GEBZE TECHNICAL UNIVERSITY, COMPUTER ENGINEERING DEPARTMENT

CSE222 - Assignment 7-8

Sorting Algorithms, Adjacency Matrix, and Graph Coloring

Due May 25, 2025

In this assignment, you are expected to implement multiple sorting algorithms and a graph class that utilizes the adjacency matrix structure. You will use your implementations in an example use case, which is a greedy graph coloring algorithm, showcasing the functionality of your implementations.

1 INTRODUCTION

This is the final assignment, and will be evaluated out of 200 points, plus 40 points for bonus objectives. See section 9 for more information about grading.

The assignment is split into 3 main parts:

1. Implement multiple generic Sorter classes.
2. Implement a graph class that stores edges in an adjacency matrix.
3. Test the functionality of your code.

Each of the above tasks is explained in its corresponding section, please read them carefully.

2 SORTING ALGORITHMS (70 POINTS)

In this section, you are expected to implement the classes GTUQuickSort, GTUInsertSort, and optionally GTUSelectSort that extend the abstract class GTUSorter, which is provided with the assignment.

2.1 GTUSorter Abstract Class

GTUSorter represents a class that can sort arrays using a given comparator instance. The class is provided as in the following:

```
public abstract class GTUSorter {
    public <T> void sort(T[] arr, Comparator<T> comparator) {
        sort(arr, 0, arr.length, comparator);
    }

    protected abstract <T> void sort(
        T[] arr,
        int start,
        int end,
        Comparator<T> comparator
    );
}
```

The sort methods are generic, allowing them to sort any array using the provided comparator. The sort methods are expected to sort the elements in ascending order according to the comparator's feedback. You can take the standard behavior of compareTo method in the Comparable interface as a reference to determine the direction. This behavior should be present in all implementations of these methods.

The abstract version of the sort method takes the start (inclusive) and end (exclusive) indices to sort subarrays. This will be needed for recursive algorithms, and will also be useful for our quick sort implementation.

2.2 GTUInsertSort Class

The first class you are expected to implement is GTUInsertSort, which uses the insertion sort algorithm to sort the array. This class should extend the GTUSorter abstract class.

2.3 GTUQuickSort Class

The second class you are expected to implement is GTUQuickSort. As the name implies, this class should use the quick sort algorithm. The implementation should satisfy the following requirements:

- Extends the GTUSorter abstract class.
- The pivot is picked at random.
- Uses another sorting algorithm for partitions smaller than the given partition limit.
- Continue using quick sort when another sorter or a partition limit is not defined.
- The Sorter and partition limit must only be given using the constructor. (To prevent the users from shooting themselves in the foot)

3 Graph Implementation Using Adjacency Matrix (70 Points)

In this section, you are expected to implement the MatrixGraph class using the GTUGraph interface. MatrixGraph will use the adjacency matrix format to store edge information. Also, you will implement the AdjacencyVect class, which represents a vector in an adjacency matrix, and use it in the implementation of MatrixGraph.

3.1 GTUGraph Interface

This interface represents a graph with basic operations. The interface is as follows:

```
public interface GTUGraph {
    // Set edge between v1 and v2.
    Boolean setEdge(int v1, int v2);

    // Check if v1 and v2 have an edge in between.
    Boolean getEdge(int v1, int v2);

    // Get the neighbors of vertex v.
    Collection<Integer> getNeighbors(int v);

    // Get the number of vertices in the graph.
    int size();

    // Reset the graph and change its size.
    void reset(int size);

    // Read edges from the file and store them in graph.
    static void readGraph(String filePath, GTUGraph graph) {...}
}
```

3.2 AdjacencyVect Class

You are expected to implement the AdjacencyVect class based on the Collection interface. This class will be used the to create the adjacency matrix in MatrixGraph class implementation in section 3.3.

An AdjacencyVect is a vector that stores binary values in each of its elements. In an adjacency matrix, it will have a single boolean element for each vertex in the graph, showing whether that vertex is adjacent to the vertex that owns this vector. For example v1 is adjacent to v2 **IFF** the value at index v2 in v1's adjacency vector is true, and vice versa.

This class implements the collection interface to allow using it as if it was a simple collection of neighbors. Abstractions of methods such as containsAll, add, remove, and iterator will prove useful for our use cases. **You will have to implement an iterator for this collection.**

The methods removeAll, retainAll, and toArray are not required to be implemented, you can leave them blank or throw exceptions, or implement them all for extra 10 points. (toArray should return an array of neighbors)

Be careful!! The iterator for this class should only iterate over the elements that are true, and not consider those that are false.

In short, your implementation of AdjacencyVect is expected to:

- Implement the Collection interface. (Implementing all methods is a bonus)
- Contains a vector of boolean values.
- Has a fixed size determined using the constructor.
- Has a custom iterator that iterates over added elements.
- Used in the MatrixGraph implementation.

3.3 MatrixGraph Class

You are expected to implement the MatrixGraph class based on the GTUGraph interface and use the adjacency matrix structure to store edge information. This class stores a single AdjacencyVect instance for each vertex in the graph.

In short, your implementation of MatrixGraph is expected to:

- Implement the GTUGraph interface.
- Store edges in an adjacency matrix.
- Use the AdjacencyVect class to construct the adjacency matrix.
- Store only undirected graphs. (Consider this when adding and querying edges)

4 A Greedy Graph Coloring Algorithm

In this section, the graph coloring algorithm that will be used in tests is explained. You are not expected to implement this algorithm, the source code will be provided.

Graph coloring is one of the most well studied optimization problems in computer science. Its basic form is the problem of assigning colors to vertices such that no 2 connected vertices have the same color.

Generally, greedy algorithms color vertices according to some ordering criteria like degree, weight, or the number of colored neighbors. In this part, what you implemented before will be used in a greedy graph coloring algorithm based on degree ordering (A degree of a vertex is the number of its neighbors). The implementation of this part will be given alongside the assignment. The pseudocode of our algorithm is demonstrated in algorithm 1.

Algorithm 1 Greedy Graph Coloring with Degree Ordering

```
Require: Graph $G = (V, E)$
Ensure: Solution $S = \{C_0, C_1, ..., C_{k-1}\}$
 1: Initialize $S \leftarrow$ None
 2: Compute degree $deg[v]$ for all $v \in V$
 3: Sort vertices $V$ into list $L$ in descending order of $deg[v]$
 4: for each vertex $v$ in $L$ do
 5:     $N(v) \leftarrow$ neighbors of $v$ in $G$
 6:     for each color $C$ in $S$ do
 7:         if $C \cap N(v) = \emptyset$ then
 8:             $C \leftarrow C \cup \{v\}$
 9:             break
10:         end if
11:     end for
12:     if $v$ was not colored then
13:         Initialize $C_{new} \leftarrow v$
14:         $S \leftarrow S \cup \{C_{new}\}$
15:     end if
16: end for
17: return $S$
```

5 Testing (60 Points)

You need to test your code on your own before submitting. The main function will be provided by the instructor. Possible test concerns:

1. Do the sorters sort correctly?
2. Does the graph store information correctly?
3. Does our work integrate correctly with the provided graph coloring algorithm?

5.1 Input

Your code will be tested using a collection of test scenarios. In each test scenario, there will a single input file. The file will contain the description of a graph. It will be in the following format:

```
<int:graph size>
<int:vertex 1> <int:vertex 2> // An edge from vertex 1 to vertex 2
<int:vertex 1> <int:vertex 2>
<int:vertex 1> <int:vertex 2>
.
.
.
```

The path to the file will be given as a command line argument. There should be no assumptions about where the file is, it must be read from the path the user provided.

5.2 Output

For each test, 3 output files are expected:

1. The sorted sums of integer pairs in the input file. Integers separated by new lines.
2. The adjacency information of the given graph but requeried from the GTUGraph class. Same format as the input file.
3. The solution of the graph coloring algorithm. Similar to the input file, but with the following differences:
   - First line is the graph size.
   - Second line is the number of colors.
   - After that, integer pairs of color ID and vertex ID, meaning that this vertex was assigned to that color. (<int:colorID> <int:vertexID>)

The paths to where these files will be written to will be given as command line arguments.

5.3 Environment

Your code will be tested in a docker container with the following specifications:

1. Ubuntu 22.04.
2. OpenJDK and OpenJRE 11.

To ensure that no problems occur during testing, please consider testing your code using the container created by the provided dockerfile. Explanation about how to test your code will be provided in the PS.

The code will be executed using the following commands:

1. make clean

2. make collect
3. make build
4. make run ARGS="<input file> <output path ending with a '/'>"

6 Documentation

You should provide comments explaining how your code works. Do not over explain, it will make reading your code harder. You should also generate the documentation using the javadoc command and provide it in your submission. Include short complexity description of each method.

7 Report

You are expected to submit a report that answers these questions:

1. Introduction: What did you do in this project? (Including bonuses)
2. Environment: What environment did you use?
3. Complexity Analysis: What is the complexity of your solutions? Why? (Short explanations)
4. Testing: How did you test your software?
5. AI usage: Where did you use AI? How?
6. Challenges: What problems did you face?
7. Conclusion What did you learn?

Your report should be:

1. Well structured.
2. Easy to read.
3. Answers the questions above. Not long, yet sufficient answers are expected.

8 Rules

- Name of the submitted archive should be: <full_name>_PA7.zip.
- The archive type should be .zip.
- Your submission should include the following:
    1. Source code directory.
    2. A makefile to build and run your code.
    3. javadoc directory including javadoc documentation.
    4. Your report as a PDF file.
- Do not include OS specific unrelated files in your submission. If any of such files causes problems during tests, you will responsible for it. (_MACOSX directory, windows identifier files, etc.)
- An example project format will be provided alongside the assignment, do not deviate from it.
- You can only use the following libraries:
    1. java.util.Comparator
    2. java.util.Random
    3. java.util.Collection

4. java.util.Scanner
5. java.util.Iterator
6. java.io.File

9 Evaluation

9.1 Grade Distribution (max of 200)

1. Sorting Algorithms (70)
2. Graph Implementation (70)
3. Testing (60)

9.2 Bonuses (max of 40)

- Implement a GTUSelectSort class. (+10)
- Implement all methods for the AdjacencyVect class. (+10)
- Do automated unit testing for all classes. (+20)

9.3 Penalties

- Cheating. (-200)
- Code not compilable. (-200)
- Bad OOP design. (-100)
- Modifying the predefined classes and interfaces. (-40)
- Forbidden library. (-40)
- Bad project format (-10)
- Incompatible input/output format. (-10)
- No javadoc documentation included. (-10)
- Invalid input/output format. (-20, and tests will be delayed)