CSE 464: Software QA and Testing

Project Part #3

Sabthagirivasan Vellingiri

1225471286

svellin3@asu.edu

Introduction:

Part 3 of the project was built using **IntelliJ** as the IDE, **Java 8** as the Java Development Kit, **Maven** for Dependency Manager, and the **graphviz-java** library shared by the professor. The **graphviz-java** was used to parse the **.dot** file into objects in Java and vice versa. In this phase 3, new classes have been created to implement different types of graph searcher algorithms such as BFS, DFS, and Random Walk using the Template design pattern, Strategy design pattern, and Factory design pattern.

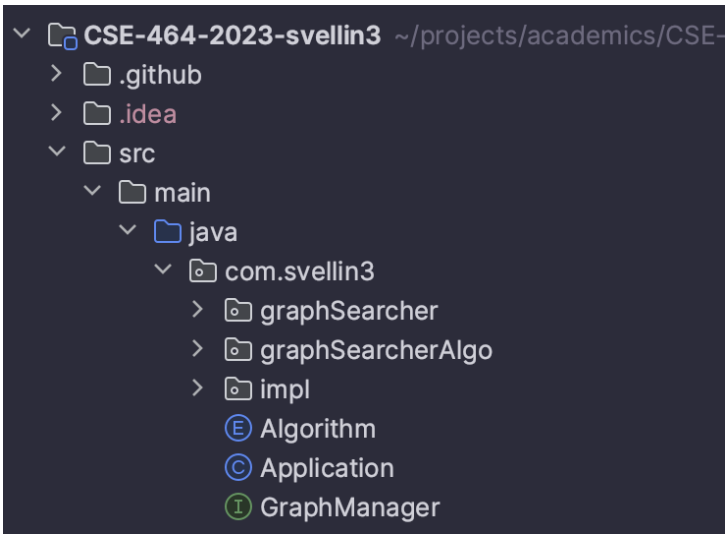
If there are any dependency issues, please resolve them using the **mvn package** in the command line terminal or resolve them using maven in IntelliJ.

All the features required in the project are added into an Interface named "**GraphManager.java**" which is in the folder: **src/main/java/com/svellin3.** The implementation of all these features is implemented in **"GraphManagerImpl.java"** which resides in the folder: **src/main/java/com/svellin3/impl.**

GitHub Link: **https://github.com/sabthagirivasanv/CSE-464-2023-svellin3.git**

How to run the application:

To run the application, run the main function that resides in the **Application.java** which can be located inside the folder: **src/main/java/com/svellin3**



The Command Line Terminal can be used to interact with the application. Initially, the application will ask for a **.dot** file to parse the graph. I have added the new input file: **input2.dot** shared by the professor in the project folder.

```
please enter the dot file to parse the graph:
input2.dot
```

Once the dot file is parsed, the application will display a list of options to perform on the parsed graph. The screenshot of the options is attached below.

```
press the below options to perform actions:
1. print the graph
2. output to file
3. add a new node
4. add a list of new nodes
5. remove a node
6. remove a list of nodes
7. add an edge
8. remove an edge
9. output as DOT graph
10. output into graphics
11. search nodes by BFS
12. search nodes by DFS
13. search nodes by Random Walk
99. exit
```

You can manipulate the structure of the graph using the options developed in Phase-1.

Option 11: Search the path from source to destination using BFS

Now, enter the source and destination nodes between which you want to find the path using BFS. Our algorithm will return the optimum path using the BFS algorithm. If no such path exists, it will return null.

```
11
Please enter the source node:
a
Please enter the destination node:
h
Searching path from a to h using algo: BFS
Visiting Path : a
Visiting Path : a -> b
Visiting Path : a -> e
Visiting Path : a -> b -> c
Visiting Path : a -> e -> f
Visiting Path : a -> e -> g
Visiting Path : a -> b -> c -> d
Visiting Path : a -> e -> f -> h
The path : a -> e -> f -> h
The path is :
a -> e -> f -> h
```

Option 12: Search the path from source to destination using DFS

Now, enter the source node and destination node between which you want to find the path using DFS. Our algorithm will return the optimum path using the DFS algorithm. If no such path exists, it will return null.

```
12
Please enter the source node:
a
Please enter the destination node:
h
Searching path from a to h using algo: DFS
Visiting Path : a
Visiting Path : a -> b
Visiting Path : a -> b -> c
Visiting Path : a -> b -> c -> d
Visiting Path : a -> e
Visiting Path : a -> e -> f
Visiting Path : a -> e -> f -> h
The path : a -> e -> f -> h
The path is :
a -> e -> f -> h
```

Option 13: Search the path from source to destination using Random Walk

Enter the source and destination nodes between which you want to find the path using DFS. Our algorithm will search for the source to the destination using a random walk algorithm. In the end, it will print the path found from the source to the destination. If no such path exists, it will return null.

Here, I attached screenshots of the search operation from **node A** to **node C** using RandomWalk Algorithm**.** In the first run and second run, we can clearly the different visiting paths when searching from source to destination.

Run 1:

```
13
Please enter the source node:
a
Please enter the destination node:
c
Searching path from a to c using algo: RANDOM_WALK
Visiting Path : a
Visiting Path : a -> b
Visiting Path : a -> e
Visiting Path : a -> e -> f
Visiting Path : a -> e -> f -> h
Visiting Path : a -> b -> c
The path : a -> b -> c
The path is :
a -> b -> c
```

Run 2:

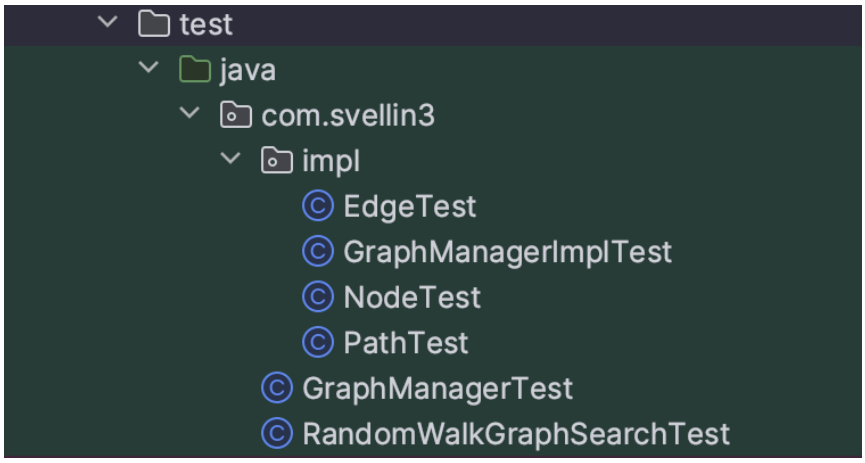
```
13
Please enter the source node:
a
Please enter the destination node:
c
Searching path from a to c using algo: RANDOM_WALK
Visiting Path : a
Visiting Path : a -> b
Visiting Path : a -> b -> c
The path : a -> b -> c
The path is :
a -> b -> c
```

How to run the Test:

All the features required in the project are added into an Interface named "**GraphManager.java**" which is in the folder: **src/main/java/com/svellin3.** The implementation of all these features is implemented in **"GraphManagerImpl.java"** which resides in the folder: **src/main/java/com/svellin3/impl.**

The unit test cases were written for all the features implemented in all three parts of the project. The test class is added inside the folder "**src/test/java/com/svellin3**"**.** All the required features for all three parts of the project are exclusively tested in the java file named "**GraphManagerTest.java**" which is located in the folder "**src/test/java/com/svellin3**". All other internal functions which are essential in achieving the required functionalities are tested in separate classes which reside inside **src/test/java/com/svellin3/impl**.

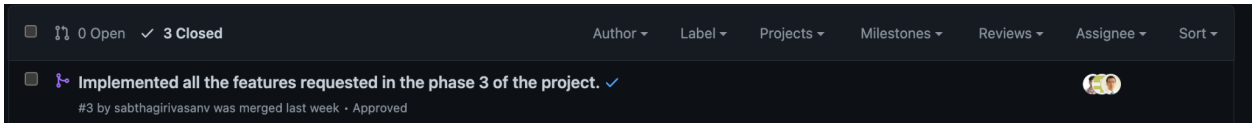
The test cases can be run using **IntelliJ** or **mvn test**. I am hereby attaching the folder structure for reference.



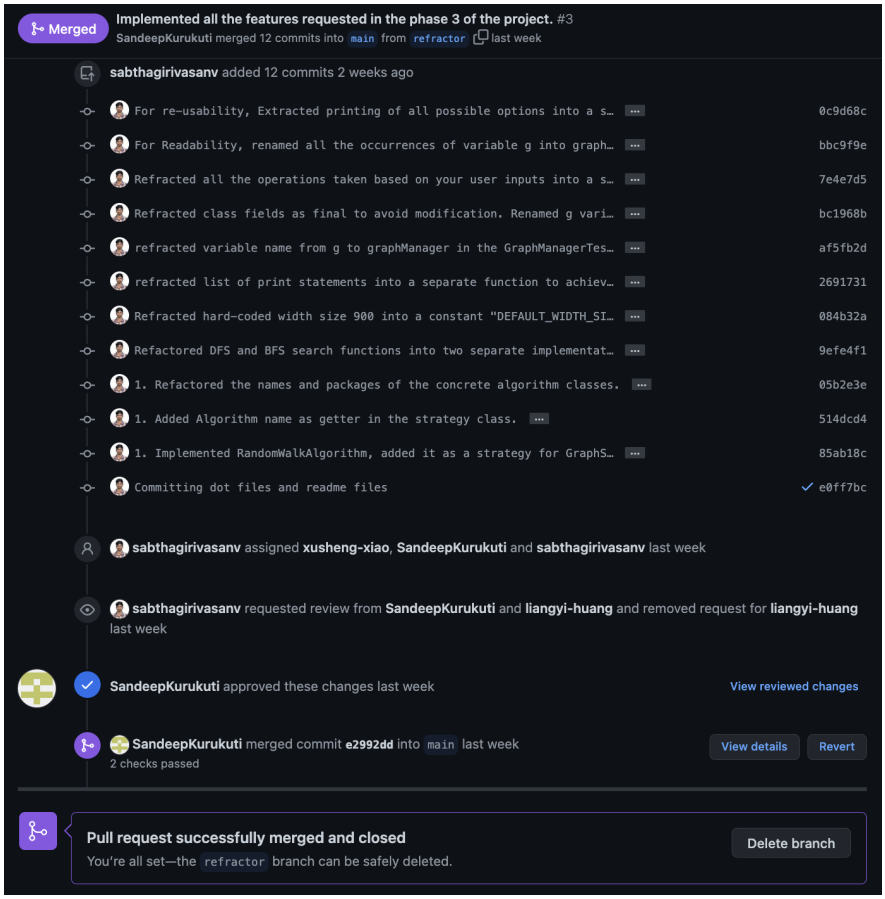
Pull Requests:

I created the pull request to merge the **refractor** branch into the main branch. Firstly, I made 7 code refactors based on the guidelines taught by the professor. Then, I implemented GraphSearcherAlgo class which uses a template design pattern to search the path from the source node to the destination node. Now, I implemented DFS, BFS, and Random Walk algorithms using this GraphSearcherAlgo abstract class. Once the algorithms are implemented using the template pattern, The algorithm is made available to the user to choose dynamically by implementing the Strategy design pattern. The GraphSearcher abstract class is the strategy class and I have implemented separate strategy classes for DFS, BFS, and Random Walk. Finally, I implemented the Factory pattern to get the respective GraphSearcherStrategy based on the Algorithm enums (such as DFS, BFS, RANDOM_WALK).

I am hereby attaching the status of the Pull request.



Pull request link: https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/pull/3



GitHub Commits:

I created a new branch: **refractor** where I made 7 code refactors and implemented the DFS, BFS, and Random Walk algorithm using the Template Design. Implemented strategy design pattern for dynamic usage of Graph Searcher strategies. Finally, implemented the Factory pattern to get the respective strategy object based on the Algorithm enum. I am hereby attaching all my commits in this branch.

Refactor commit links:

1. For re-usability, Extracted printing of all possible options into a separate function called "printAllOptions" in Application.java

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/0c9d68c8fbe54f23c065327d9f202360f1355db8

2. For Readability, renamed all the occurrences of variable g into graphManager in the GraphManagerImplTest.java

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/bbc9f9eca8914deba75a1395528f9498da45ba0d

3. Refracted all the operations taken based on your user inputs into a separate functions for readability and re-usability.

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/7e4e7d5be713f9196333b70bb71f16424e472508

4. Refracted class fields as final to avoid modification. Renamed g variable into mutableGraph to provide readability.

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/bc1968b015780656fc0a43d992ef19ec7359ab96

5. Refracted variable name from g to graphManager in the GraphManagerTest.java to add more readability to the code.

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/af5fb2d301504e12c3cfa9d06ec5d7b1eb2ce695

6. Refracted list of print statements into a separate function to achieve re-usability.

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/2691731f8b62059c6ef201079f38ad4a8e14eed7

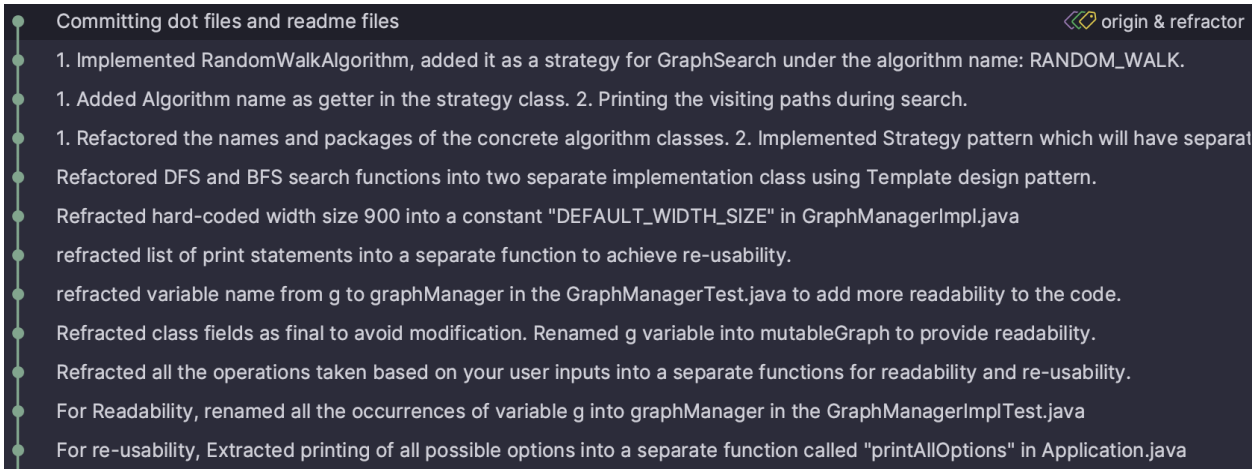
7. Refracted hard-coded width size 900 into a constant "DEFAULT_WIDTH_SIZE" in GraphManagerImpl.java

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/084b32af9bba92fccd9599a2908c1f0e06c0e54c

Design pattern and Random Walk implementation commit links:

1. https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/9efe4f1ad59901efe13381fb47c68eca4b6865bb
2. https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/05b2e3eadaf704977f15d7a27739733de7a466d7
3. https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/514dcd4c0046072aac3386e04cc14988dbd56a07
4. https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/commit/85ab18cc02a59174329e8499c7a1618abb0f09e0

Git Commit Logs:



GitHub workflow results:

GitHub Action link:

https://github.com/sabthagirivasanv/CSE-464-2023-svellin3/actions/workflows/maven.yml