SUMMER RESEARCH INTERSHIP PRESENTATION

RAJ KARIYA
202103048

"REGRESSION TEST CASE GENERATION USING PRE-TRAINED LARGE LANGUAGE MODELS"

PROJECT INTRODUCTION

- Regression Test Case Generation refers to the process of creating and updating test cases specifically designed to ensure that new changes in a software application do not break or adversely affect existing functionality.

- The goal of regression testing is to verify that the software continues to perform as expected after updates, enhancements, or bug fixes.

PROJECT INTRODUCTION

- **Challenges:**
  - Manual test case generation is time-consuming and prone to human error.
  - Automated tools may generate test cases that are irrelevant or insufficient, leading to gaps in coverage.
  - Managing and maintaining a large number of test cases can be resource-intensive.

PROJECT OBJECTIVE

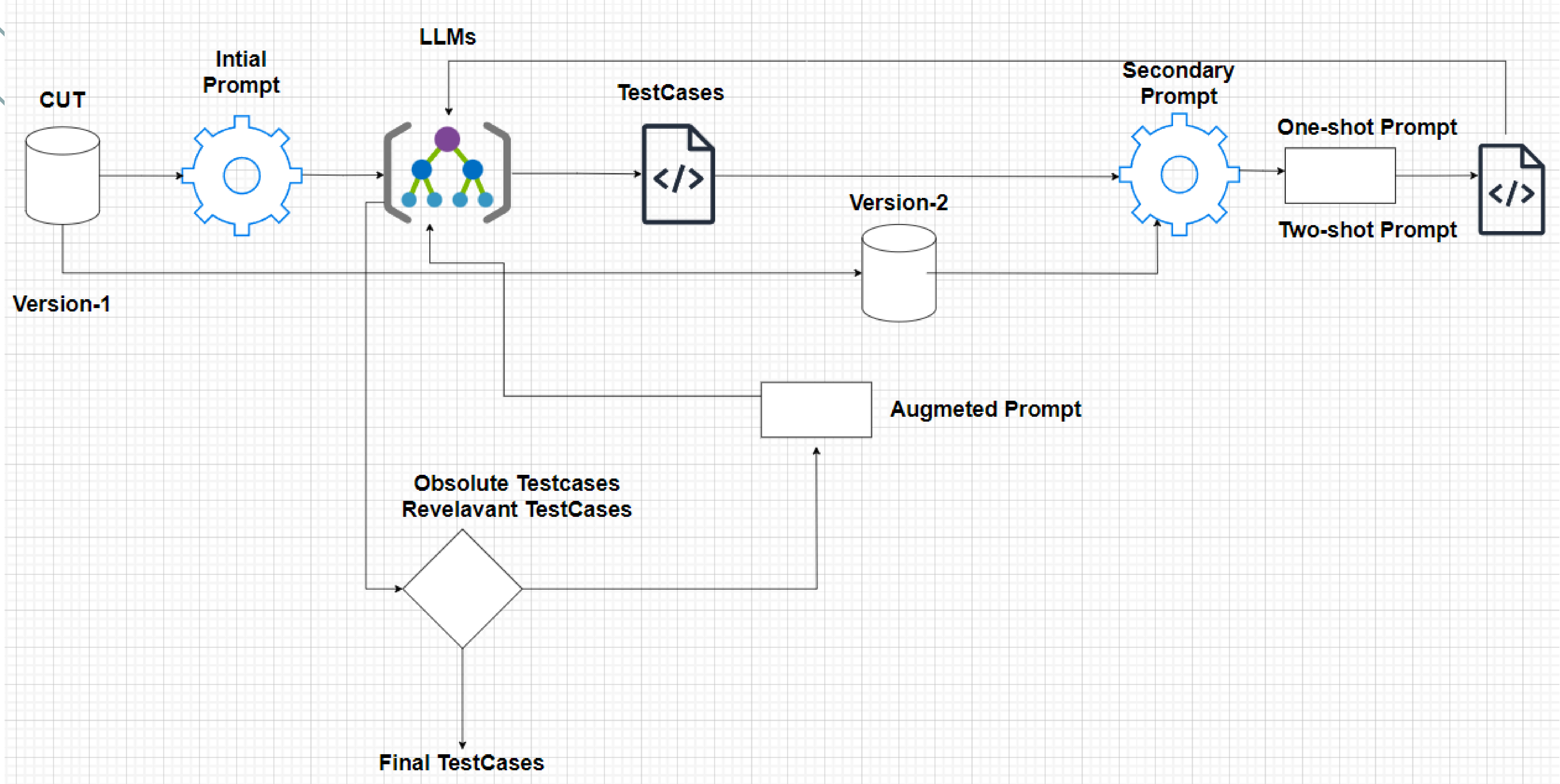
- Develop a concise Prompting Technique for better Testcase generation.
- Using Structral Comparsion of Control Flow graphs to differentiate between the different paths and branches of given code.
- Validation of the testcase through Control flow graph for Obsolute and revelant Testcases.

PROPOSED APPROACH

Combining LLMs and CFGs

- **LLMs (Gemini):** Automatically generate diverse and relevant test cases targeting new and modified code paths.
- **CFGs:** Analyze the structure of the code to identify critical paths and verify the relevance of generated test cases.
- **Develop Concise Prompt** for LLMs for Testcase genration for better Coverage.
- **Dual Verification Process:** Use CFGs to classify test cases as obsolete, relevant, or new, ensuring comprehensive coverage and as-well as checking with LLMS based response.

FLOWCHART REPRESENTATION



TERMINOLOGY

- Obsolete test cases are those that are no longer applicable or relevant because the parts of the code they were designed to test have been removed.

- Relevant test cases are those that remain valid and useful after code changes. They continue to correctly test the functionality of the software, including both unchanged and newly added code paths.

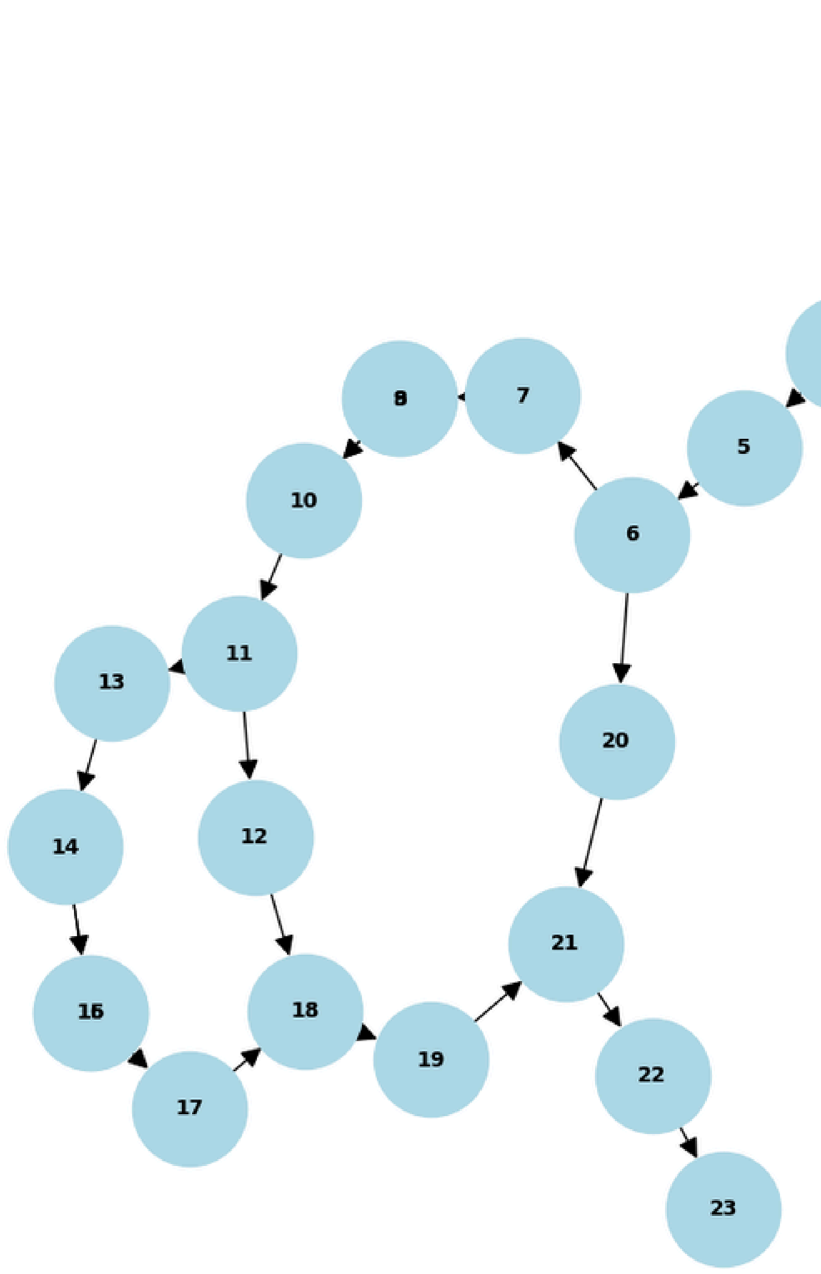
ROLE OF CONTROL FLOW GRAPH

Verification with CFGs of generated testcases

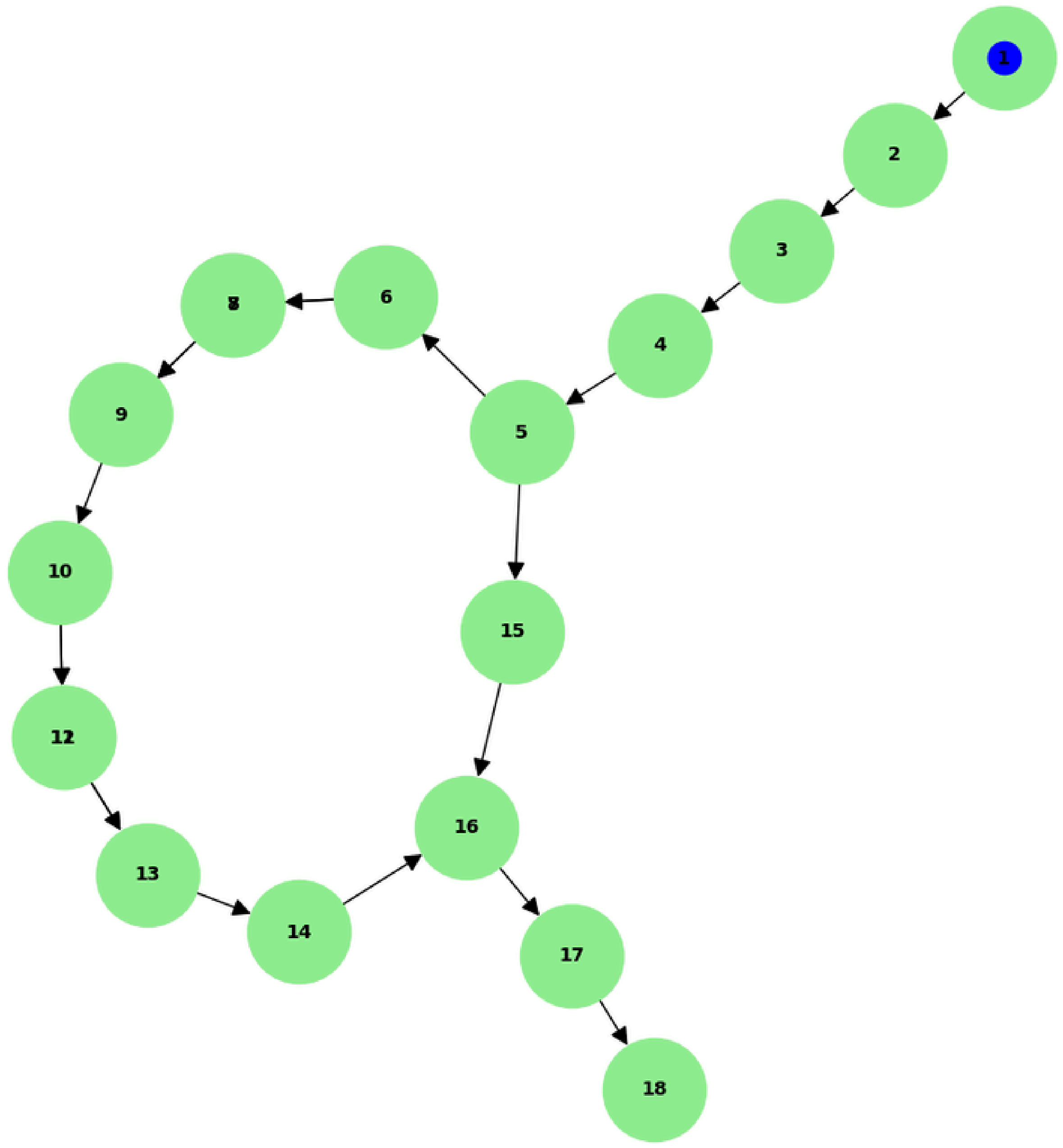
- Generate CFGs for both the original and modified code versions.
- Represent code blocks and control flow between them.
- Identify all possible execution paths within the code.
- Compare original and modified CFGs to detect changes.
- Map generated test cases to paths in these CFGs.
- Classify test cases as obsolete, relevant, or new.

DEMONSTRATION

CFG for Version 1 (Corrected)



CFG for Version 2 (Corrected)



DEMONSTRATION

```
Node Number to Syntax Mapping:
1: Equilateral = 0
2: Isosceles = 1
3: Scalene = 2
4: Right_Triangle = 3
5: entry
6: if a < b + c and b < a + c and (c < a + b)
7: if a ** 2 + b ** 2 == c ** 2 or b ** 2 + c ** 2 == a ** 2 or c ** 2 + a ** 2 ==
8: return Right_Triangle
9: else
10: join
11: if a == b == c
12: return Equilateral
13: else
14: if a == b or b == c or a == c
15: return Isosceles
16: else
17: join
18: join
19: return Scalene
20: else
21: join
22: return None
23: exit
Node Number to Syntax Mapping:
1: Equilateral = 0
2: Scalene = 1
3: Right_Triangle = 2
4: entry
5: if a < b + c and b < a + c and (c < a + b)
6: if a ** 2 + b ** 2 == c ** 2 or b ** 2 + c ** 2 == a ** 2 or c ** 2 + a ** 2 ==
7: return Right_Triangle
8: else
9: join
10: if a == b == c
11: return Equilateral
12: else
13: join
14: return Scalene
15: else
16: join
17: return None
18: exit
Number of independent paths for Version 1 (Corrected): 5
Number of independent paths for Version 2 (Corrected): 4
```

DEMONSTRATION

Orginal Version

```
regression_program_code_1 = """
Equilateral = 0
Isosceles = 1
Scalene = 2
Right_Triangle=3

def triangle_type(a, b, c):
  if a < b + c and b < a + c and c < a + b:
    if  a**2 + b**2 == c**2 or b**2 + c**2 == a**2 or c**2 + a**2 == b**2:
      return Right_Triangle
    if a == b == c:
      return Equilateral
    elif a == b or b == c or a == c:
      return Isosceles
    return Scalene
  return None
"""
```

Modified Version

```
regression_program_code_2 = """
Equilateral = 0
Scalene = 1
Right_Triangle=2

def triangle_type(a, b, c):
  if a < b + c and b < a + c and c < a + b:
    if  a**2 + b**2 == c**2 or b**2 + c**2 == a**2 or c**2 + a**2 == b**2:
      return Right_Triangle
    if a == b == c:
      return Equilateral
    return Scalene
  return None
"""
```

DEMONSTRATION

Orginal Version’s Testcase by LLM

```
self.assertEqual(triangle_type(5, 5, 5), Equilateral),
self.assertEqual(triangle_type(5, 5, 3), Isosceles),
self.assertEqual(triangle_type(3, 4, 5), Scalene),
self.assertEqual(triangle_type(3, 4, 5), Right_Triangle),
self.assertIsNone(triangle_type(1, 2, 5)),
self.assertIsNone(triangle_type(0, 3, 4)),
self.assertIsNone(triangle_type(-1, 3, 4)),
self.assertEqual(triangle_type(3.0, 4.0, 5.0), Right_Triangle),
self.assertEqual(triangle_type(1000, 1000, 1000), Equilateral),
self.assertEqual(triangle_type(0.1, 0.1, 0.1), Equilateral),
self.assertEqual(triangle_type(5, 3, 5), Isosceles),
self.assertEqual(triangle_type(5, 4, 3), Right_Triangle)
```

Modified Version Testcase by LLM

```
self.assertEqual(triangle_type(5, 5, 5), Equilateral)
self.assertEqual(triangle_type(3, 4, 5), Scalene)
self.assertEqual(triangle_type(3, 4, 5), Right_Triangle)
self.assertIsNone(triangle_type(1, 2, 5))
self.assertIsNone(triangle_type(0, 3, 4))
self.assertIsNone(triangle_type(-1, 3, 4))
self.assertEqual(triangle_type(3.0, 4.0, 5.0), Right_Triangle)
self.assertEqual(triangle_type(1000, 1000, 1000), Equilateral)
self.assertEqual(triangle_type(0.1, 0.1, 0.1), Equilateral)
self.assertEqual(triangle_type(5, 5, 3), Scalene)  # Isosceles but n
self.assertEqual(triangle_type(5, 3, 5), Scalene)  # Isosceles but n
self.assertEqual(triangle_type(5, 4, 3), Right_Triangle)
```

DEMONSTRATION

Two shot prompting

```
# Here is the Regression function:
{regression_program_code_1} and {regression_program_code_2}

# Task -1 : Understand the Context before Generating the test cases. :
  Obselete Test Cases: The Testcase which are not getting affected because of change in code.
  New Testcase : The Testcase which introduced by the updated version of the function.
  Regression Test Cases: The Testcase which include the new test cases that cover the updated functionality, as well as the existing test cases that are not obsolete.

# Task 2 - check by passing testcase in function wherther the following testcase passes all the conditions and expected value and acutal value are same or not.
# Task 3 - identify the obselete test cases that are not getting affected because of change.
# Task 4 - identify the Revelant test cases.
# Task 5- = Now generate new test cases according to modified version and check manually too in detail calculation and also append to orginal passed testcases if it passed manually in task -1.
# meniton under **Complete Set of Test Cases:**
# Note : format of testcase shoud be assert funtion_name(paramters)..
"""
```

```
## Task 3: Obselete Test Cases

* `self.assertEqual(triangle_type(5, 5, 3), Isosceles)`
* `self.assertEqual(triangle_type(5, 3, 5), Isosceles)`

## Task 4: Relevant Test Cases

* `self.assertEqual(triangle_type(5, 5, 5), Equilateral)`
* `self.assertEqual(triangle_type(3, 4, 5), Scalene)`
* `self.assertEqual(triangle_type(3, 4, 5), Right_Triangle)`
* `self.assertIsNone(triangle_type(1, 2, 5))`
* `self.assertIsNone(triangle_type(0, 3, 4))`
* `self.assertIsNone(triangle_type(-1, 3, 4))`
* `self.assertEqual(triangle_type(3.0, 4.0, 5.0), Right_Triangle)`
* `self.assertEqual(triangle_type(1000, 1000, 1000), Equilateral)`
* `self.assertEqual(triangle_type(0.1, 0.1, 0.1), Equilateral)`
* `self.assertEqual(triangle_type(5, 4, 3), Right_Triangle)`
```

DEMONSTRATION

Finding Obsolute and Revelant using CFGs

```python
# Find obsolete test cases
obsolete_cases = []
for case in test_cases:
    if path_map_v1[case] != path_map_v2[case]:
        obsolete_cases.append(case)

obsolete_cases
```

```
[(5, 5, 3), (5, 3, 5)]
```

```python
# Compare paths
Relavant_cases = []
for case in test_cases:
    if path_map_v1[case] == path_map_v2[case]:
        Relavant_cases.append(case)

print("Relavant test cases:")
for case in Relavant_cases:
    print(case)
```

```
Relavant test cases:
(5, 5, 5)
(3, 4, 5)
(3, 4, 5)
(1, 2, 5)
(0, 3, 4)
(-1, 3, 4)
(3.0, 4.0, 5.0)
(1000, 1000, 1000)
(0.1, 0.1, 0.1)
(5, 4, 3)
```

CONCLUSION AND FUTURE WORK

- **Effective Integration:** LLMs and CFGs together provide a robust framework for regression test case generation.
- Leveraging the **potential of LLMs** and Concise promts for better Testcases.
- **Improvements:** Enhanced coverage, efficiency, and accuracy in regression testing.

FUTURE WORK

- **Future Directions:**
  - **Scalability:** Extend the approach to handle larger and more complex codebases.
  - **Advanced Techniques:** Explore integration with dynamic analysis and more sophisticated LLM models for further improvement.
  - Extend the study to compare the performance of different LLMs (e.g., **GPT-4, BERT, T5**) in generating test cases and fine tune LLMs on domain-specific codebases to improve the relevance and accuracy of generated test cases for particular industries
  - Provide insights into which models are best suited for specific types of code or testing scenarios.

THANK YOU

https://colab.research.google.com/drive/ 19HoNcfckq2vcKBdp6zcWWSZnwrdlNc4O?usp= sharing