

CSE 464 Project Part 1

This Maven project, named "CSE-464-2023-amehta65", is a graph manipulator project which allows you to manipulate and work with graphs in the DOT file format. It offers several features to parse, modify, and visualize graphs. Below are instructions and example code to run the Maven project.

Prerequisites

Before running this project, ensure that you have the following prerequisites installed on your system:

1. Java Development Kit (JDK)
2. Apache Maven
3. Git (for cloning the project repository)

Getting Started

Follow these steps to get started with the Graph Manipulator project:

1. **Clone the Repository (optional):** You can clone the project repository from [GitHub](#) or download the source code as a ZIP archive.

➤ `git clone https://github.com/AviMehta90/CSE-464-2023-amehta65.git`

2. **Navigate to the Project Directory:** Open a terminal or command prompt, and change your current directory to the project's root folder.

➤ `cd CSE-464-2023-amehta65`

3. **Compile and Build the Project:** Use Maven to compile and build the project. This command will download the required dependencies and create an executable JAR file.

➤ `mvn clean package`

4. **Run the maven project (JUnitTesting)**

➤ `mvn test`

Features in GraphManipulator Class

Feature 1: Parse a DOT Graph File

```
parseGraph(String filePath)
```

This function parses a DOT graph file to create a graph. It takes the file path as an argument and reads the DOT file to build the graph. If successful, it sets the graph `g` and provides information about the number of nodes and edges in the parsed graph.

Information Functions

```
getNumberOfNodes()
```

This function returns the total number of nodes in the parsed graph.

```
getNodeLabels()
```

Returns a set of node labels in the parsed graph, including a label indicating the number of nodes.

```
getNumberOfEdges()
```

Returns the total number of edges in the parsed graph.

```
getEdgeInfo()
```

Returns a set of edge information in the parsed graph, where each entry represents an edge in the format "Source Node -> Destination Node."

```
toGraphString()
```

Returns a formatted string that combines information about the nodes and edges of the parsed graph.

```
outputGraph(String filePath)
```

This function writes the information about nodes and edges to a text file at the specified file path and returns the contents of the created file as a string.

Feature 2: Adding Nodes

```
addNode(String label)
```

Adds a node with the specified label to the graph. If a node with the same label already exists, it won't be added again. Returns `true` if the node is added successfully, or `false` if the node already exists.

```
addNodes(String[] labels)
```

Adds multiple nodes with the labels provided in the `labels` array. If any of the provided labels already exist in the graph, the function won't add them. Returns `true` if at least one new node is added, or `false` if all provided labels already exist.

Feature 3: Adding Edges

```
addEdge(String srcLabel, String dstLabel)
```

Adds an edge between two nodes in the graph based on their labels. The function ensures that both the source and destination nodes exist in the graph before adding the edge.

Feature 4: Output the Imported Graph

```
outputDOTGraph(String filename)
```

This function renders the imported graph as a DOT file and saves it with the specified filename. The rendered DOT file will be located in the specified directory.

```
outputGraphics(String filePath)
```

Visualizes the graph as a PNG image. It takes a path to a DOT file as input, reads the graph from that file, and generates a PNG image file. The generated image will be located in the specified directory.

Feature 5: Part 2 Changes

```
removeNode(String label)
```

To remove a node, use the `removeNode` method and provide the label of the node you want to remove.

```
removeNodes(String[] label)
```

To remove multiple nodes, pass an array of strings and use the `removeNodes` method and provide the labels of the nodes you want to remove.

```
removeEdge(String srcLabel, String dstLabel)
```

To remove an edge, use the `removeEdge` method and provide the label of the source and destination nodes you want to remove.

```
graphSearch(srcNode, dstNode)
```

You can perform a graph search between two nodes using either a breadth-first or depth-first search algorithm. Use the `graphSearch` method with the desired search algorithm based on the enum in the main class. The path of the search algorithm will be stored in a class named `Path` which will output in the format "a -> b -> c"

Running the Application

Now that you've built the project, you can run the `GraphManipulator` application. Here are some examples of how to use it:

1. Parsing a DOT File:

- To parse a DOT graph file and display node and edge information, use the following code:

```
GraphManipulator manipulator = new GraphManipulator();
manipulator.parseGraph("path/to/your/graph.dot");
System.out.println(manipulator.toGraphString());
```

- **Console Output**

```
Feature 1: Parsing DOT Graph File
Dot File Parsed at src/main/resources/test1.dot

Dot File looks like:
digraph "G" {
"a" -> "b"
"b" -> "c"
"c" -> "a"
}

Number of Nodes: 3
Nodes: [a, b, c]
Number of Edges: 3
Edges: [b -> c, c -> a, a -> b]
Output String Graph Info at src/main/resources/GraphInfoAsString.txt
```

2. Adding Nodes:

- You can add nodes to the imported graph using the `addNode` method. If the label already exists, it won't be added again.

```
GraphManipulator manipulator = new GraphManipulator();
manipulator.parseGraph("path/to/your/graph.dot");
manipulator.addNode("NewNode");
```

- **Console Output**

```
Feature 2: Adding Node(s): d, e, f  
New node(s) added: true
```

3. Adding Edges:

- To add edges between existing nodes in the graph, use the `addEdge` method. Provide the labels of the source and destination nodes.

```
GraphManipulator manipulator = new GraphManipulator();  
manipulator.parseGraph("path/to/your/graph.dot");  
manipulator.addEdge("Node1", "Node2");
```

- Console Output

Feature 3: Adding Edges

Edge created: a -> d

Edge created: e -> c

Edge created: f -> a

New edge(s) added

```
digraph "G" {  
  "a" -> "d"  
  "a" -> "b"  
  "b" -> "c"  
  "c" -> "a"  
  "e" -> "c"  
  "f" -> "a"  
}
```

4. Output as DOT File:

- You can output the imported graph as a DOT file with the following code. This will create a DOT file in the specified location.

```
GraphManipulator manipulator = new GraphManipulator();  
manipulator.parseGraph("path/to/your/graph.dot");  
manipulator.outputDOTGraph("output_graph.dot");
```

5. Output as Graphics:

- To visualize the graph as a PNG image, provide the path to the DOT file and specify the output file path.

```
GraphManipulator manipulator = new GraphManipulator();  
manipulator.outputGraphics("path/to/your/graph.dot");
```

- **Console Output**

```
Feature 4: Output the Imported Graph as DOT File  
DOT file created at: graph_for_graphics.dot
```

```
Feature 4: Output the Imported Graph as Graphics
```

```
15:26:50.661 [main] INFO guru.nidi.graphviz.engine.GraphvizCmdLineEngine  
15:26:50.663 [main] INFO guru.nidi.graphviz.service.CommandLineExecutor  
15:26:50.665 [main] DEBUG guru.nidi.graphviz.service.CommandLineExecutor  
15:26:50.769 [main] INFO guru.nidi.graphviz.engine.GraphvizCmdLineEngine  
Graph graphics generated at : src/main/resources/graph_for_graphics.dot
```

```
Process finished with exit code 0
```

6. Remove Node(s):

- To check node(s) removal functionality.

```
GraphManipulator manipulator = new GraphManipulator();  
manipulator.parseGraph("path/to/your/graph.dot");  
manipulator.addNode("Node1");  
manipulator.addNodes(String[]{"Node2", "Node3"});  
manipulator.removeNode("Node1");  
manipulator.removeNode("NonexistentNode");  
manipulator.removeNodes(String[]{"Node2", "Node3"});  
manipulator.removeNodes(String[]{"NonExNode2", "NonExNode3"});
```

- Console Output Remove Node(Test Case)

✓ GraphManipulator 522 ms	Dot File Parsed at src/main/resources/test1.dot
✓ testParseGraph() 65 ms	Number of Nodes: 3
✓ testOutputDOTG 16 ms	Nodes: [a, b, c]
✓ testToGraphString 6 ms	Number of Edges: 3
✓ testAddEdge() 1 ms	Edges: [b -> c, c -> a, a -> b]
✓ testAddNode() 1 ms	Adding node d
✓ testOutputGraph() 1 ms	Number of Nodes: 4
✓ testRemoveNodes 2 ms	Nodes: [a, b, c, d]
✓ testAddNodes() 1 ms	Number of Edges: 3
✓ testRemoveEdge() 1 ms	Edges: [b -> c, c -> a, a -> b]
✓ testRemoveNode(1 ms	Node d removed.
✓ testOutputGrap 420 ms	Number of Nodes: 3
✓ testGraphSearch() 7 ms	Nodes: [a, b, c]
	Number of Edges: 3
	Edges: [b -> c, c -> a, a -> b]
	Node z not found.

- Console Output Remove Nodes(Test Case)

✓ GraphManipulator 522 ms	Dot File Parsed at src/main/resources/test1.dot
✓ testParseGraph() 65 ms	Adding nodes e, f
✓ testOutputDOTG 16 ms	Number of Nodes: 5
✓ testToGraphString 6 ms	Nodes: [a, b, c, e, f]
✓ testAddEdge() 1 ms	Number of Edges: 3
✓ testAddNode() 1 ms	Edges: [b -> c, c -> a, a -> b]
✓ testOutputGraph() 1 ms	Removing nodes: e, f
✓ testRemoveNodes 2 ms	Node e removed.
✓ testAddNodes() 1 ms	Node f removed.
✓ testRemoveEdge() 1 ms	Number of Nodes: 3
✓ testRemoveNode(1 ms	Nodes: [a, b, c]
✓ testOutputGrap 420 ms	Number of Edges: 3
✓ testGraphSearch() 7 ms	Edges: [b -> c, c -> a, a -> b]
	Removing non-existent nodes: x, y
	Node x not found.
	Node y not found.

7. Remove Edge:

- To check remove edge functionality.


```

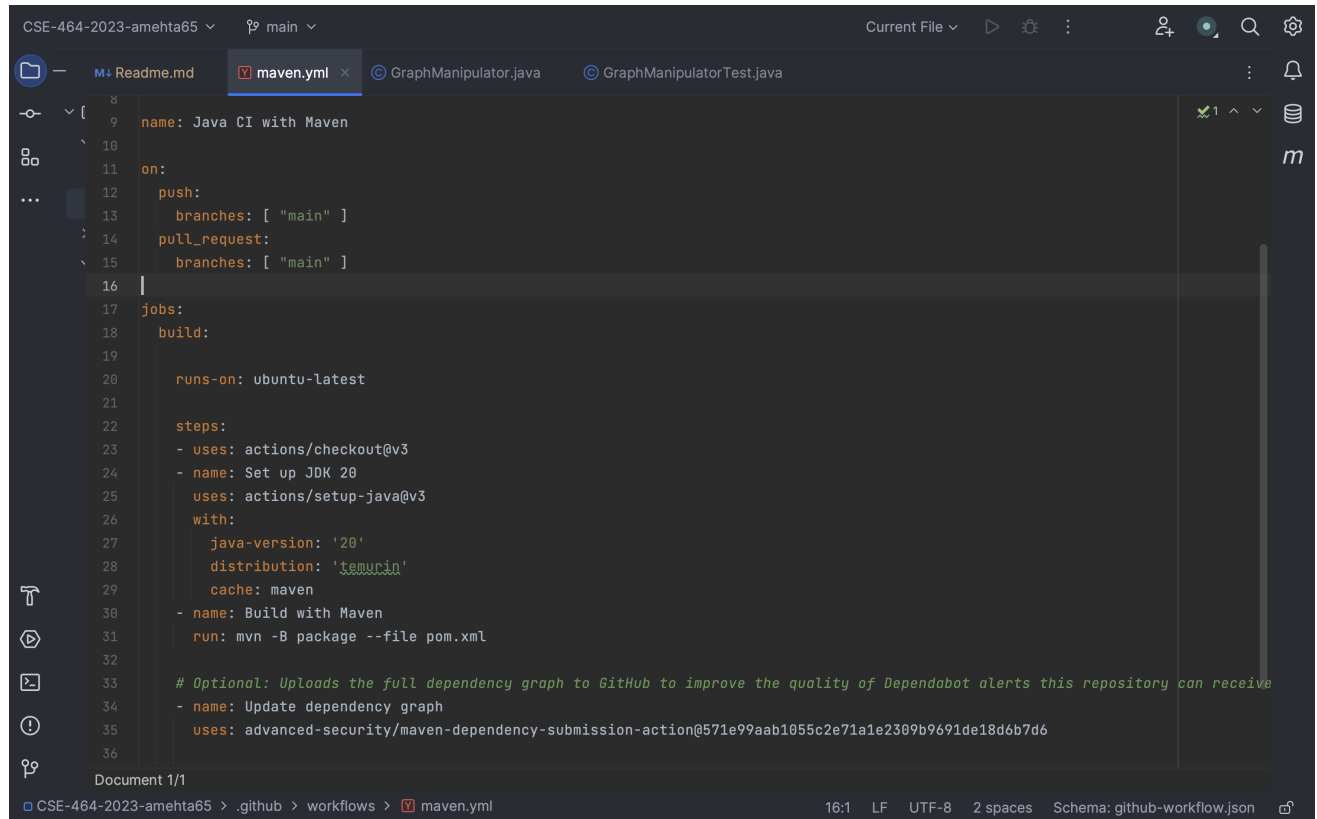
GraphManipulator manipulator = new GraphManipulator();
manipulator.parseGraph("path/to/your/graph.dot");
manipulator.addNode("Node1");
manipulator.addEdge(String[]{"Node1", "Node2"});
manipulator.removeEdge(String[]{"Node1", "Node2"});
manipulator.removeEdge(String[]{"NonExNode2", "NonExNode3"});

```

- **Console Output Remove Edge(Test Case)**

✓ GraphManipulator	512 ms	Dot File Parsed at src/main/resources/test1.dot
✓ testParseGraph()	63 ms	Edge created: a -> d
✓ testOutputDOTG	14 ms	Edge created: e -> b
✓ testToGraphString	4 ms	Number of Nodes: 3
✓ testAddEdge()	3 ms	Nodes: [a, b, c]
✓ testAddNode()	1 ms	Number of Edges: 5
✓ testOutputGraph()	2 ms	Edges: [c->a, e->b, a->b, b->c, a->d]
✓ testRemoveNode	3 ms	Edge e -> b removed.
✓ testAddNodes()	1 ms	Number of Nodes: 3
✓ testRemoveEdge()	1 ms	Nodes: [a, b, c]
✓ testRemoveNode	1 ms	Number of Edges: 4
✓ testOutputGraph	412 ms	Edges: [c->a, a->b, b->c, a->d]
✓ testGraphSearch()	7 ms	Removing non-existent edge: m -> n
		Edge m -> n not found.

8. Creating maven.yml for native CI/CD on github (under actions tab in the repository)



```
8
9 name: Java CI with Maven
10
11 on:
12   push:
13     branches: [ "main" ]
14   pull_request:
15     branches: [ "main" ]
16
17 jobs:
18   build:
19
20     runs-on: ubuntu-latest
21
22     steps:
23     - uses: actions/checkout@v3
24     - name: Set up JDK 20
25       uses: actions/setup-java@v3
26       with:
27         java-version: '20'
28         distribution: 'temurin'
29         cache: maven
30     - name: Build with Maven
31       run: mvn -B package --file pom.xml
32
33     # Optional: Uploads the full dependency graph to GitHub to improve the quality of Dependabot alerts this repository can receive
34     - name: Update dependency graph
35       uses: advanced-security/maven-dependency-submission-action@571e99aab1055c2e71a1e2309b9691de18d6b7d6
36
```

Document 1/1

CSE-464-2023-amehta65 > .github > workflows > maven.yml 16:1 LF UTF-8 2 spaces Schema: github-workflow.json

The screenshot shows the GitHub Actions interface for a repository named 'CSE-464-2023-amehta65'. The 'Actions' tab is selected, displaying a workflow named 'Java CI with Maven' (maven.yml). A list of 4 workflow runs is shown, each with a status icon, title, commit hash, branch, and duration.

Event	Status	Branch	Actor
Merged changes with enum and added test case	Success	main	AviMehta90
Conflicts merge	Success	main	AviMehta90
Update maven.yml	Success	main	AviMehta90
Create maven.yml	Failure	main	AviMehta90

The screenshot shows the details of a specific workflow run titled 'Merged changes with enum and added test case #4'. The 'Summary' tab is selected, showing a list of jobs. The 'build' job is highlighted, and its details are shown on the right, including a list of steps and their durations.

Job	Status	Duration
build	Success	29 minutes ago in 23s

Step	Status	Duration
Set up job	Success	3s
Run actions/checkout@v3	Success	1s
Set up JDK 20	Success	8s
Build with Maven	Success	6s
Update dependency graph	Success	3s
Post Set up JDK 20	Success	0s
Post Run actions/checkout@v3	Success	0s
Complete job	Success	0s

9. Creating branches and Merging Conflicts:

- Creating bfs and dfs branch and merging with main branch

➤ `git checkout -b bfs`

Do the changes in bfs branch and type the following command:

➤ `git add .`
`git commit -m "Added BFS Search Algorithm"`

```
git push origin bfs
```

Go back to main branch

```
➤ git checkout main
```

Create dfs branch

```
➤ git checkout -b dfs
```

Do the changes in dfs branch and type the following command:

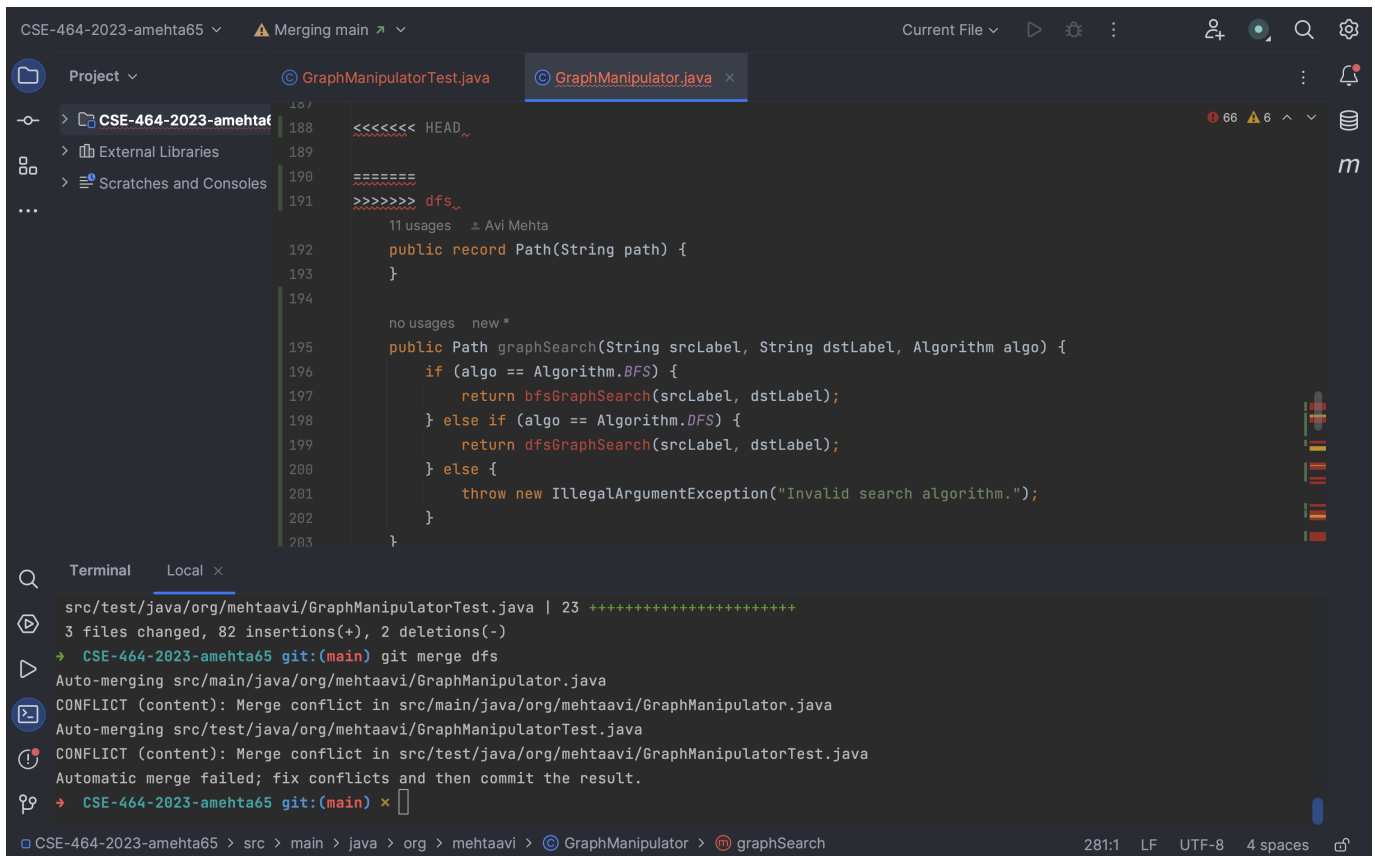
```
➤ git add .  
git commit -m "Added DFS Search Algorithm"  
git push origin dfs
```

Now merge the dfs and bfs with main branch by checking in individual branches

```
➤ git checkout main  
git merge bfs  
git merge dfs
```

Merge conflicts and then commit and push to main origin

```
➤ git add .  
git commit -m "Resolved merge conflicts and added algorithm selection in graphSearch"
```



10. Graph Search(BFS):

- Checking the BFS search algorithm.

```
GraphManipulator gM = new GraphManipulator();
gM.addNode("d");
gM.addNodes(new String[]{"e", "f"});
gM.addEdge("a", "d");
gM.addEdge("e", "c");
gM.addEdge("f", "a");
GraphManipulator.Path path1 = gM.graphSearch("a", "c", GraphManipulator.Algorithm.BFS);
System.out.println("Performing BFS");
System.out.println(path1);
```

11. Graph Search(DFS):

- Checking the DFS search algorithm.

```
GraphManipulator gM = new GraphManipulator();
gM.addNode("d");
gM.addNodes(new String[]{"e", "f"});
gM.addEdge("a", "d");

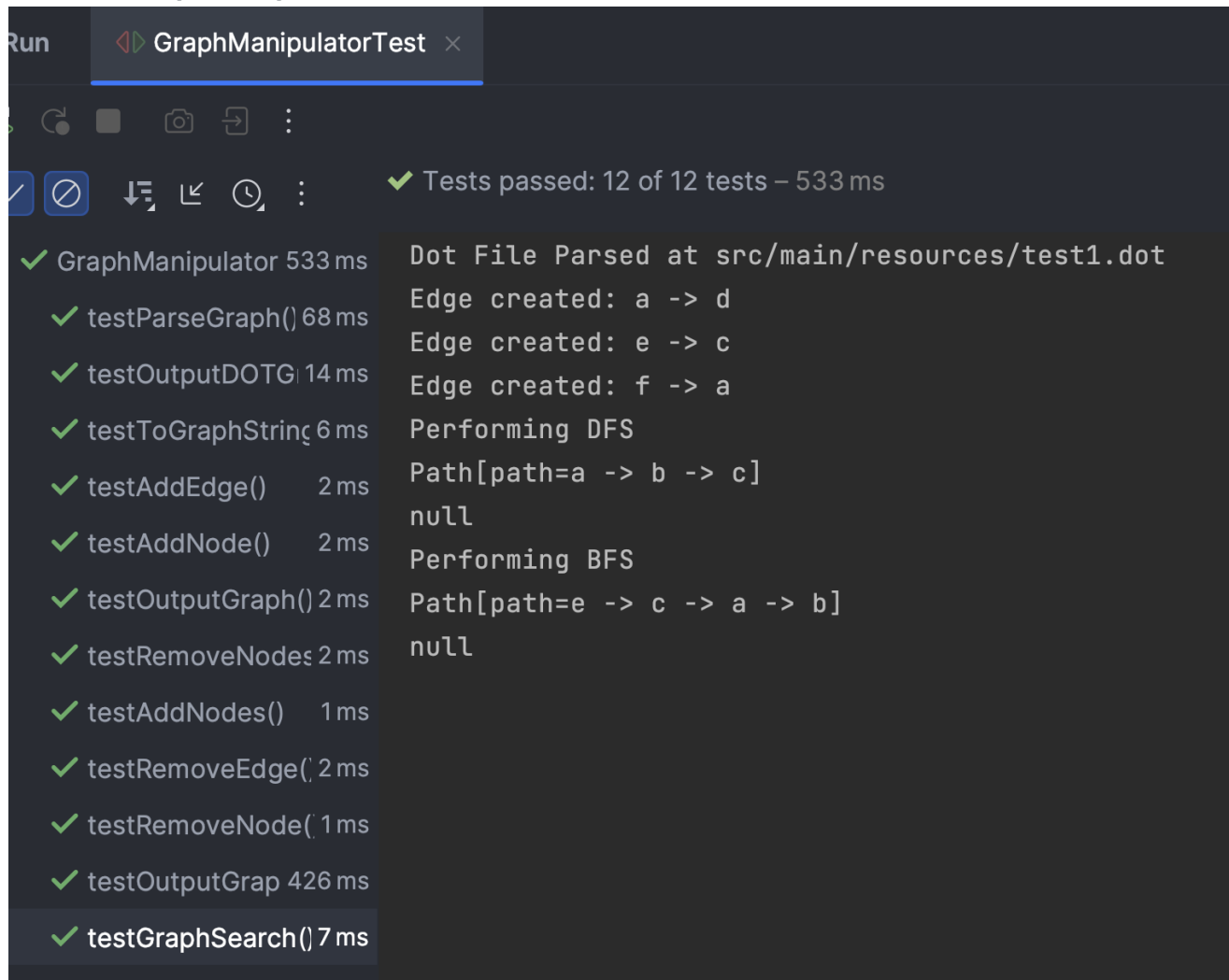
gM.addEdge("e", "c");
gM.addEdge("f", "a");
```

```

GraphManipulator.Path path1 = gM.graphSearch("a", "c", GraphManipulator.Algorithm.D
System.out.println("Performing DFS");
System.out.println(path1);

```

- **Console Output Graph Search(Test Case)**



Refactoring Changes:

1. Filename change

- **File Changes** new_graph_image.png
- **Reason:** Snake Casing for better readability

2. Instantiated Global String Variables

- **Variables Added** EDGE_DELIMITER = "->" and PATH_PREFIX = "src/main/resources/"
- **Reason:** Accessible to all parts of the program. The intent is to protect data from being changed.

3. Error Handling

- **Changes Made:** Error message in outputDOTGraph changed to "Error creating DOT file: " + e.getMessage()
- **Reason:** Easy Identification of Program Code and Error-Handling Code.

4. Extracted Method:

- **Method Renamed:** modifyNodes Method added
- **Reason:** Added to check whether adding or deleting node in a single method making the original methods less complex and readable.

5. String Format

- **Changes Made:** String.format("%s%s%s", srcLabel, EDGE_DELIMITER, dstLabel);
- **Reason:** The format string provides a clear template for the resulting string, making it easier to understand the structure. The String.format method supports localization by allowing you to specify different format patterns based on the locale.

Code Patterns

Template Pattern

This branch of the project introduces a refactoring of the `GraphManipulator` class by applying the template pattern. The goal is to abstract common steps in the BFS (Breadth-First Search) and DFS (Depth-First Search) algorithms and provide a more modular and maintainable solution.

Changes Made

1. Base Class - `GraphSearchAlgorithm`:

- Created a new base class `GraphSearchAlgorithm` to define the common steps for both BFS and DFS.
- Moved the common initialization, path retrieval, and overall search structure to this base class.

2. Subclasses - `BFSAlgorithm` and `DFSAlgorithm`:

- Created two subclasses, `BFSAlgorithm` and `DFSAlgorithm`, that extend `GraphSearchAlgorithm`.
- Implemented algorithm-specific steps such as retrieving the next node and processing neighbors.
- Implemented a single linked list structure for both the classes. The difference stands in the handling of neighbours where in BFS the behaviour of the linkedlist is queue and in DFS it is a stack.

3. Refactoring in `GraphManipulator`:

- Updated the `GraphManipulator` class to use the new template pattern.
- The `graphSearch` method now creates an instance of the appropriate algorithm based on the selected type (BFS or DFS).

Usage

1. BFS Search:

```
GraphManipulator graphManipulator = new GraphManipulator();  
GraphManipulator.Path path = graphManipulator.graphSearch("sourceLabel", "destinati
```

2. DFS Search:

```
GraphManipulator graphManipulator = new GraphManipulator();  
GraphManipulator.Path path = graphManipulator.graphSearch("sourceLabel", "destinati
```

Strategy Pattern

Strategy Interface (GraphSearchStrategy)

- **Purpose:** To define a common interface for all graph search strategies.
- **Methods:** Declares the method `graphSearch(String srcLabel, String dstLabel)` .
- **Usage:** Implemented by `BFSAlgorithm` and `DFSAlgorithm` .

Concrete Strategies (BFSAlgorithm and DFSAlgorithm)

- **BFSAlgorithm:**
 - **Inherits:** `GraphSearchAlgorithm` .
 - **Implements:** `GraphSearchStrategy` .
 - **Behavior:** Implements the BFS algorithm for graph searching.
 - **Method:** `graphSearch` overridden to provide BFS specific logic.
- **DFSAlgorithm:**
 - **Inherits:** `GraphSearchAlgorithm` .
 - **Implements:** `GraphSearchStrategy` .
 - **Behavior:** Implements the DFS algorithm for graph searching.
 - **Method:** `graphSearch` overridden to provide DFS specific logic.

Context Class (GraphManipulator)

- **Responsibility:** Manages the graph and delegates the search operation to the current strategy.
- **Key Method:**
 - `setSearchStrategy(GraphSearchStrategy strategy)` : Sets the current search strategy.
 - `graphSearch(String srcLabel, String dstLabel)` : Delegates the search to the chosen strategy.

Test Code

- **Setting Strategy:** We can dynamically set the desired search strategy (BFS or DFS) using `setSearchStrategy` .
- **Searching:** Once the strategy is set, calling `graphSearch` on `GraphManipulator` executes the search based on the selected strategy.

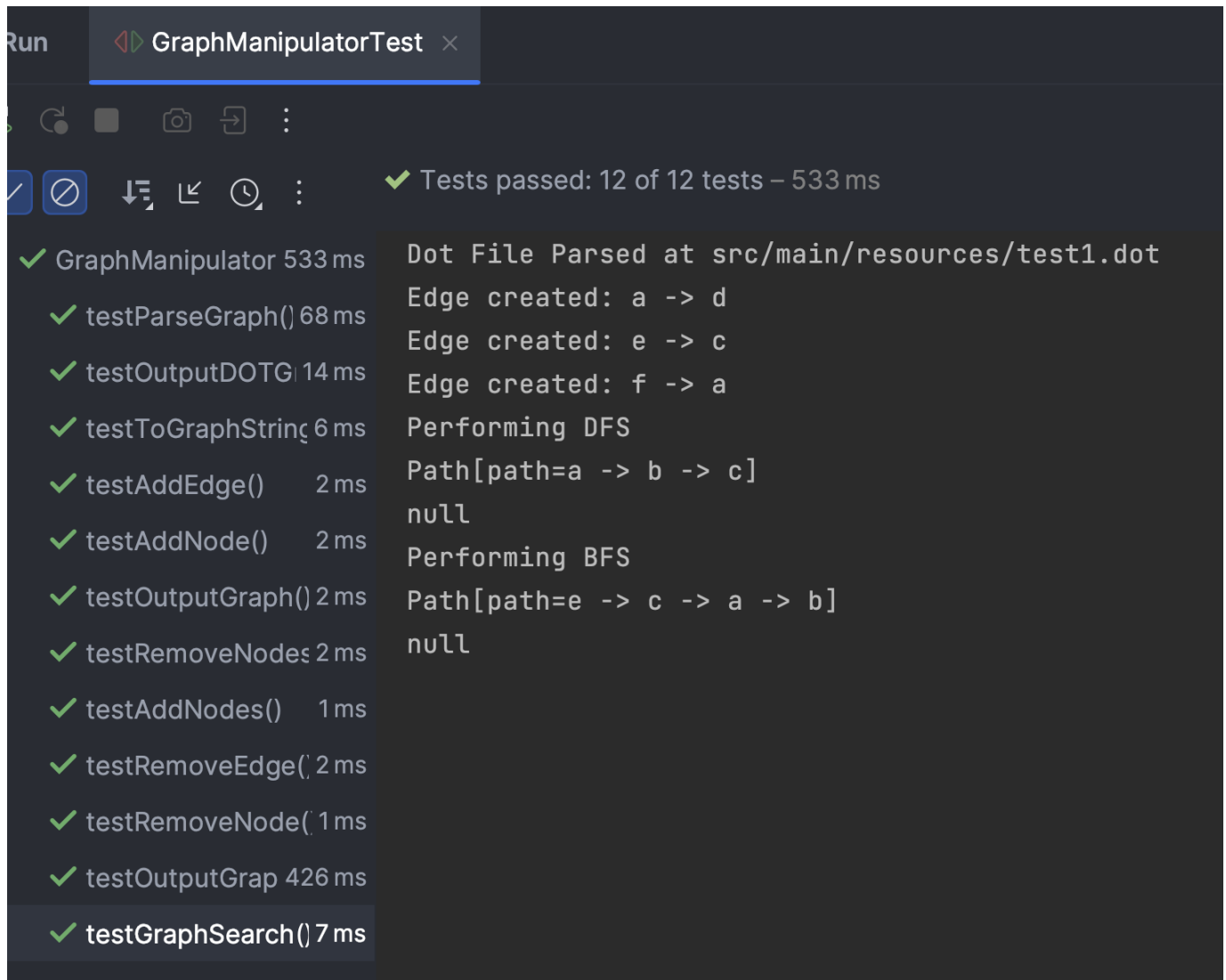
Usage Example

```
GraphManipulator manipulator = new GraphManipulator();

// Setting BFS as the strategy
manipulator.setSearchStrategy(new BFSAlgorithm(graph));
Path bfsPath = manipulator.graphSearch("source", "destination");

// Switching to DFS
manipulator.setSearchStrategy(new DFSAlgorithm(graph));
Path dfsPath = manipulator.graphSearch("source", "destination");
```

Output



```
Run GraphManipulatorTest x
✓ Tests passed: 12 of 12 tests - 533 ms

✓ GraphManipulator 533 ms
  ✓ testParseGraph() 68 ms
  ✓ testOutputDOTG 14 ms
  ✓ testToGraphString 6 ms
  ✓ testAddEdge() 2 ms
  ✓ testAddNode() 2 ms
  ✓ testOutputGraph() 2 ms
  ✓ testRemoveNodes 2 ms
  ✓ testAddNodes() 1 ms
  ✓ testRemoveEdge() 2 ms
  ✓ testRemoveNode() 1 ms
  ✓ testOutputGrap 426 ms
  ✓ testGraphSearch() 7 ms

Dot File Parsed at src/main/resources/test1.dot
Edge created: a -> d
Edge created: e -> c
Edge created: f -> a
Performing DFS
Path[path=a -> b -> c]
null
Performing BFS
Path[path=e -> c -> a -> b]
null
```

Advantages

- **Flexibility:** Easily switch between different algorithms at runtime without modifying the core logic.
- **Scalability:** New search strategies can be added without changing the existing codebase.
- **Maintainability:** Each algorithm is encapsulated in its own class, making it easier to manage and modify.

Random Walk Search

This feature introduces a Random Walk Search algorithm to explore paths in a graph. The Random Walk algorithm starts from a source node and randomly selects neighbors to traverse until the destination node is reached or a certain condition is met.

How to Use

1. **Create an iterated method of the RandomWalkAlgorithm:**

```
public String randomWalkSearchProcess(String srcLabel, String dstLabel, int numIter)
    StringBuilder result = new StringBuilder("random testing\n");

    for (int i = 0; i < numIterations; i++) {
        System.out.println("Iteration: "+ (i+1));
        Path path = searchStrategy.graphSearch(srcLabel, dstLabel);
        result.append("visiting iteration ").append(i + 1).append(" ").append(path)
    }

    return result.toString();
}
```

2. Run the Random Walk Search:

```
MutableGraph test_graph = new Parser().read(new FileInputStream("src/main/resources
gM.setSearchStrategy(new RandomWalkAlgorithm(test_graph));
System.out.println("Performing Random Walk Search");
GraphManipulator.Path rwsPath = gM.graphSearch("a", "c");
System.out.println(rwsPath.toString());
```

Output:

Performing Random Walk Search

Iteration: 1

Path[path=a]

Path[path=a -> b]

Iteration: 2

Path[path=a]

Path[path=a -> e]

Path[path=a -> e -> g]

Path[path=a -> e -> g -> h]

Iteration: 3

Path[path=a]

Path[path=a -> e]

Path[path=a -> e -> f]

Path[path=a -> e -> f -> h]

random testing

visiting iteration 1 Path[path=a -> b -> c]

visiting iteration 2 Path[path=No path found]

visiting iteration 3 Path[path=No path found]

Project Structure

The project's source code is organized as follows:

- `src/main/java/org/mehtaavi` : Contains the Java source code.
- `src/test/java/org/mehtaavi` : Contains the Java test code.
- `src/main/resources` : Place your DOT files here and specify the output directory for graphics.

Commit Links

1. [First Commit](#)
2. [First Feature Added](#)
3. [Second Feature Added](#)
4. [Third Feature Added](#)

5. Fourth Feature Added
6. Deleted 'example' directory
7. code formatted
8. Final Upload
9. Added 3 APIs to support node and edge removal with test unit
10. Create maven.yml
11. Update maven.yml
12. Added BFS graph search API
13. Name corrected for BFS API graphsearch
14. Implemented DFS graph search API
15. Conflicts merge
16. Merged changes with enum and added test case
17. Formatting changes
18. Refactoring changes
19. Template Pattern
20. Strategy Pattern
21. Random Walk Search Implementation