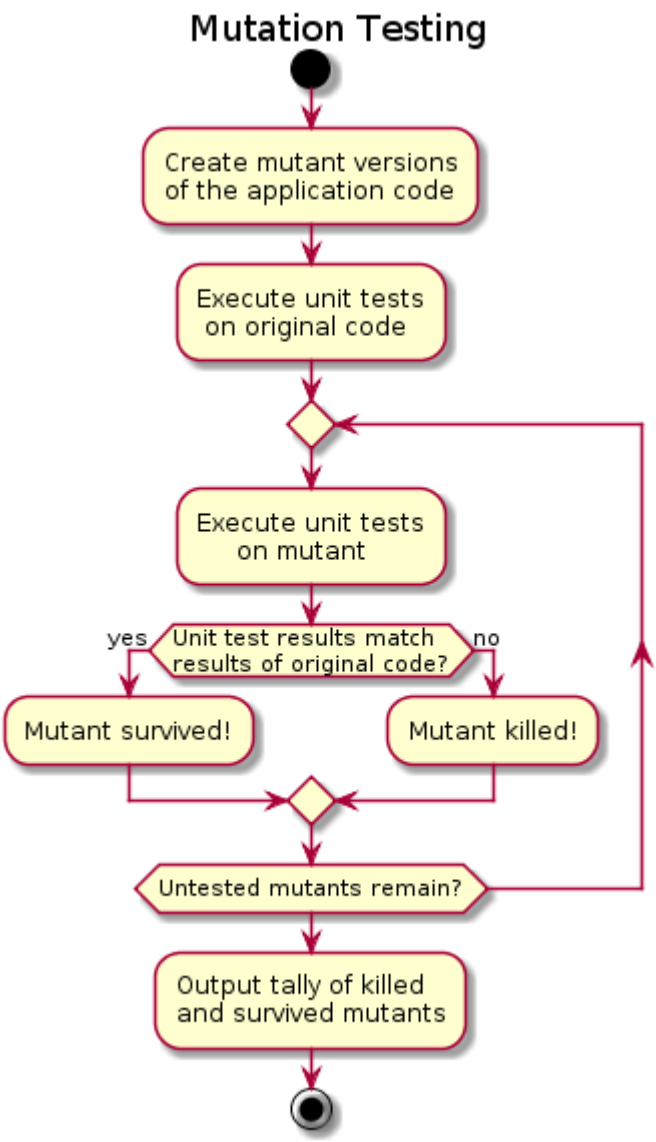
CSE 731: Software Testing - Project Work

Team Members

Name	Roll Number
Abhinav Kumar	IMT2022079
Aaditya Joshi	IMT2022092

Introduction to Mutation Testing



Mutation testing is a software testing technique that evaluates the quality and effectiveness of a test suite by deliberately introducing small changes called *mutations* into the program's source code. These mutations simulate common programming errors such as changed operators, incorrect conditions, or altered constants. The modified versions of the program, known as *mutants*, are then executed against the existing tests.

If the test suite detects the change and fails appropriately, the mutant is considered **"killed."** If the test suite passes despite the injected defect, the mutant **"survives,"** indicating a potential weakness or gap in the tests.

What Mutation Testing Is Used For

- **Assessing test-suite strength:** It provides a more rigorous measure of test quality than code coverage alone.
- **Revealing weak or missing test cases:** Surviving mutants highlight areas where tests aren't sufficiently validating behavior.
- **Encouraging better test design:** Developers gain insights into how to write more precise, meaningful, and robust tests.
- **Improving software reliability:** By ensuring tests can catch realistic errors, mutation testing increases confidence in the system's correctness.
- **Benchmarking testing tools and strategies:** Useful in research and engineering teams evaluating different test approaches.

Types of Mutants

In mutation testing, mutants are classified into several behavioral categories that describe how they behave during generation and testing. The four primary types are:

---

1. Stillborn Mutants

Stillborn mutants are mutants that **cannot run at all** because the mutation introduces invalid code.

- They fail to compile or cause syntax errors.
- They are discarded before execution.
- They do not contribute to the mutation score.

---

2. Trivial Mutants

Trivial mutants are mutants that are **very easy to kill**, typically because the mutation causes an obvious and immediate change in program behavior.

- Even simple or minimal tests can detect them.
- They provide limited insight into overall test suite strength.

---

3. Equivalent Mutants

Equivalent mutants are mutants that **behave exactly like the original program for all possible inputs**, despite the syntactic change.

- They **cannot** be killed by any test.
- They increase computational cost but provide no value.
- Detecting them automatically is difficult and sometimes undecidable.

---

4. Dead Mutants

Dead mutants (also called **killed mutants**) are mutants that the test suite **successfully detects**, meaning at least one test fails when executed against this mutant.

- They are the desirable outcome in mutation testing.
- A high number of dead mutants indicates a strong and effective test suite.

Strong and Weak Killing in Mutation Testing

In mutation testing, the terms **strong killing** and **weak killing** describe how effectively a test suite detects the effects of a mutation.

Weak Killing

Weak killing occurs when the effect of a mutation is detected **at the point where the mutated statement is executed**, regardless of whether this difference influences the program's final output. In other words, the mutation causes an observable deviation in the program's internal state during execution, but this deviation does not necessarily propagate to the test's final assertion.

Weak killing is primarily concerned with **state infection** whether the mutation changes the internal values or control flow immediately after the mutated operation. It indicates that the test suite reaches and exercises the mutated code, but it does not guarantee that the test would fail in a real execution scenario.

Strong Killing

Strong killing requires that the mutation not only affect the program's state but also **propagate to an externally observable outcome** such as a function's return value, printed output, or a value checked by a test assertion. A mutant is considered strongly killed when this observable difference causes an actual test failure.

Strong killing therefore captures the full chain of mutation effects:

1. **Reachability** – the mutated code is executed.
2. **Infection** – the mutation alters the program state.
3. **Propagation** – the altered state leads to a difference in the final observable behavior.
4. **Detection** – the test identifies the difference and fails.

Because strong killing reflects the real-world ability of tests to detect faults, it is the standard used in most practical mutation testing tools.

Project Overview

This project applies **mutation testing** to a large Python codebase (~1000+ LOC) consisting of algorithmic, mathematical, data-structure, utility, and integration modules.

The goal is to demonstrate:

- Strongly killing mutants at both **unit** and **integration** levels
- Use of **multiple mutation operators**
- Automated mutation analysis using open-source tools
- Clear mapping of test cases to mutation-based adequacy criteria

The codebase implements multiple complete functionalities spread across modules such as:

- Geometry computations (`geometry.py`)
- Banking operations (`banking.py`)
- Graph algorithms (`graph_algos.py`)
- Data-structure manipulation (`data_structures.py`)
- Searching, sorting, statistics, matrix operations
- End-to-end workflows in integration modules (`integration.py` , `integration_2.py` , `integration_3.py`)
- matrix operations (`matrix_ops.py`)
- search algorithms (`search_algos.py`)
- sorting algorithms (`sorting_algos.py`)
- string operations (`string_utils.py`)
- statistics (`stats_lib.py`)

- utility functions (`utils.py`)
- set operations (`set_ops.py`)

folder	files	code
src	15	1388
tests	15	1095

$$\text{Mutation Score} = \frac{\text{Killed Mutants} + \text{Timeout Mutants}}{\text{Total Mutants}} \times 100\%$$

$$\text{Mutation Score} = \frac{M_k + M_{\text{to}}}{M_t} \times 100\%$$

Repository Link

A full version of the project and test suite is available at:

https://github.com/Abhinav-Kumar012/ST_project.git

This repository contains:

- The `src/` and `tests/` folders
- Mutation testing configuration and scripts
- HTML reports for unit and integration mutation testing

Source Code Description

The `src/` directory contains the following modules:

- **geometry.py** — distance, area, polygon checks, slope calculations
- **banking.py** — account operations, interest calculations, validations
- **graph_algos.py** — BFS, DFS, topological sort, shortest-path utilities
- **data_structures.py** — stack, queue, linked list primitives

- **matrix_ops.py** — matrix addition, multiplication, transformations
- **sorting_algos.py** — merge, insertion, quick sort
- **search_algos.py** — linear search, binary search
- **set_ops.py** — set union, intersection, difference
- **string_utils.py** — snake case, camel case, palindrome, anagram
- **stats_lib.py** — mean, median, mode, variance computations
- **utils.py** — general helper functions
- **integration.py**, **integration_2.py**, **integration_3.py** — multi-module workflows combining utils, geometry, banking, stats_lib, string_utils, sorting_algos, search_algos, graph_algos, matrix_ops, set_ops

The integration modules are used specifically for **integration-level mutation testing**, as required in the project statement .

Test Case Design Strategy (Mutation Testing)

We implemented a mutation-based test strategy for **unit testing** and **integration testing**, satisfying the project requirement of using mutation operators at both levels with strong mutant killing .For **unit testing**, we have tried using node coverage (99%).

Code coverage

code coverage report

Unit-Level Mutation Operators Used

1. **Statement Deletion (SDL):** Deletes a statement to test if the missing logic is detected.
2. **Logical Connector Replacement (LCR):** Replaces logical connectors (e.g., `and` with `or`).
3. **Conditional Operator Deletion / Negation (COD):** Deletes a conditional operator.

4. **Conditional Operator Insertion (COI):** Adds Not statements to invert consitional statements.

Integration-Level Mutation Operators Used

1. **Statement Deletion (SDL):** Deletes a statement to test if the missing logic is detected.
2. **Logical Connector Replacement (LCR):** Replaces logical connectors (e.g., `and` with `or`).
3. **Relational Operator Replacement (ROR):** Replaces relational operators (e.g., `==` with `!=`).
4. **Conditional Operator Insertion (COI):** Adds Not statements to invert consitional statements.

Results

name	killed	timeout	survived	total	percentage
integration	86	6	19	111	82.88
unit	746	130	62	938	93.39
total	832	136	81	1049	92.28

Tooling: `mut.py`

We are using `mut.py` for performing mutation testing. The `commands.sh` script demonstrates its usage.

Setup

create a venv in the root directory and install the dependencies using the following command:

```
python3 -m venv .venv

source .venv/bin/activate

pip install -r requirements.txt
```

Usage

for indivisual module's report genration

```
mut.py --target src.utils --unit-test tests.test_utils_2 --operator SDL L
```

for full project report genration

```
bash commands.sh
```

Explanation of Flags

- `--target` : The module to mutate (source code).
- `--unit-test` : The test module to run against the mutants.
- `--operator` : Specific mutation operators to apply.
- `--report-html [DIR]` : Generates an HTML report.
- `--path .` : Adds the current directory to the Python path.

Individual Contributions

Team Member	Contribution Details
Abhinav Kumar	unit testing for banking, data_structures, graph_algos, matrix_ops,search_algos, utils, geometry,set_ops
Aaditya Joshi	integration testing and unit testing for sorting_algos, stats_lib, string_utils

AI Disclosure

AI was used in this project to implement the following:

- Ensure we had node coverage in unit testing.
- Added conditional statements in integration testing for increasing logical mutant count.