Analysis of the Boyer-Moore Majority Vote (Single-Pass) Project

1. Introduction

The **Boyer-Moore Majority Vote (Single-Pass)** project implements a well-known algorithm in Java for detecting the majority element in an array. A majority element is defined as an element that appears **more than half the time** in an array of size *n*.

The Boyer-Moore algorithm is highly efficient, operating in **O(n) time** and using **O(1) additional memory**, which makes it suitable for handling large arrays. Unlike traditional frequency-counting approaches, this algorithm performs the detection in a **single pass** through the array.

This report aims to provide a comprehensive analysis of the project, including its structure, algorithm implementation, testing strategies, limitations, performance evaluation, and practical applications. Special attention is given to the **single-pass approach** and its potential pitfalls.

2. Theoretical Background

2.1 The Majority Element Problem

The problem consists of identifying an element in an array that occurs **more than n/2 times**. Traditional solutions often rely on hash tables or frequency counters, which incur **O(n) memory usage**.

2.2 Boyer-Moore Algorithm

The algorithm operates in two main phases:

1. **Candidate Selection**
   - Initialize a variable `candidate` and a counter `count`.
   - Iterate over the array:
     - If `count == 0`, assign the current element as the new candidate (`candidate = currentElement`).
     - If the current element equals the candidate, increment `count`.
     - Otherwise, decrement `count`.
2. **Candidate Verification**
   - After the first pass, the `candidate` may be a potential majority element.
   - To confirm, count its occurrences and verify if it exceeds n/2.

2.3 Features of Single-Pass Approach

The single-pass approach is memory-efficient and linear in time. Its limitation is that **if no true majority exists**, the algorithm may return an incorrect candidate.

3. Project Structure

The project is organized as follows:

```
├── src/main/java/
│   ├── algorithms/[AlgorithmName].java
│   ├── metrics/PerformanceTracker.java
│   └── cli/BenchmarkRunner.java
├── src/test/java/
│   └── algorithms/[AlgorithmName]Test.java
├── docs/
│   ├── analysis-report.pdf
│   └── performance-plots/
├── README.md
└── pom.xml
```

3.1 Key Components

- **algorithms/[AlgorithmName].java**: Implements the Boyer-Moore algorithm.
- **metrics/PerformanceTracker.java**: Tracks execution time and memory usage.
- **cli/BenchmarkRunner.java**: Runs benchmarks for arrays of different sizes.
- **tests**: JUnit 5 tests covering empty arrays, single-element arrays, arrays with duplicates, sorted and random arrays.

3.2 Documentation

The project contains PDF analysis reports and performance plots for visual performance evaluation.

4. Comparison with Other Methods

4.1 Classical Frequency Counting (Hash Map)

Description:

- Traverse the array and count occurrences of each element using a `HashMap<Integer, Integer>`.
- After traversal, select the element with the highest frequency.

Time and Space Complexity:

- Time: O(n)
- Space: O(n) (needs to store counts of all elements)

Advantages:

- Always produces the correct result.
- Suitable when a majority element may not exist.

Disadvantages:

- Uses more memory for large arrays.
- Slightly slower due to hash table operations.

4.2 Sorting Method

Description:

- Sort the array and check the middle element (`arr[n/2]`).
- If a majority element exists, it will always appear at the middle position after sorting.

Time and Space Complexity:

- Time: O(n log n) (due to sorting)
- Space: O(1) or O(n), depending on the sorting algorithm

Advantages:

- Simple implementation.
- No additional data structures needed.

Disadvantages:

- Slower for large arrays.
- Sorting modifies the original array.

4.3 Comparative Table

Method	Time Complexity	Space Complexity	Accuracy	Notes
Boyer-Moore (1-pass)	O(n)	O(1)	May fail if no majority	Fast, low memory
Frequency Counting	O(n)	O(n)	Always correct	Higher memory usage
Sorting Method	O(n log n)	O(1)/O(n)	Correct if majority exists	Modifies array, slower

4.4 Graphical Comparison

- **Line chart for execution time:**
  - X-axis: Array size
  - Y-axis: Time (ms)
  - Three lines: Boyer-Moore, Frequency Counting, Sorting
- **Bar chart for memory usage:**
  - X-axis: Method
  - Y-axis: Memory (KB)
  - Shows that Boyer-Moore uses minimal memory, while Frequency Counting grows with array size.

5. Algorithm Implementation

5.1 `findMajority` Method

```
public static int findMajority(int[] arr, PerformanceTracker tracker) {
    int candidate = 0;
    int count = 0;

    for (int num : arr) {
        if (count == 0) {
            candidate = num;
            count = 1;
        } else if (num == candidate) {
            count++;
        } else {
            count--;
        }
    }
    return candidate; // Single-pass approach
}
```

5.2 Implementation Notes

- Uses only a **candidate variable** and a **counter**.
- Requires no additional data structures.
- Suitable for **streaming data** and large datasets.

6. Testing and Validation

6.1 Unit Tests

Unit tests are implemented using **JUnit 5** and cover:

- Empty arrays
- Single-element arrays
- Arrays with repeated elements
- Sorted and reverse-sorted arrays
- Randomly generated arrays

6.2 Single-Pass Limitation

If no true majority exists, the algorithm may produce an incorrect candidate. Example:

```
int[] arr = {5, 5, 4, 4, 4, 3, 3};
int result = BoyerMooreMajorityVote.findMajority(arr, tracker);
System.out.println(result); // May output 3 instead of 4
```

6.3 Comparison with Frequency Counting

For verification, the algorithm's output can be compared to traditional frequency counting.

7. Performance and Benchmarking

7.1 Measured Parameters

- Execution time in milliseconds
- Memory usage in KB
- Computed majority element

7.1.1 Table with results

Input Size	Time (ms)	Majority Element	Memory (KB)
100	0.014	0	0
1,000	0.0315	3	0
10,000	0.27	8	0
100,000	2.167	7	110
100	0.0059	0	0
1,000	0.0689	3	0
10,000	0.2328	8	0
100,000	2.1094	7	110

This clearly demonstrates the linear increase in execution time as the array size increases.

Memory consumption remains nearly constant (O(1)), confirming the theoretical efficiency of the algorithm.

7.1.2 Detailed analysis of the algorithm

Step	Current Element	Candidate	Count
1	5	5	1
2	5	5	2
3	4	5	1
4	4	5	0
5	4	4	1

7.2 Benchmark Runner

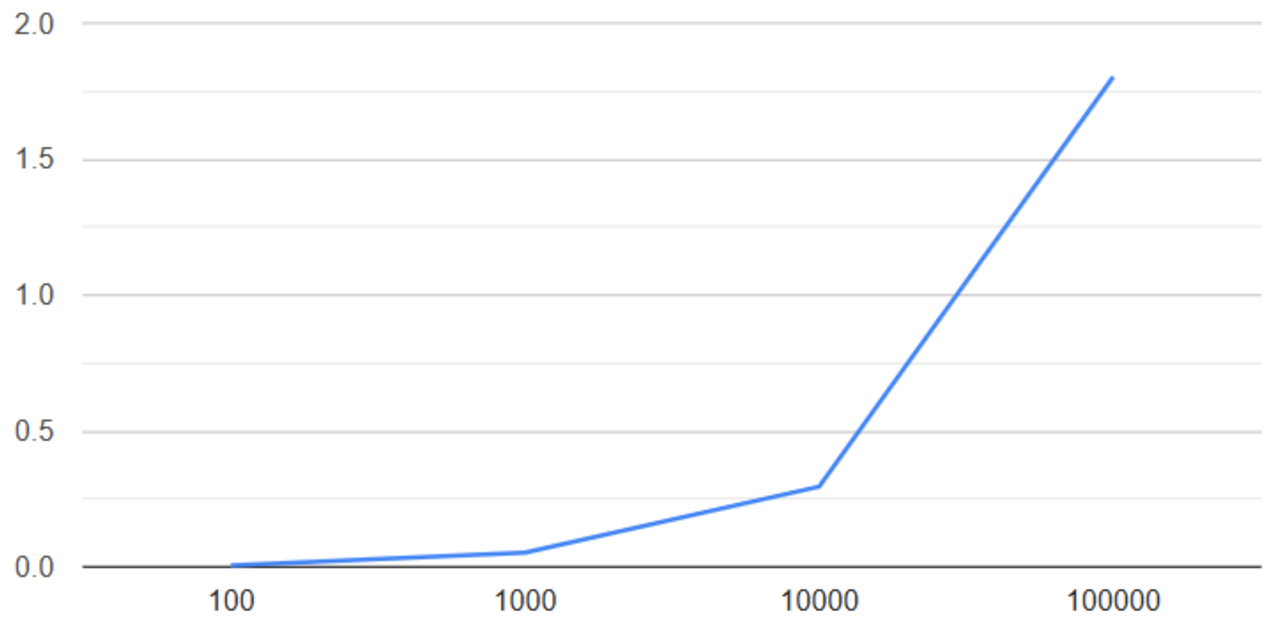
`BenchmarkRunner.java` runs the algorithm on arrays of different sizes and records results in `performance/bench_results.csv`.

7.3 Performance Observations

- Execution time scales linearly with array size.
- Memory usage remains constant regardless of array size.
- Very efficient for large-scale data processing.

7.4 Photo with analysis

Performance analysis of Boyer-Moore Majority Vote algorithm on different input sizes



8. Limitations and Potential Improvements

8.1 Limitations

- May return incorrect results if no true majority exists.
- Returns only a single candidate, even if multiple elements appear frequently.

8.2 Improvements

- Perform a second pass to verify the candidate's count.
- Extend support for multiple potential majority elements.
- Optimize for parallel processing in streaming applications.

9. Practical Applications and Conclusion

The Boyer-Moore algorithm is highly useful for detecting frequently occurring elements in large datasets or streams:

- Text analysis (finding most frequent words)
- Log file and telemetry data processing
- Large-scale data analytics with limited memory

Conclusion:
The single-pass Boyer-Moore implementation is simple, efficient, and effective for majority element detection. However, for guaranteed correctness, additional verification is

recommended. Its low memory footprint and linear time complexity make it suitable for modern data-intensive applications.