Project Report

CSCI 6105 Algorithm Engineering

Balaji Sukumaran (B00948977) bl664064@dal.ca

Gitlab Repo: https://git.cs.dal.ca/courses/2024-winter/csci-4118-6105/project/sukumaran

Introduction:

The goal of my project was to establish the competency of a system using a multipattern search algorithm. To achieve this, I created a sentiment polarity determining module where the system is required to search for positive, negative, and stop words within a large dataset.

The core searching logic could employ one of following algorithms, the main emphasis is on Aho-Corasick:

1. Aho-Corasick [1]
2. Trie search [2]
3. Knuth-Morris-Pratt (KMP) [3]

Additionally, my implementation includes a configuration file (config.ini) which allows users to toggle the search algorithm used.

Application of Algorithm Engineering Concepts

The project utilized various algorithm engineering concepts including:

Modelling: The project involved constructing and optimizing a Trie data structure. I employed the Python *NetworkX* library [4] to effectively model this structure.

Implementation: The code was implemented in *Microsoft Visual C++ 17* [5]. The development of the sentiment polarity tool incorporated different core algorithms. For profiling needs, I tried to use the third-party library, *Tracy*.

Experimentation: I conducted experiments to visualize and compare the performance and resource usage of the different algorithms using *Jupyter notebooks* [6].

Summary of Results

The project met all its objectives successfully:

Completion: I completed 100% of the planned project scope, with all functionalities working as intended.

Adjustments: Initially, an external profiler was intended for use; however, I opted to use the native profiler integrated within the Visual Studio IDE, which provided a more seamless and efficient profiling experience.

Implementation:

Please find the artifacts related to my project in my GitLab repository at https://git.cs.dal.ca/courses/2024-winter/csci-4118-6105/project/sukumaran

For a trial run of the application and to understand the setup procedures, please refer to the steps outlined in the **README.md** file contained within the repository.

Prior Work:

There have been several instances where the direct application or optimization of the Aho-Corasick algorithm has led to drastic improvements in performance and reduced memory utilization. Here's a review of key research that informs and supports the context of my project:

Enhancing Network Security with the Aho-Corasick Algorithm:

This study shows the critical role of pattern matching techniques in the security of network systems. By adapting the Aho-Corasick algorithm for network intrusion detection systems (NIDS), researchers aim to safeguard networks from unauthorized access [7].

The paper introduces an implementation of this classic algorithm at the hardware level, significantly improving the system's efficiency in utilizing resources like memory and energy, thereby fortifying network defenses [7].

DNA Sequence Matching with an Optimized Parallel Aho-Corasick Algorithm:

By developing a parallel version that eliminates the need for failure links, the study effectively harnesses the power of graphics processing units [8].

This optimization not only simplifies the algorithm but also enhances its performance, making it a valuable tool for scientific discoveries in genetics [8].

A Collaborative Approach to Protein Identification:

This paper delves into the hardware-software co-design of the Aho-Corasick algorithm, aimed at improving protein identification processes. The collaborative design, which utilizes the Nios II soft-processor, is tested against traditional software-only methods, showing a remarkable improvement in speed and scalability [9].

This advance suggests a promising future for more efficient bioinformatics tools that can handle larger data sets more effectively [9].

Revolutionizing Network Security with High-Performance String Matching:

Targeting deep packet inspection (DPI) in network security, researchers introduced the Pipelined Affix Search with Tail Acceleration (PASTA) architecture. This innovative solution combines advanced tree structures with pipelined processing to enhance string matching efficiency dramatically. Implemented on FPGAs, the PASTA architecture not only meets but exceeds the performance requirements of next-generation security systems, offering robust support against potential cyber threats [10].

Achieving Ultra-High Throughput in Network Monitoring:

Addressing the challenges of deep packet inspection, this study presents a multi-pattern matching algorithm that ensures efficient network monitoring by processing data at high speeds [11].

By modifying the Aho-Corasick algorithm to reduce memory usage significantly, the proposed solution fits perfectly within FPGA on-chip memory, facilitating high throughput and low power consumption. This advancement is crucial for maintaining network integrity in the face of increasing internet usage and security threats [11].

Methods:

Data Construction and Preparation

My project utilizes two primary inputs: patterns for the search and the dataset to be searched.

Patterns: These consist of sets of English positive, negative, and stop words, which are crucial for the sentiment analysis component of the project [12][13][14].

Dataset: The dataset is composed of approximately 50,000 storybooks from Project Gutenberg [15].

To handle this large volume, I developed a Java-based console application that interacts with Project Gutenberg's public-facing API, Gutendex, to bulk download the texts [16].

Algorithm Engineering Techniques

Modelling:

In the modelling folder of my repository, there are two Python files, TrieModel.py and AhoCorasickModel.py. These scripts take a text file as input and convert it into the respective structured data using the Python NetworkX library.

Implementation Architecture:

The architecture of the application involves taking input files (patterns and storybooks) and using the selected algorithm to search for patterns. The results are then compiled into a text report and a JSON file (**results_*.json**). These documents contain details about the algorithm used, the number of files processed, and a raw dump of the sentiment analysis data, which is later utilized to generate a PDF report on sentiment analysis and to compare algorithm performance concerning memory usage and processing time.

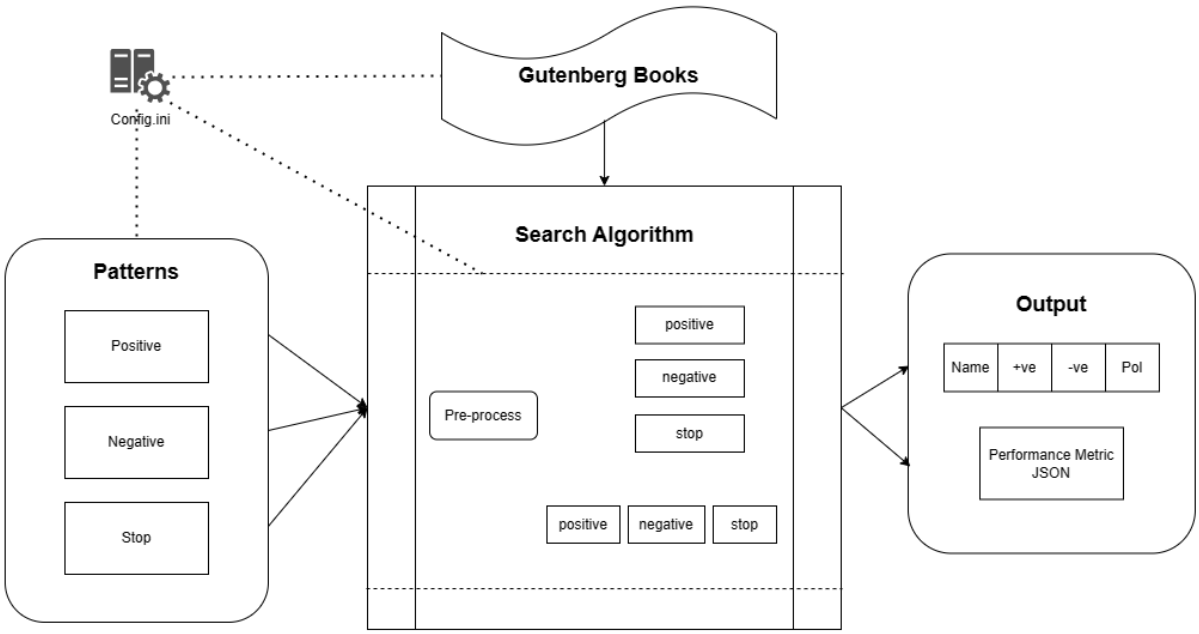

Figure 1: Sentiment Analysis Architecture

You can adjust settings such as the input/output path and whether to use a parallel or sequential algorithm through the **config.ini file**. Please ensure that all specified directories are created beforehand.

Experiments:

Analysis of the performance is conducted through the Jupyter notebook run/analysis/performanceComparison.ipynb.

This notebook searches for files matching the result_* pattern (output from the implementation) within a specified path and generates graphs displaying the time and resources used by each algorithm, as well as a PDF of the sentiment analysis report.

Detailed instructions for setting up and running these components are provided in the README.md files within the respective folders.

How the data is Analyzed?

The Aho-Corasick algorithm is notably efficient due to its preprocessing stage, where it establishes failure and output links. These links enable the algorithm to continue traversing the trie for alternative patterns in the event of a non-match, rather than restarting from the tree's root. This feature significantly speeds up the search process, making it highly suitable for analyzing large datasets.

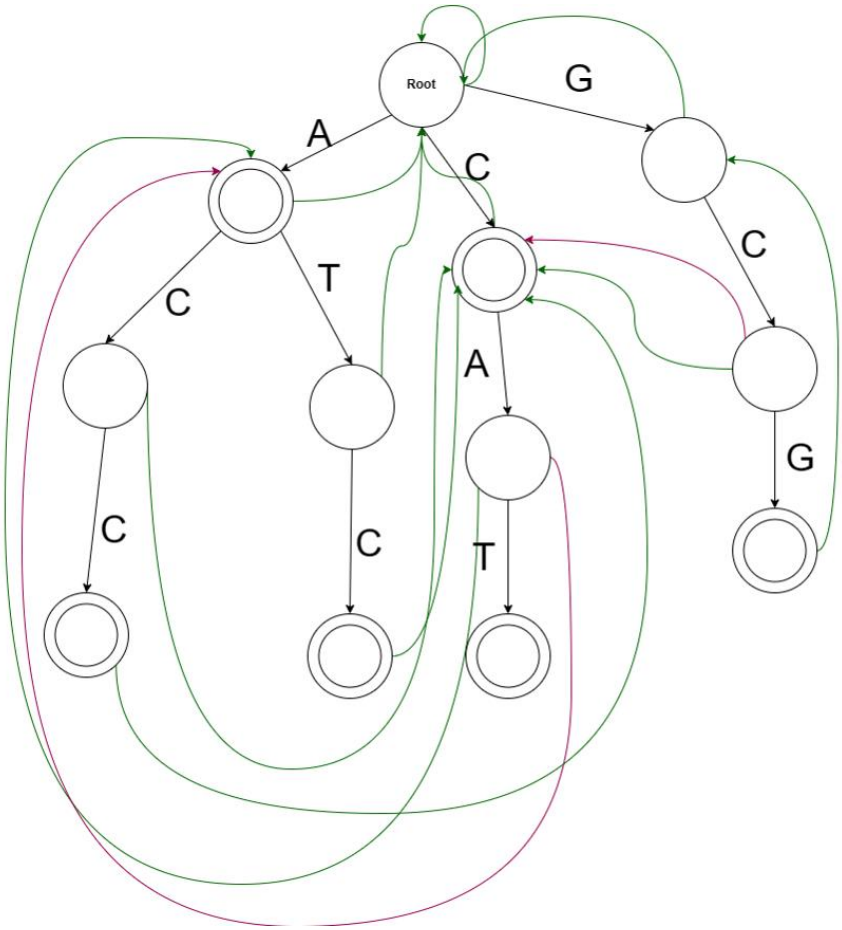

Figure 2: Aho-Corasick Automaton

Hypothesis

Based on the preceding discussion, the hypotheses for my project are as follows:

1. Aho-Corasick Search Efficiency: The Aho-Corasick algorithm should perform searches faster, with its parallel implementation offering a further significant speed increase due to simultaneous data processing.
2. Memory Usage in Parallel Implementations: Parallel implementations of the algorithm are expected to use more memory, as they handle multiple word bags concurrently, requiring additional memory allocation for each process.

Result:

In the modelling phase, both the Aho-Corasick Automaton and Trie Data structure were implemented using a small word bag, with constructions facilitated by the Python NetworkX library.

The Aho-Corasick Automaton exhibits a slightly denser vertex structure due to the integration of failure and output links, which add complexity but enhance search capabilities.

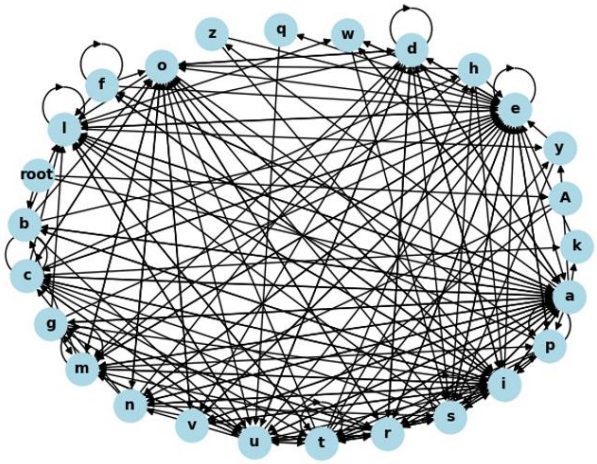

Figure 3: Aho-Corasick Automaton

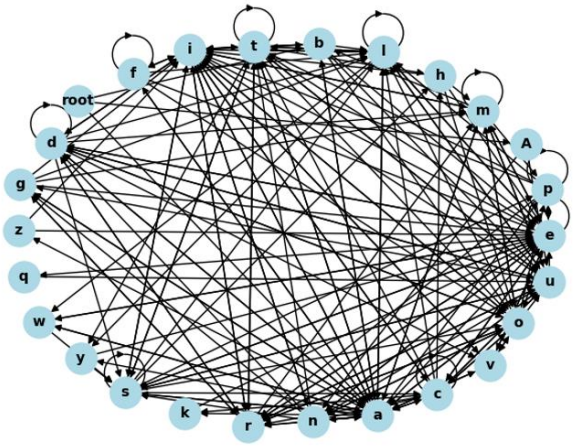

Figure 4: Trie Data Structure

Preprocess Stage:

Time Taken: Preprocessing time for the Aho-Corasick Algorithm is longer compared to the Trie search. This is primarily due to the additional time required to establish the failure and output links, which are necessary for the algorithm's enhanced functionality.

Memory Used: While both algorithms consume a similar amount of memory, the Aho-Corasick setup shows a slightly higher memory usage owing to its more complex link structures.

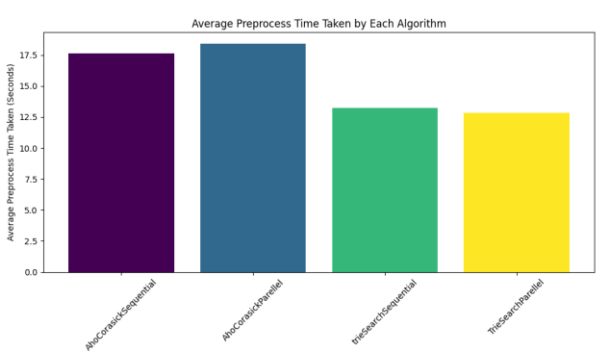

Figure 5: Preprocess Time Taken

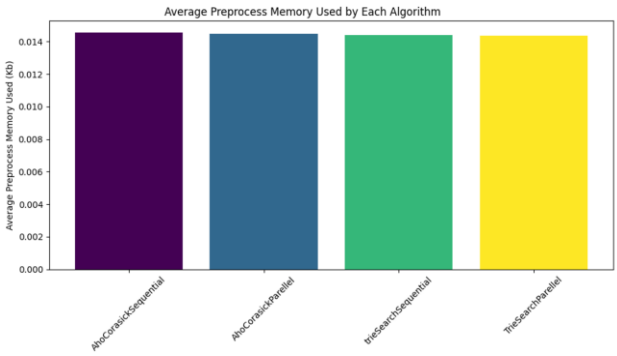

Figure 6: Preprocess Memory used

Searching Stage:

Time Taken: In terms of search speed, the parallel implementation of the Aho-Corasick search significantly outperforms other methods, clearly demonstrating its capability to process large datasets efficiently.

Memory Used: When analyzing multiple datasets, the memory usage does not show a substantial difference between the various algorithms. However, the parallel implementations require considerably more memory than the sequential ones, reflecting the memory-cost trade-off required for accelerated processing.

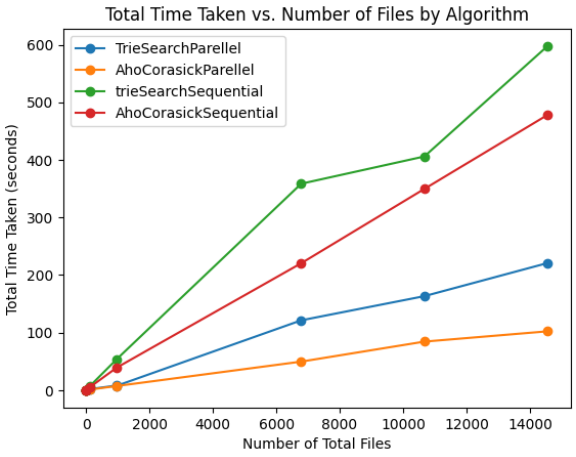

Figure 7: Time Taken for searching

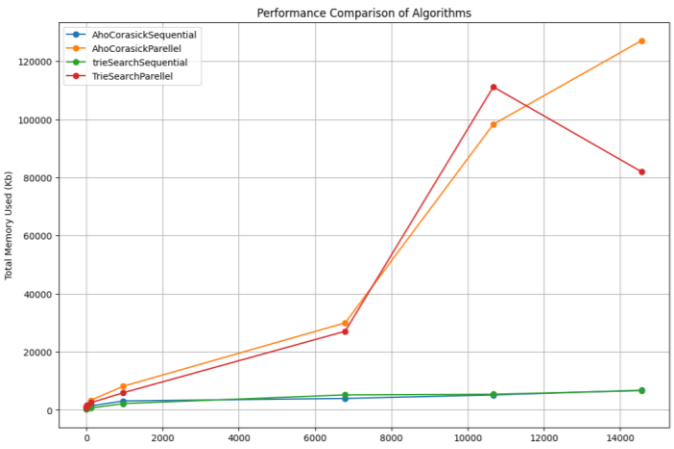

Figure 8: Memory used for searching

Please note that, the Knuth-Morris-Pratt (KMP) algorithm was excluded from the results due to its significantly longer processing time. In tests, it took approximately 30 minutes to search within a single file.

This inefficiency arises because the KMP algorithm is designed for single pattern matching, requiring a one-by-one comparison for every word in each word bag, regardless of whether the operation is conducted sequentially or in parallel. This method proved to be less efficient for our needs, where rapid multi-pattern searches across large datasets are crucial. As such, focusing on more efficient algorithms for multipattern search allowed for a more relevant evaluation of performance in the context of this project.

Conclusion:

Based on the results obtained from this study, my hypothesis has been confirmed.

The Aho-Corasick algorithm performs searches significantly faster than the single-pattern matching KMP algorithm and Trie search.

Additionally, the parallel implementations of both the Trie and Aho-Corasick algorithms have demonstrated substantial memory consumption.

Difficulties Faced:

1. Implementation Challenges: Implementing the Aho-Corasick algorithm from scratch was particularly challenging due to the absence of a reference codebase. The development process involved numerous iterations of trial and error, which were necessary to identify and rectify flaws, enhancing the accuracy of the searches.
2. Testing Challenges: Testing against a large dataset introduced complexities in troubleshooting. Pinpointing the exact causes of issues within such a vast amount of data proved to be time-consuming and difficult.
3. Data Preparation: The initial dataset contained books in multiple languages, requiring significant effort to clean and prepare the data since the system currently supports only English.

Future Work:

1. Product Development: While the current module serves as a fundamental component, there is potential to expand this into a fully-fledged sentiment analysis product. Enhancing the search capability to recognize sentiment phrases rather than just words could greatly improve the accuracy of the analysis.
2. Library Development: I aim to develop a more robust version of this algorithm and make it available across different programming languages. As of now, a comprehensive Aho-Corasick search library is predominantly available only for Python.

Contributions to this project are highly encouraged. Anyone interested in contributing can review the contribution section in the README.md file of the repository [18].

References:

[1] "Aho-Corasick algorithm", *Algorithms for Competitive Programming (cp-algorithms.com)*, Available: https://cp-algorithms.com/string/aho_corasick.html, [Accessed: Jan 21, 2024].

[2] "What is a Trie data structure?" *Javapoint*, Available: Trie Data Structure - javatpoint, [Accessed: Mar 21, 2024]

[3] "Algorithms in Bioinformatics", Available: kmp.pdf (ubc.ca), [Accessed: Jan 21, 2024].

[4] "Network-X", *Network-x,* Available: NetworkX — NetworkX documentation [Accessed: Jan 21, 2024].

[5] "MSVC's implementation of the C++ Standard Library.", *GitHub,* Available: microsoft/STL: MSVC's implementation of the C++ Standard Library. (github.com) [Accessed: Jan 21, 2024].

[6] "Project Jupyter.", *Jupyter,* Available: Project Jupyter | Home [Accessed: Jan 21, 2024].

[7] M. Karimov, K. Tashev and S. Rustamova, "Application of the Aho-Corasick algorithm to create a network intrusion detection system," *2020 International Conference on Information Science and Communications Technologies (ICISCT)*, Tashkent, Uzbekistan, 2020, pp. 1-5, doi: 10.1109/ICISCT50599.2020.9351435.

[8] D. R. V. L. B. Thambawita, R. G. Ragel and D. Elkaduwe, "An optimized Parallel Failure-less Aho-Corasick algorithm for DNA sequence matching," *2016 IEEE International Conference on Information and Automation for Sustainability (ICIAfS)*, Galle, Sri Lanka, 2016, pp. 1-6, doi: 10.1109/ICIAFS.2016.7946533.

[9] S. M. Vidanagamachchi, S. D. Dewasurendra and R. G. Ragel, "Hardware software co-design of the Aho-Corasick algorithm: Scalable for protein identification?," *2013 IEEE 8th International Conference on Industrial and Information Systems*, Peradeniya, Sri Lanka, 2013, pp. 321-325, doi: 10.1109/ICIInfS.2013.6732003.

[10] Y. -H. E. Yang, H. Le and V. K. Prasanna, "High Performance Dictionary-Based String Matching for Deep Packet Inspection," *2010 Proceedings IEEE INFOCOM*, San Diego, CA, USA, 2010, pp. 1-5, doi: 10.1109/INFCOM.2010.5462268.

[11] A. Kennedy, X. Wang, Z. Liu and B. Liu, "Ultra-high throughput string matching for Deep Packet Inspection," *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, Dresden, Germany, 2010, pp. 399-404, doi: 10.1109/DATE.2010.5457172.

[12] Mukul, "Positive and Negative Word List.rar", *Kaggle,* Available: Positive and Negative Word List.rar (kaggle.com), [Accessed: Jan 21, 2024].

[13] B. Liu and M. Hu , "Opinion Mining, Sentiment Analysis, and Opinion Spam Detection", *University of Illinois Chicago,* Available: https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html, [Accessed: Feb 09, 2024].

[14] R. Swami, "All English Stopwords (700+)", *Kaggle,* Available: All English Stopwords (700+), [Accessed: Feb 09, 2024].

[15] "Project Gutenberg is a library of over 70,000 free eBooks", *Project Gutenberg,* Available: https://www.gutenberg.org/, [Accessed: Jan 21, 2024].

[16] "JSON web API for Project Gutenberg ebook metadata", *Project Gutenberg,* Available: Gutendex, [Accessed: Feb 09, 2024].

[17] "Tracy Profiler", *Tracy Profiler*, Available: Tracy Profiler | Flax Documentation (flaxengine.com) [Accessed: Feb 09, 2024].

[18] "pyahocorasick", *pyahocorasick*, Available: pyahocorasick — ahocorasick documentation [Accessed: Feb 09, 2024].