CSE222-Assignment7-8 Sorting Algorithms, Adjacency Matrix, and Graph Coloring Report

---

1. Introduction

In this project, I implemented various algorithmic and structural components to build a complete graph-based solution. The work included:

- Development of three sorting algorithm classes: GTUInsertSort, GTUQuickSort, and the bonus GTUSelectSort, all extending the abstract GTUSorter class.
- Design and implementation of the AdjacencyVect class, a custom Collection that holds boolean vectors to represent adjacency relations in a graph.
- Implementation of the MatrixGraph class, based on the adjacency matrix model, using the AdjacencyVect as its building block.
- Integration and validation of the above components with a provided greedy graph coloring algorithm.
- Creation of a modular, reusable unit testing framework to ensure the correctness of all implemented classes.

All bonus features were successfully implemented:

- GTUSelectSort
- Full Collection method implementation in AdjacencyVect
- Automated testing for all major components

---

2. Environment

The development and testing of the project were conducted in the following environment:

- **Operating System:** Ubuntu 22.04 (also tested using the provided Docker container)
- **Java Development Kit:** OpenJDK 11
- **IDE:** Visual Studio Code
- **Build System:** Manual Makefile with custom commands (make clean, make collect, make build, make run)
- **Testing Environment:** Local execution with simulated input/output scenarios and Docker validation

---

3-Complexity Analysis

1. AdjacencyVect Complexity

- **Constant Time O(1):**
  size(), isEmpty(), contains(), add(), remove(), and creating an iterator.
  These methods operate by direct array access or simple field operations.
- **Linear in Input Size O(k):**
  Bulk operations such as containsAll(), addAll(), and removeAll() scale with the size of the input collection.
- **Linear in Vertex Count O(n):**
  Constructor, clear(), retainAll(), and toArray() involve scanning or initializing the entire adjacency boolean array.
- **Iterator Operations:**
  Average O(1) if true entries are dense, worst-case O(n) when scanning many false entries.

2. MatrixGraph Complexity

- **Edge Operations:**
  setEdge(v1, v2) and getEdge(v1, v2) are O(1) since they map directly to AdjacencyVect's constant-time add/contains methods.
- **Neighbor Retrieval:**
  getNeighbors(v) returns the adjacency vector for vertex v in O(1).
- **Reset:**
  reset(size) initializes the adjacency vectors for all vertices, costing $O(n^2)$ overall since each AdjacencyVect is size n and there are n vertices.

3. Sorting Algorithms Complexity (Brief)

- **Insertion Sort:**
  Best O(n), worst O(n^2). Efficient for small or nearly sorted data.
- **Quick Sort:**
  Average and best O(n log n), worst O(n^2). Uses randomized pivot and may switch to insertion sort for small partitions.
- **Selection Sort:**
  Always O(n^2). Simple but inefficient for large data.

---

4. Testing

I created a **manual unit testing framework** to verify the correctness of each module without relying on JUnit. Key testing strategies included:

- Writing and executing test classes for each sorting algorithm to verify correctness across sorted, reverse, and randomized arrays.
- Testing AdjacencyVect for add, remove, contains, iterator, and toArray functionalities including edge cases like invalid indices or duplicate entries.
- Validating MatrixGraph by setting edges, querying them, and checking neighbor sets in graphs of different sizes and densities.
- Testing file input/output paths manually, and validating integration with the provided greedy coloring logic.

Test results were printed to the console, with PASS/FAIL indications based on assertions, making the testing repeatable and extensible.

---

5. AI Usage

AI tools were used in several non-coding and supporting tasks to increase productivity, readability, and consistency throughout the project. Specifically, AI assisted in the following areas:

- Generating inline comments for complex methods to enhance code readability and explain non-obvious logic.
- Designing and structuring unit test cases by helping define test objectives, expected outcomes, and coverage strategies.
- Reviewing edge cases and identifying missing scenarios in the test suite.
- Improving the clarity, structure, and formatting of this report to ensure a professional and coherent presentation.

All core implementations, algorithm design, and debugging processes were performed manually. AI was used strictly as a documentation and productivity enhancer—not for solving the assignment directly.

---

6. Challenges

During the implementation and testing phases of the project, I encountered several challenges that required additional effort and careful debugging:

- **Testing and Debugging:** Building reliable unit tests without using standard libraries like JUnit was time-consuming. Ensuring that all test cases executed correctly and produced meaningful results required a robust custom test design.
- **AdjacencyVect Methods:** Implementing all methods of the Collection interface in AdjacencyVect, especially containsAll, removeAll, and retainAll, was challenging. Maintaining consistency in the internal boolean array while complying with interface semantics took careful attention.
- **Custom Iterator Logic:** Designing a custom iterator that correctly traverses only the true entries in the adjacency vector was another tricky part. Ensuring that the iterator skipped false values without skipping valid true values required multiple testing iterations.
- **Quick Sort with Partition Limit:** Implementing GTUQuickSort with a configurable partition limit, and switching to another sorting algorithm (such as GTUInsertSort) for small partitions introduced complexity in design and constructor handling. Ensuring fallback behavior worked seamlessly for edge cases took time to get right.

Each of these challenges improved my problem-solving ability and understanding of data structure behavior under real use cases.

---

7. Conclusion

Through this assignment, I significantly strengthened both my theoretical understanding and practical skills in algorithm design, data structures, and object-oriented programming. The task required a blend of abstract thinking and hands-on implementation, offering a comprehensive experience in developing modular and maintainable Java software.

Specifically, I deepened my knowledge in:

- **Generic Programming in Java:** Implementing generic sorter classes based on an abstract base class improved my understanding of type abstraction, interface design, and reusable component architecture.
- **Graph Representation:** Working with adjacency matrices for undirected graphs provided insight into space-time trade-offs in graph implementations. The AdjacencyVect design, combining boolean arrays with the Collection interface, offered both efficiency and flexibility.
- **Custom Iterator and Interface Compliance:** Building a compliant, functional Collection implementation from scratch—including a custom iterator that only traverses active elements—strengthened my understanding of internal Java library structures and interface-driven development.
- **Algorithm Design and Optimization:** Implementing different sorting algorithms and integrating logic such as partition limits in QuickSort taught me how algorithmic choices affect performance and complexity in real-world scenarios.
- **Software Quality and Testing:** Designing a manual unit testing framework helped me appreciate the role of automated testing in ensuring robustness and correctness. Writing test cases early also improved code clarity and modularity.
- **Integration and System Thinking:** Combining various components (sorters, graphs, iterators, and input/output handlers) to support a higher-level algorithm (graph coloring) helped me think in terms of software systems rather than isolated code blocks.

Overall, this assignment was a complete software engineering exercise—requiring not just working code, but thoughtful design, correctness, testability, and extensibility. It closely mirrors real-world development workflows and served as a valuable preparation for future academic or industry-level projects.

.