



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2024 年春季学期 计算学部《软件工程》课程

Lab 1 实验报告

姓名	班级/学号	联系方式
张智雄	2103601/2021112845	18328310316
韩晨烨	2103601/2021111247	13858456312

目 录

1	实验要求.....	1
1.1	结对编程.....	1
1.2	Git 实战	1
2	待求解问题描述.....	1
3	算法与数据结构设计.....	3
3.1	设计思路与算法流程图.....	3
3.2	数据结构设计.....	7
3.3	算法时间复杂度分析.....	8
4	实验与测试.....	8
4.1	读取文本文件并展示有向图.....	8
4.2	查询桥接词.....	11
4.3	根据桥接词生成新文本.....	11
4.4	计算最短路径.....	12
4.5	随机游走.....	14
5	编程语言与开发环境.....	15
6	结对编程.....	16
6.1	分组依据.....	16
6.2	角色切换与任务分工.....	16
6.3	工作照片.....	17
6.4	工作日志.....	17
7	Git 操作过程	19
7.1	实验场景(1): 仓库创建与提交	19
7.2	实验场景(2): 分支管理	23
8	在 IDE 中使用 Git Plugin	27
9	小结.....	29

1 实验要求

1.1 结对编程

- 练习结对编程，体验敏捷开发中的两人合作；
 - 两人一组，自由组合；
 - 使用一台计算机，共同编码，完成实验要求；
 - 在工作期间，两人的角色至少切换 6 次；
 - 选择一门支持面向对象的编程语言，推荐 Java
 - 提交一个.java 文件，其中至少包含以下函数：
 - `main(...)`: 主程序入口，接收用户输入文件，生成图，并允许用户选择后续各项功能；
 - `void showDirectedGraph(type G, ...)`: 展示有向图
 - `String queryBridgeWords(String word1, String word2)`: 查询桥接词
 - `String generateNewText(String inputText)`: 根据 bridge word 生成新文本
 - `String calcShortestPath(String word1, String word2)`: 计算两个单词间的最短路径
 - `String randomWalk()`: 随机游走
 - 除了 `main()` 之外，上述其他函数应尽可能保持与用户输入/系统输出的独立性（所有输入输出均应在 `main` 函数中完成；如果采用 GUI，则在 GUI 框架中完成）；
 - 不能改变函数的 specification（参数列表/类型、返回值类型、函数名）；
 - 例外 1: 函数 `void showDirectedGraph(type G,...)` 的输入参数 `G` 的类型 `type`，由开发者自行定义；可根据需要增加其他参数。
 - 例外 2: 函数 `main(String[] args)` 的输入参数个数与具体含义由开发者自定义。
 - 必要时可增加其他辅助函数，但须在实验报告中列清楚各函数的作用；
- 避免使用任何第三方 Java 外部算法库完成上述功能

1.2 Git 实战

- 熟练掌握 Git 的基本指令和分支管理指令；
- 掌握 Git 支持软件配置管理的核心机理；
- 在实践项目中使用 Git /Github 管理自己的项目源代码；
- 本部分实验由个人单独完成。

2 待解决问题描述

实验描述: 从文本生成图并在图上进行运算

输入数据:

1. 文本文件: 包含英文文本数据，文本文件将作为输入。
 - 文本中的换行符和回车符将被视为空格。
 - 文本中的任何标点符号也将被视为空格。

- 忽略所有非字母字符（仅保留 A-Z 和 a-z）。

输出数据：生成的有向图：根据输入的文本文件，生成一个有向图，并展示在屏幕上。

- 节点：文本中的每个单词（不区分大小写）。
- 边：若两个单词在文本中相邻出现，则在它们之间创建一条有向边。
- 边权重：边的权重为这两个单词相邻出现的次数。

功能需求

1. **生成有向图：**
 - 用户输入文本文件位置和文件名，或通过启动参数提供文件路径和文件名。
 - 程序读取文本文件并生成有向图。
2. **展示有向图：**
 - 以某种形式展示生成的有向图，可以是命令行输出或图形化界面显示。
 - 可选：将有向图保存为图形文件。
3. **查询桥接词（Bridge Words）：**
 - 用户输入任意两个单词，程序查询并返回它们之间的桥接词。
 - 桥接词：图中存在两条边 **word1 -> word3** 和 **word3 -> word2**。
 - 处理输入单词不在图中或没有桥接词的情况，并输出相应提示信息。
4. **生成新文本：**
 - 用户输入一行新文本，根据生成的图计算相邻单词的桥接词，将桥接词插入两个单词之间，并输出新文本。
 - 若无桥接词，则保持原样；若有多个桥接词，则随机选择一个。
5. **计算最短路径：**
 - 用户输入两个单词，程序计算并显示它们之间的最短路径（边权重之和最小），同时展示路径长度。
 - 若有多条最短路径，只需展示一条。
 - 处理不可达情况并提示。
6. **随机游走：**
 - 程序随机选择一个节点作为起点，进行随机遍历，记录经过的所有节点和边，直到出现重复边或无出边为止。
 - 输出遍历路径，并将结果写入文件。
 - 允许用户随时停止遍历。

约束条件

1. **语言选择：**必须使用支持面向对象的编程语言，推荐 Java。
2. **代码组织：**
 - 主函数 **main(...)** 作为程序入口，处理用户输入和输出，调用其他功能。
 - 其余函数尽可能保持与用户输入/系统输出的独立性。
 - 避免使用第三方 Java 外部算法库。
3. **函数规范：**
 - 保持实验手册指定的函数名、参数列表和返回类型。
 - 允许为 **showDirectedGraph** 函数定义自定义的图类型参数。
 - 允许为 **main(String[] args)** 函数定义自定义的输入参数个数和含义。
4. **附加功能：**实现可选功能可以获得附加分，例如图形化界面、保存图形文件、计算所有最短路径等。

提交要求

1. **代码文件：**提交 **.java** 文件，包含规定的函数和其他必要的辅助函数。

2. **实验报告**: 提交实验报告, 详细说明各函数的作用及实现过程。
3. **评审标准**: 结果的正确性、健壮性、算法执行时间、代码质量、结对编程配合度及可选功能支持程度。

3 算法与数据结构设计

3.1 设计思路与算法流程图

1. 文本生成图并展示

首先, 通过读取文本文件的内容, 并将其规范化处理 (如替换非字母字符、转换为小写、按空格分割成单词数组)。然后, 根据处理后的单词数组构建有向图, 对每对相邻单词添加节点和有向边。接着, 生成 DOT 文件来描述有向图的结构, 最后利用 Graphviz 将 DOT 文件转换为图像文件, 实现图的可视化展示。

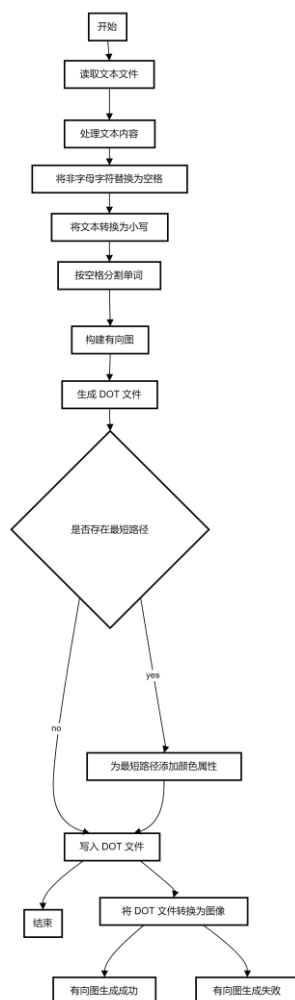


图 1 文本生成图流程图

2. 查询桥接词

通过 BFS 遍历了 start 的所有直接邻居, 并检查这些邻居是否有直接路径到 end, 从而找到所有可能的桥接词。BFS 的特性保证了找到的桥接词是从 start 出发最

早可以到达 end 的中间节点。

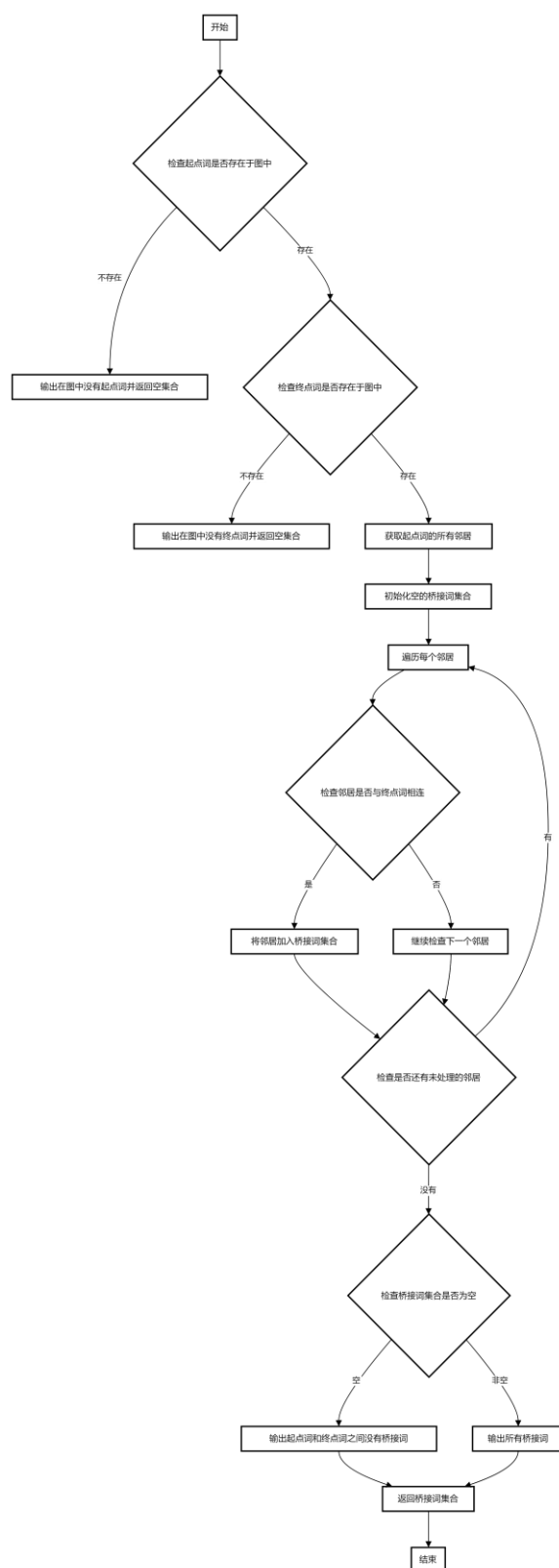


图2 桥接词查询流程图

3. 根据桥接词生成新文本

首先，将输入文本按空格分割成单词数组，然后初始化一个用于构建新文本的 `StringBuilder` 和一个随机数生成器 `Random`。接着，遍历单词数组，对于每对相邻

单词，先将当前单词添加到新文本中，然后调用 `queryBridgeWords` 方法查询当前单词和下一个单词之间的桥接词。如果存在桥接词，则随机选择一个桥接词并插入到新文本中。遍历结束后，将最后一个单词添加到新文本中，最后打印并返回生成的新文本。

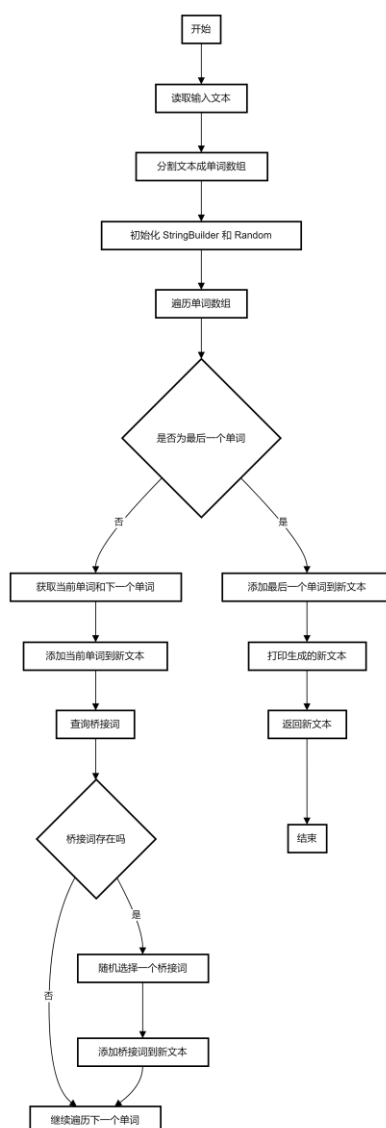


图 3 根据桥接词生成新文本流程图

4. 计算最短路径

首先，初始化一个路径列表和一个已访问节点的集合，通过调用 `findAllPaths` 方法获取所有从起始单词到目标单词的路径。如果没有路径，返回 `null`。然后，找到所有路径中最短的路径长度，并筛选出所有最短路径。接着，生成一个以当前计数器 `i` 作为后缀的 DOT 文件路径和图像文件路径，通过调用 `createDotFile` 方法生成标记最短路径的 DOT 文件，再调用 `convertDotToImage` 方法将 DOT 文件转换为图像文件，最后返回所有最短路径。

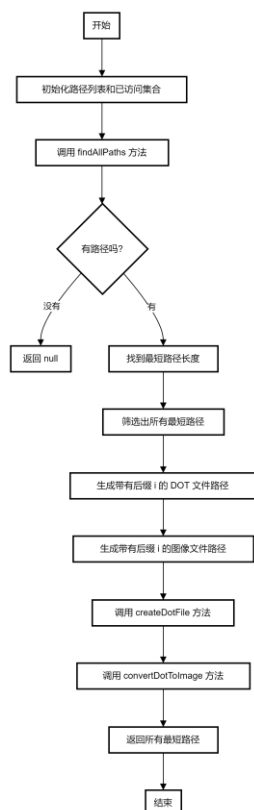


图 4 计算最短路径流程图

5. 随机游走

该算法通过在有向图中进行随机游走，从一个随机起点开始，不断随机选择下一步直到无法继续为止。游走过程中记录访问过的节点和边，最终将访问过的节点记录输出为文本文件。具体步骤包括选择随机起点、初始化记录结构、在图中随机游走（模拟延迟以便于演示）、记录节点和边、将结果写入文本文件。

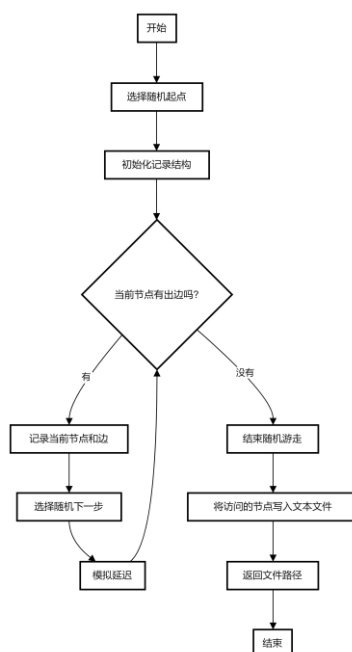


图 5 随机游走流程图

3.2 数据结构设计

1. 有向图数据结构

- `Map<String, Map<String, Integer>> directedGraph`
 - 定义: 一个嵌套的哈希映射结构, 外层 `Map` 的键是顶点, 值是另一个 `Map`。内层 `Map` 的键是与外层顶点相邻的顶点, 值是两个顶点之间的边的权重。
 - 用途: 用于存储有向图, 其中每个顶点都映射到一个包含其出边信息的 `Map`。内层的 `Map` 存储了每个相邻顶点以及它们之间的边的权重。

2. 程序控制数据结构

- `volatile boolean stopWalk`
 - 定义: 一个 `volatile` 关键字修饰的布尔变量。
 - 用途: 用于标记是否停止随机游走。使用 `volatile` 可以确保多线程环境中对该变量的修改能及时被其他线程看到。
- `CountDownLatch latch`
 - 定义: 一个 `CountDownLatch` 对象。
 - 用途: 用于协调主线程和随机游走线程之间的同步。`CountDownLatch` 可以使一个线程等待其他线程完成各自的工作后再继续执行。

3. 方法中的局部数据结构

- `List<String> path`
 - 定义: 一个 `List`, 保存当前路径中的顶点。
 - 用途: 在深度优先搜索 (DFS) 中用于记录当前路径。
- `Set<String> visited`
 - 定义: 一个 `Set`, 用于记录已访问的顶点。
 - 用途: 在 DFS 和广度优先搜索 (BFS) 中用于避免重复访问顶点。
- `Queue<String> queue`
 - 定义: 一个 `Queue`, 用于广度优先搜索。
 - 用途: 用于在 BFS 中存储待访问的顶点。
- `StringBuilder content`
 - 定义: 一个 `StringBuilder` 对象。
 - 用途: 用于构建从文件中读取的文本内容。
- `List<List<String>> allPaths`
 - 定义: 一个 `List`, 包含多个路径, 每个路径又是一个 `List`。
 - 用途: 用于存储从一个顶点到另一个顶点的所有路径。
- `List<List<String>> shortestPaths`
 - 定义: 一个 `List`, 包含多个最短路径, 每个路径又是一个 `List`。
 - 用途: 用于存储从一个顶点到另一个顶点的所有最短路径。

4. 其他辅助数据结构

- `Random random`
 - 定义: 一个 `Random` 对象。
 - 用途: 用于生成随机数, 例如在随机游走中选择下一个顶点。
- `String[] words`
 - 定义: 一个字符串数组。
 - 用途: 用于存储从文本文件中读取并处理后的所有单词。

3.3 算法时间复杂度分析

1. 根据文本生成图
 - 时间复杂度: $O(n)$
 - 原因: 读取文件和处理文本的时间复杂度为 $O(n)$, 遍历单词数组构建有向图的时间复杂度为 $O(n)$ 。
2. 展示图
 - 时间复杂度: $O(E)$
 - 原因: 生成 DOT 文件的时间复杂度为 $O(E)$, 将 DOT 文件转换为图像文件的时间复杂度可以近似看作 $O(1)$ 。
3. 查询桥接词
 - 时间复杂度: $O(E)$
 - 原因: 广度优先搜索 (BFS) 的时间复杂度为 $O(E)$, 因为每个顶点和每条边都会被访问一次。
4. 根据桥接词生成新文本
 - 时间复杂度: $O(k \cdot E)$
 - 原因: 对于输入文本中的每对相邻单词, 调用一次桥接词查询, 时间复杂度为 $O(E)$, 输入文本中的单词对数为 k 。
5. 计算最短路径
 - 时间复杂度: $O(E \cdot P)$
 - 原因: 查找所有路径在最坏情况下的时间复杂度为 $O(E \cdot P)$, 其中 P 是从起点到终点的所有路径数。
6. 随机游走
 - 时间复杂度: $O(E)$
 - 原因: 在最坏情况下, 如果图是一个强连通图且随机游走不停, 时间复杂度为 $O(E)$, 因为每条边可能会被访问一次。

4 实验与测试

设计 1 个至少包含 50 个单词的输入文本文件, 使之可覆盖本题目中关于输入文件和功能的各种特殊情况, 作为你开发的程序的输入。

针对在有向图上操作的每项功能, 为其设计各种可能的输入数据。输入数据的数量不限, 以测试程序的充分性为评判标准 (下面各节中的表格的行数请自行扩展)。

记录程序的输出结果, 判断输出结果是否与期望一致, 并记录程序运行截图。

4.1 读取文本文件并展示有向图

完成可选功能: 将生成的有向图以图形文件形式保存到磁盘

文本文件中包含的内容:

As the sun dipped below the horizon, casting a warm glow across the serene landscape, the weary travelers made their way through the winding paths of the ancient city, marveling at the intricate architecture that stood as a testament to centuries of history and culture, while the distant sound of laughter and music floated on the evening breeze, filling the air with a sense of joy and anticipation for the adventures that awaited them. The sun dipped below the horizon, casting a warm glow across the serene landscape

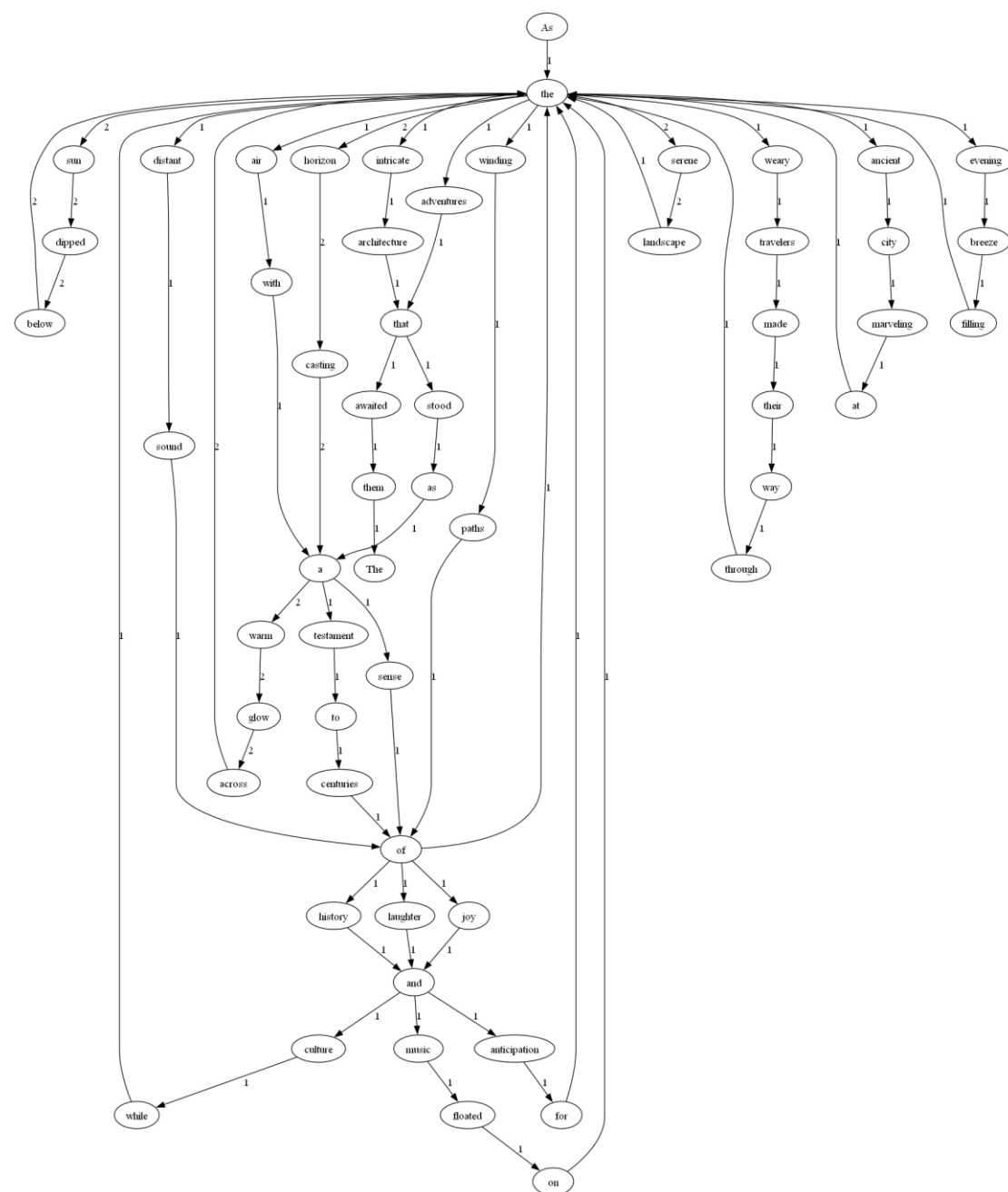
期望生成的图（手工计算得到）:

图 6 期望生成的图

程序实际生成的图:

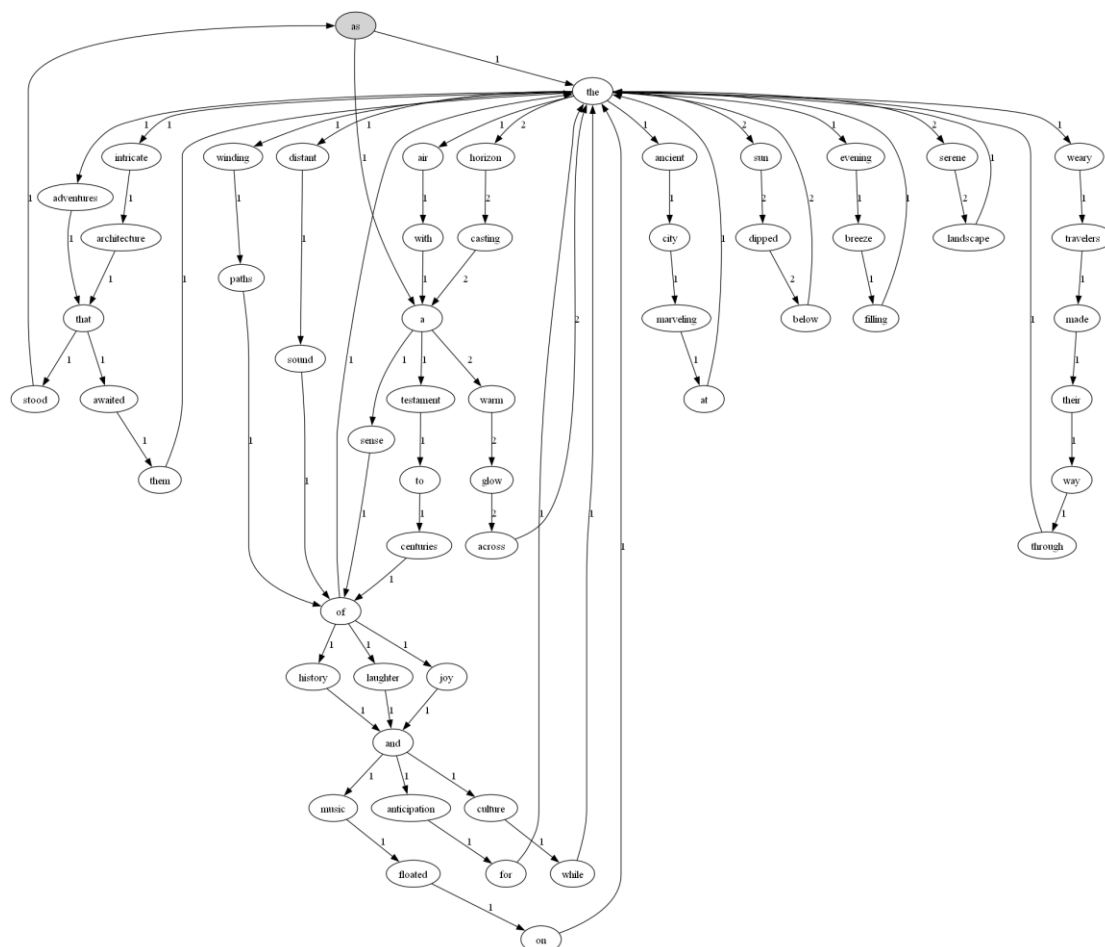


图 7 程序实际生成的图

二者是否一致:

虽然两张图中各个节点的位置可能不同,但是仔细分析每对节点可以发现两者是一致的。

给出实际运行得到结果的界面截图。

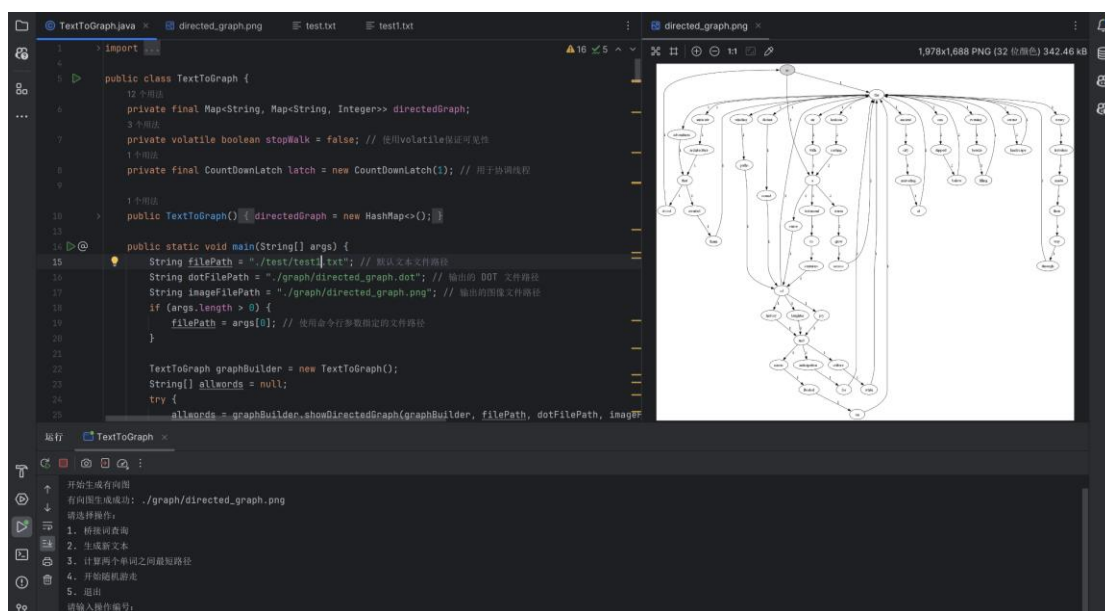


图 8 实际运行得到结果的界面截图

4.2 查询桥接词

序号	输入（2 个单词）	期望输出	实际输出	运行是否正确
1	distant of	桥接词为：sound	桥接词为：sound	正确
2	air horizon	air 和 horizon 之间没有桥接词	air 和 horizon 之间没有桥接词	正确
3	sun below	桥接词为：dipped	桥接词为：dipped	正确

给出实际运行得到结果的界面截图。

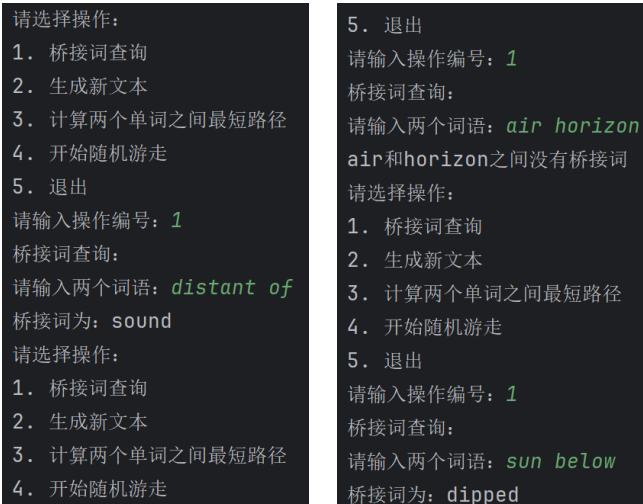
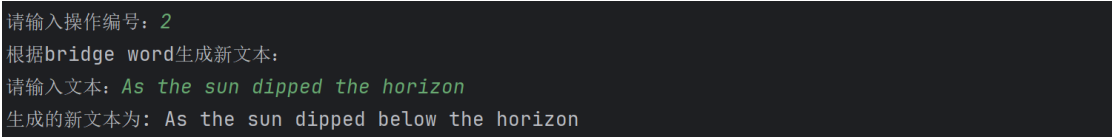


图 9 实际运行得到结果的界面

4.3 根据桥接词生成新文本

序号	输入（一行文本）	期望输出	实际输出	运行是否正确
1	As the sun dipped the horizon	As the sun dipped below the horizon	As the sun dipped below the horizon	正确
2	casting a warm across the landscape	casting a warm glow across the serene landscape	casting a warm glow across the serene landscape	正确
3	the travelers made their way the winding paths of the city	the weary travelers made their way through the winding paths of the ancient city	the weary travelers made their way through the winding paths of the ancient city	正确

给出实际运行得到结果的界面截图。



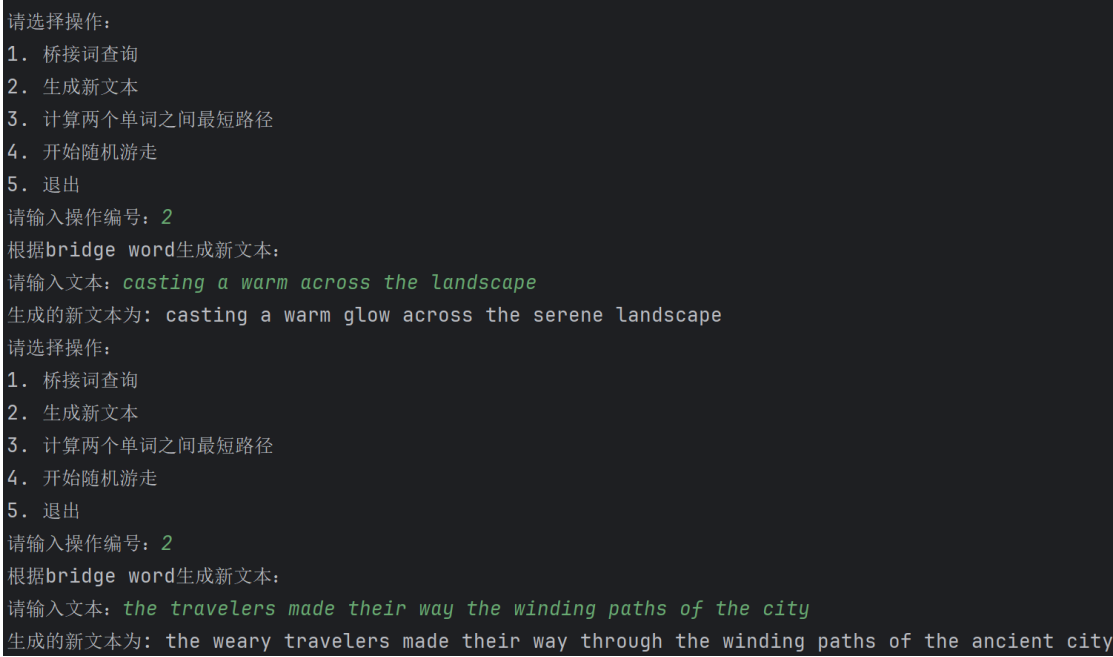


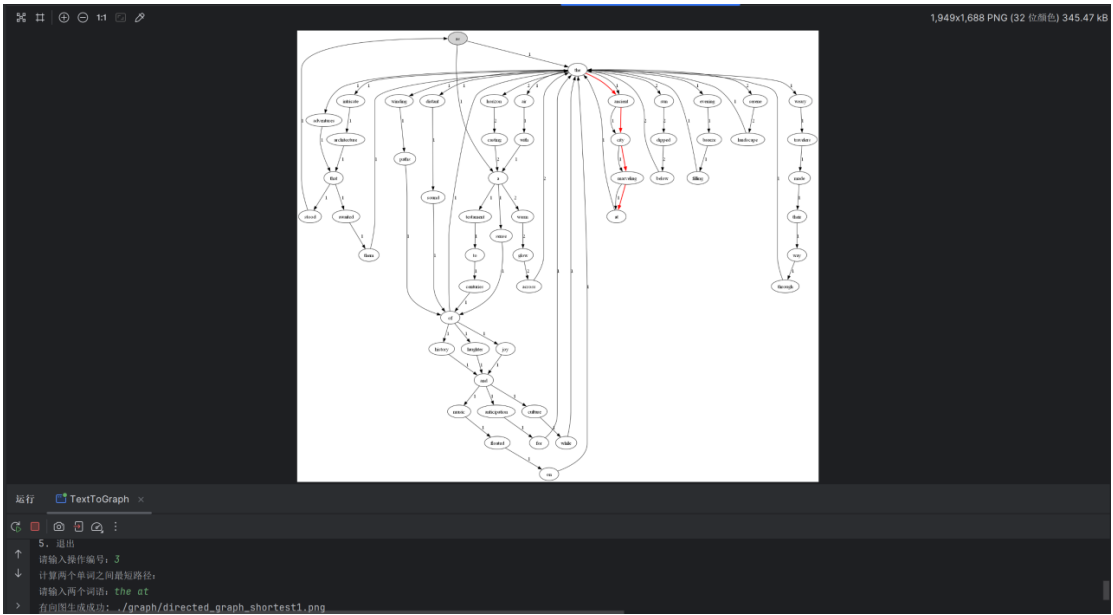
图 10 实际运行得到结果的界面

4.4 计算最短路径

完成可选功能：计算出所有最短路径，并以不同的突出显示方式展示出来。

序号	输入（两个单词、或一个单词）	期望输出	实际输出	运行是否正确
1	the at	一条长度为 4 的路径	一条长度为 4 的路径	正确
2	the of	两条长度为 3 的路径	两条长度为 3 的路径	正确
3	with laughter	一条长度为 4 的路径	一条长度为 4 的路径	正确

给出实际运行得到结果的界面截图。



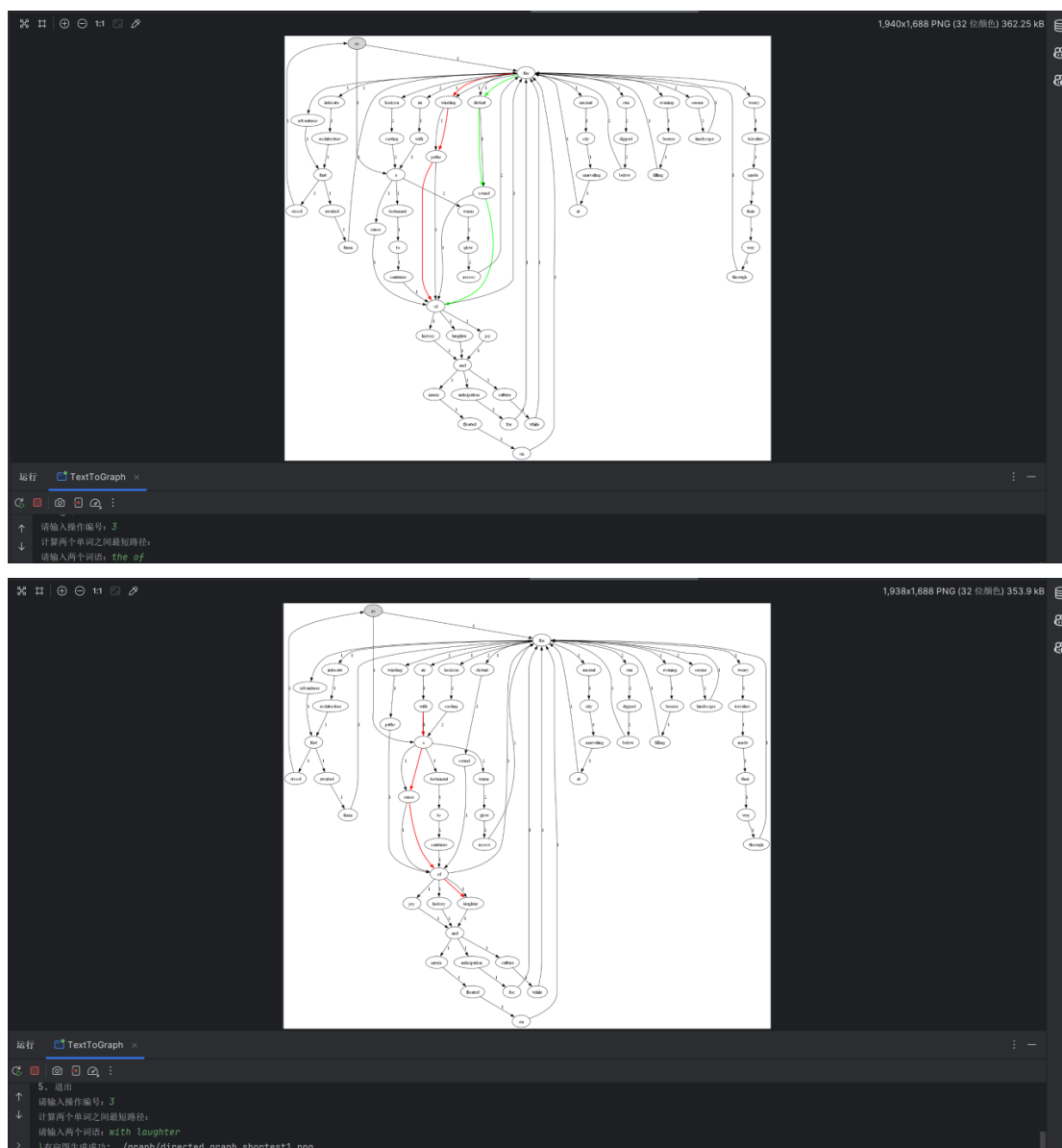


图 11 运行得到结果

完成可选功能：如果用户只输入一个单词，则程序计算出该单词到图中其他任一单词的最短路径，并逐项展示出来。

以单词 the 为例，输出其到图中其他任一单词的最短路径如下：

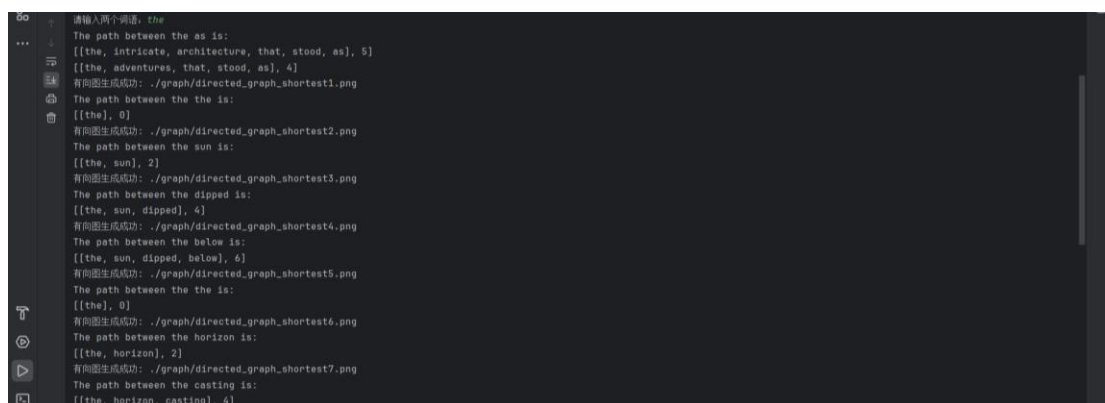


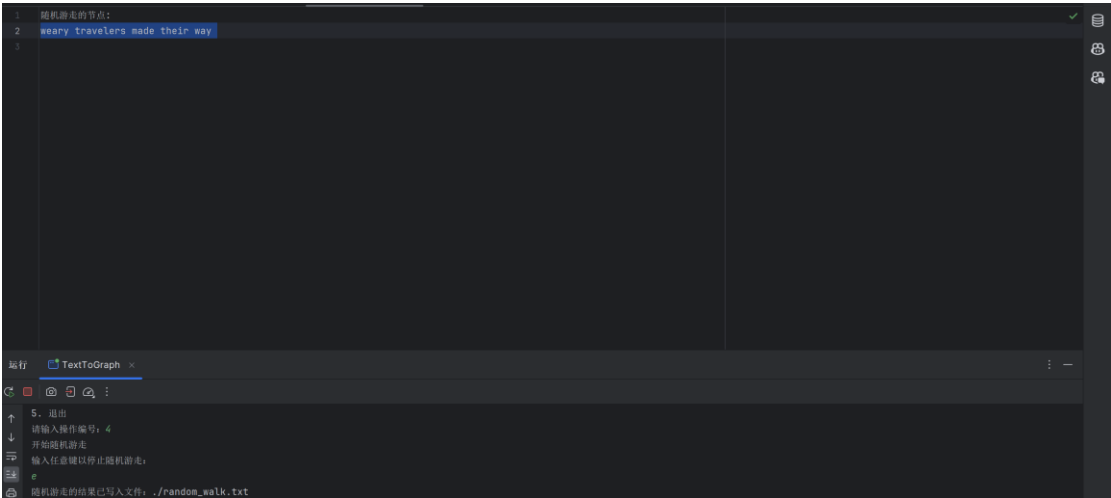
图 12 运行得到结果

4.5 随机游走

该功能无输入，让你的程序执行多次，分别记录结果。

序号	实际输出	程序运行是否正确
1	随机游走的节点: weary travelers made their way	正确
2	随机游走的节点: dipped below the ancient city marveling at the evening breeze filling the distant sound	正确
3	随机游走的节点: joy and anticipation for the intricate architecture that awaited them the winding paths of history and music floated on the winding paths of joy and music floated on the ancient city marveling at the horizon casting a sense of the winding paths of joy and culture while the distant sound of history and anticipation for the winding paths of laughter and music floated on the horizon casting a testament to centuries of history and music floated on the evening breeze filling the weary travelers made their way through the serene landscape the sun dipped below the weary travelers made their way through the ancient city marveling at the air with a warm glow across the weary travelers made their way through the adventures that stood as a testament to centuries of laughter and anticipation for the sun dipped below the serene landscape the adventures that stood as a sense of the winding paths of joy and anticipation for the evening breeze filling the serene landscape the serene landscape	正确

给出实际运行得到结果的界面截图。



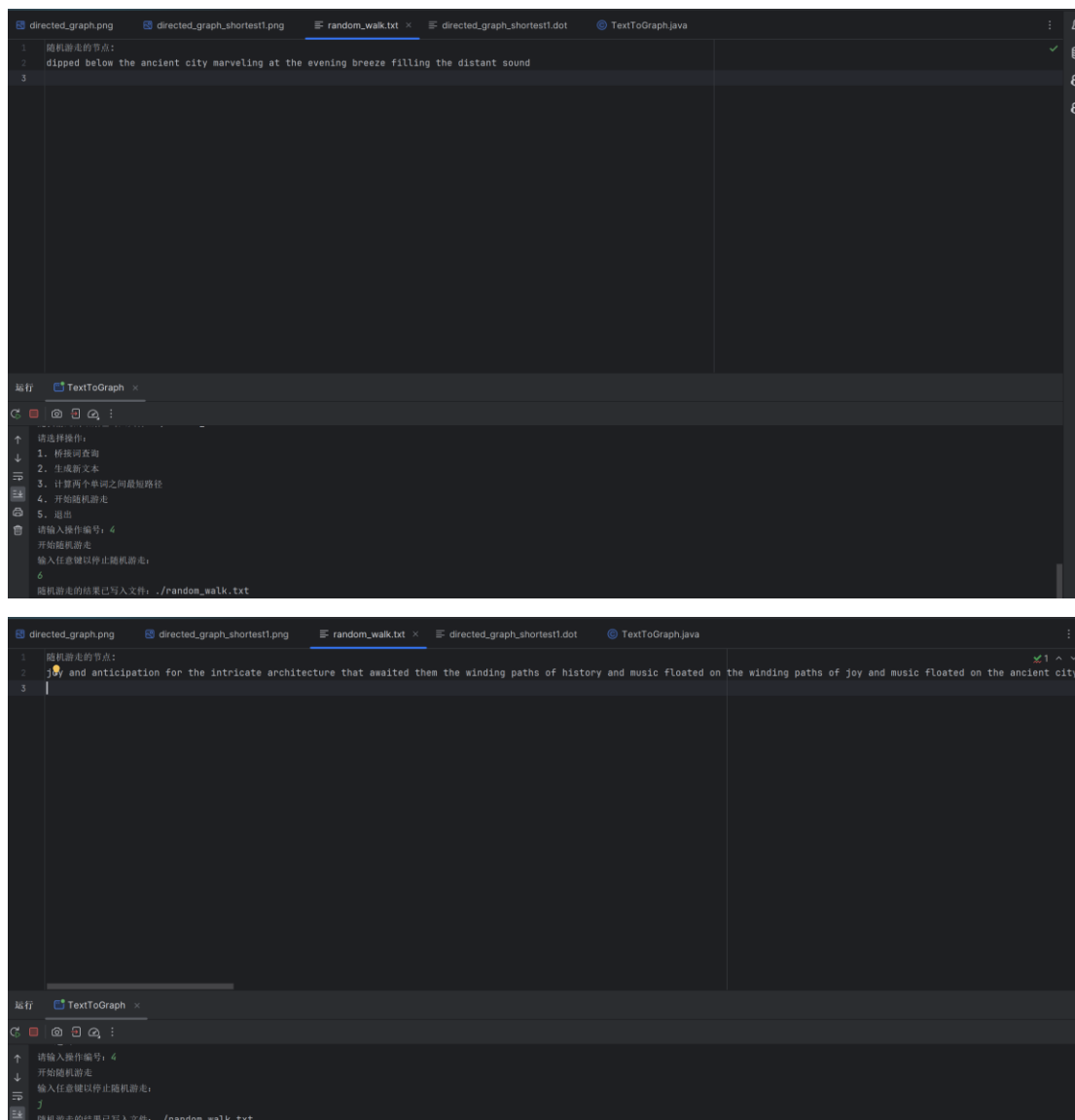


图 13 运行得到的结果

5 编程语言与开发环境

Java DK 版本: Java DK8

IDE 版本: IntelliJ IDEA 2024.1.1 (Community Edition)

Graphviz version: 11.0.0

6 结对编程

6.1 分组依据

1. 性格：
- **韩晨烨：**韩晨烨性格沉稳细致，善于思考和分析问题，喜欢独立解决复杂的问题。他在团队中通常扮演策划者和分析师的角色，能够仔细考虑每个步骤的细节，确保每个环节都能顺利进行。
 - **张智雄：**张智雄性格外向活跃，擅长沟通和团队协作，具有较强的执行力和创新能力。他在团队中通常扮演执行者和推动者的角色，能够快速响应问题，提出创造性的解决方案，并带动团队士气。
2. 能力：
- **韩晨烨：**韩晨烨有较强的逻辑思维能力和问题解决能力，擅长处理复杂的算法和数据结构。他注重代码的质量和可维护性，能够编写高效、可靠的代码。
 - **张智雄：**张智雄具备较强的项目管理和团队协调能力，能够有效地组织和分配任务，确保项目按时完成。他有丰富的实践经验，能应对实际开发过程中遇到的各种挑战。
3. 编程技能：
- **韩晨烨：**韩晨烨擅长后端开发，特别是在算法和数据结构方面有深入的研究。他对复杂系统的设计和实现有丰富的经验，能够编写高效、可靠的代码。他擅长进行代码优化和性能调优，确保系统的高效运行。
 - **张智雄：**张智雄擅长文本处理、文件操作和多线程编程。他在处理大规模数据和复杂逻辑方面有丰富的经验，能够高效地实现和优化各种功能模块。
4. 结对编程的优势：
- **互补性：**韩晨烨和张智雄在性格和能力上形成了互补，韩晨烨的沉稳细致和张智雄的活跃创新相得益彰，使得团队在遇到问题时能够既有深度分析又有快速响应。
 - **全面技能覆盖：**两人共同拥有 Java 开发的全面技能，韩晨烨专注于后端逻辑处理，张智雄专注于文本处理和多线程编程，这使得他们能够独立完成从后端到具体功能实现的完整开发流程，提高了项目的整体效率和质量。
 - **协同工作：**通过结对编程，两人能够互相检查和改进代码，提高代码的质量和可靠性。同时，他们能够分享彼此的经验和知识，促进相互学习和成长。

6.2 角色切换与任务分工

日期	时 间 （ HH:MM -- HH:MM)	“驾驶员”	“领航员”	本段时间的任务
2024-05-13	15:45 -- 16:30	韩晨烨	张智雄	编写基本的有向图构建代码和文件读取功能
2024-05-13	16:30 -- 17:30	张智雄	韩晨烨	实现桥接词查询功能并进行初步测试

2024-05-20	15:45 -- 16:30	韩晨烨	张智雄	实现根据桥接词生成新文本的功能
2024-05-20	16:30 -- 17:30	张智雄	韩晨烨	实现最短路径计算（单个和两个单词输入）
2024-05-27	15:45 -- 16:30	韩晨烨	张智雄	实现随机游走功能并处理线程协调和终止
2024-05-27	16:30 -- 17:30	张智雄	韩晨烨	测试和调试所有功能模块，修复发现的问题

6.3工作照片



图 14 工作照片

6.4工作日志

由领航员负责记录，记录结对编程期间的遇到的问题、两人如何通过交流合作解决每个问题的。可增加表格的行。

日期/时间	问题描述	最终解决方法	两人如何通过交流找到解决方法
5.13/15:45-16:30	无法正确读取文本文件的内容	检查文件路径和读取逻辑，修复文件读取代码	韩晨烨首先检查了文件路径和读取逻辑，发现了一个文件路径写错了，并进行了修复。
5.13/16:30-17:30	桥接词查询功能无法正常运行	修改了桥接词查询算法的逻辑，确保其正确性	刚开始以为桥接词是找到两点间所有的词，后来张智雄重新看了 PPT 发现桥接词只能为一个词
5.20/15:45-16:30	桥接词生成新文本不能生成所有缺失的词	在查询桥接词时，确保所有可能的桥接词都被考虑	两人经过分析代码，发现是在选择桥接词时的逻辑不够全面，未考虑到所有可能的情况。
5.20/16:30-17:30	最短路径计算功能所得最短路径错误	重新修改了 findallpath 函数，将其返回值改为包含路径	张智雄重新阅读了代码，发现 findallpath 函数只返回了路径，而没有路径长

		长度	度, 因此计算最短路径的时候默认了每一条边都是 0, 告诉韩晨烨后, 韩晨烨重新修改了代码
5.27/15:45-16:30	随机游走功能无法正确终止	修复了随机游走功能中的线程终止逻辑	张智雄和韩晨烨一起分析了随机游走功能中的线程终止逻辑, 发现监听终止符号的代码有问题。他们进行了代码重构, 并添加了正确的线程终止条件。
5.27/16:30-17:30	发现代码健壮性不够	在选择菜单输入的代码中增加了关于错误输入的判断代码	张智雄在测试代码时发现输入菜单输入字母会直接报错, 韩晨烨在代码中加入了对于错误输入的判断代码, 从而解决了该问题。

7 Git 操作过程

7.1 实验场景(1): 仓库创建与提交

R0: 在进行每次 git 操作之前, 随时查看工作区、暂存区、git 仓库的状态, 确认项目里的各文件当前处于什么状态;

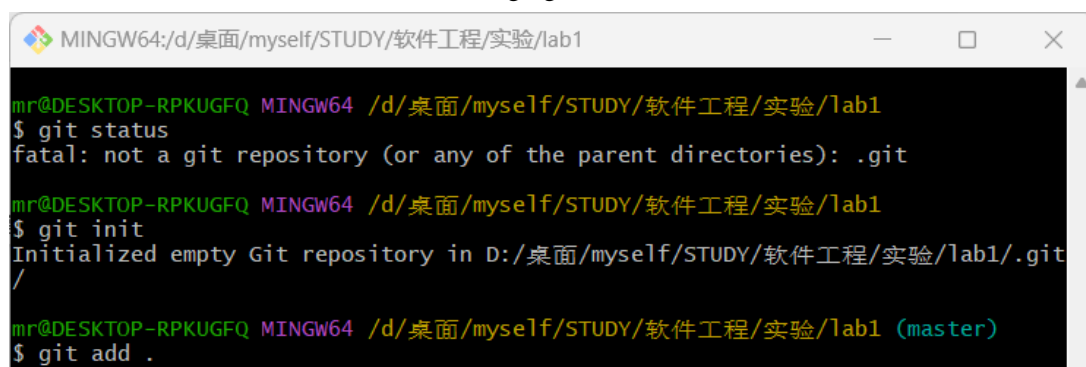
操作命令: **git status**

执行界面的截图附在后面每次 git 操作前, 查看工作区、暂存区、git 仓库的状态

R1: 本地初始化一个 git 仓库, 将自己在 Lab1 中所创建项目的全部源文件加入进去, 纳入 git 管理。

操作命令: 依次输入 **git status**, **git init** 和 **git add .**

由于还未初始化仓库, 输入 git status 查看状态时提示缺少 .git 目录。而后 git init 输入后提示成功初始化本地仓库。git add . 将当前目录及其子目录中的所有更改(新文件、修改过的文件、删除的文件)添加到暂存区(staging area)



```
MINGW64:/d/桌面/myself/STUDY/软件工程/实验/lab1
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1
$ git status
fatal: not a git repository (or any of the parent directories): .git

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1
$ git init
Initialized empty Git repository in D:/桌面/myself/STUDY/软件工程/实验/lab1/.git/

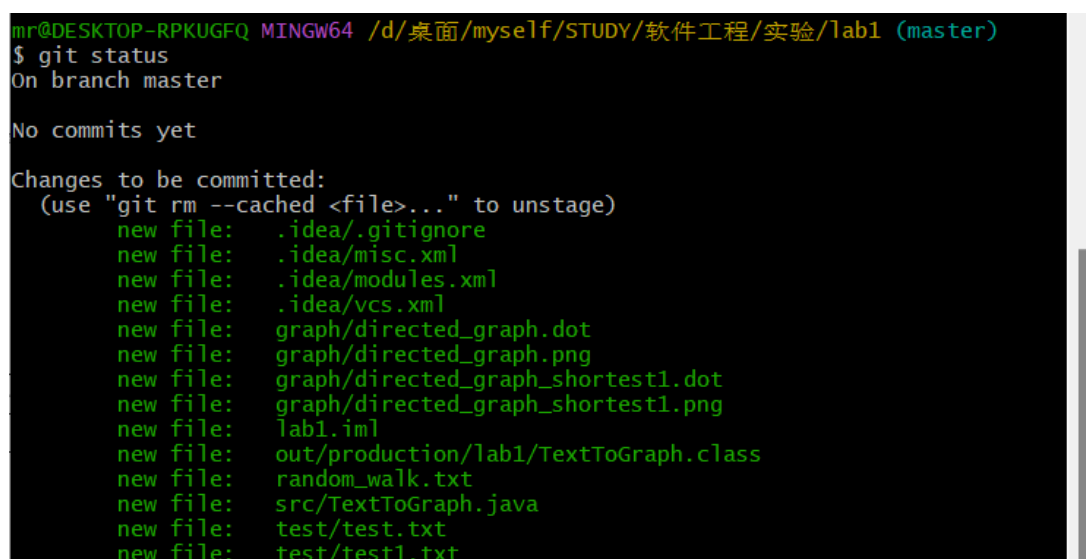
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git add .
```

图 15 仓库初始化截图

R2: 提交;

操作命令: 依次输入 **git status**, **git commit -m "Initial commit"**

首先输入 git status 时, 显示当前没有提交, 所有新加入的文件等待提交; 而后输入提交命令提交后, 显示状态更改的文件。



```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .idea/.gitignore
    new file:   .idea/misc.xml
    new file:   .idea/modules.xml
    new file:   .idea/vcs.xml
    new file:   graph/directed_graph.dot
    new file:   graph/directed_graph.png
    new file:   graph/directed_graph_shortest1.dot
    new file:   graph/directed_graph_shortest1.png
    new file:   lab1.iml
    new file:   out/production/lab1/TextToGraph.class
    new file:   random_walk.txt
    new file:   src/TextToGraph.java
    new file:   test/test.txt
    new file:   test/test1.txt
```

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git commit -m "Initial commit"
[master (root-commit) 6ce0257] Initial commit
14 files changed, 507 insertions(+)
create mode 100644 .idea/.gitignore
create mode 100644 .idea/misc.xml
create mode 100644 .idea/modules.xml
create mode 100644 .idea/vcs.xml
create mode 100644 graph/directed_graph.dot
create mode 100644 graph/directed_graph.png
create mode 100644 graph/directed_graph_shortest1.dot
create mode 100644 graph/directed_graph_shortest1.png
create mode 100644 lab1.iml
create mode 100644 out/production/lab1/TextToGraph.class
create mode 100644 random_walk.txt
create mode 100644 src/TextToGraph.java

```

图 16 提交截图

R3: 查看上次提交之后都有哪些文件修改、具体修改内容是什么;

操作命令: 依次输入 **git status** 和 **git diff**

随意修改 test.txt 文件内的内容, 可以看到显示有修改, 但修改并未提交; 同时查看修改内容, 可以看到在文末增加了一行 test git, 与实际相符。

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test/test.txt

no changes added to commit (use "git add" and/or "git commit -a")

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git diff
diff --git a/test/test.txt b/test/test.txt
index 1ff6c37..1234162 100644
--- a/test/test.txt
+++ b/test/test.txt
@@ -1,2 @@
-To explore strange new worlds, To seek out new life and new civilizations
\ No newline at end of file
+To explore strange new worlds, To seek out new life and new civilizationsAM
+test git
\ No newline at end of file

```

图 17 查看修改截图

R4: 重新提交;

操作命令: 依次输入 **git status**, **git add .** 和 **git commit -m "Change txt"**

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test/test.txt

no changes added to commit (use "git add" and/or "git commit -a")

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git add .

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git commit -m "Change txt"
[master 561815c] Change txt
1 file changed, 2 insertions(+), 1 deletion(-)

```

图 18 重新提交截图

R5: 再次对部分文件修改后重新提交

操作命令: `git add .` 和 `git commit -m "Change another txt"`

由于此处我先执行了 `git add .` 再执行 `git status`, 所以显示的是绿色等待被提交, 而不是没有提交:

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git add .

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   test/test1.txt

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git add .

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git commit -m "Change another txt"
[master c6dbbfe] Change another txt
1 file changed, 3 insertions(+), 1 deletion(-)
```

图 19 再次修改提交截图

R6: 把最后一次提交撤销

操作命令: `git reset HEAD~1`

这会把 HEAD 指针移回上一个提交 (HEAD~1), 并清除最后一次提交的内容。

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git status
On branch master
nothing to commit, working tree clean

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git reset HEAD~1
Unstaged changes after reset:
M       test/test1.txt
```

图 20 撤销截图

R7: 查询该项目的全部提交记录

操作命令: `git log`

可以看到第二次的修改还没提交, 提交记录一共两次

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test/test1.txt

no changes added to commit (use "git add" and/or "git commit -a")

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git log
commit 561815c209e1c93d8b10a57ea7ed749c7ec6b334 (HEAD -> master)
Author: rookiexiong7 <1557588606@qq.com>
Date:   Mon May 27 14:46:19 2024 +0800

    Change txt

commit 6ce0257766c14e5aed1cafdaae927ce69e5f6f8c
Author: rookiexiong7 <1557588606@qq.com>
Date:   Mon May 27 14:37:09 2024 +0800

    Initial commit
```

图 21 查看提交记录

R8: 在 GitHub 上创建名为“Lab1-学号”的仓库，并在本地仓库建立对应的远程仓库
操作命令: `git remote add origin https://github.com/rookiexiong7/Lab1-2021112845.git`

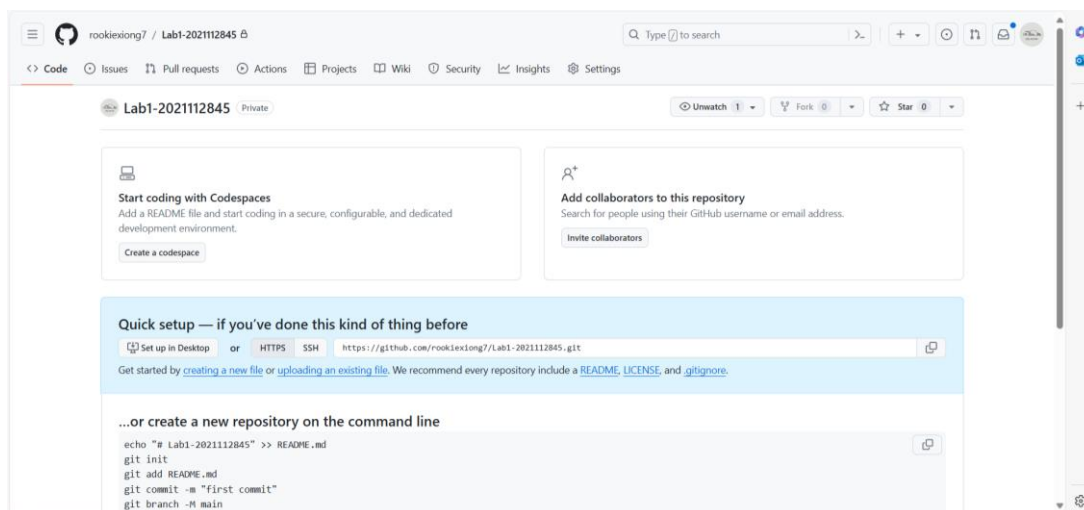


图 22 建立仓库

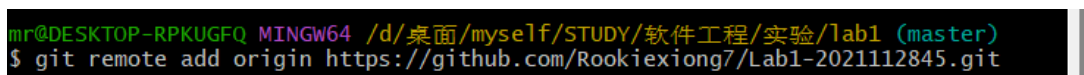


图 23 在本地建立对应的远程仓库

R9: 将之前各步骤得到的本地仓库全部内容推送到 GitHub 的仓库中;

操作命令: `git push -u origin master`

推送后, 可以在 Github 界面上查看到本地仓库的全部内容

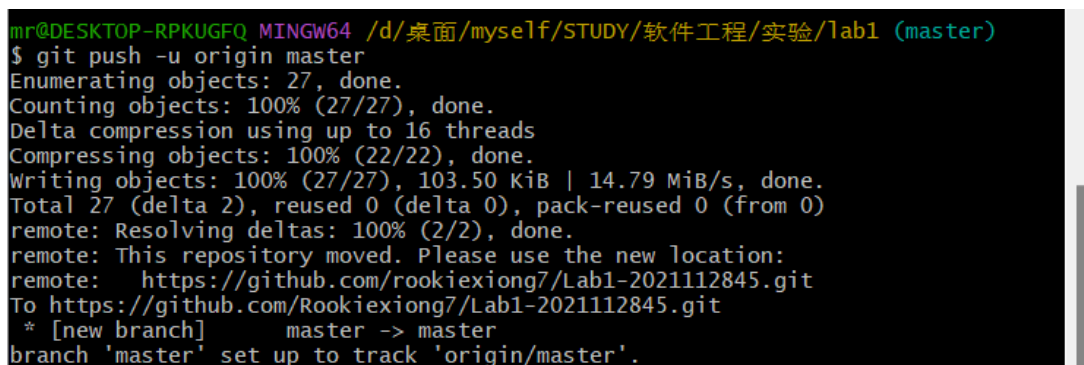


图 24 推送截图

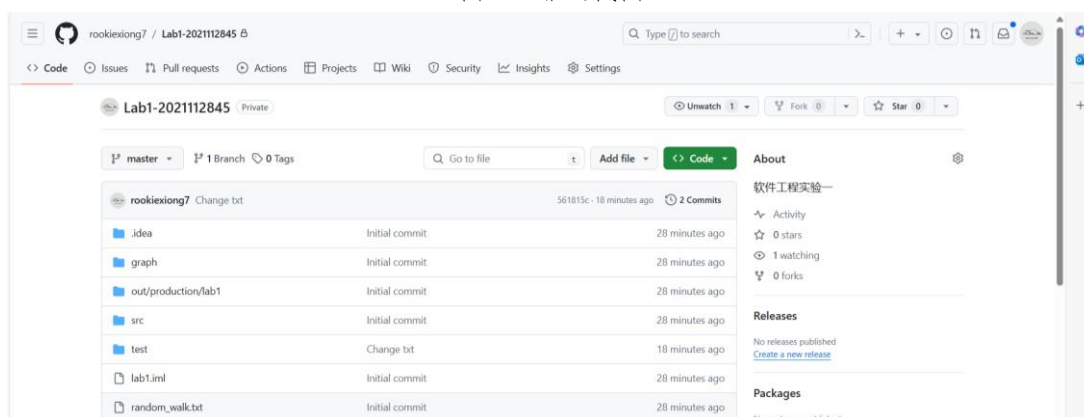


图 25 web 端结果展示

7.2 实验场景(2): 分支管理

R1: 获得本地 Lab1 仓库的全部分支, 切换至分支 master;

操作命令: `git branch` 和 `git checkout master`

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git branch
* master

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git checkout master
Already on 'master'
M   test/test1.txt
Your branch is up to date with 'origin/master'.
```

图 26 操作截图

R2: 在 master 基础上建立两个分支 B1、B2;

操作命令: `git checkout -b B1`, `git checkout master` 和 `git checkout -b B2`

执行上述命令后, 再次执行 `git branch` 命令, 可以看到新建的分支 B1 和 B2。

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git checkout -b B1
Switched to a new branch 'B1'

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git checkout master
Switched to branch 'master'
M   test/test1.txt
Your branch is up to date with 'origin/master'.

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (master)
$ git checkout -b B2
Switched to a new branch 'B2'

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B2)
$ git branch
B1
* B2
master
```

图 27 操作截图

R3: 在 B2 分支基础上创建一个新分支 C4;

操作命令: 依次执行 `git checkout B2` 和 `git checkout -b C4`

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B2)
$ git checkout B2
Already on 'B2'
M   test/test1.txt

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B2)
$ git checkout -b C4
Switched to a new branch 'C4'
```

图 28 操作截图

R4: 在 C4 上, 对 2 个文件进行修改并提交;

操作命令: 依次执行 `git add .` 和 `git commit -m "modify for C4"`

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (C4)
$ git add .
```

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (C4)
$ git commit -m "modify for C4"
[C4 0571dfe] modify for C4
3 files changed, 5 insertions(+), 1 deletion(-)
create mode 100644 "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\C41.txt"
create mode 100644 "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\C42.txt"

```

图 29 操作截图

R5: 在 B1 分支上对同样的 2 个文件做不同修改并提交;

操作命令: 输入 `git checkout B1`, `git add .` 和 `git commit -m "modify for B1"`

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (C4)
$ git checkout B1
Switched to branch 'B1'
A      "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\B11.txt"
A      "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\B12.txt"

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git add .

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git commit -m "modify for B1"
[B1 016e625] modify for B1
2 files changed, 2 insertions(+)
create mode 100644 "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\B11.txt"
create mode 100644 "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\B12.txt"

```

图 30 操作截图

R6: 将 C4 合并到 B1 分支, 若有冲突, 手工消解;

操作命令: 依次输入 `git checkout B1` 和 `git merge C4`

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git checkout B1
Already on 'B1'

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git merge C4
hint: Waiting for your editor to close the file...      0 [sig] bash 1947! sigp
acket::process: Suppressing signal 18 to win32 process (pid 18532)
Merge made by the 'ort' strategy.
 test/test1.txt                                         | 4 +++-
 .../C41.txt                                           | 1 +
 .../C42.txt                                           | 1 +
3 files changed, 5 insertions(+), 1 deletion(-)
create mode 100644 "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\C41.txt"
create mode 100644 "\\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271\C42.txt"

```

图 31 操作截图

R7: 在 B2 分支上对 2 个文件做修改并提交;

操作命令: 输入 `git checkout B2`, `git add .` 和 `git commit -m "modify for B2"`

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B2)
$ git add .

```

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B2)
$ git commit -m "modify for B2"
[B2 9484323] modify for B2
2 files changed, 2 insertions(+)
create mode 100644 "\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271/B21.txt"
create mode 100644 "\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271/b22.txt"

```

图 32 操作截图

R8: 查看目前哪些分支已经合并、哪些分支尚未合并;

操作命令: 依次执行 `git checkout B1`, `git branch --merged` 和 `git branch --no-merged`

```

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git checkout B1
Already on 'B1'

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git branch --merged
* B1
  C4
  master

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git branch --no-merged
B2

```

图 33 操作截图

R9: 将已经合并的分支删除, 将尚未合并的分支合并到一个新分支上, 分支名字为你的学号;

操作命令: 执行 `git branch -d C4` 以删除已合并分支, 执行 `git checkout -b 2021112845` 创建新分支, 而后执行 `git merge B2` 将 B2 合并到新分支中。

```

MINGW64:/d/桌面/myself/STUDY/软件工程/实验/lab1
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git branch -d C4
Deleted branch C4 (was 0571dfe).

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (B1)
$ git checkout -b 2021112845
Switched to a new branch '2021112845'

mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (2021112845)
$ git merge B2
Merge made by the 'ort' strategy.
"\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271/B21.txt" | 1 +
"\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271/b22.txt" | 1 +
2 files changed, 2 insertions(+)
create mode 100644 "\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271/B21.txt"
create mode 100644 "\346\226\260\345\273\272\346\226\207\344\273\266\345\244\271/b22.txt"

```

图 34 操作截图

R10: 将本地以你的学号命名的分支推送到 GitHub 上自己的仓库内;

操作命令: `git push -u origin 2021112845`

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (2021112845)
$ git push -u origin 2021112845
Enumerating objects: 26, done.
Counting objects: 100% (26/26), done.
Delta compression using up to 16 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (23/23), 1.62 KiB | 1.62 MiB/s, done.
Total 23 (delta 10), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (10/10), completed with 2 local objects.
remote: This repository moved. Please use the new location:
remote:   https://github.com/rookiexiong7/Lab1-2021112845.git
remote:
remote: Create a pull request for '2021112845' on GitHub by visiting:
remote:   https://github.com/rookiexiong7/Lab1-2021112845/pull/new/2021112845
remote:
To https://github.com/Rookiexiong7/Lab1-2021112845.git
* [new branch]      2021112845 -> 2021112845
branch '2021112845' set up to track 'origin/2021112845'.
```

图 35 操作截图

R11: 查看完整的版本变迁树;

操作命令: `git log --graph --all --oneline -decorate`

```
mr@DESKTOP-RPKUGFQ MINGW64 /d/桌面/myself/STUDY/软件工程/实验/lab1 (2021112845)
$ git log --graph --all --oneline --decorate
* 196535f (HEAD -> 2021112845, origin/2021112845) Merge branch 'B2' into 2021112845
|
| * 9484323 (B2) modify for B2
| * 4c1b869 (B1) Merge branch 'C4' into B1
| * | 0571dfe modify for C4
| * | 016e625 modify for B1
|
| * 561815c (origin/master, master) Change txt
| * 6ce0257 Initial commit
```

图 36 操作截图

R12: 在 Github 上以 web 页面的方式查看你的 Lab1 仓库的当前状态。

在 web 页面查看 2021112845 分支, 界面如下:

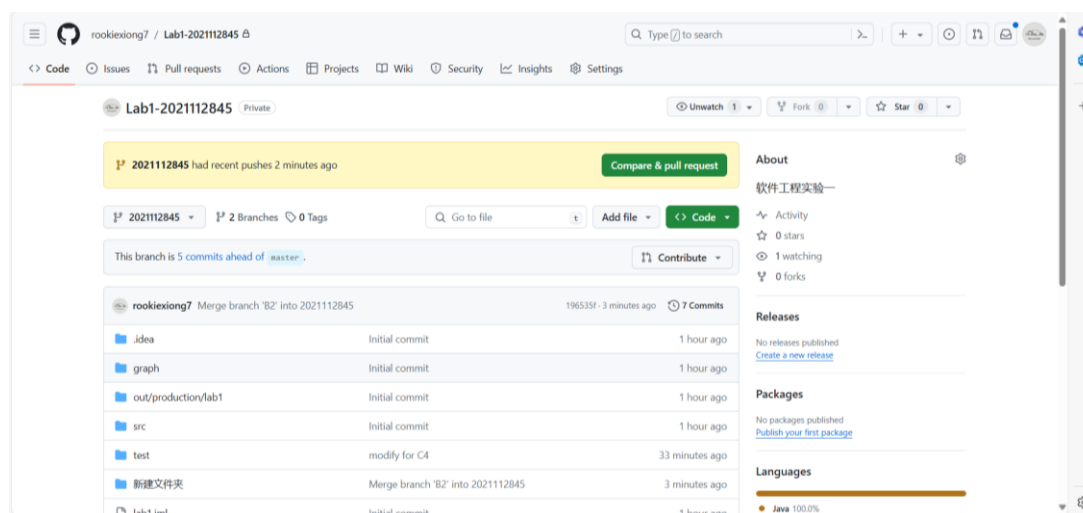


图 37 界面截图

8 在 IDE 中使用 Git Plugin

R1: 在 IDE 中，将自己的 Lab1 纳入 Git 管理；

本实验使用 IDEA 作为开发环境 IDE，点击底部导航栏的 Git 选项，可以看到当前的分支状态和版本历史。

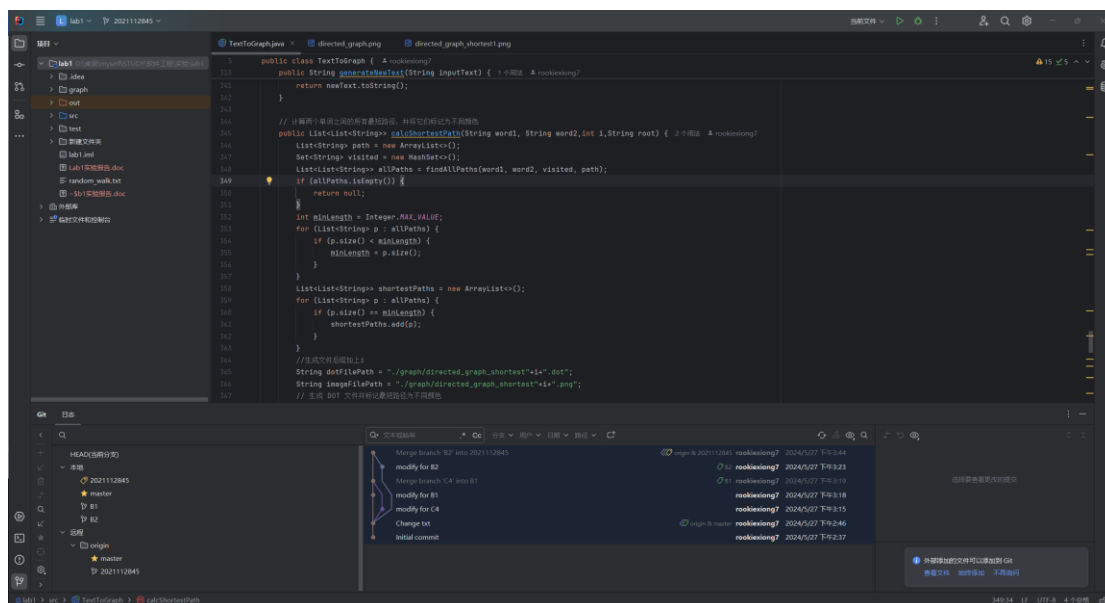


图 38 当前的分支状态和版本变迁历史

R2: 对 Lab1 进行若干修改，对其进行本地仓库提交操作；

修改代码后点击左侧任务栏进行提交：

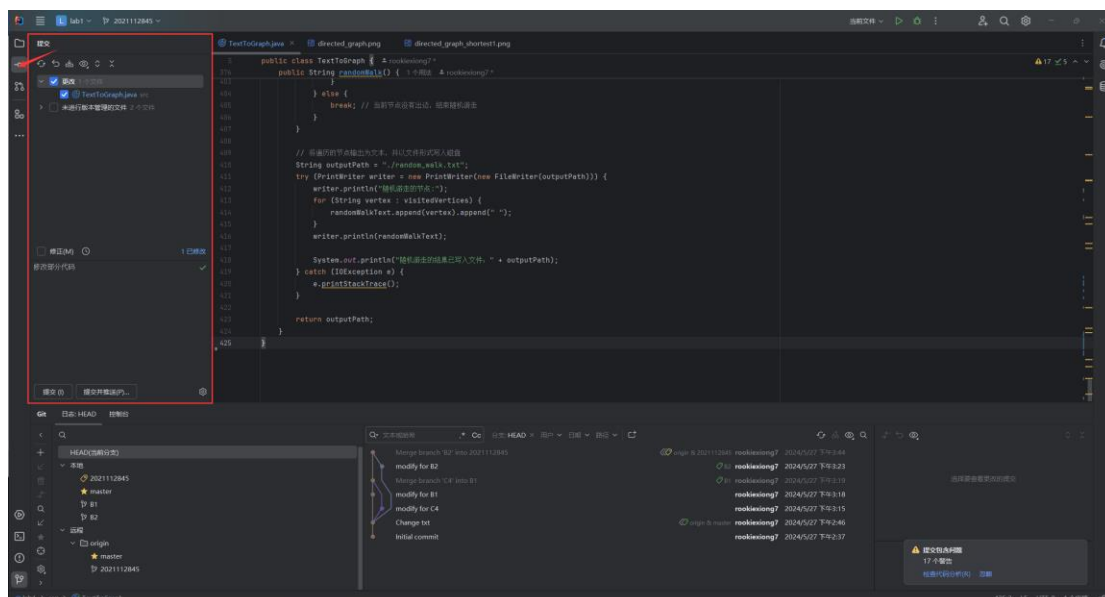


图 39 提交本地仓库

而后可以看到已经提交至本地仓库：

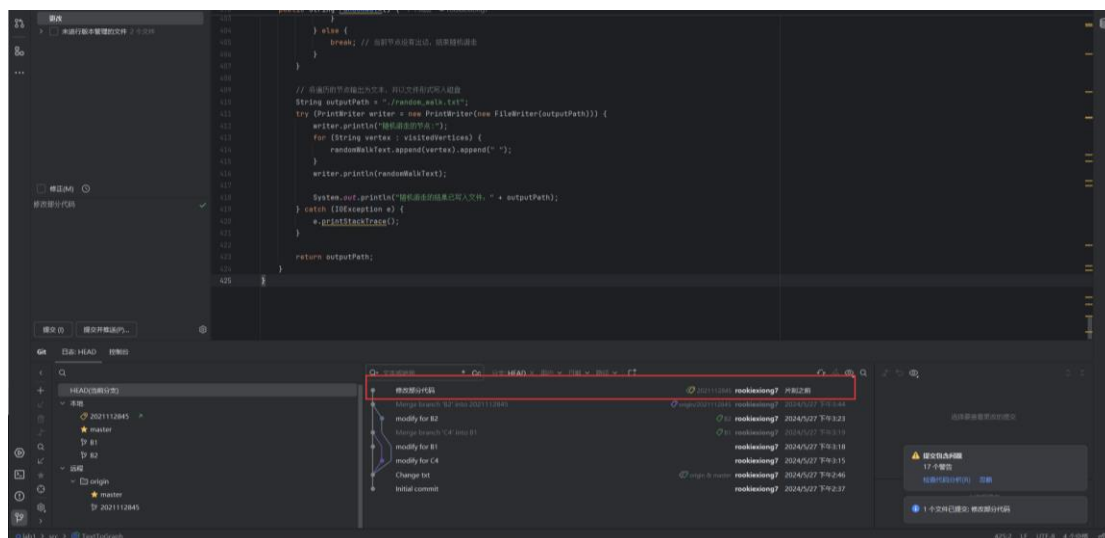


图 40 结果截图

R3: 将 Lab1 内容推送至个人 GitHub 的 Lab1 仓库。

切换到 master 分支，将推送提交至远程仓库，显示推送成功。

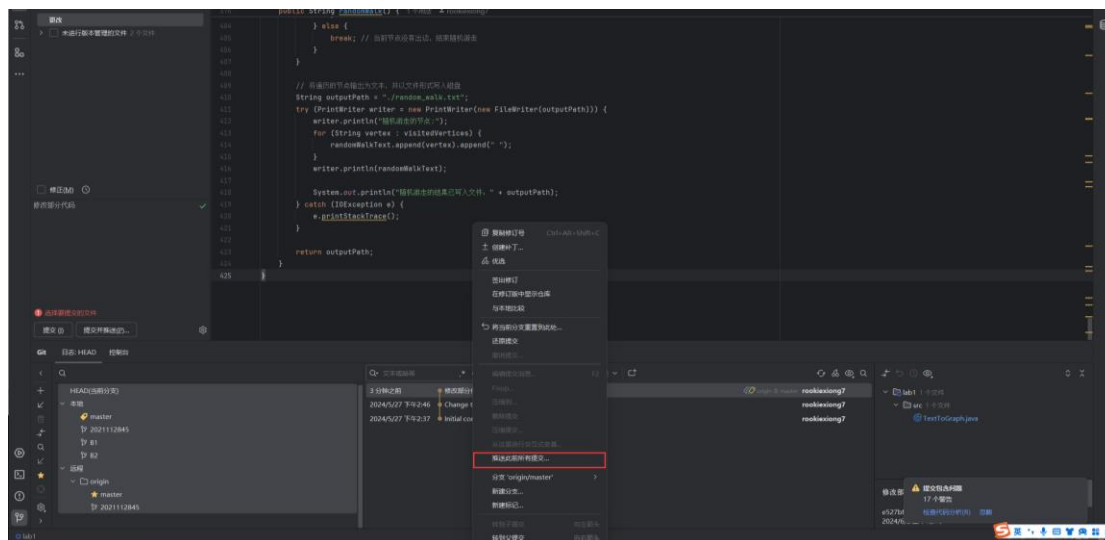


图 41 推送远程仓库

在 Github 中可以看到新提交的更改。

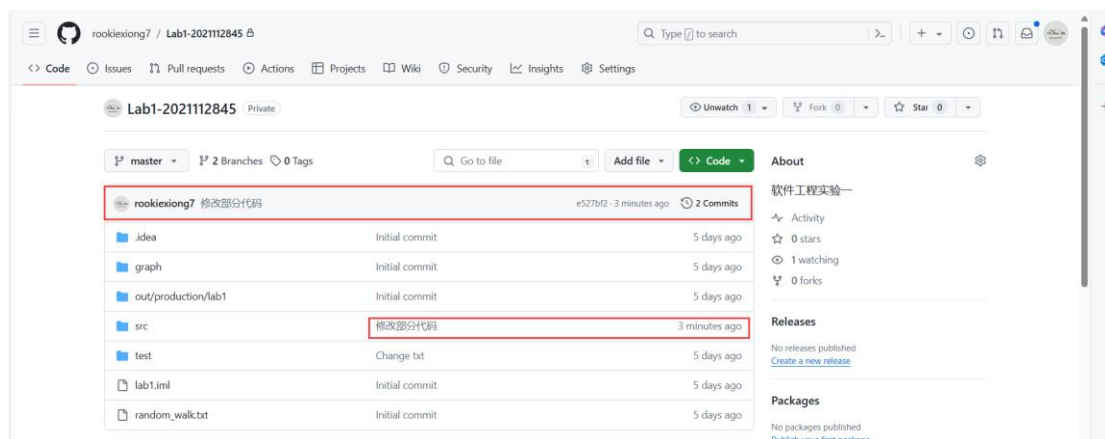


图 42 界面截图

9 小结

在本次实验中，结对编程实验让我们深刻体验了敏捷开发中的协作模式。通过与队友共同编码，我们学会了有效地分工合作、及时沟通交流，并且在代码设计和重构中培养了良好的团队合作意识。通过这种方式，我们不仅提升了编程技能，还锻炼了问题解决和设计能力。

同时，Git 实战让我们更好地掌握了版本控制和团队协作的技巧。通过使用 Git 管理项目源代码，我们能够更轻松地进行代码版本管理、分支管理和团队协作，提高了项目的可维护性和开发效率。

这两项实验的结合使我们不仅加深了对敏捷开发和软件配置管理的理解，也在实践中积累了宝贵的团队协作经验，为今后的软件开发项目奠定了坚实的基础。