Adin Dorf
4/21/2023

CSE 464 Project Part 3

Repo: https://github.com/AdinDorf/CSE4642023adorf

1. Refactors:

   Reference to the online catalogue of refactor types: https://refactoring.com/catalog/
   So I did my refactors a bit weird. Each commit has some combination of Replace Function with Command, Rename Variable, Replace Error Code with Exception, Remove Middle-Man, Remove Dead Code, Combine Functions into Class, more that I'm failing to specify. Throughout parts 1 and 2 of the project, I **struggled** to correctly use the GraphViz library to correctly implement the necessary methods in a clean, efficient, and readable way. The library is structured in such a way that it's very difficult to access individual nodes/edges from other nodes. Directly manipulating graphs was also sophisticated and needlessly difficult due to unfortunate encapsulation decisions by the developers. The goal of this series of refactors was to change the internal structure of the program for the better (in terms of readability, efficiency, ease of modification) without changing the external behavior. Technically, this entire process could probably be considered one refactor with many different parts, but considering the bar for refactoring is as high as "Renaming a Variable" I figured it would be ok to break this apart into several commits. I started by disabling most of the functionality of the program, and gradually brought features back online through each commit.

   a. Refactor 1:
      Commit:
      https://github.com/AdinDorf/CSE4642023adorf/commit/f091102fbc095ff9f335e645cb8c1b2e3bf30226
      Types: Remove Dead Code, Replace Error Code with Exception
      Description: This code adds the baseline parts for my refactor. Prior to this commit, my code used custom classes extending the MutableGraph/MutableNode classes of the GraphViz package to fulfil the addNode, removeNode, BFS, DFS, etc. requirements of the previous parts of the project. Here I create new structures Graph.java, Node.java, and Edge.java for this purpose that are **not** reliant on GraphViz. More about GraphViz is still tied in is explained later. I also added two new custom Exceptions, EdgeNotFound and NodeNotFound, to be thrown when those cases got hit as opposed to a null-pointer or more generalized exception.

   b. Refactor 2:
      Commit:
      https://github.com/AdinDorf/CSE4642023adorf/commit/52dc02e71d4c3f1dc92b2261d5f3a7984ef28c97
      Types: Replace Error Code with Exception, Remove Dead Code
      Description: In commit 2, I reworked the dot parser. It's honestly a bit janky, but it consolidates the janky code into one method instead of letting it be scattered throughout the project, so I consider that an improvement. The new 'dot' function still uses the GraphViz parser to take in a file and create a MutableGraph, but it now takes the MutableGraph and converts it into the much-more-simplified 'Graph' class that I created in refactor commit 1. I consider this 'removing dead code' as all the unused parts of MutableGraph and now no-longer being parsed or kept track of.

c. Refactor 3:
   Commit:
   https://github.com/AdinDorf/CSE4642023adorf/commit/81157684c7b293a7a0adf2b3833eb21e3127d61f
   Types: Remove Dead Code
   Description: Commit 3 re-enables the toPNG() method by converting my Graph class back into a MutableGraph, then into a PNG through the GraphViz parser.
d. Refactor 4:
   Commit:
   https://github.com/AdinDorf/CSE4642023adorf/commit/95ea3450f903113e43577322caa8261a4d45e438
   Types: Substitute Algorithm
   Description: This addresses the primary reason for most of these commits. This commit rewrites the graph search algorithms to make them much faster. Prior to this commit, each iteration of the graph search required findNode() to be run, drastically increasing the runtime of each algorithm. Using my own class, I was able to access each node from the previous node making for a much quicker search. I also modified the Path class to account for this.
e. Refactor 5:
   Commit:
   https://github.com/AdinDorf/CSE4642023adorf/commit/88b60ede7e16bc5f142e3335503f36347b70c3ea
   Types: Rename Variable
f. Description: This commit includes clarification via a renamed variable (p -> shortestPath) and adds output showing the entire path traversed as opposed to only the shortest path found.

2. Template Design Pattern

   Commit:
   https://github.com/AdinDorf/CSE4642023adorf/commit/1d68a8aa9139a8ff31221b6d8e01026b2c2aeb13

   The template pattern is applied through the new Search.java abstract class. GraphSearch now passes in a Search object to Graph.java, which can either be a BFS or a DFS. BFS.java and DFS.java are classes that inherit Search.java, and each override the addNode and removeNode functionality.

3. Strategy Design Pattern

   Commit:
   https://github.com/AdinDorf/CSE4642023adorf/commit/16befa703d4f917ba25e0b0e79e70cc1b68c6c19

   The Strategy pattern is applied through the following new classes: AddBehavior, BFSAddNode, DFSAddNode, RemoveBehavior, BFSRemoveNode, DFSRemoveNode, WalkBehavior, DFSWalk, BFSWalk

The Search subclasses (BFS, DFS, RWF) now use AddBehavior, RemoveBehavior, and WalkBehavior, to call their respective functions. Each of these are interfaces which have subclasses that implement them depending on which algorithm is chosen.

Important note: Prior to writing the code for part 3, I had the realization that I might have already fulfilled the requirements in part 2 by passing in the abstract Search class into the GraphSearch method. I was unsure of whether the goal of part 3 was to undo the work of part 2 to implement a true strategy design pattern with multiple interfaces or to do what I already did for part 2 but in two different commits and call it a strategy pattern. I later merged these two branches to create a commit that uses both the Strategy and Template patterns, but that was before this commit was created. If you would like to see the implementation of both patterns simultaneously, see the commit for Random Walk Search using the Strategy pattern.

4. Random Walk Search

Using the Template design pattern:

Commit:
https://github.com/AdinDorf/CSE4642023adorf/commit/eb50d77953b5d5cc19ec3feec602a6efb1ce4d07

This commit builds on-top of the part 2 branch. It utilizes the template design pattern outlined in that part, but with a few modifications to allow for a completely modular template. Since the main difference between Random Walk Search and BFS/DFS is how nodes get chosen after each iteration, I added an abstract walk() method that refills the data structure with new nodes. The BFS and DFS walk methods are the same, but the Random Walk Search walk() chooses a random node via a random number generator to add to the data structure. Alternatively, if an end node is reached, the start node is re-added to the queue.

Using the Strategy design pattern:

Commit:
https://github.com/AdinDorf/CSE4642023adorf/commit/c2aaf2da992a2098d26ecc9e16a9e6cf2a951ef0
This commit builds on-top of the refactor-strategy branch I created for part 3. It implements both the strategy and template patterns, but with added AddNode, RemoveNode, and Walk implementations for Random Walk Search.

Instructions:
When running the program, a menu should be the first thing displayed on the command line. You need to start by either running addNode to create a node on an empty graph, or by running ‘dot’ and inputting the path to your test graph. Included in the project is the test graph for project 3 at *src/test/java/org/example/test3.dot*. Adding nodes can be done by inputting addNode, then inputting the name of the node. It is assumed that node names are unique. The same goes for

adding edges. Removing a node can be accomplished by inputting removeNode at the menu and typing the name of the node. RemoveEdge can be used to get rid of an edge given the source and destination nodes. Attempting to remove a non-existent node or edge will throw an exception. Using the BFS, DFS, and RWS algorithms can be done by inputting graphSearch, the source node, the destination node, then which algorithm to use (one of bfs, dfs, or rand). The result of the search will then be printed above the next menu print.

Each example below uses the test3.dot file included in the repository (which is also same test file posted on canvas for use testing project 3)

Random Walk Search Examples:

```
graphSearch: return a path between two nodes

graphSearch
Enter the source node name
a
Enter the destination node name
h
Enter the search algorithm to use (bfs, dfs, rand)
rand
Path determined by rand: a -> e -> g -> h
Order of traversal: a -> b -> c -> d -> a -> b -> c -> d -> a -> b -> c -> d -> a -> b -> c -> d -> a -> e -> g -> h
```

```
graphSearch
Enter the source node name
a
Enter the destination node name
h
Enter the search algorithm to use (bfs, dfs, rand)
rand
Path determined by rand: a -> e -> g -> h
Order of traversal: a -> e -> g -> h
```

```
graphSearch
Enter the source node name
a
Enter the destination node name
h
Enter the search algorithm to use (bfs, dfs, rand)
rand
Path determined by rand: a -> e -> g -> h
Order of traversal: a -> b -> c -> d -> a -> e -> f -> h
```

DFS Examples

Adin Dorf
4/21/2023

```
graphSearch
Enter the source node name
a
Enter the destination node name
c
Enter the search algorithm to use (bfs, dfs, rand)
dfs
Path determined by dfs: a -> b -> c
Order of traversal: a -> e -> g -> h -> f -> b -> c
```

```
graphSearch
Enter the source node name
a
Enter the destination node name
h
Enter the search algorithm to use (bfs, dfs, rand)
dfs
Path determined by dfs: a -> e -> g -> h
Order of traversal: a -> e -> g -> h
```

BFS Examples

```
graphSearch
Enter the source node name
a
Enter the destination node name
c
Enter the search algorithm to use (bfs, dfs, rand)
bfs
Path determined by bfs: a -> b -> c
Order of traversal: a -> b -> e -> c
```

Adin Dorf
4/21/2023

```
graphSearch
Enter the source node name
a
Enter the destination node name
h
Enter the search algorithm to use (bfs, dfs, rand)
bfs
Path determined by bfs: a -> e -> f -> h
Order of traversal: a -> b -> e -> c -> f -> g -> d -> h
```