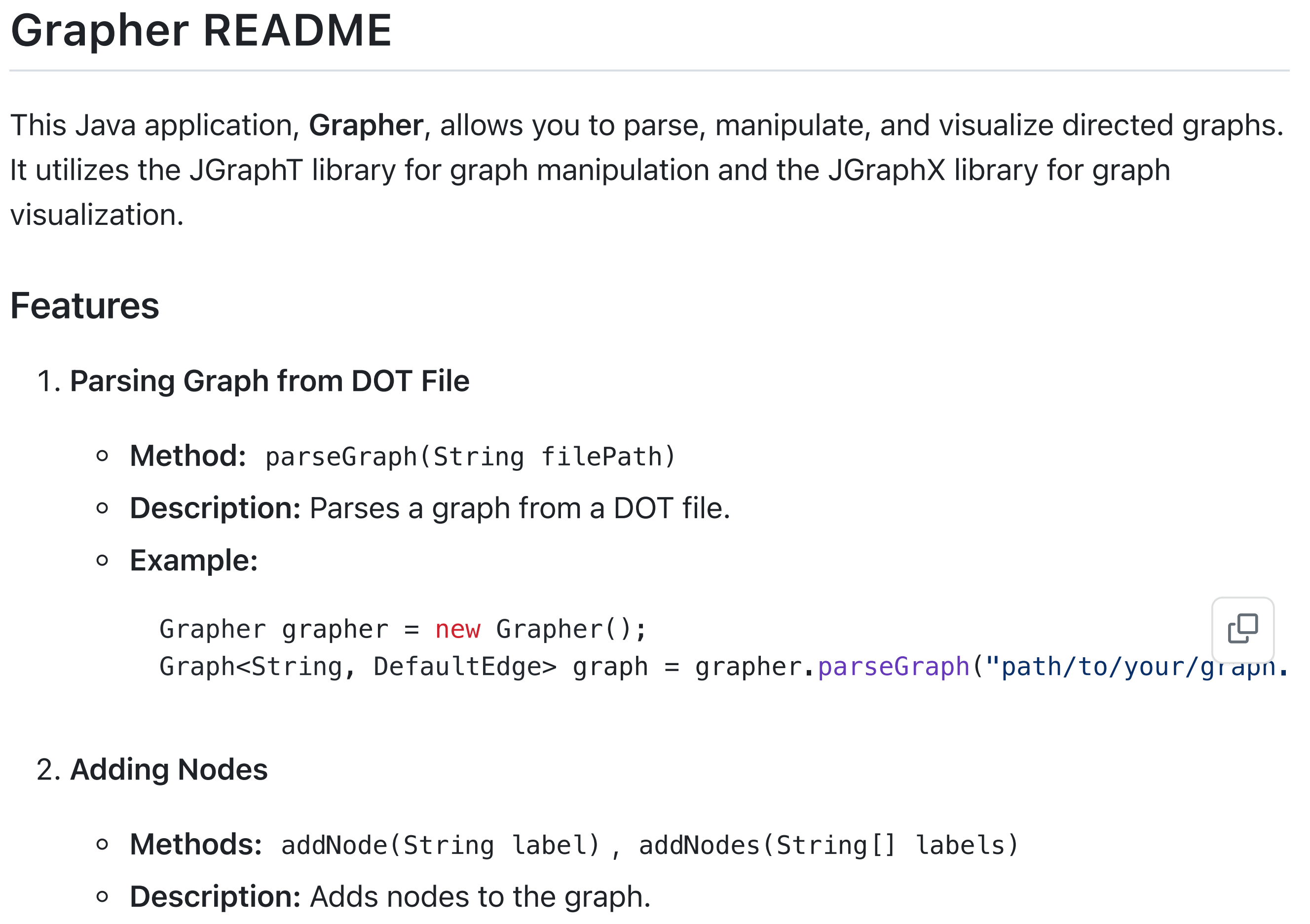


Grapher README

This Java application, **Grapher**, allows you to parse, manipulate, and visualize directed graphs. It utilizes the JGraphT library for graph manipulation and the JGraphX library for graph visualization.

Features

1. **Parsing Graph from DOT File**
   - **Method:** `parseGraph(String filePath)`
   - **Description:** Parses a graph from a DOT file.
   - **Example:**

     ```
     Grapher grapher = new Grapher();
     Graph<String, DefaultEdge> graph = grapher.parseGraph("path/to/your/graph.
     ```

2. **Adding Nodes**
   - **Methods:** `addNode(String label)` , `addNodes(String[] labels)`
   - **Description:** Adds nodes to the graph.

- **Example:**

  ```java
  grapher.addNode("A");
  grapher.addNodes(new String[]{"B", "C"});
  ```

3. **Adding Edges**

   - **Method:** `addEdge(String srcLabel, String dstLabel)`
   - **Description:** Adds directed edges between nodes.
   - **Example:**

     ```java
     grapher.addEdge("A", "B");
     ```

4. **Exporting Graph to DOT Format**

   - **Method:** `outputDOTGraph(String filePath)`
   - **Description:** Exports the graph in DOT format to a file.
   - **Example:**

     ```java
     grapher.outputDOTGraph("path/to/save/graph.dot");
     ```

5. **Exporting Graph as Image**

   - **Method:** `outputGraphics(String filePath)`
   - **Description:** Exports the graph as an image file (PNG format).
   - **Example:**

     ```java
     grapher.outputGraphics("path/to/save/graph.png");
     ```

6. **Generating Graph Information**

   - **Method:** `toString()`
   - **Description:** Generates a string containing graph information, including nodes and edges.
   - **Example:**

     ```java
     String graphInfo = grapher.toString();
     ```

7. **Writing Graph Information to File**

   - **Method:** `writeToFile(String filePath)`
   - **Description:** Writes graph information to a text file.
   - **Example:**

```
grapher.writeToFile("path/to/save/graphInfo.txt");
```

8. **Removing a Node**

    - **Method:** `removeNode(String label)`
    - **Description:** Removes a node from the graph.
    - **Example:**

    ```
    grapher.removeNode("B");
    ```

9. **Removing multiple Nodes**

    - **Method:** `removeNodes(String[] labels)`
    - **Description:** Removes multiple nodes from the graph
    - **Example:**

    ```
    String[] nodesToRemove1 = {"A", "B"};
    grapher.removeNodes(nodesToRemove1);
    ```

10 **Removing an Edge**

- **Method:** `removeEdge(String srcLabel, String dstLabel)`
- **Description:** Removes an edge from the graph.
- **Example:**

```
grapher.removeEdge("A", "B");
```

11. **Searching for a path in the graph**

- **Method:** `Path graphSearch(String src, String dst, Algorithm algo)`
- **Description:** Finds a path from a source node to destination node in BFS or DFS as specified.
- **Example:**

```
grapher.graphSearch("A", "E", Algorithm.BFS);
```

12. **Parsing through the graph to get a random path from source to destination**

- **Method:** `Grapher.Path randomWalkPath = grapher2.graphSearch("a", "h");`
- **Description:** Outputs a random path from source to destination.
- **Example:**

```
Grapher grapher2 = new Grapher();
grapher2.setStrat(new RandomWalk(grapher2.parseGraph("src/main/resources/input
Grapher.Path randomWalkPath = grapher2.graphSearch("a", "h");
System.out.println(randomWalkPath);
```

How to Run

1. Clone the repository - Link
2. **Compile the Code:**

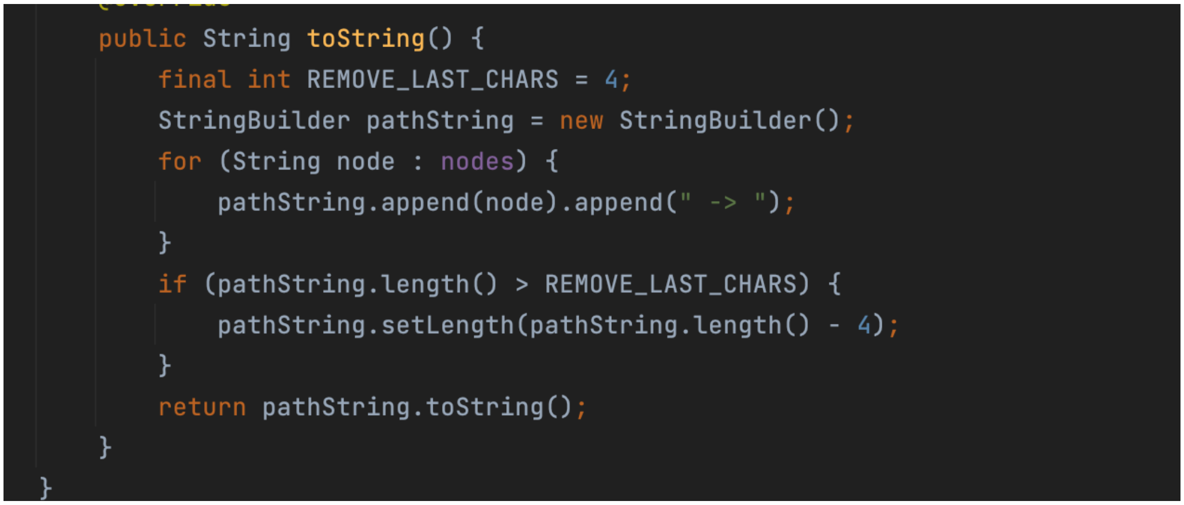
   ```
   mvn package
   ```

3. **Run Tests:**

   ```
   mvn test
   ```

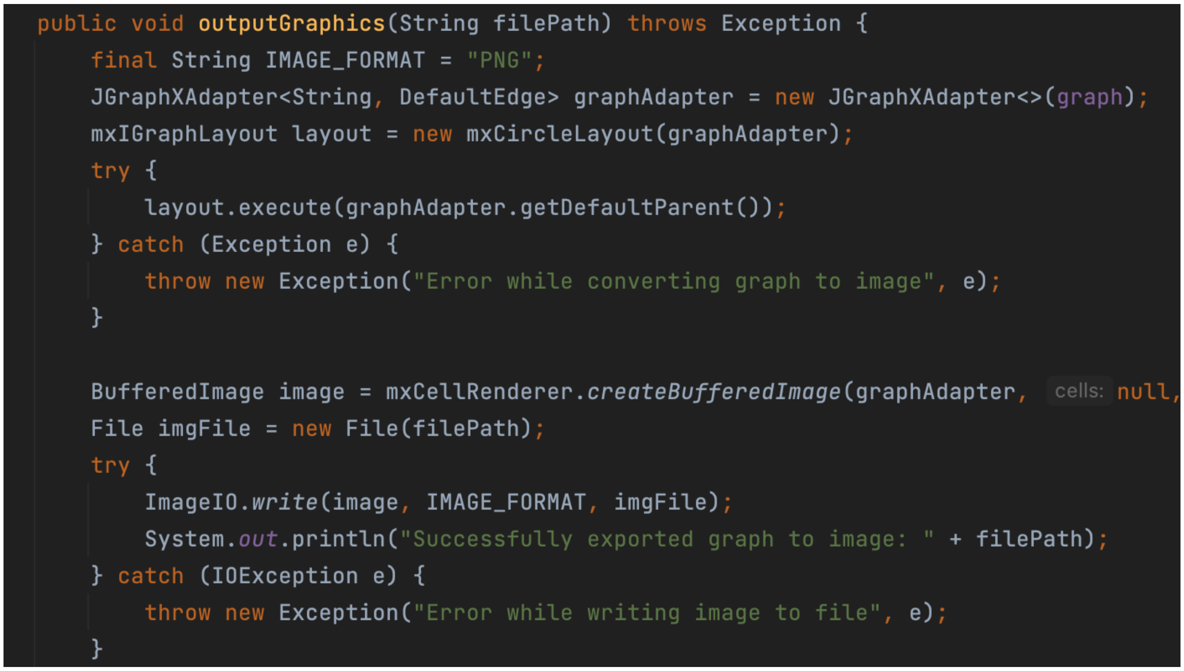
Refactor branch

1. Performing 5 refactors
   - Removing unused imports - Unused imports have been removed in order to reduce some clutter in the code
   - Used a variable instead of a direct numerical value - Using the number 4 did not make it clear why 4 was used, giving 4 as a value to a variable helps understand the purpose because of the variable name

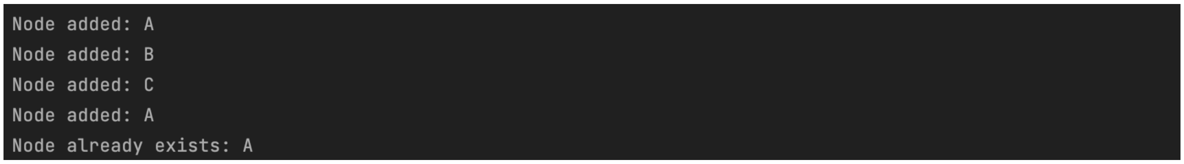
     ```
         @Override
         public String toString() {
             final int REMOVE_LAST_CHARS = 4;
             StringBuilder pathString = new StringBuilder();
             for (String node : nodes) {
                 pathString.append(node).append(" -> ");
             }
             if (pathString.length() > REMOVE_LAST_CHARS) {
                 pathString.setLength(pathString.length() - 4);
             }
             return pathString.toString();
         }
     }
     ```

   - Used a variable instead of a direct value - Using the word PNG might not immediately make it clear why PNG was used, giving PNG as a value to a variable helps understand the purpose because of the variable name



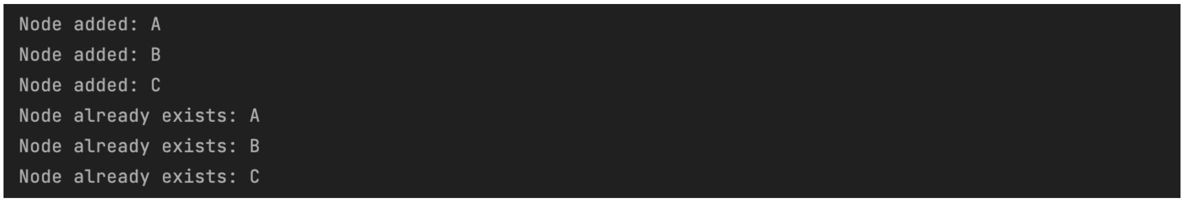
```java
public void outputGraphics(String filePath) throws Exception {
    final String IMAGE_FORMAT = "PNG";
    JGraphXAdapter<String, DefaultEdge> graphAdapter = new JGraphXAdapter<>(graph);
    mxIGraphLayout layout = new mxCircleLayout(graphAdapter);
    try {
        layout.execute(graphAdapter.getDefaultParent());
    } catch (Exception e) {
        throw new Exception("Error while converting graph to image", e);
    }

    BufferedImage image = mxCellRenderer.createBufferedImage(graphAdapter,  cells: null,
    File imgFile = new File(filePath);
    try {
        ImageIO.write(image, IMAGE_FORMAT, imgFile);
        System.out.println("Successfully exported graph to image: " + filePath);
    } catch (IOException e) {
        throw new Exception("Error while writing image to file", e);
    }
```

- Modified testAddNode to show it handles duplicate node being added - There was no test case to see how duplicate node was handled in testAddNode



```
Node added: A
Node added: B
Node added: C
Node added: A
Node already exists: A
```

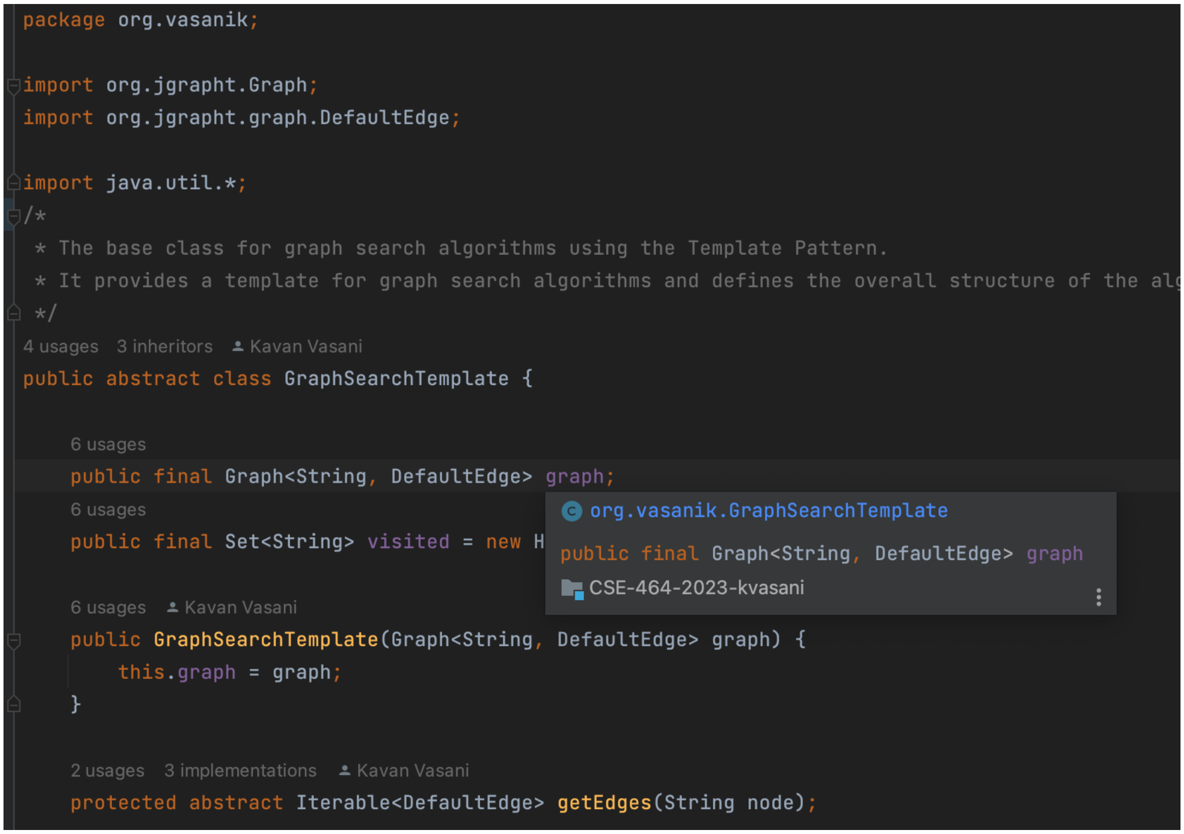
- Modified testAddNodes to show it handles duplicate nodes being added - There was no test case to see how duplicate nodes were handled in testAddNode



```
Node added: A
Node added: B
Node added: C
Node already exists: A
Node already exists: B
Node already exists: C
```

2. Implementation of Template pattern
    - The Template Pattern is a design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
    - In this code, the GraphSearchTemplate class is the template. It provides the overall structure of a graph search algorithm but leaves specific steps (like getting edges,

creating collections, and getting the next path) to be implemented by subclasses.

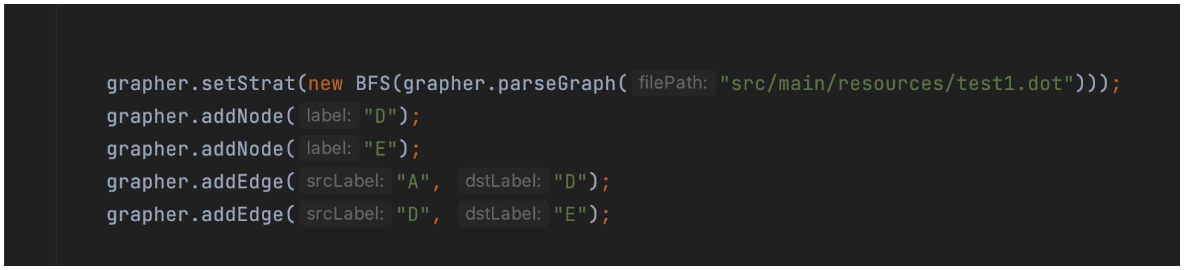


```
package org.vasanik;

import org.jgrapht.Graph;
import org.jgrapht.graph.DefaultEdge;

import java.util.*;
/*
 * The base class for graph search algorithms using the Template Pattern.
 * It provides a template for graph search algorithms and defines the overall structure of the alg
 */
4 usages   3 inheritors   Kavan Vasani
public abstract class GraphSearchTemplate {

    6 usages
    public final Graph<String, DefaultEdge> graph;
    6 usages
    public final Set<String> visited = new H

    6 usages   Kavan Vasani
    public GraphSearchTemplate(Graph<String, DefaultEdge> graph) {
        this.graph = graph;
    }

    2 usages   3 implementations   Kavan Vasani
    protected abstract Iterable<DefaultEdge> getEdges(String node);
```

```
org.vasanik.GraphSearchTemplate
public final Graph<String, DefaultEdge> graph
CSE-464-2023-kvasani
```

- BFS, DFS and RandomWalk classes extend GraphSearchTemplate, providing implementations for the specific steps needed for Breadth-First Search, Depth-First Search and Random Walk respectively.
- BFS uses a queue (LinkedList) for the collection, and DFS uses a stack.

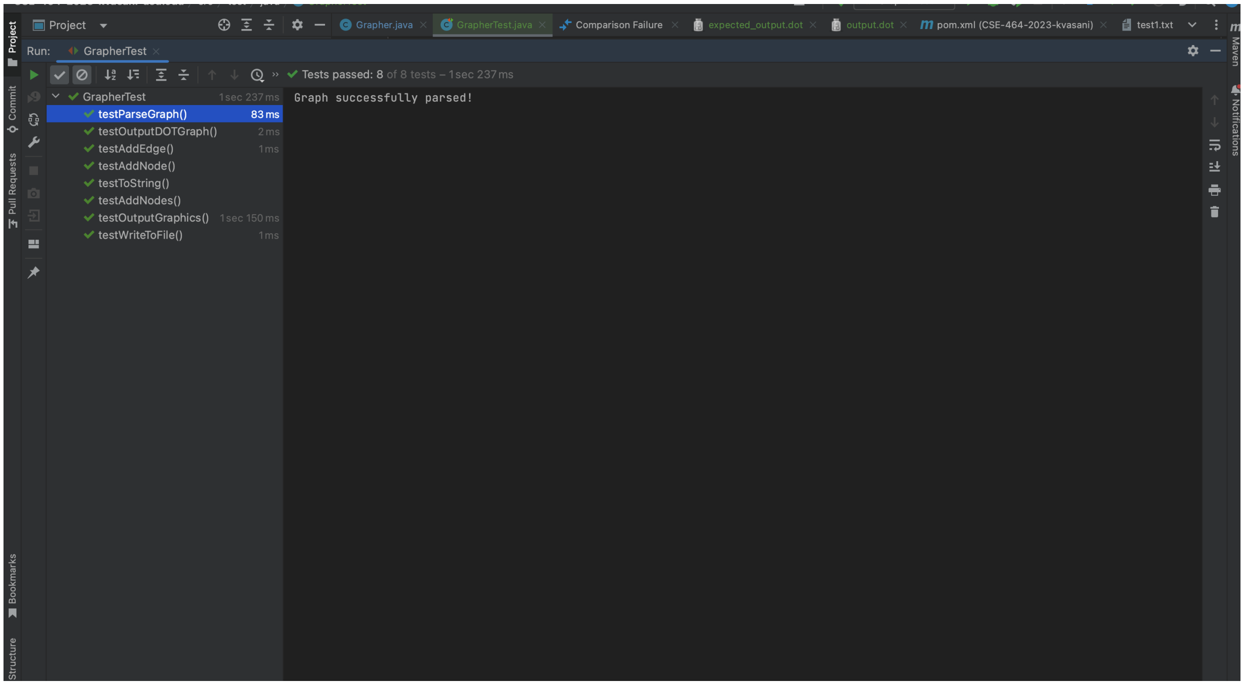
3. Implementation of Strategy pattern
   - The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the client to choose the appropriate algorithm at runtime.
   - The BFS and DFS classes extend GraphSearchTemplate, providing concrete implementations for the specific steps of Breadth-First Search and Depth-First Search, respectively.
   - The Grapher class has a member variable strat of type IGraphSearch, which represents the current strategy for graph search.
   - IGraphSearch interface defines the contract that all graph search algorithms must follow. It has a single method graphSearch for conducting the search and returning a path.



```
grapher.setStrat(new BFS(grapher.parseGraph( filePath: "src/main/resources/test1.dot")));
grapher.addNode( label: "D");
grapher.addNode( label: "E");
grapher.addEdge( srcLabel: "A",  dstLabel: "D");
grapher.addEdge( srcLabel: "D",  dstLabel: "E");
```

Screenshots

- **Parsed Graph Information:**

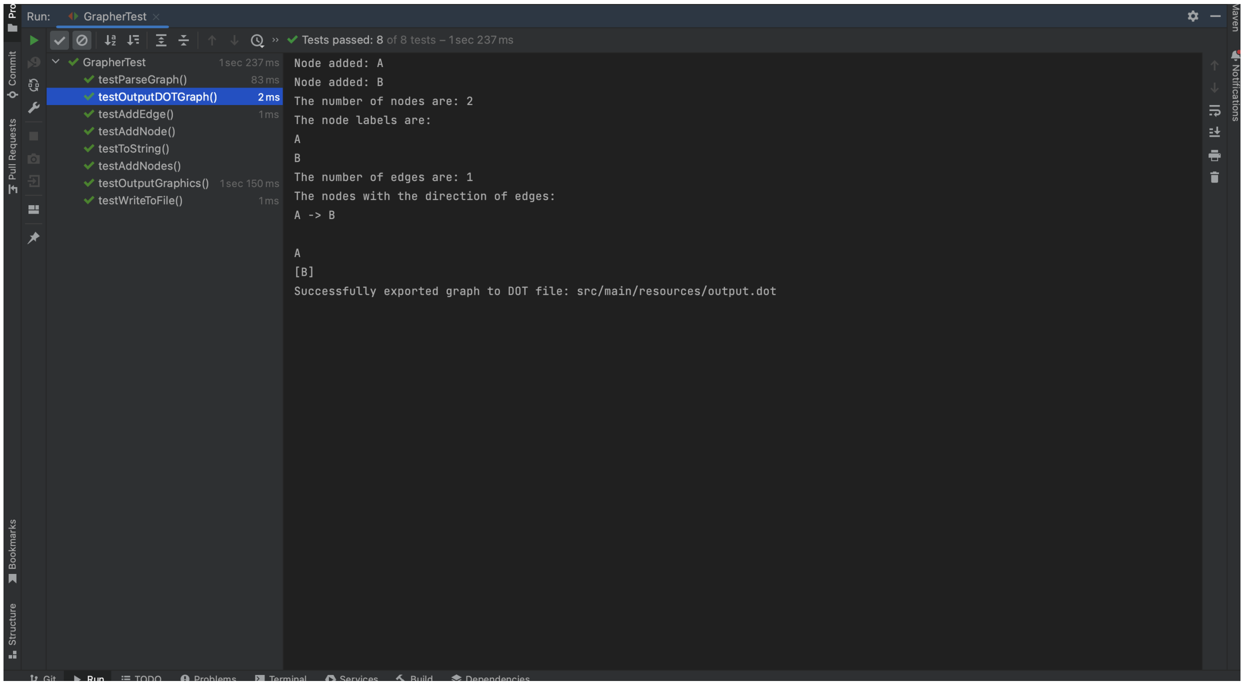


✓ Tests passed: 8 of 8 tests – 1 sec 237 ms

```
✓ GrapherTest              1 sec 237 ms
  ✓ testParseGraph()              83 ms
  ✓ testOutputDOTGraph()           2 ms
  ✓ testAddEdge()                  1 ms
  ✓ testAddNode()
  ✓ testToString()
  ✓ testAddNodes()
  ✓ testOutputGraphics()   1 sec 150 ms
  ✓ testWriteToFile()              1 ms
```

```
Graph successfully parsed!
```

- **Exported Graph (DOT Format):**

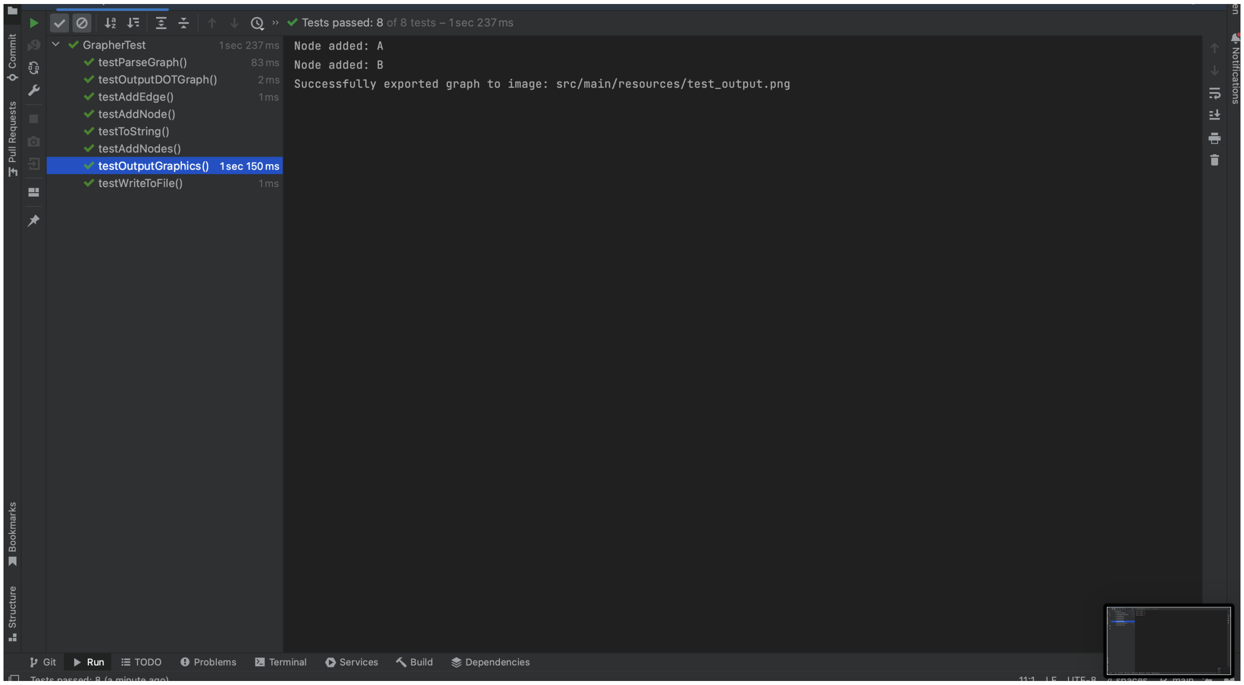


✓ Tests passed: 8 of 8 tests – 1 sec 237 ms

```
✓ GrapherTest              1 sec 237 ms
  ✓ testParseGraph()              83 ms
  ✓ testOutputDOTGraph()           2 ms
  ✓ testAddEdge()                  1 ms
  ✓ testAddNode()
  ✓ testToString()
  ✓ testAddNodes()
  ✓ testOutputGraphics()   1 sec 150 ms
  ✓ testWriteToFile()              1 ms
```

```
Node added: A
Node added: B
The number of nodes are: 2
The node labels are:
A
B
The number of edges are: 1
The nodes with the direction of edges:
A -> B

A
[B]
Successfully exported graph to DOT file: src/main/resources/output.dot
```

- **Exported Graph (Image):**

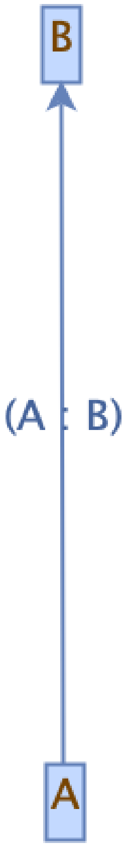


Tests passed: 8 of 8 tests – 1 sec 237 ms

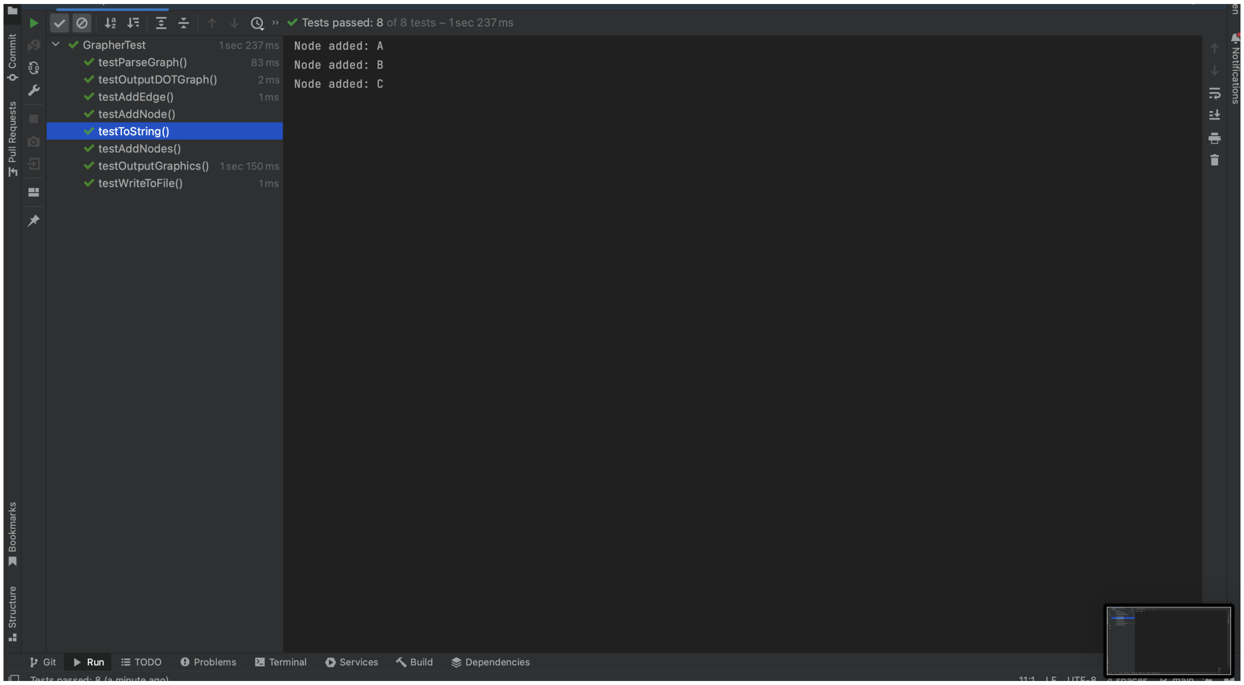
- GrapherTest 1 sec 237 ms
  - testParseGraph() 83 ms
  - testOutputDOTGraph() 2 ms
  - testAddEdge() 1 ms
  - testAddNode()
  - testToString()
  - testAddNodes()
  - testOutputGraphics() 1 sec 150 ms
  - testWriteToFile() 1 ms

```
Node added: A
Node added: B
Successfully exported graph to image: src/main/resources/test_output.png
```

- **Exported Graph (Image):**

- 

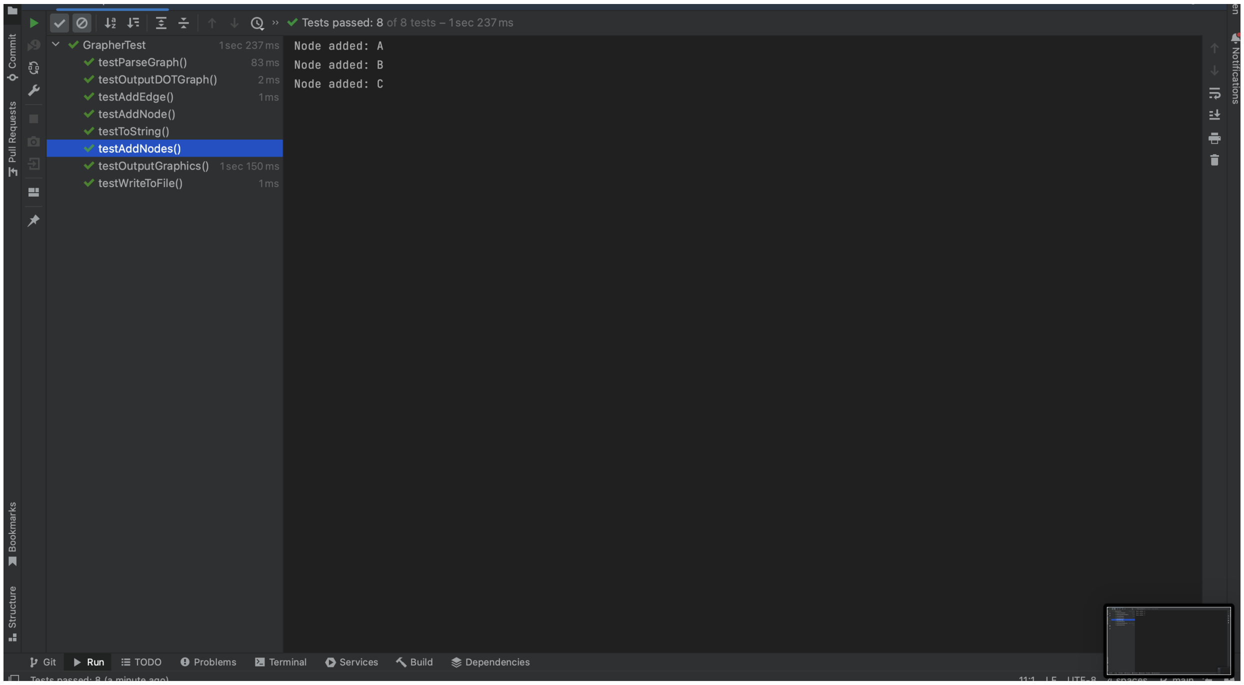
- **Output to String:**

- 

```
Tests passed: 8 of 8 tests – 1 sec 237 ms
GrapherTest                 1 sec 237 ms
  testParseGraph()                 83 ms
  testOutputDOTGraph()              2 ms
  testAddEdge()                     1 ms
  testAddNode()
  testToString()
  testAddNodes()
  testOutputGraphics()      1 sec 150 ms
  testWriteToFile()                 1 ms
```

```
Node added: A
Node added: B
Node added: C
```

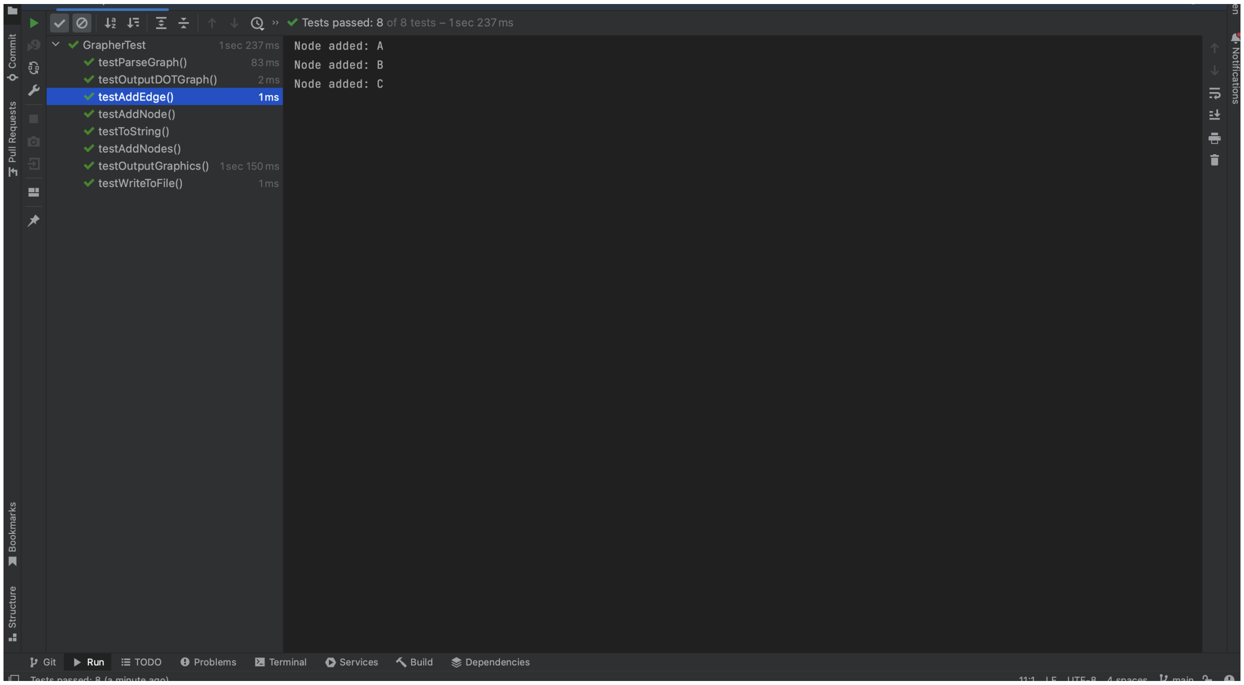
- ## Adding list of nodes:

- 

```
Tests passed: 8 of 8 tests – 1 sec 237 ms
GrapherTest                 1 sec 237 ms
  testParseGraph()                 83 ms
  testOutputDOTGraph()              2 ms
  testAddEdge()                     1 ms
  testAddNode()
  testToString()
  testAddNodes()
  testOutputGraphics()      1 sec 150 ms
  testWriteToFile()                 1 ms
```

```
Node added: A
Node added: B
Node added: C
```

- **Added Edges:**

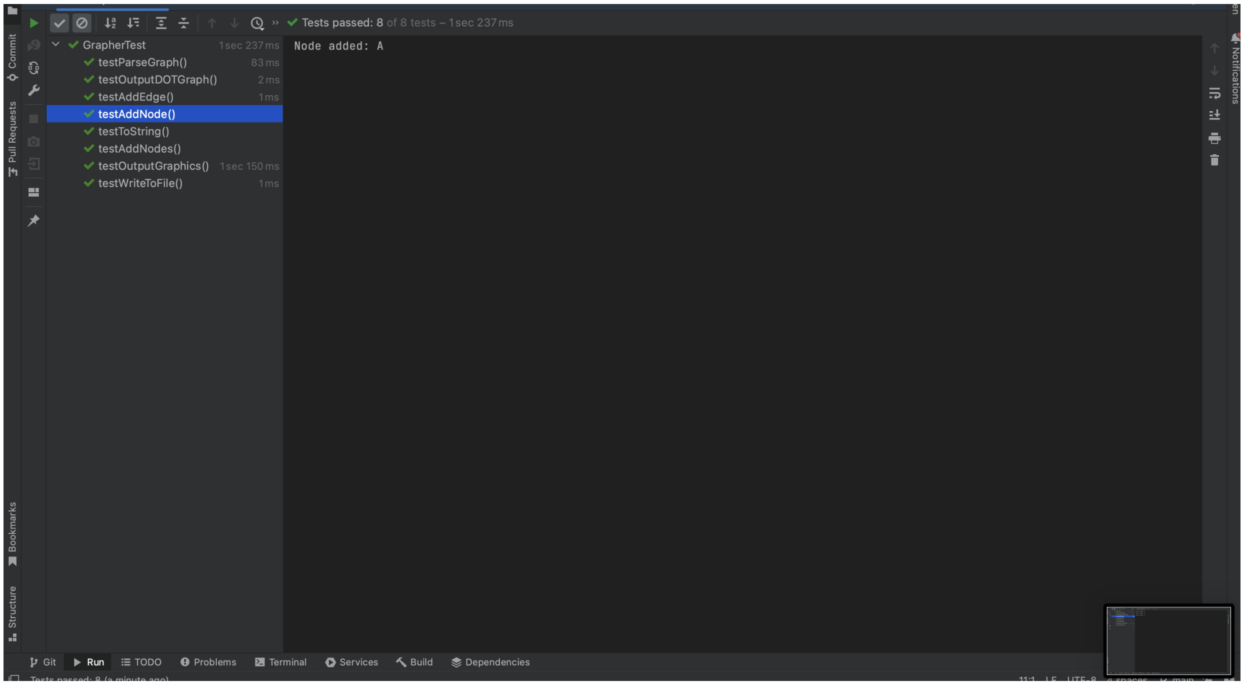


Tests passed: 8 of 8 tests – 1 sec 237 ms

- GrapherTest 1 sec 237 ms
  - testParseGraph() 83 ms
  - testOutputDOTGraph() 2 ms
  - testAddEdge() 1 ms
  - testAddNode()
  - testToString()
  - testAddNodes()
  - testOutputGraphics() 1 sec 150 ms
  - testWriteToFile() 1 ms

```
Node added: A
Node added: B
Node added: C
```

- **Added Node:**

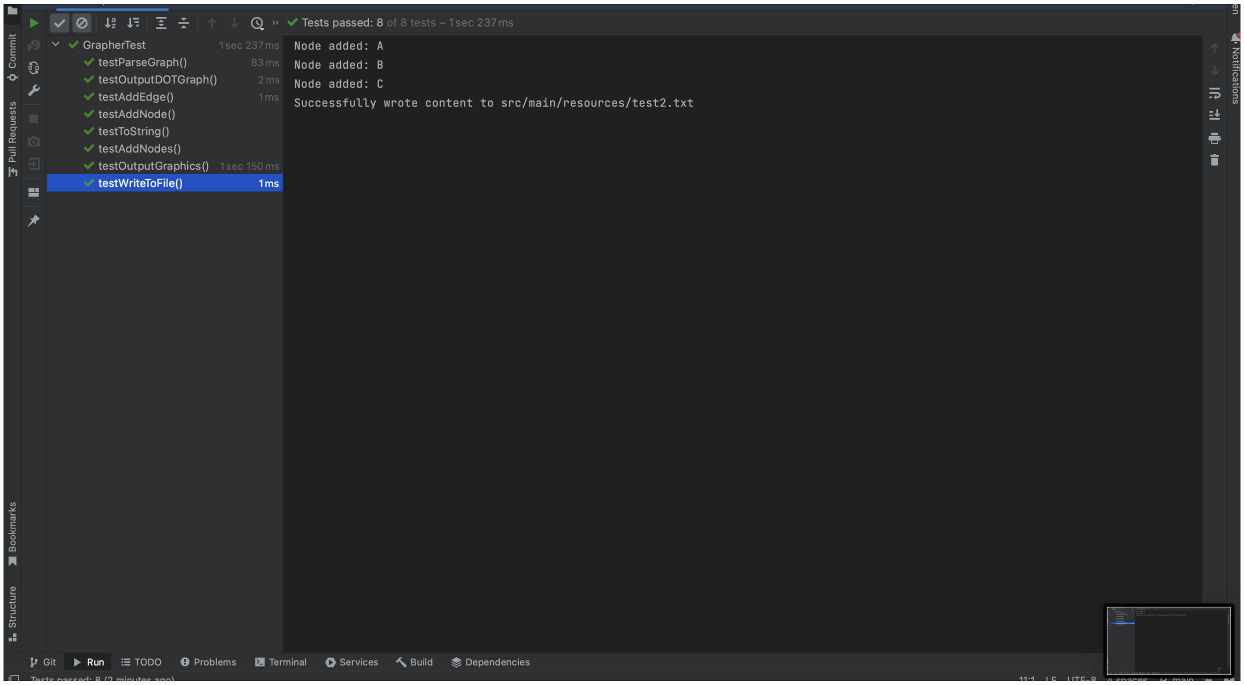


Tests passed: 8 of 8 tests – 1 sec 237 ms

- GrapherTest 1 sec 237 ms
  - testParseGraph() 83 ms
  - testOutputDOTGraph() 2 ms
  - testAddEdge() 1 ms
  - testAddNode()
  - testToString()
  - testAddNodes()
  - testOutputGraphics() 1 sec 150 ms
  - testWriteToFile() 1 ms

```
Node added: A
```

- **Write Graph to text file:**



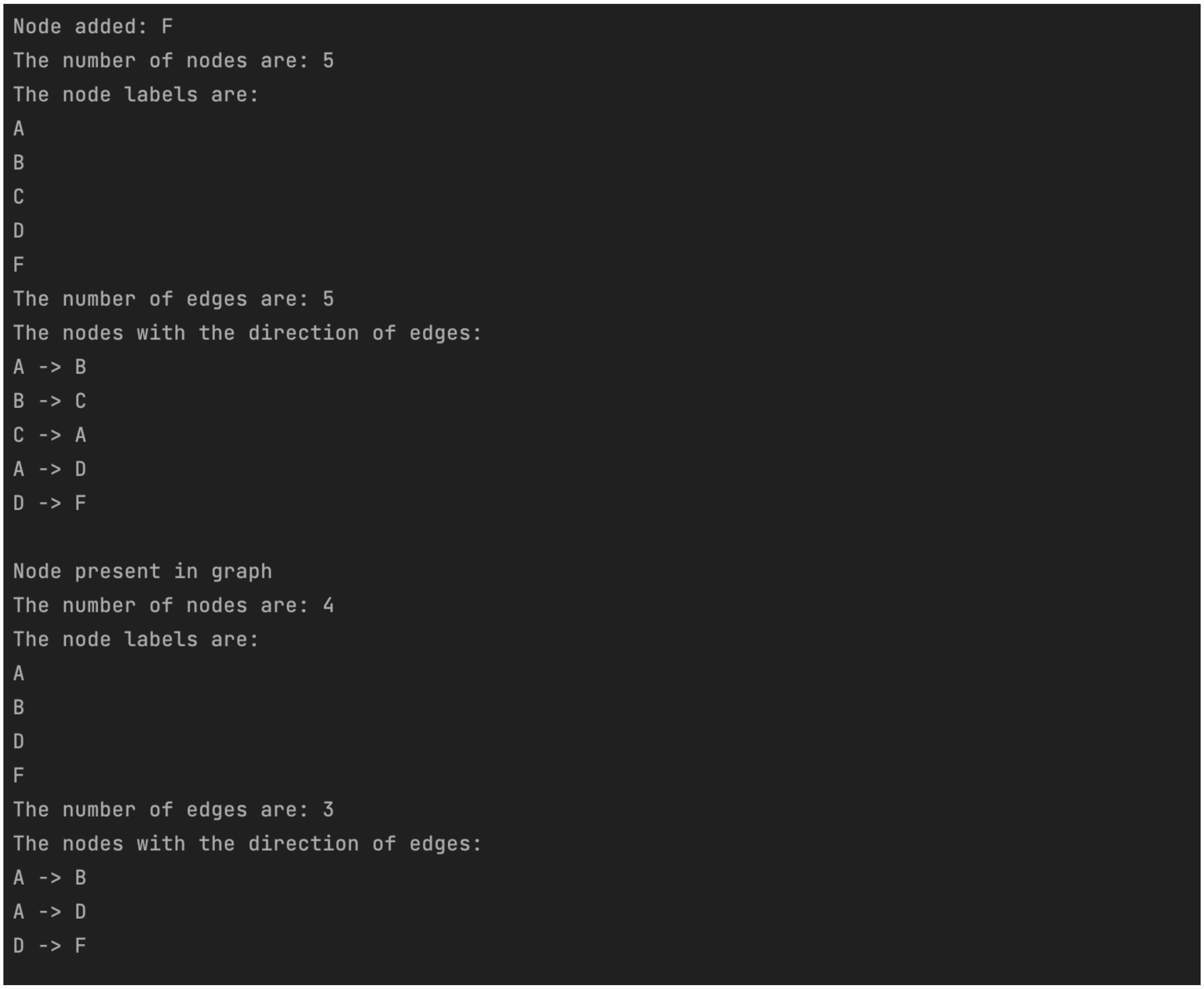
Tests passed: 8 of 8 tests – 1 sec 237 ms

```
GrapherTest                1 sec 237 ms
  testParseGraph()               83 ms
  testOutputDOTGraph()            2 ms
  testAddEdge()                   1 ms
  testAddNode()
  testToString()
  testAddNodes()
  testOutputGraphics()   1 sec 150 ms
  testWriteToFile()               1 ms
```

```
Node added: A
Node added: B
Node added: C
Successfully wrote content to src/main/resources/test2.txt
```

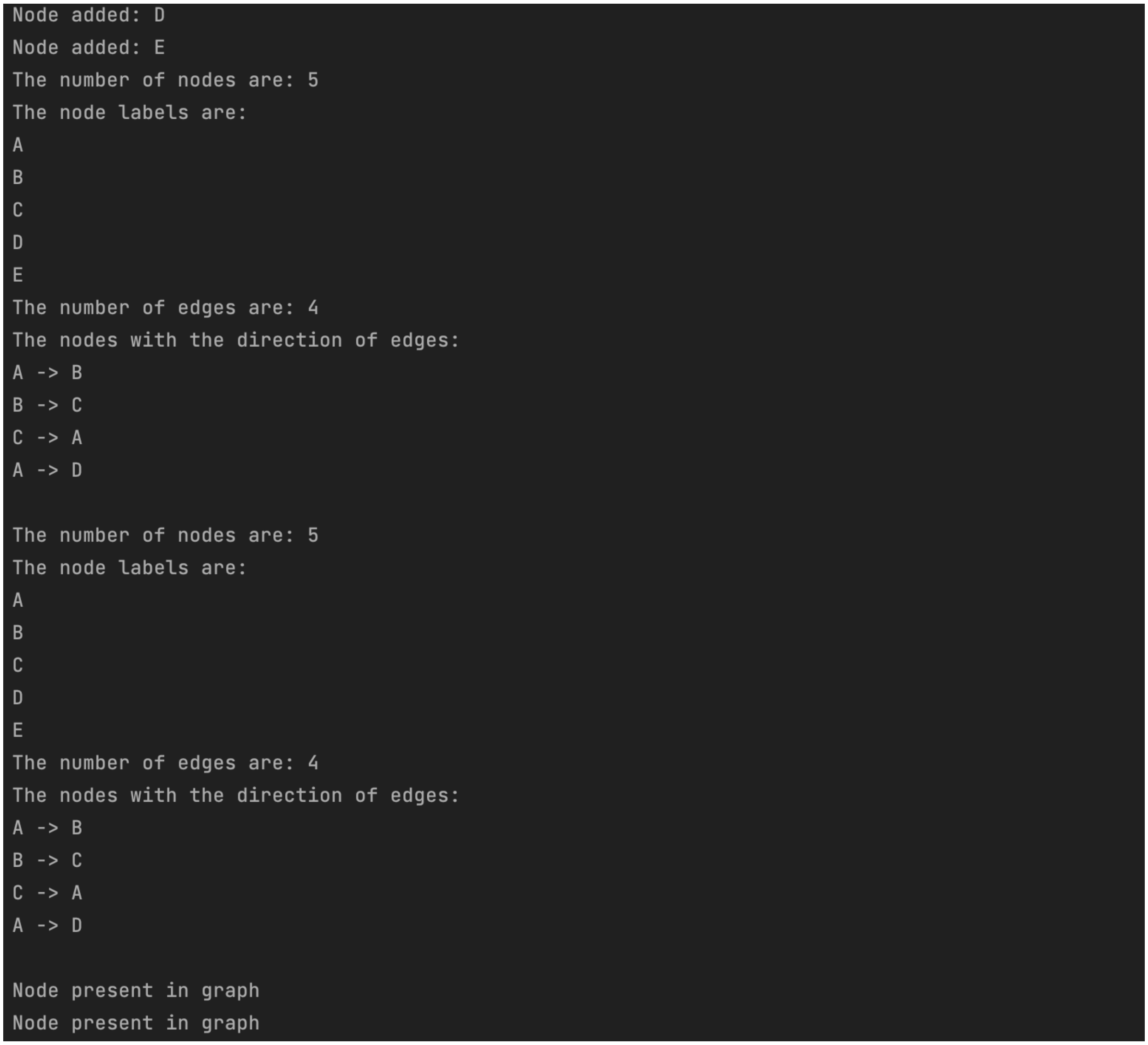
Tests passed: 8 (2 minutes ago)

- **Remove a Node from the Graph:**

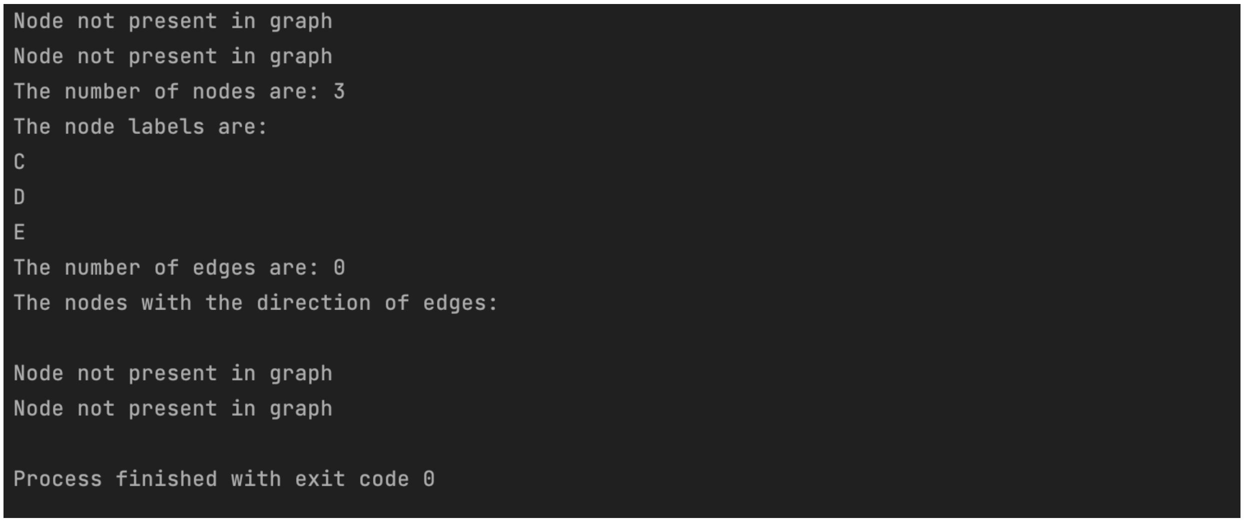


```
Node added: F
The number of nodes are: 5
The node labels are:
A
B
C
D
F
The number of edges are: 5
The nodes with the direction of edges:
A -> B
B -> C
C -> A
A -> D
D -> F

Node present in graph
The number of nodes are: 4
The node labels are:
A
B
D
F
The number of edges are: 3
The nodes with the direction of edges:
A -> B
A -> D
D -> F
```

- Remove multiple Nodes from the Graph:

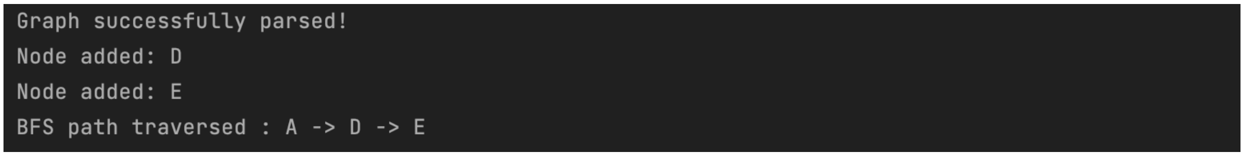


```
Node added: D
Node added: E
The number of nodes are: 5
The node labels are:
A
B
C
D
E
The number of edges are: 4
The nodes with the direction of edges:
A -> B
B -> C
C -> A
A -> D

The number of nodes are: 5
The node labels are:
A
B
C
D
E
The number of edges are: 4
The nodes with the direction of edges:
A -> B
B -> C
C -> A
A -> D

Node present in graph
Node present in graph
```



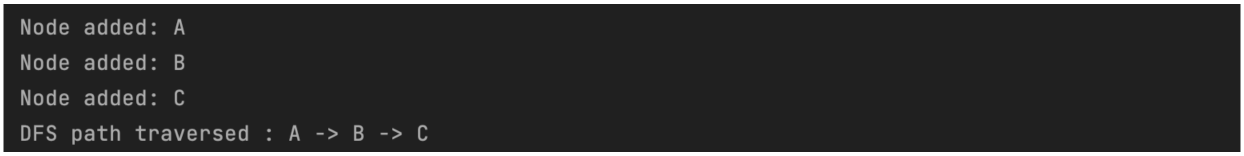
```
Node not present in graph
Node not present in graph
The number of nodes are: 3
The node labels are:
C
D
E
The number of edges are: 0
The nodes with the direction of edges:

Node not present in graph
Node not present in graph

Process finished with exit code 0
```

- Created a BFS branch:



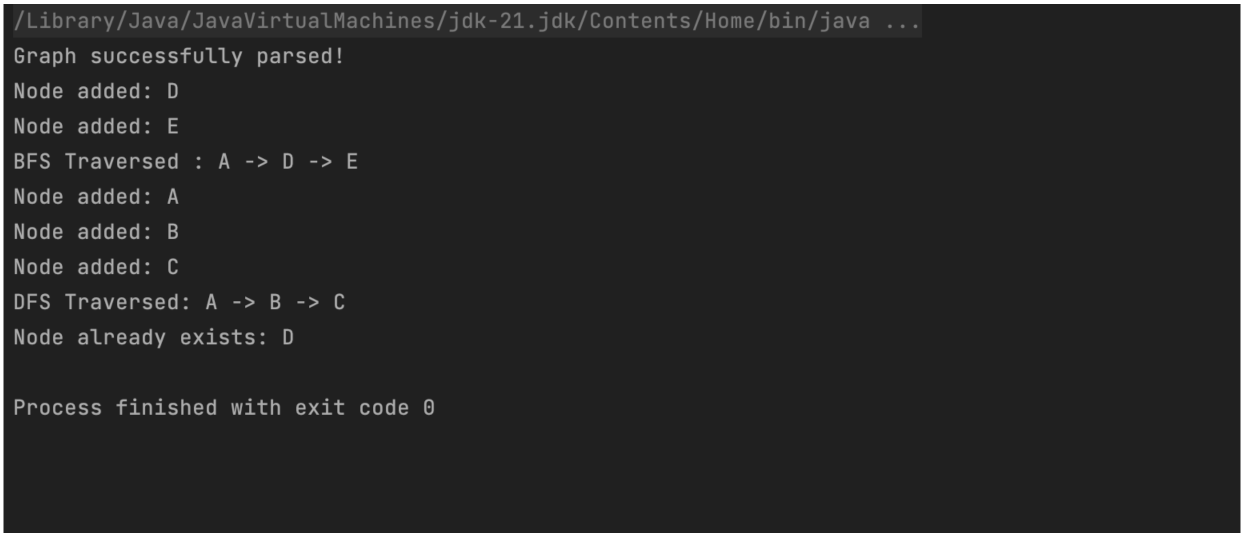
```
Graph successfully parsed!
Node added: D
Node added: E
BFS path traversed : A -> D -> E
```

- **Created a DFS branch:**



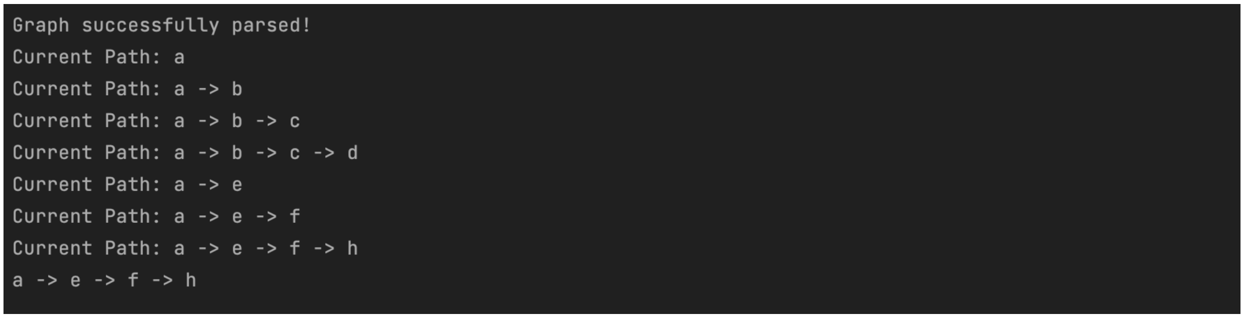
```
Node added: A
Node added: B
Node added: C
DFS path traversed : A -> B -> C
```

- **Searching for a path from source node to destination node (merged conflicts):**

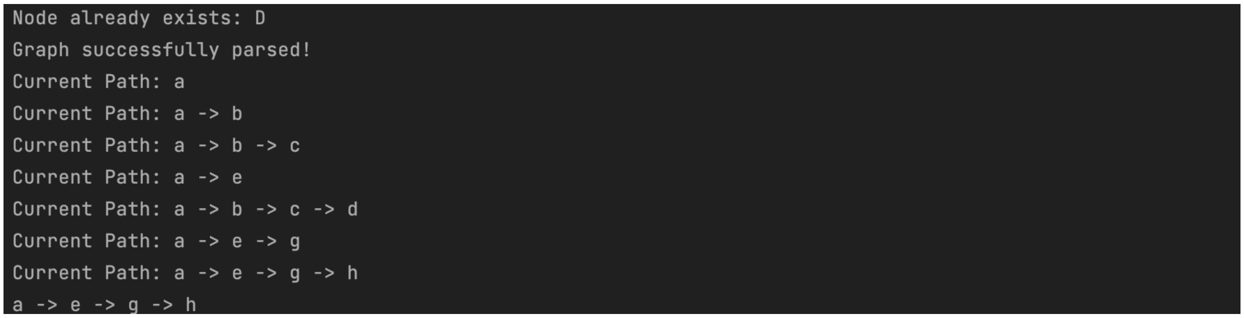


```
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java ...
Graph successfully parsed!
Node added: D
Node added: E
BFS Traversed : A -> D -> E
Node added: A
Node added: B
Node added: C
DFS Traversed: A -> B -> C
Node already exists: D

Process finished with exit code 0
```

- **Implementing a random walk path from a source node to destination node**



```
Graph successfully parsed!
Current Path: a
Current Path: a -> b
Current Path: a -> b -> c
Current Path: a -> b -> c -> d
Current Path: a -> e
Current Path: a -> e -> f
Current Path: a -> e -> f -> h
a -> e -> f -> h
```



```
Node already exists: D
Graph successfully parsed!
Current Path: a
Current Path: a -> b
Current Path: a -> b -> c
Current Path: a -> e
Current Path: a -> b -> c -> d
Current Path: a -> e -> g
Current Path: a -> e -> g -> h
a -> e -> g -> h
```

Commits

- Initial commit
- Built Maven. Also added feature 1
- Finished feature 1. This commit outputs the graph and writes it to a text file.
- Finished feature 2. Node and list of nodes can now be added. The result is reflected in the output of the graph.

- Finished Feature 3. The Edges are added to the graph and it is reflected when the graph is outputed.
- Finished Feature 4. The graph is visible in the dot file and a png image is also formed to visualize the graph.
- Added all the tests and the finishing touches
- Added APIs for removing a node, removing multiple nodes and removing an edge
- Created maven.yml
- Uncommented previous test cases
- Merged the maven into main
- Added bfs branch
- Added dfs branch
- Merged conflicts between dfs and bfs branches
- Performed 5 refactorings in the code
- Template pattern added for BFS and DFS
- Strategy pattern implemented for DFS BFS
- Random walk implemented
- Added comments to explain project structure better