

**2025年春季学期  
计算学部《软件工程》课程**

**实验报告**

**Lab3 代码评审与单元测试**

|  |  |  |
| --- | --- | --- |
| **姓名** | **学号** | **联系方式** |
| 彭锦杰 | 2022112885 | 1491797710@qq.com |

**目 录**

[1 实验要求 1](#_Toc168176490)

[2 在IDE中配置代码审查与分析工具 1](#_Toc168176491)

[2.1 Checkstyle 1](#_Toc168176492)

[2.2 SpotBugs 1](#_Toc168176493)

[2.3 EclEmma 1](#_Toc168176494)

[2.4 Junit 1](#_Toc168176495)

[3 Checkstyle所发现的代码问题清单及原因分析 1](#_Toc168176496)

[4 SpotBugs所发现的代码问题清单及原因分析 1](#_Toc168176497)

[5 针对Lab1的黑盒测试 2](#_Toc168176498)

[5.1 所选的被测函数及其需求规约 2](#_Toc168176499)

[5.2 等价类划分结果 2](#_Toc168176500)

[5.3 测试用例设计 2](#_Toc168176501)

[5.4 JUnit测试代码 3](#_Toc168176502)

[5.5 JUnit单元测试结果 3](#_Toc168176503)

[5.6 未通过测试的原因分析及代码修改 4](#_Toc168176504)

[6 针对Lab1的白盒测试 5](#_Toc168176505)

[6.1 所选的被测函数 5](#_Toc168176506)

[6.2 程序流程图 6](#_Toc168176507)

[6.3 控制流图 6](#_Toc168176508)

[6.4 圈复杂度计算与基本路径识别 6](#_Toc168176509)

[6.5 测试用例设计 6](#_Toc168176510)

[6.6 JUnit测试代码 7](#_Toc168176511)

[6.7 JUnit单元测试结果 7](#_Toc168176512)

[6.8 代码覆盖度分析 8](#_Toc168176513)

[6.9 未通过测试的原因分析及代码修改 8](#_Toc168176514)

[7 计划与实际进度 9](#_Toc168176515)

[8 小结 9](#_Toc168176516)

[文档全部完成之后，请更新上述区域]

# 实验要求

▪ 针对Lab1所完成的代码，进行代码评审(走查) ，从代码规范性和正确

性角度对代码进行评价；

▪ 使用以下两个工具完成实验：

– Checkstyle

– SpotBugs

# 在IDE中配置代码审查与分析工具

我使用的是pycharm中的插件

## Flake8

工具：Flake8

安装方式：使用 pip 安装（pip install flake8）

配置方法：在 PyCharm External Tools 中配置，或使用 PyCharm 的 Code Inspection 功能。

## Pylint

工具：Pylint

安装方式：pip install pylint

配置方式：在 PyCharm External Tools 中配置，也可在终端中运行：

pylint your\_module.py

## Pytest

工具：Pylint

安装方式：pip install pylint

配置方式：在 PyCharm External Tools 中配置，也可在终端中运行：

pylint your\_module.py

## Converge

工具：coverage.py

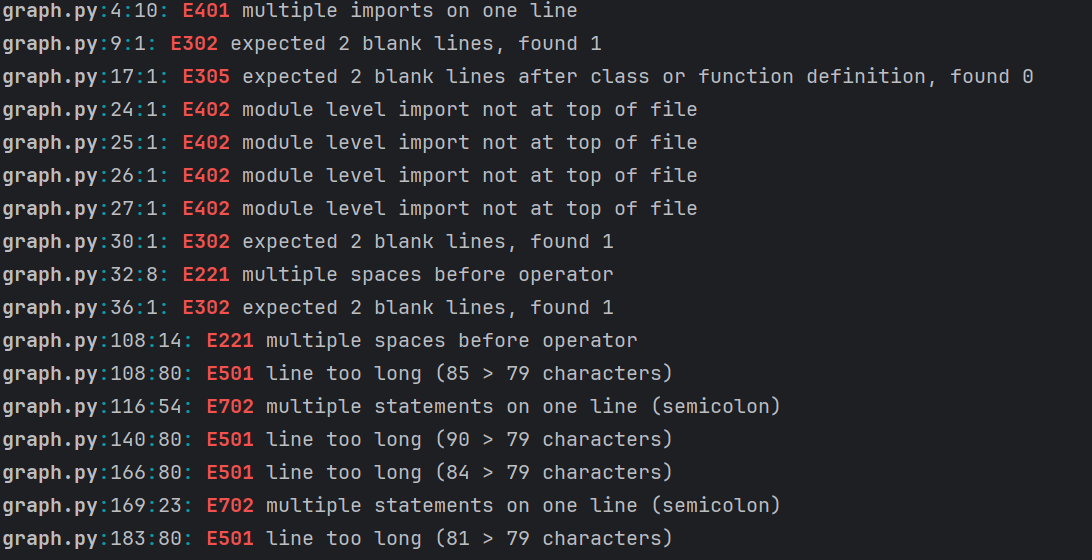
安装方式：pip install coverage

使用方式：

1. coverage run -m pytest

2. coverage report 或 coverage html

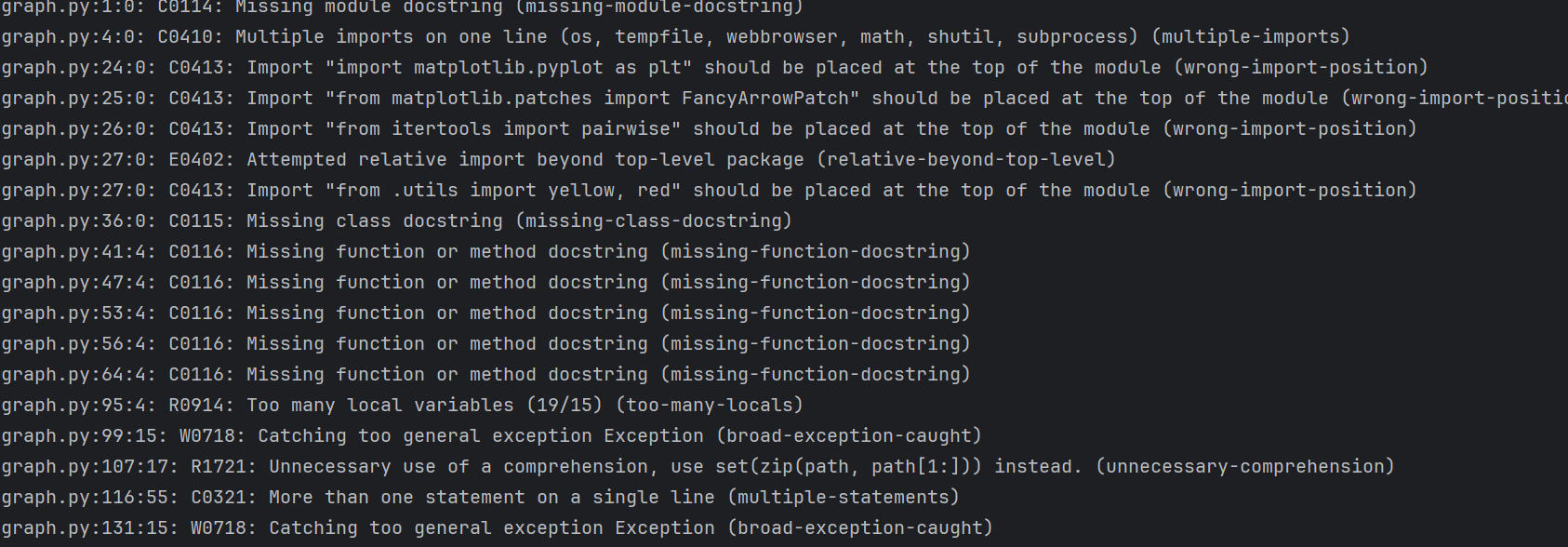
# Flake8所发现的代码问题清单及原因分析



针对同种类型的问题，只需要列出一个典型代表即可。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 问题描述 | 类型 | 所在代码行号 | 修改策略 |
| 1 | multiple imports on one line | E401 | 4 | 拆分多模块导入为多行，例如 import os、import sys |
| 2 | expected 2 blank lines, found 1 | E302 | 9 | 函数或类定义前应添加空行，使其前后有两行空行 |
| 3 | module level import not at top of file | E402 | 24-27 | 所有 import 语句应写在文件顶部 |
| 4 | multiple spaces before operator | E221 | 32 | 运算符前后应只保留一个空格 |
| 5 | line too long (>79 characters) | E501 | 第108/140等行 | 拆行或使用 \ 折行以避免超出限制 |
| 6 | multiple statements on one line (semicolon) | E702 | 第116/169等行 | 一行只写一个语句，避免使用分号分隔 |

# Pylint所发现的代码问题清单及原因分析



针对同种类型的问题，只需要列出一个典型代表即可。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 优先级 | 问题描述 | 违反的规则集 | 所在代码行号 | 修改策略 |
| 高 | 定义局部变量过多（>15） | R0914 too-many-locals | 第95/156 行 | 简化函数逻辑，抽取子函数，减少局部变量数量 |
| 中 | 捕获了过于宽泛的异常（Exception） | W0718 broad-exception-caught | 多处 | 捕获具体异常类型，如 ValueError, KeyError 等 |
| 低 | 不必要地使用了 set 推导式 | R1721 unnecessary-comprehension | 第127行 | 简化为 set(zip(path, path[1:])) |

# 针对Lab1的黑盒测试

## 所选的被测函数及其需求规约

**函数名称**：generate\_new\_text(g: DirectedGraph, text: str) -> str

**功能描述**：将桥接词随机插入到输入文本相邻单词之间，形成新的句子。

**输入描述**：

g：包含单词关系的有向图（DirectedGraph）

text：原始英文文本，单词之间以空格分隔

**输出描述**：

插入桥接词后的新文本（单词以空格拼接）

## 等价类划分结果

请根据自己的情况扩展该表格，给各个等价类唯一的编号。

可根据需要，增加下表的行数。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 约束条件说明 | 有效等价类及其编号 | | 无效等价类及其编号 | |
| 输入单词数大于等于2 | 输入单词数大于等于2且存在两个单词中间有桥接词 | (1) | 输入单词数小于2 | (4) |
|  | 输入单词数大于等于2且不存在两个单词中间有桥接词 | (2) | 输入单词数大于等于2且输入的单词均不为原文本中的单词 | (5) |
|  |  | (3) |  | (6) |

## 测试用例设计

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例编号 | 输入 | 期望输出 | 所覆盖的等价类编号 |
|  | word | word | 4 |
|  | the fox | the quick fox | 1 |
|  | the quick fox | the quick fox | 2 |
|  | (空) | （空） | 4 |
|  | x y | x y | 5 |

## Pytest单元测试代码

针对5.3中的每一个测试用例，把其测试代码粘贴如下，代码必须是完整的。

import pytest  
from .graph import DirectedGraph  
from .bridge import generate\_new\_text  
  
# 辅助打印函数  
def log\_case(name, input\_text, output\_text):  
 print(f"\n[{name}] 输入: {input\_text}")  
 print(f"[{name}] 输出: {output\_text}")  
  
# 构造测试图  
@pytest.fixture  
def sample\_graph():  
 g = DirectedGraph()  
 g.add\_edge("the", "quick")  
 g.add\_edge("quick", "fox")  
 g.add\_edge("the", "brown")  
 g.add\_edge("brown", "fox")  
 g.add\_edge("lazy", "dog")  
 return g

|  |  |
| --- | --- |
| 测试用例  编号 | Pytest测试代码 |
|  | def test\_tc1\_single\_word(sample\_graph):  input\_text = "word"  result = generate\_new\_text(sample\_graph, input\_text)  log\_case("TC1", input\_text, result)  assert result == "word" |
|  | def test\_tc2\_has\_bridge(sample\_graph):  input\_text = "the fox"  result = generate\_new\_text(sample\_graph, input\_text)  log\_case("TC2", input\_text, result)  words = result.split()  assert words[0] == "the" and words[-1] == "fox"  assert len(words) in [2, 3] |
|  | def test\_tc3\_chain\_words(sample\_graph):  input\_text = "the quick fox"  result = generate\_new\_text(sample\_graph, input\_text)  log\_case("TC3", input\_text, result)  words = result.split()  assert words[0] == "the" and words[-1] == "fox"  assert len(words) >= 3 |
|  | def test\_tc4\_empty\_text(sample\_graph):  input\_text = ""  result = generate\_new\_text(sample\_graph, input\_text)  log\_case("TC4", input\_text, result)  assert result == "" |
|  | def test\_tc5\_unknown\_words(sample\_graph):  input\_text = "x y"  result = generate\_new\_text(sample\_graph, input\_text)  log\_case("TC5", input\_text, result)  assert result == "x y" |

## Pytest单元测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例编号 | 期望输出 | 实际输出 | 是否通过测试，请给出屏幕截图 |
|  | word | word |  |
|  | the fox | the quick fox |  |
|  | the quick fox | the quick fox |  |
|  | （空） | （空） |  |
|  | x y | x y |  |

## 未通过测试的原因分析及代码修改（均通过测试）

请简要分析自己的Lab1代码为何未通过5.5节表格中某些测试用例的原因，并通过修改代码消除此类不符合需求的Bug，必要时给出修改后的代码。

若5.5节表格中没有未通过的测试用例，本节可空。

注意：虽然本部分为黑盒测试，但发现错误之后仍然需要对代码进行修改。此时，需要根据测试结果对代码加以修改，目的是“满足需求”。

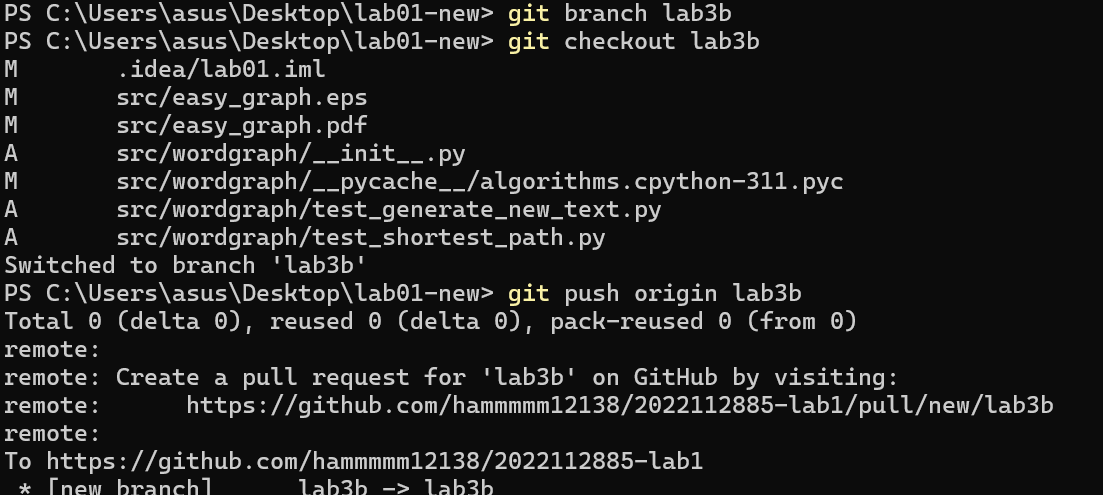
此外，此部分是否发现Lab1的Bug，不影响Lab1实验结果的评判。

修改代码之后，请重新填写下表，尽可能保证所有测试用例都能通过测试。

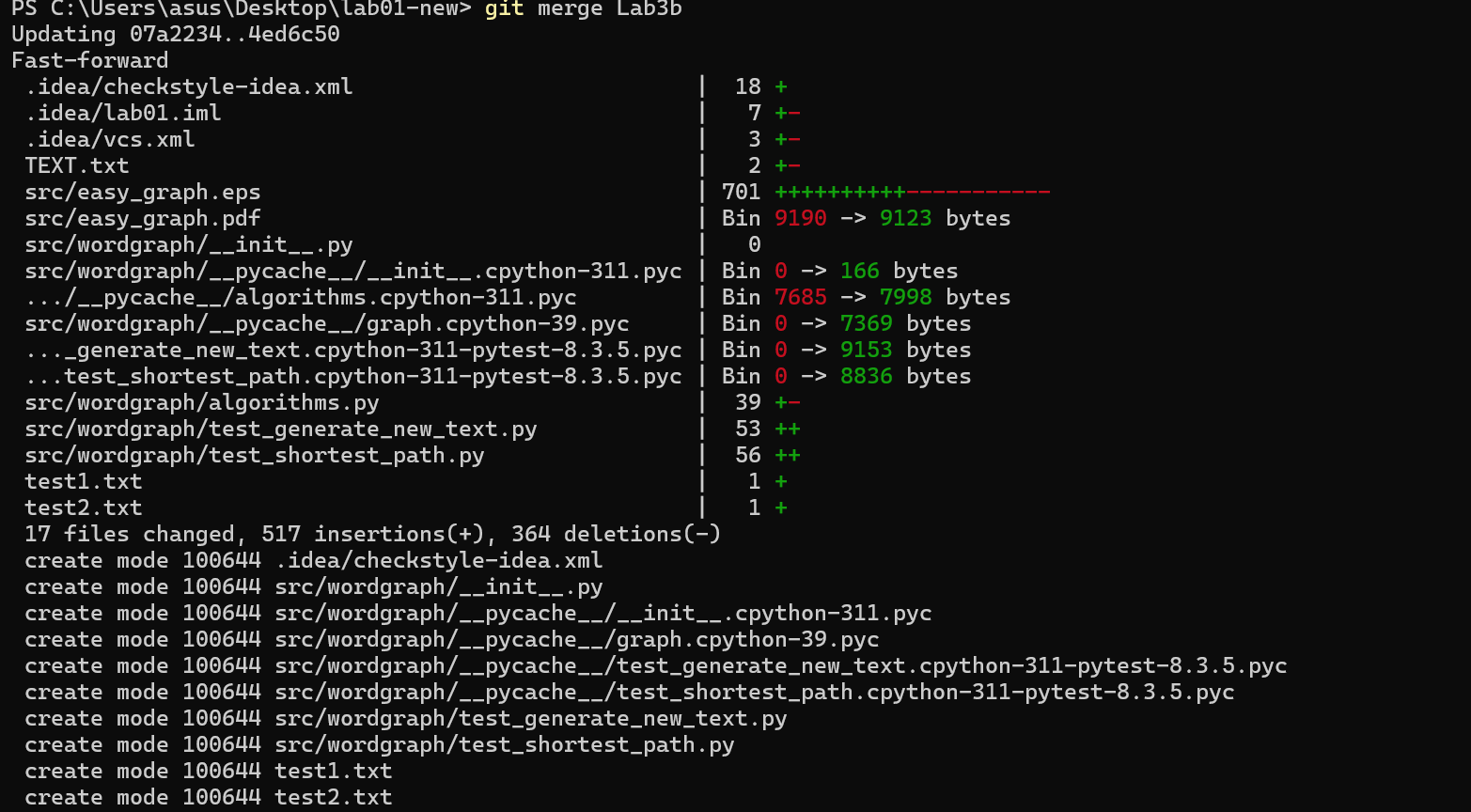
|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例编号 | 期望输出字符串 | 实际输出字符串 | 是否通过测试，请给出屏幕截图 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

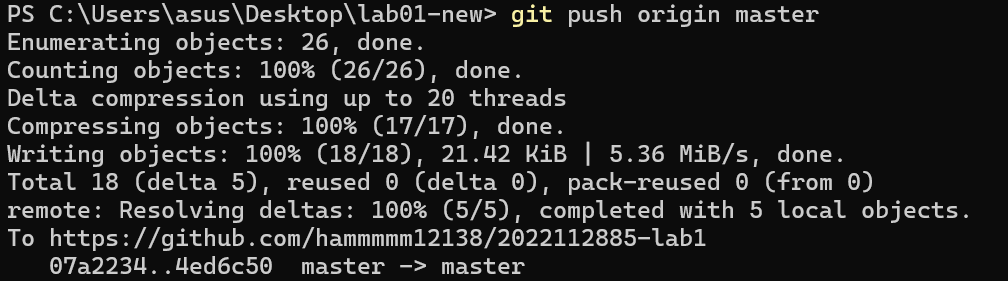
## Git操作记录

给出本地创建Lab3b分支，以及推送到Github上操作命令的截图；



给出本地合并Lab3b分支到master分支，以及推送到Github上操作命令的截图。





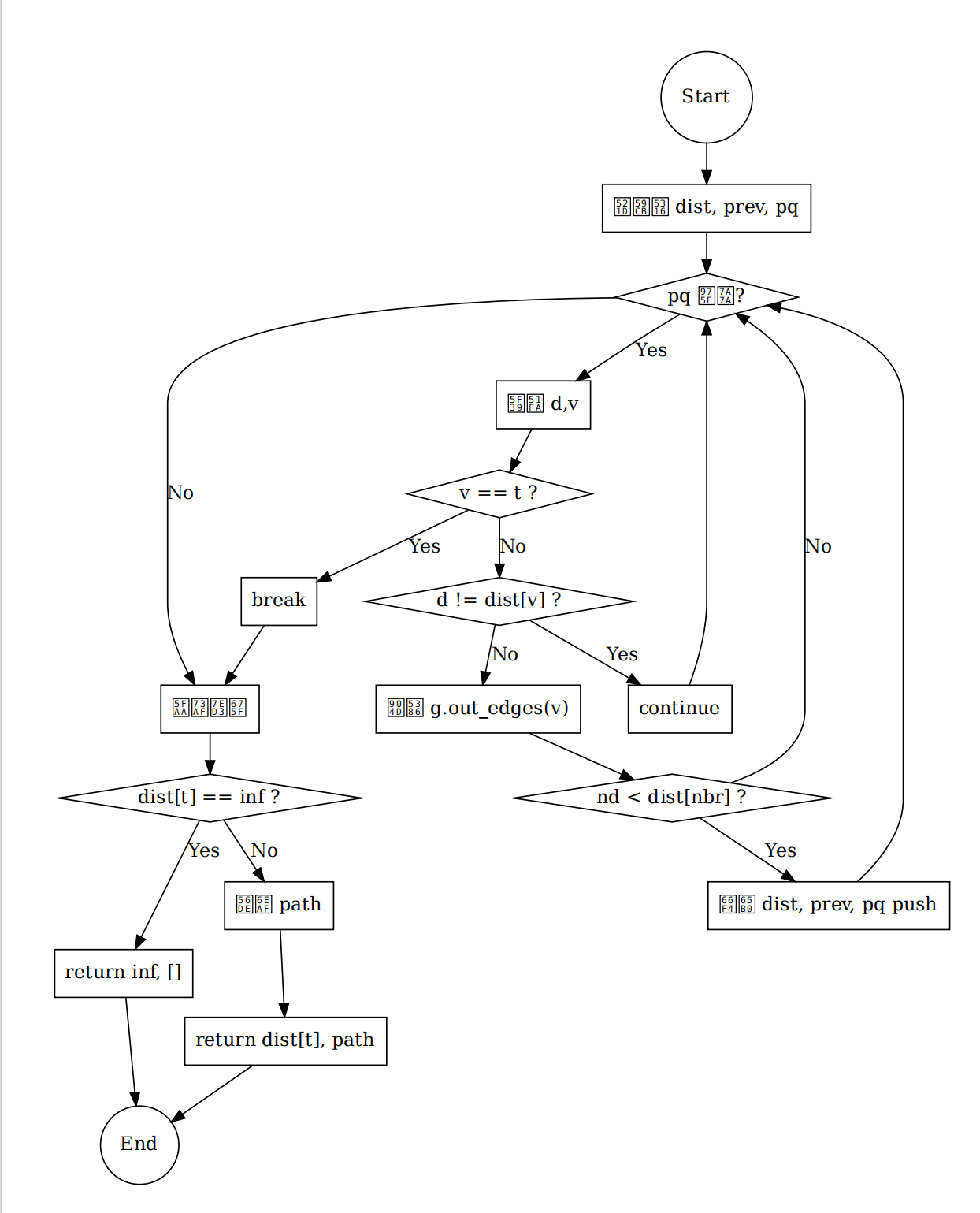
# 针对Lab1的白盒测试

## 所选的被测函数

注意：不能与5.1节所选函数重复。

|  |  |  |  |
| --- | --- | --- | --- |
| 被测函数的名称 | shortest\_path(g: DirectedGraph, s: str, t: str) | | |
| 功能描述 | 使用 Dijkstra 算法，计算图 g 中从源点 s 到目标点 t 的最短路径与距离 | | |
| 被测函数的代码  （以IDE环境下的截图方式给出，确保能看清楚，并保留IDE为每行代码分配的行号，后续各部分均以此行号为准。如果一屏截取不下，可以分多屏截取，均插入右侧格中） |  | | |
| 输入参数列表 | 参数名 | 含义 | 数据类型 |
| g | 图对象 | DirectedGraph |
| s  t | 起点单词  重点单词 | Str  Str |
| 输出参数 | 含义 | | 数据类型 |
| Dist 最短距离（不可达为 ∞）  Path 最短路径节点序列 | | Float  List[str] |
| 代码总行数 | 16 | | |
| 包含的循环数 | 3 | | |
| 包含的判定数 | 4 | | |

## 程序流程图



## 图示 AI 生成的内容可能不正确。控制流图

## 圈复杂度计算与基本路径识别

圈复杂度为：（请给出计算过程）

**节点数 N = 12** **边数 E = 15** **出口 P = 1**

V(G)=E−N+2P=15−12+2=5

| **基本路径编号** | **关键行号序列** | **覆盖的判定分支** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **P1** | 11 ➜ 12 ➜ 13(T) ➜ 20(F) ➜ 21 ➜ 22(loop F) ➜ 23 | 起点弹出即 v==t，成功回溯路径并返回 |

|  |  |  |
| --- | --- | --- |
| **P2** | 11 ➜ 12 ➜ 13(F) ➜ 14(T → continue) ↺ 11 … pq 为空 ➜ 20(T) (不可达返回) | 触发 d != dist[v] 分支，最终 unreachable |

|  |  |  |
| --- | --- | --- |
| **P3** | 11 ➜ 12 ➜ 13(F) ➜ 14(F) ➜ 15–19 (一次松弛) ↺ 11 ➜ 12 ➜ 13(T) ➜ 20(F) ➜ 21–22 ➜ 23 | 正常松弛后再次出队即命中目标 |

|  |  |  |
| --- | --- | --- |
| **P4** | 11 ➜ 12 ➜ 13(F) ➜ 14(F) ➜ 15–17(F) (松弛失败) ↺ 15(for 结束) ↺ 11 … pq 为空 ➜ 20(T) | for-loop 未更新任何边，导致不可达 |

|  |  |  |
| --- | --- | --- |
| **P5** | 11 ➜ 12 ➜ 13(T) ➜ 20(F) ➜ 21 ➜ 23 | 特殊情形：s == t（while 22 不进入循环） |

注意：各基本路径要使用6.1节表格里给出的行号。

## 测试用例设计

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例编号 | 输入数据 | 期望的输出 | 所覆盖的基本路径编号 |
|  | A → A（起点与终点相同） | dist = 0 path = ["A"] | p1 |
|  | A → F（默认图） | dist = 3 path= ["A","B","C","F"] | P3 |
|  | A → E（默认图） | dist = 2 path = ["A","D","E"] 或 ["A","B","E"] | P3 |
|  | A → Y 并 向图添加孤立边 X → Y | dist = inf path = [] | P4 |
|  | A → F 并 添加劣路径 A → X → E | dist = 3 path = ["A","B","C","F"] | P2 |

## Pytest测试代码

针对6.5中的每一个用例，把其测试代码粘贴如下，代码必须是完整的。

import pytest

from .graph import DirectedGraph

from .algorithms import shortest\_path

# 构造用于路径覆盖的测试图

@pytest.fixture

def sample\_graph():

g = DirectedGraph()

g.add\_edge("A", "B")

g.add\_edge("B", "C")

g.add\_edge("C", "D")

g.add\_edge("A", "D")

g.add\_edge("B", "E")

g.add\_edge("D", "E")

g.add\_edge("E", "F")

g.add\_edge("C", "F")

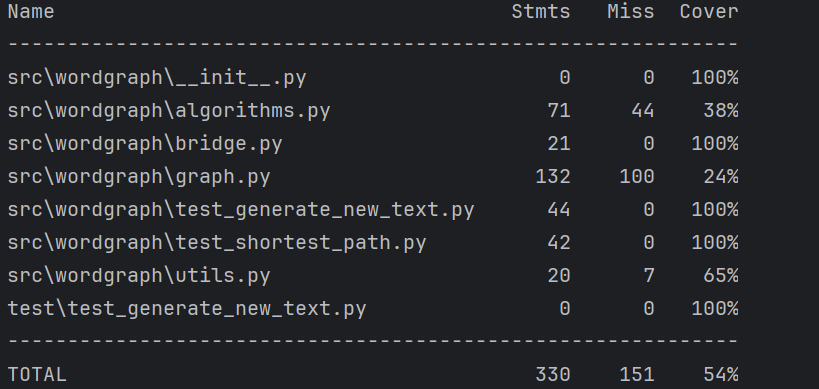
return g

|  |  |
| --- | --- |
| 测试用例编号 | Pytest测试代码 |
|  | def test\_p1\_full\_path(sample\_graph):  dist, path = shortest\_path(sample\_graph, "A", "F")  print(f"\n[P1] 距离: {dist}, 路径: {path}")  assert path == ["A", "B", "C", "F"]  assert dist == 3.0 |
|  | def test\_p2\_direct\_path(sample\_graph):  dist, path = shortest\_path(sample\_graph, "A", "E")  print(f"\n[P2] 距离: {dist}, 路径: {path}")  assert dist == 2.0  assert path in [["A", "D", "E"], ["A", "B", "E"]] |
|  | def test\_p3\_self\_loop(sample\_graph):  dist, path = shortest\_path(sample\_graph, "A", "A")  print(f"\n[P3] 距离: {dist}, 路径: {path}")  assert dist == 0.0  assert path == ["A"] |
|  | def test\_p4\_unreachable(sample\_graph):  sample\_graph.add\_edge("X", "Y") # 添加孤立子图  dist, path = shortest\_path(sample\_graph, "A", "Y")  print(f"\n[P4] 距离: {dist}, 路径: {path}")  assert dist == float("inf")  assert path == [] |
|  | def test\_p5\_dist\_check\_continue(sample\_graph):  # 构造双路径结构以触发 pq 中的低优先路径被跳过  sample\_graph.add\_edge("A", "X")  sample\_graph.add\_edge("X", "E") # A→X→E（劣路径）  dist, path = shortest\_path(sample\_graph, "A", "F")  print(f"\n[P5] 距离: {dist}, 路径: {path}")  assert path == ["A", "B", "C", "F"] |

## Pytest单元测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例编号 | 期望输出 | 实际输出 | 是否通过测试，请给出屏幕截图 |
|  | [P1] 距离: 3.0, 路径: ['A', 'B', 'C', 'F'] | [P1] 距离: 3.0, 路径: ['A', 'B', 'C', 'F'] |  |
|  | [P2] 距离: 2.0, 路径: ['A', 'B', 'E']或['A','D','E'] | [P2] 距离: 2.0, 路径: ['A', 'B', 'E'] |  |
|  | [P3] 距离: 0.0, 路径: ['A'] | [P3] 距离: 0.0, 路径: ['A'] |  |
|  | [P4] 距离: inf, 路径: [] | [P4] 距离: inf, 路径: [] |  |
|  | [P5] 距离: 3.0, 路径: ['A', 'B', 'C', 'F'] | [P5] 距离: 3.0, 路径: ['A', 'B', 'C', 'F'] |  |

## 代码覆盖度分析



## 未通过测试的原因分析及代码修改（均通过测试）

分析自己的Lab1代码为何未通过6.7节表格中某些测试用例的原因，并通过修改代码消除此类BUG。必要时给出修改后的代码。

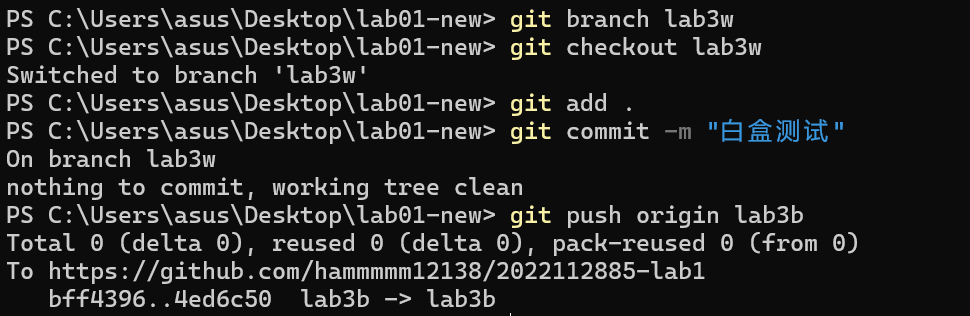
若6.7节表格中没有未通过的测试用例，本节可空。

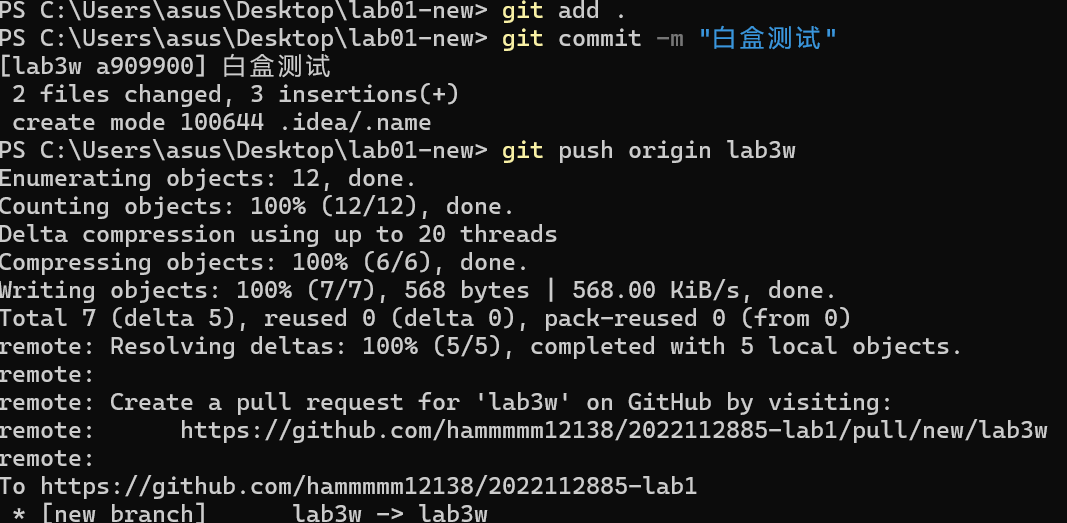
修改代码之后，请重新填写下表，保证所有测试用例都能通过测试。

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例编号 | 期望输出 | 实际输出 | 是否通过测试，请给出屏幕截图 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

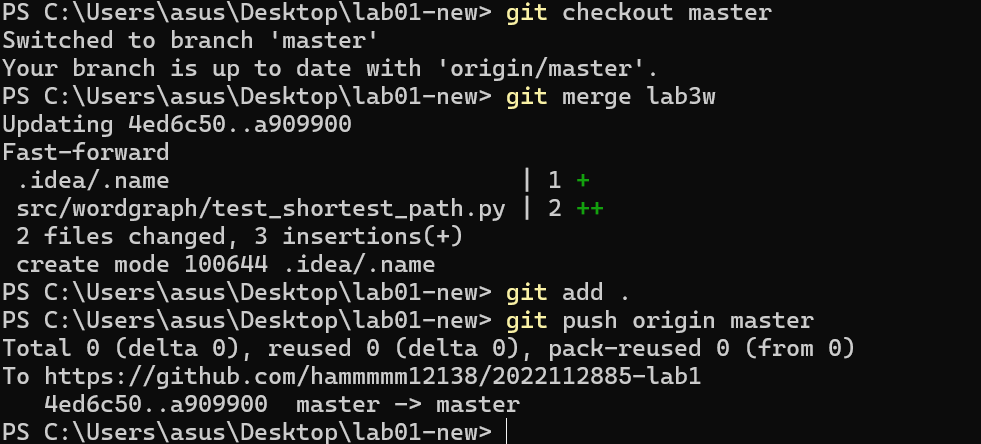
## Git操作记录

给出本地创建Lab3w分支，以及推送到Github上操作命令的截图；





给出本地合并Lab3w分支到master分支，以及推送到Github上操作命令的截图。



# 计划与实际进度

|  |  |  |  |
| --- | --- | --- | --- |
| 任务名称 | 计划时间长度（分钟） | 实际耗费时间（分钟） | 提前或延期的原因分析 |
| 代码评审 | 60 | 40 | 操作并不复杂 |
| 单元测试 | 120 | 180 | 原有的代码不完全适合进行单元测试，对单元测试的认知也不够全面，并且由于先前实验使用的是python代码，导致这里需要找python相关插件，使用了一些时间。 |
|  |  |  |  |

# 小结

在本次实验中，我学习与实践了代码评审和单元测试的工作。代码评审教会了我使用自带的工具进行初步的分析，对自己的代码风格进行提升，减少代码中隐含的bug有非常多的好处。单元测试则让我有机会实践黑盒测试和白盒测试的方法。从划分数据的等价类，到计算代码的路径，都使我能更好地理解代码检查的必要性。在这次实验中，我受益颇丰。