CSE 464: Software QA and Testing - Project Part 1

Author: Saurabh Dingwani

GitHub Repo: https://github.com/sdingwan/CSE-464-2024-sdingwan.git

Table of Contents

1. Project Overview

This project implements a Java application that reads, manipulates, and outputs graphs in the DOT format. The project includes the following features:

- Parsing a DOT graph.
- Adding nodes and edges to the graph.
- Exporting the graph to DOT and PNG formats.
- Writing unit tests to validate the features.

2. Project Setup

Prerequisites:

- Java JDK 11+
- Maven
- Git
- Graphviz

Setup Instructions:

Clone the Repository
Navigate to the Project Directory
Ensure Maven and Java are installed
Install Required Libraries

```
mvn clean install
```

3. Instructions to Run the Code

To compile and run the code, execute the following command:

```
mvn package
```

This will compile the project and run the tests. The project uses JUnit for testing and Maven for managing dependencies and building the application.

To run the tests, use the command:

```
mvn test
```

The DOT file input/output methods can be executed via the following:

Parse a Graph:

```
GraphParser parser = new GraphParser();
parser.parseGraph("path/to/sample.dot");
```

Add Nodes:

```
parser.addNode("A");
```

Add Edges:

```
parser.addEdge("A", "B");
```

Export to DOT Format:

```
parser.outputDOTGraph("path/to/output.dot");
```

Export to PNG Format:

```
parser.outputGraphics("path/to/output.png", "png");
```

4. Test Execution

JUnit tests are provided for each feature. After running `mvn test`, the results of the test cases will be shown in the console.

5. Expected Outputs

Feature 1: Parsing a DOT Graph

- Input: A graph in DOT format.
- Output: Graph details such as number of nodes and edges, and the graph representation.

Feature 2 and 3: Adding Nodes and Edges

- Input: A set of nodes and edges.
- Output: The graph with the new nodes and edges.

Feature 4: Export to DOT and PNG

- Input: A graph structure.
- Output: DOT and PNG files.

6. GitHub Commits and Branches

Each feature is committed to GitHub as follows:

- **Feature 1: Parse DOT Graph**: https://github.com/sdingwan/CSE-464-2024-sdingwan/commit/ef52a47bee151eb4789d25db78a028a34b72a3e2
- **Feature 2: Add Nodes**: https://github.com/sdingwan/CSE-464-2024-sdingwan/commit/736874515c679519d6459ede00ba97a02478ee7f
- **Feature 3: Add Edges**: https://github.com/sdingwan/CSE-464-2024-sdingwan/commit/00a4b319e0ea95f7b54a62f24e430fec450cb09e

- **Feature 4: Output to DOT/PNG**: https://github.com/sdingwan/CSE-464-2024-sdingwan/commit/0b39af6e3f10c08fa009e1feb60b6c73ca49fbb5

You can find the main branch and continuous integration here.

https://github.com/sdingwan/CSE-464-2024-sdingwan/commits/main/

GraphParserTest to run the code

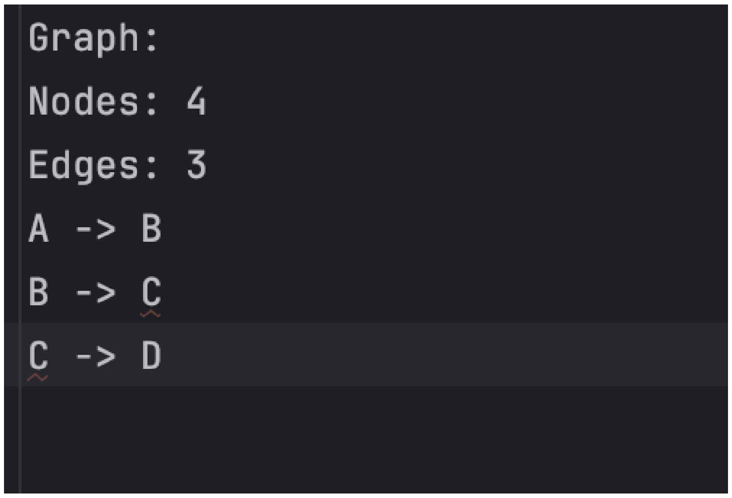
```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import static org.junit.jupiter.api.Assertions.*;


public class GraphParserTest {


   private GraphParser parser;


   // Setup method to initialize the GraphParser object before each test
   @BeforeEach
   public void setUp() {
       parser = new GraphParser();
   }


   // Test Feature 1: Parsing a DOT graph file
   @Test
   public void testParseGraph() throws IOException {
       parser.parseGraph("src/test/resources/sample.dot"); // Input DOT file


       String expectedOutput = Files.readString(Paths.get("src/test/resources/expected-output.txt")).trim();
       String actualOutput = parser.toString().trim().replaceAll("\\s+", " ").replaceAll("\r\n", "\n").replaceAll("\n", " ").trim();


       System.out.println("Expected: \n" + expectedOutput);
       System.out.println("Actual: \n" + actualOutput);


       assertEquals(expectedOutput.replaceAll("\\s+", " ").replaceAll("\r\n", "\n").replaceAll("\n", " ").trim(), actualOutput, "The outputs should match when formatted uniformly.");
   }




   // Test Feature 2: Adding a single node
   @Test
   public void testAddNode() {
       parser.addNode("A");
       parser.addNode("B");
       parser.addNode("A");  // Adding duplicate node


       // Check that the number of nodes is 2 (no duplicates)
       assertEquals(2, parser.getGraph().vertexSet().size());
```

```java
}


// Test Feature 2: Adding a list of nodes

@Test

public void testAddNodes() {

    String[] nodes = {"A", "B", "C"};

    parser.addNodes(nodes);


    // Check that all nodes were added

    assertEquals(3, parser.getGraph().vertexSet().size());

}


// Test Feature 3: Adding an edge

@Test

public void testAddEdge() {

    parser.addNode("A");

    parser.addNode("B");

    parser.addEdge("A", "B");

    parser.addEdge("A", "B");  // Adding duplicate edge


    // Check that only 1 edge was added (no duplicates)

    assertEquals(1, parser.getGraph().edgeSet().size());

}


// Test Feature 4: Output the graph to DOT file

@Test
```

```
    public void testOutputDOTGraph() throws IOException {

        // Adding nodes

        parser.addNode("A");

        parser.addNode("B");

        parser.addNode("C");

        parser.addNode("D");


        // Adding edges

        parser.addEdge("A", "B");

        parser.addEdge("B", "C");

        parser.addEdge("C", "D");


        // Output the graph to a DOT file

        String outputPath = "src/test/resources/output.dot";

        parser.outputDOTGraph(outputPath);


        // Read the expected and actual DOT file content

        String expectedDOT =
Files.readString(Paths.get("src/test/resources/expected.dot"));

        String actualDOT = Files.readString(Paths.get(outputPath));


        // Normalize newline characters across different environments

        expectedDOT = expectedDOT.replace("\r\n", "\n");

        actualDOT = actualDOT.replace("\r\n", "\n");


        // Assertion to check if the contents are the same
        assertEquals(expectedDOT, actualDOT, "The DOT files should match
expected structure.");

    }




    // Test Feature 4: Output the graph to PNG file

    @Test

    public void testOutputGraphics() throws IOException {

        parser.addNode("A");

        parser.addNode("B");

        parser.addEdge("A", "B");


        // Output the graph to a PNG file

        parser.outputGraphics("src/test/resources/output.png", "png");


        // Verify that the PNG file was created

        File outputFile = new File("src/test/resources/output.png");

        assertTrue(outputFile.exists());

    }


    // Additional Test: Ensure correct graph string representation (toString)

    @Test

    public void testGraphToString() {

        parser.addNode("A");

        parser.addNode("B");

        parser.addEdge("A", "B");
        String expectedString = "Graph: \nNodes: 2\nEdges: 1\nA -> B\n";

        assertEquals(expectedString, parser.toString());

    }

}
```

7. Expected Output Screenshots (use sample.dot as input file)

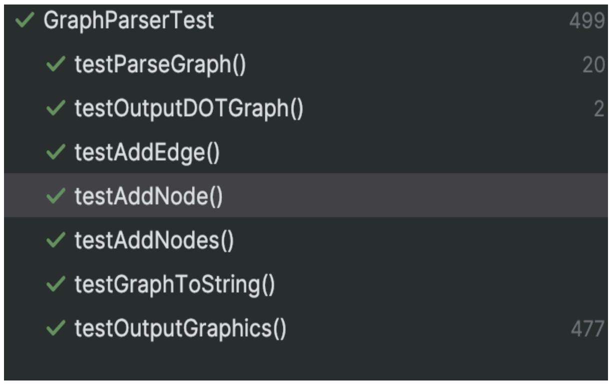
Feature 1: Parsed Graph



```
Graph:
Nodes: 4
Edges: 3
A -> B
B -> C
C -> D
```

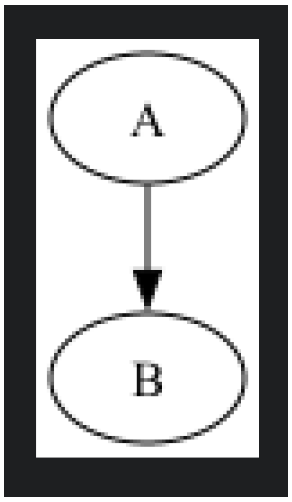
Feature 2 and 3: Added Nodes and edges

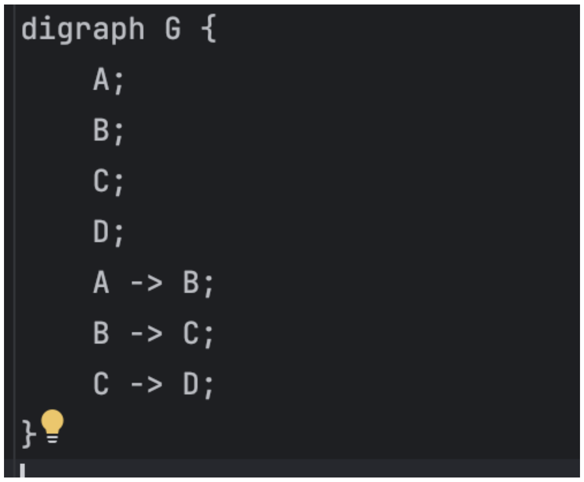
This feature does not specify that we have to produce an output here. It just tells us to add nodes and edges. It does this as expected as it passes the unit test.



```
✓ GraphParserTest          499
  ✓ testParseGraph()        20
  ✓ testOutputDOTGraph()     2
  ✓ testAddEdge()
  ✓ testAddNode()
  ✓ testAddNodes()
  ✓ testGraphToString()
  ✓ testOutputGraphics()   477
```

Feature 4: Export to PNG and DOT





```
digraph G {
    A;
    B;
    C;
    D;
    A -> B;
    B -> C;
    C -> D;
}
```