Efficient Text Search and Replace Tool

Project Report

Submitted by

Ahmed Sakr	ID: 4
Ashraf Mohamed	ID: 6
Mohamed Ashraf	ID: 18
Mohamed Sakr	ID: 21

Department of Computer Science

1 Project Overview

This project implements a high-performance command-line text search and replacement tool in C++. The application loads a large text file into memory once and supports fast interactive queries with advanced matching options, including case sensitivity, match modes (Whole Word, Prefix, Substring), highlighted output, and optional non-destructive replacement with timing statistics.

Key goals achieved:

- Efficient single-pass file loading for repeated queries.
- Linear-time string searching using an advanced algorithm.
- Flexible query syntax supporting real-world search needs.
- Accurate replacement without modifying the original in-memory data.
- Performance measurement for both search and replace operations.

The tool is particularly suited for analyzing log files, source code, or large documents where repeated searches are common.

2 Team Members and Responsibility Division

ID	Name	Responsibility
18	Mohamed Ashraf	Advanced Search Algorithm
6	Ashraf Mohamed	Replacement & Performance Reporting
21	Mohamed Sakr	User Interface & Command Parsing
4	Ahmed Sakr	Core Architecture & File I/O

Detailed contributions:

- **Core Architecture & File I/O**
  - Designed the central `TextSearch` class as the main engine.
  - Implemented efficient one-time file loading into a vector of lines.
  - Added robust error handling for missing or inaccessible files with clear user feedback.
- **Advanced Search Algorithm**
  - Integrated the Knuth–Morris–Pratt (KMP) algorithm for guaranteed linear-time performance.
  - Supported configurable case sensitivity and three distinct match modes.
  - Ensured correct boundary detection using alphanumeric separators.
- **User Interface & Command Parsing**
  - Built an interactive REPL-style command loop.
  - Implemented robust tokenization supporting quoted keywords with spaces.
  - Designed clean flag-based syntax for optional parameters.

  - Added visual highlighting of matches using brackets in output.
- **Replacement & Performance Reporting**
  - Developed safe and efficient text replacement that writes to a new file.
  - Integrated high-resolution timing to measure processing speed.
  - Provided detailed statistics on matches found and replacements made.

3 Key Features

- **Interactive Mode**: Multiple searches on the same file without reloading.
- **Advanced Matching**:
  - `Whole`: Complete words only (bounded by non-alphanumeric characters).
  - `Prefix`: Words starting with the keyword.
  - `Substring`: Matches anywhere in the line.
- **Case Control**: Sensitive (default) or insensitive (`c=i`).
- **Quoted Keywords**: Support for multi-word phrases (e.g., "machine learning").
- **Visual Feedback**: Matches highlighted with `[ ]` brackets.
- **Replacement**: Non-destructive, outputs to a new file.
- **Performance Metrics**: Execution time reported in microseconds.

4 Usage Instructions

4.1 Compilation

Compile using a modern C++ compiler:

```
g++ main.cpp TextSearch.cpp utils.cpp -o textsearch
```

Listing 1: Compilation command

4.2 Running the Program

```
./textsearch --file=input.txt
```

Listing 2: Launching the tool

4.3 Command Syntax

```
Query <keyword> [c=i|s] [m=Whole|Prefix|Substring] [r=replace_text] [o=
    output_path]
```

Listing 3: General query format

<keyword> Required search term (use quotes for phrases containing spaces).

c=i|s Case sensitivity: `i` = insensitive, `s` = sensitive (default: `s`).

m=... Match mode (default: `Whole`).

r=... Optional replacement text.

o=... Output file path (required when using `r=`).

4.4 Examples

- `Query error`
- `Query Error c=i m=Substring`
- `Query bug r=feature o=fixed.txt`
- `Query "access denied" c=i`

5 Design Decisions and Trade-offs

- **Memory vs. Speed**: Entire file loaded into RAM once for ultra-fast repeated queries.
- **KMP Algorithm**: Chosen for guaranteed $O(N + M)$ worst-case time and simple multi-match handling.
- **Line-based Storage**: Facilitates boundary checks and clean context display.
- **Non-overlapping Replacements**: Avoids issues with overlapping patterns.
- **Non-destructive Design**: Original data preserved for safe multi-query usage.

6 Complexity Analysis

6.1 Search Operation

- Time Complexity: $O(N + M)$ per query (N = text length, M = keyword length)
- Auxiliary Space: $O(M)$

6.2 Replace Operation

- Time Complexity: $O(N)$ (single pass over text)

6.3 Memory Usage

- Dominant: $O(N)$ for storing the full text
- Suitable for files up to several hundred MB on modern systems

7 Performance Considerations

The tool is optimized for repeated queries on large datasets (e.g., log analysis, code navigation). Tests consistently show sub-millisecond search times on multi-megabyte files, demonstrating the effectiveness of the KMP algorithm combined with in-memory storage.