CSE 464 README

GitHub Repo Link: https://github.com/divyavijayakumar72/CSE_464_2023

CI → https://github.com/divyavijayakumar72/CSE_464_2023/actions

Refactor branch: https://github.com/divyavijayakumar72/CSE_464_2023/tree/refactor

Pull Request: https://github.com/divyavijayakumar72/CSE_464_2023/pull/5

Main branch: https://github.com/divyavijayakumar72/CSE_464_2023

Bfs branch: https://github.com/divyavijayakumar72/CSE_464_2023/tree/bfs

Dfs branch: https://github.com/divyavijayakumar72/CSE_464_2023/tree/dfs

INTRODUCTION:

The goal of this project is to convert a DOT file to a graph object and perform graph manipulations on it. I have created a custom graph class, **GraphManager.java** where I have used a Hashmap to represent the graph. I used **Graphviz-java api library** to convert DOT file to a graph object. I have used **Main.java** class to run the project.

As per instructions, I have created a new branch **"refactor"**.

STEP 1: REFACTORING

Refactor 1:

Commit Link:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/8620d42e795279bcb963404e8e32bef870448043

Action workflow Link:

https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4579352642

Refactor 2:

Commit Link:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/45cdea11d916fbcf467da9c0b78a9542f101989d

Action workflow Link:

https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4579470613

Refactor 3:

Commit Link:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/2a0f28caffa8cfe1ab8e5faeeb33c28dfcad9261

Action workflow Link:

https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4584844373

Refactor 4:

Commit Link:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/8aa65ca01e747b7bdb5b29ff134b81c7ec87c461

Action workflow Link:
https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4585254617

Refactor 5:
Commit Link:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/4fd3708270c568d58b84bf1a289594e70831aad3

Action Workflow Link:
https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4585273095

STEP 2: TEMPLATE PATTERN
Action Workflow:
https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4593925405

Commit:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/b6451242209b58b808184f5558af55aa10310eb8

The Template Pattern is a design pattern that defines the skeleton of an algorithm in a class, allowing subclasses to implement specific steps of the algorithm. I have defined an abstract class **GraphTemplatePattern.java** that defines the common steps of both the search algorithms. It contains a method **traverse()** that initializes HashSet, Queue, and HashMap. This method calls the abstract method **traverseHelperFunc ()** that performs graph traversal.

I have created two java class **BFS.java** and **DFS.java** which extend the abstract class and implement **traverseHelperFunc()** for each class. The **BFS.java** uses a queue that stores the visited nodes while the **DFS.java** uses recursion to reach the destination node from the source node. The **getNeighbors()** method will obtain the neighbors of the current node from the graph. The **traverse()** method constructs the path from source node to destination node, and returns result in the form of Path object.

```
48
49          GraphTemplatePattern bfs = new BFS();
50          System.out.println("Step 2 BFS");
51          bfs.traverse(graph.map,  source: "a",  destination: "d");
52
53          GraphTemplatePattern dfs = new DFS();
54          System.out.println("Step 2 DFS");
55          dfs.traverse(graph.map,  source: "b",  destination: "d");
56
```

The above lines of code in **Main.java** will run the BFS and DFS using Template Pattern, and the output will be printed in console as follows. I have used **input.dot** as input file and I have given source node and destination node as **a** and **d** respectively for BFS. I have given source and destination nodes as **b** and **d** for DFS.

The following is the output:

```
Step 2 BFS
The path using Template Pattern is a -> b -> c -> d
Step 2 DFS
The path using Template Pattern is b -> c -> d
```

STEP 3: STRATEGY PATTERN

Action Workflow:

https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4706091921

Commit:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/7e3628f72e4d4f8076391ac5d571c81c5132a587

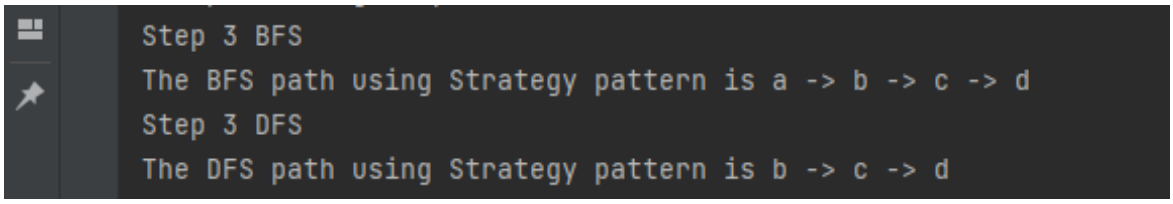
The strategy pattern is a design pattern that defines a family of algorithms, encapsulates each one, and enables the algorithms to be selected at runtime. I have defined an interface **SearchAlgorithm.java** that has a method **search().** The classes **BFSStrategy.java** and **DFSStrategy.java** implement this interface and implement their version of the search() method in their respective classes.
The **BFSStrategy.java,** like the name suggests, implements BFS graph traversal using Strategy Pattern. It initializes a Queue and a HashSet, then it gets the first node from the queue in order and checks if that element is destination node. If the check is false, the neighbors of the node that are not visited yet are added to the Queue and HashSet. When the destination node is obtained, the Path object that returns the path from source node to destination node Is returned.
The **DFSStrategy.java** implements DFS graph traversal using Strategy Pattern. It initializes a HashSet and an ArrayList before using recursion to explore the neighbors of the current node and adds them to the HashSet and ArrayList. If the destination node is not found in one path, the method backtracks and removes the last node until an unvisited node is obtained. When the destination node is obtained, the Path object that returns the path from source node to destination node Is returned.
The **enum** values **BFS** and **DFS** are used to select which algorithm is to be executed.

```
System.out.println("Step 3 BFS");
graph.GraphSearch( src: "a",  dst: "d", Algorithm.BFS);

System.out.println("Step 3 DFS");
graph.GraphSearch( src: "b",  dst: "d", Algorithm.DFS);
```

The above lines of code in **Main.java** will execute the **GraphSearch()** method implemented in the **GraphManager.class** which in turn will invoke the **BFSStrategy.java** and **DFSStrategy.java** classes. I have used **input.dot** as input file and I have given source node and destination node as **a** and **d** respectively for BFS. I have given source and destination nodes as **b** and **d** for DFS.

The following is the output printed in the console:



STEP 4: RANDOM WALK SEARCH

Action Workflow:
https://github.com/divyavijayakumar72/CSE_464_2023/actions/runs/4710346574

Commit:
https://github.com/divyavijayakumar72/CSE_464_2023/commit/6a4ffaf80516f166fbba07671015abdf5967b48f

I implemented this algorithm using both Template and Strategy Pattern in the source code. For the Template Pattern I defined **RandomWalkTemplate.java** class that extends the existing **GraphTemplatePattern.java** class. For the Strategy Pattern I implemented **RandomWalkStrategy.java** class that implements the existing interface **SearchAlgorithm.java.**

```
/* TEMPLATE PATTERN RANDOM WALK SEARCH*/
GraphTemplatePattern randomWalk = new RandomWalkTemplate();
System.out.println("Step 4 Random Walk Search with Template Pattern");
graph.parseGraph( filepath: "random-walk-input.dot");
randomWalk.traverse(graph.map, source: "a", destination: "c");

/* STRATEGY PATTERN RANDOM WALK SEARCH*/
System.out.println("Step 4 Random Walk Search with Strategy Pattern");
graph.GraphSearch( src: "a", dst: "c", Algorithm.RWS);
```

The above code will execute the random search using both the Template and Strategy patterns with the input file as random-walk-input.dot. We can also use the below line of code in line 71 to view the outputs for the path **a -> h** using Template Pattern.

```
randomWalk.traverse(graph.map, "a", "h");
```

We can use the below line of code in line 75 to view the outputs for the path **a -> h** using Strategy Pattern.

```
graph.GraphSearch("a", "c", Algorithm.RWS);
```

I have used **random-walk-input.dot** file uploaded in the source code as the example graph and **a -> c** as the example path. I have also tested with the path **a -> h** to ensure that the code is working as required.

I ran the code for Random Walk Search multiple times and got the below outputs when I ran the code with the source node as **a** and destination node as **c**. The results indicate the randomness of the search every time I run the code.

```
Random walk search using the input file provided in Assignment instructions
Visiting Path {nodes = [a]}
Visiting Path {nodes = [a, b]}
Visiting Path {nodes = [a, b, c]}
The random path is a -> b -> c
```

```
Random walk search using the input file provided in Assignment instructions
Visiting Path {nodes = [a]}
Visiting Path {nodes = [a, e]}
Visiting Path {nodes = [a, e, g]}
Visiting Path {nodes = [a, e, g, h]}
Visiting Path {nodes = [a, e, f]}
Visiting Path {nodes = [a, b]}
Visiting Path {nodes = [a, b, c]}
The random path is a -> b -> c
```

```
Random walk search using the input file provided in Assignment instructions
Visiting Path {nodes = [a]}
Visiting Path {nodes = [a, e]}
Visiting Path {nodes = [a, e, f]}
Visiting Path {nodes = [a, e, f, h]}
Visiting Path {nodes = [a, e, g]}
Visiting Path {nodes = [a, b]}
Visiting Path {nodes = [a, b, c]}
The random path is a -> b -> c
```