CSE222 - Assignment 7-8
Sorting Algorithms, Adjacency Matrix, and Graph Coloring

Introduction:

In this assignment, both sorting algorithms were implemented and an undirected graph structure based on an adjacency matrix was created.

In the sorting part, different sorting types were implemented through an abstract class called GTUSorter. This class is generic and allows sorting arrays of different types using a comparator.GTUInsertSort uses the insertion sort algorithm. GTUQuickSort uses the quick sort algorithm. GTUSelectSort is a sorter that works with the selection sort algorithm.

In the graph part, the MatrixGraph class was developed based on the GTUGraph interface. This class stores connections in the graph using an adjacency matrix. Each row is represented by a boolean array, and these rows are stored using the AdjacencyVect class. The AdjacencyVect class implements the Collection interface in Java and has an `iterator()` method that only returns the neighbors marked as `true`. In this way, accessing a node's neighbors becomes possible.

Environment:

For this assignment, VS Code was used as the development environment. The code was written and compiled using Java 20. The operating system used was Macos and tested on Ubuntu 24.04.

The project was compiled and executed through the terminal using a Makefile. JavaDoc documentation was also generated using Makefile. No external libraries were used.

Complexity Analysis:

The GTUInsertSort algorithm uses the insertion sort method. In the worst and average cases, its time complexity is $O(n^2)$. This is because, for each element, the algorithm may need to compare it with all the elements before it and shift them to insert the current element into the correct position.

The GTUQuickSort algorithm uses the quick sort technique with random pivot selection. On average, its time complexity is $O(n \log n)$, but in the worst case (for example, if the pivot is always the smallest or largest element), it can become $O(n^2)$. To make it more efficient for small subarrays, it uses another sorting algorithm like GTUInsertSort as a fallback sorter.

The GTUSelectSort algorithm is based on selection sort. Its time complexity is always $O(n^2)$ because in each step it searches the remaining part of the array to find the smallest element.

In the graph part, the MatrixGraph class uses an adjacency matrix to store edges. The methods `setEdge(v1, v2)` and `getEdge(v1, v2)` both run in $O(1)$ time because they use direct indexing into a boolean array. The `getNeighbors(v)` method runs in $O(n)$ time because it has to check every column in the row to find which neighbors exist.

The AdjacencyVect class also provides an iterator. The total time complexity of the iterator is $O(n)$, because in the worst case it may need to go through the entire array to find the elements marked as `true`.

Testing:

I tested each sorting algorithm separately. I created specific test functions for each one inside the MyTest class. I used sample integer arrays and compared the result after sorting with the array that was supposed to be sorted. Each of the GTUInsertSort, GTUQuickSort, and GTUSelectSort algorithms was tested with the same input to check whether they sorted the array correctly.

The GTUQuickSort algorithm was also tested with different sorters (GTUInsertSort and GTUSelectSort) and different partition limit values. In this way, it was checked whether the fallback sorting mechanism given through the constructor worked correctly.

For the graph structure, I created a MatrixGraph and added edges to it. I checked whether these edges were really added using the getEdge function. Using the getNeighbors function, I tested whether each node returned only the correct neighbors. I also tested the reset function by shrinking the graph after creating it and checking if the edges were cleared.

In addition, I tested the AdjacencyVect class independently. I checked whether the add, remove, contains, and the custom iterator() methods worked correctly.

I ran all the tests through the main method and printed the result outputs.

AI Usage:

While working on this assignment, I used AI to understand the difference between the two `toArray` methods in the Collection interface. Based on that, I implemented and overridden the methods myself.

I also used AI while writing JavaDoc comments, to make sure they were in the correct format and style. Additionally, I asked for help to understand the causes of some small errors I encountered during the implementation.

Challanges:

One of the challenges I faced was understanding how sorting algorithms work and designing them correctly. Before writing the code, I needed to fully understand the logic behind the algorithms. So, I used the course book for help and studied how sorting algorithms work. After I understood the logic, I started coding.

Another problem was that I didn't fully understand how to override the functions in the Collection interface. I learned the structure of these functions with the help of AI. Some of the functions had parameters of type `Object`, but in our case, we needed

to work with `Integer`. This was a challenge for me because I wasn't sure if using casting would go against OOP principles. I tried to find other ways, but in the end, I realized that casting was necessary and acceptable in this case, and it did not cause any issues. I also had some trouble while writing the iterator, but I was able to complete it with the help of external resources.

I developed the project on a macOS system, but I wanted to test if it worked in other environments as well. On my own system with Ubuntu 18, I got a compile error when I ran the `make build` command. At first, I thought the problem was in my code, but when I tried the same project on a system with Ubuntu 24.04, everything worked fine. Overall, even though I had some small difficulties, I believe I completed the assignment successfully.

Conclusion:

With this assignment, I learned how sorting algorithms work not just in theory but also in practice. Working with generic structures, using comparators, and designing different algorithms helped me understand these concepts better.

Writing my own iterator, overriding different methods, and working with generic types helped me improve my programming skills. Turning the logic of the algorithms into real code also made me understand how they actually work.

In general, I improved myself in turning theoretical knowledge into practical work. By solving the technical problems I faced, I gained a stronger and more organized way of thinking in software development.

Sümeyye Dalga

220104004091