hazel

helix

hello

henry

heron

hertz

hilum

hoagy

hogan

human

humid

humus

hyena

hymen

ideal

idyll

igloo

ileum

iliac

image

imbue

impel

inane

incur

index

infix

infra

ingot

inlay

input

inure

issue

ivory

jazzy

jiffy

jimmy

joust

julep

junco

junta

juror

kapok

karma

kazoo

keyed

khaki

kinky

kiosk

knead

knife

known

knurl

koala

kodak

kombu

kudzu

kulak

labia

laden

ladle

laity

lanky

lapse

larva

laugh

laura

layup

leggy

lemma

lewis

lilac

limbo

limit

livre

loath

lobar

loess

logic

lucky

lunar

lunge

lyric

madam

mambo

maple

matte

mauve

maxim

maybe

mayst

meant

melee

merit

metro

mezzo

midst

mimic

minim

mitre

mixup

modus

moire

monad

motif

motto

movie

mucus

mugho

multi

murky

murre

music

mynah

myrrh

nabla

nadir

naiad

naked

ninth

noble

nomad

nonce

nylon

obese

objet

occur

ocean

octal

octet

offal

offer

often

ohmic

olden

omega

opera

optic

orbit

order

organ

osier

ought

ouvre

ouzel

ovary

oxeye

oxide

ozone

pampa

panda

panel

pasha

pearl

pecan

penis

peril

petit

petri

pewee

phlox

photo

phyla

piano

picky

piggy

pinto

pious

pique

pixel

pizza

plasm

podia

poesy

polka

posey

posit

prior

prism

privy

proof

proud

psalm

psych

puffy

pygmy

quaff

quail

quake

qualm

query

queue

rabat

rabbi

radar

radon

rainy

raise

rajah

ranch

razor

realm

recur

relic

resin

rheum

rhino

rifle

rigid

rinse

risky

rival

robin

robot

rodeo

rupee

sabra

salad

samba

satyr

scion

scowl

scuba

seamy

sedan

seize

sepal

serif

setup

sheik

shoal

shoji

sibyl

sidle

sigma

sinew

siren

sisal

skimp

smith

snafu

soapy

sober

solid

somal

spasm

spawn

sperm

sprig

spume

stoic

study

sugar

sulfa

sumac

super

supra

sushi

swath

synod

syrup

taboo

tacit

talus

tawny

tenet

tepee

tepid

theft

their

theta

thigh

third

thorn

thrum

tibet

tibia

tidal

tiger

tilde

timid

toady

today

topaz

torso

torus

totem

truly

truth

tulip

turvy

tutor

tweak

tweed

twist

ulcer

ultra

umbra

unary

uncle

under

until

upper

upset

urban

usage

usher

usual

valet

velar

veldt

venom

vicar

video

vigil

vinyl

virus

visit

visor

vista

vitro

vixen

voice

vomit

waltz

weary

weber

weird

welsh

wharf

wheel

whiff

whirl

whoop

widen

widow

width

windy

wrath

xerox

xylem

yacht

young

yucca

zazen

zebra

zloty

任务三子任务二

```
import java.io.*;
import java.util.*;

public class WordLadderInteractive {

    private static Map<String, List<String>> adjList = new
HashMap<>();
    private static List<String> wordList = new ArrayList<>();

    public static void main(String[] args) throws IOException {
        String inputFile = "words5.txt"; //  单词列表文件

        // 1. 读取单词列表并生成邻接表
        readWords(inputFile);
        buildAdjacencyList();

        // 2. 启动交互式字梯游戏
        playGame();
```

```
    }

    // 读取单词列表
    private static void readWords(String fileName) throws
IOException {
        BufferedReader br = new BufferedReader(new
FileReader(fileName));
        String line;
        while ((line = br.readLine()) != null) {
            wordList.add(line.trim());
        }
        br.close();
    }

    // 构建邻接表
    private static void buildAdjacencyList() {
        for (String word1 : wordList) {
            adjList.putIfAbsent(word1, new ArrayList<>());
            for (String word2 : wordList) {
                if (differByOne(word1, word2)) {
                    adjList.get(word1).add(word2);
                }
```

```
                }
            }
        }

        // 启动交互式字梯游戏
        private static void playGame() {
            Scanner scanner = new Scanner(System.in);

            // 随机选择两个单词
            String startWord, endWord;
            do {
                startWord = getRandomWord();
                endWord = getRandomWord();
            } while (!isPathExists(startWord, endWord)); // 确保两单词之间有路径

            System.out.println("字梯游戏，启动!");
            System.out.println("起始单词: " + startWord);
            System.out.println("结束单词: " + endWord);

            // 计算最短路径长度
```

```
        int shortestPathLength = findShortestPathLength(startWord,
endWord);

        // 游戏交互
        String currentWord = startWord;
        int steps = 0;

        System.out.println("逐个改变字符以到达结束单词。");
        while (true) {
            System.out.print("你的输入: ");
            String userWord = scanner.nextLine().trim();

            // 检查用户输入是否有效
            if (!wordList.contains(userWord)) {
                System.out.println("输入无效！该单词不在字典中,
请重新输入。");
                continue;
            }

            if (!differByOne(currentWord, userWord)) {
                // 给出正确的下一个单词
```

```
                String nextValidWord =
getNextValidWord(currentWord, endWord);
                System.out.println("输入的单词与当前单词相差超
过一个字母！下一个有效单词是: " + nextValidWord);
                continue;
            }

            currentWord = userWord;
            steps++;

            if (currentWord.equals(endWord)) {
                System.out.println("恭喜！你成功完成了字梯游戏，
步数为 " + steps + " 步。");
                if (steps == shortestPathLength) {
                    System.out.println("太棒了！你找到了最短路
径！");
                } else {
                    System.out.println("不过，最短路径是 " +
shortestPathLength + " 步。");
                }
                break;
            }
```

```
        }

        scanner.close();
    }

    // 检查两个单词是否仅相差一个字母
    private static boolean differByOne(String word1, String word2) {
        if (word1.length() != word2.length()) return false;
        int diffCount = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diffCount++;
                if (diffCount > 1) return false;
            }
        }
        return diffCount == 1;
    }

    // 判断是否存在从 start 到 end 的路径（BFS）
    private static boolean isPathExists(String start, String end) {
        if (start.equals(end)) return true;
```

```
        Set<String> visited = new HashSet<>();
        Queue<String> queue = new LinkedList<>();
        queue.add(start);
        visited.add(start);

        while (!queue.isEmpty()) {
            String current = queue.poll();

            for (String neighbor : adjList.get(current)) {
                if (neighbor.equals(end)) {
                    return true;
                }
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(neighbor);
                }
            }
        }

        return false;
    }
```

```
    // 计算从 start 到 end 的最短路径长度（BFS）
    private static int findShortestPathLength(String start, String end)
{
        if (start.equals(end)) return 0;

        Set<String> visited = new HashSet<>();
        Queue<String> queue = new LinkedList<>();
        queue.add(start);
        visited.add(start);

        int level = 0;

        while (!queue.isEmpty()) {
            int size = queue.size();
            level++;

            for (int i = 0; i < size; i++) {
                String current = queue.poll();

                for (String neighbor : adjList.get(current)) {
                    if (neighbor.equals(end)) {
                        return level;
```

```
                        }
                        if (!visited.contains(neighbor)) {
                            visited.add(neighbor);
                            queue.add(neighbor);
                        }
                    }
                }
            }

            return -1; // 无法到达
        }

        // 随机获取一个单词
        private static String getRandomWord() {
            Random random = new Random();
            return wordList.get(random.nextInt(wordList.size()));
        }

        // 获取与当前单词相差一个字母并且接近目标单词的下一个有效单词
        private static String getNextValidWord(String currentWord, String endWord) {
```

```
            // 遍历与当前单词相差一个字母的所有单词
            for (String neighbor : adjList.get(currentWord)) {
                // 返回与目标单词距离最短的那个单词
                if (findShortestPathLength(neighbor, endWord) <
findShortestPathLength(currentWord, endWord)) {
                    return neighbor;
                }
            }
            return currentWord; // 如果没有找到合适的单词，返回当前
单词
        }
}
```