Інструментальні засоби розробки програмного забезпечення

Лабораторна робота №1

Мета: Опанувати принципи та підходи модульного тестування шляхом розробки та впровадження unit-тестів для наявного програмного коду, забезпечити перевірку коректності роботи окремих функцій, підвищити надійність, підтримуваність і якість програмного забезпечення.

Ювженко Назарій Олександрович ІПС-21

Теоретичні відомості:

Модульне тестування — це метод тестування програмного забезпечення, який перевіряє найменші окремі частини коду, такі як функції або класи, ізольовано одна від одної

Google Test – це фреймворк для модульного тестування коду мовою C++, розроблений Google. Він дозволяє створювати та запускати тести для перевірки окремих частин програми, використовуючи макроси для визначення тестів, тверджень та фікстур.

Опис вхідного коду:

Графи на основі списку суміжності, матриці суміжності (збереження даних у вершинах та ребрах графів). Додавання та видалення вершин/ребер. Перевірка на зв’язність графу. Визначення відстані між двома вершинами графу. Збереження графу в файл.

Розробка unit-тестів:

Для створення тестів використовувався Google Test як основний фреймворк тестування

Для гарантії незмінності вхідного коду було створено допоміжний клас, дочірній основному, TestableGraph. До цього класу було додано деякі функції, модернізовані через особливості тестів.

```
class TestableGraph : public Graph {
public:
    using Graph::DFS;
    using Graph::setVert;
    using Graph::setDir;
    using Graph::getMatrix;
    using Graph::isConnected;
    using Graph::addEdge;
    using Graph::removeEdge;
    using Graph::distanceBetween;
    using Graph::isStronglyConnected;
    using Graph::getGraphData;
    using Graph::listToFile;
    using Graph::matrixToFile;

    vector<vector<int>>& data() { return graph; }
    void setMatrix(bool m) { matrix = m; }
    void setData(const vector<vector<int>>& g) { graph = g; }

    bool checkVertices(int& u, int& v) {
        cin >> u >> v;
        if (cin.fail()) {
            cin.clear();
            cin.ignore(32767, '\n');
            cout << "Invalid vertex!\n";
            return false;
        }
        if (u == -1 && v == -1)
            return false;
        if (u < 0 || u >= vertices || v < 0 || v >= vertices) {
            cout << "Invalid vertex!\n";
            return false;
        }
        return true;
    }

    TestableGraph getTranspose() {
        TestableGraph gT;
        gT.setVert(vertices);
        gT.setDir(directed);
        if (matrix) {
            gT.matrix = true;
            gT.graph.assign(vertices, vector<int>(vertices, 0));
            for (int i = 0; i < vertices; i++)
                for (int j = 0; j < vertices; j++)
                    if (graph[i][j])
                        gT.graph[j][i] = 1;
        }
        else {
            gT.matrix = false;
            gT.graph.resize(vertices);
            for (int i = 0; i < vertices; i++)
                for (int u : graph[i])
                    gT.graph[u].push_back(i);
        }
        return gT;
    }
};
```

В ході тестування було використано можливість зміни буфері std::cin та std::cout для іммітації вводу користувача, та запобігання виводу повідомлень тестованими функціями в концоль і перевірки вмісту цих повідомлень

```
stringstream dummy;
streambuf* orig_cout = cout.rdbuf(dummy.rdbuf());

stringstream input("0 1\n");
streambuf* orig_cin = std::cin.rdbuf(input.rdbuf());
g.addEdge();

cin.rdbuf(orig_cin);
cout.rdbuf(orig_cout);
```

```
stringstream dummy;

streambuf* orig_cout = cout.rdbuf(dummy.rdbuf());
g.isStronglyConnected();
cout.rdbuf(orig_cout);

string output = dummy.str();
EXPECT_NE(output.find("Graph is Connected"), string::npos);
```

Усього було додано 38 тестових сценаріїв, які охоплюють наступні напрямки:

1) Операції з ребрами (Edge Manipulation)
   - Додавання ребер (орієнтовані/неорієнтовані, петлі, дублікати)
   - Видалення ребер (включно з self-loop та ребрами, що не існують)
   - Перевірка консистентності внутрішнього стану графа після змін

2) Перевірка введення користувача (Input Validation)
   - Тести на коректне введення, вихід за межі, текстове введення
   - Перевірка наявності повідомлень про помилки у cout
   - Контроль відсутності побічних ефектів на граф

3) Обхід графа (Traversal)
   • DFS для матриці та списку суміжності
   • Перевірка коректного покриття лише досяжних вершин

4) Перевірка зв’язності (Connectivity)
   • isConnected() та isStronglyConnected()
   • Випадки: без ребер, без вершин, часткова зв’язність
   • Перевірка відповідних повідомлень у cout

5) Пошук найкоротшої відстані (Distance Between)
   • Перевіряються існуючі та недосяжні маршрути
   • Контроль коректних повідомлень у cout

6) Збір параметрів графа (Graph Configuration Input)
   • Читання кількості вершин та орієнтації
   • Стійкість до некоректного текстового вводу

7) Транспонування графа (Transpose Operation)
   • Правильність перенесення напрямків
   • Перевірка логіки для матриці та списку
   • Врахування self-loop

8) Виведення графа (Printing)
   • Перевірка форматованого виводу adjacency list та adjacency matrix

9) Збереження у файл (Data Export)
   • Тести для listToFile() та matrixToFile()
   • Перевірка структури файлу та коректного заголовку

10) Інтеграційні тести

• Додавання ребер → перевірка зв’язності

• Емуляція реального використання

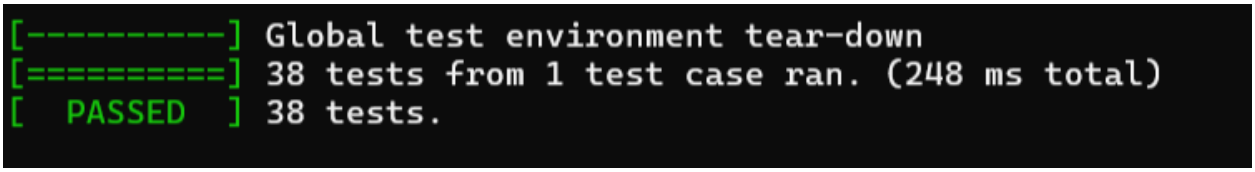
11)Performance-тести

• Перевірка швидкості роботи на великих графах

• Smoke-тест без ризику stack overflow

Усі тести успішно проходять та забезпечують:

- Повне базове покриття логіки Graph
- Коректність поведінки в різних режимах
- Перевірку граничних випадків некоректного вводу
- Надійність структури даних
- Можливість подальшого масштабування функціоналу



Висновок: вході цієї лабораторної роботи я опанував принципи та підходи модульного тестування шляхом розробки та впровадження unit-тестів для наявного програмного коду

Посилання:

Код на github

Pull request на github

Документація google test