Efficiency of String Matching Algorithms

By: Jeremy Hopkins, Terry Houston, Tochi Okwudire

Requirement Analysis

- Problem Statement: We are trying to analyze the efficiencies of various different String Matching Algorithms
- Objective: Evaluate different string matching algorithms based on their preprocessing time, pattern and text length, focusing on performance in searching in large texts.

Requirement Analysis

- Functional Requirements: The string matching algorithms we employ must be able to, at the very least, match a given text with a given pattern correctly. It’s crucial that KMP, Rabin-Karp, and Boyer-Moore are all able to provide the exact same results.
- Non-functional Requirements: Runtime, time and space complexity, scalability with different string sizes

Knuth-Morris-Pratt Algorithm Design

A string search algorithm in O(n + m) time where n = text length and m = pattern length.

- **Longest Prefix Suffix (LPS) Array**
  - Stores the longest prefix that matches a suffix up to each position in the pattern
  - Prevents wasting of time by preprocessing
  - “abcdef” and “efghij” share “ef”
  - The array changes as you proceed through the text
- **Multiple Instances**
  - The algorithm should be able to recognize multiple instances of a pattern in a text

KMP Development Strategy

We need a method that builds the LPS array as well as another method that searches through the array using two user-given parameters, the text and the pattern. The build method stores the lengths of the LPS matches for each position in the pattern to avoid redundant comparisons, backtracking when a mismatch occurs.

The search method uses the previously built LPS array to perform the KMP algorithm, given a text and a pattern. As it traverses the text it checks for matches between pattern and text; if a mismatch occurs, it uses the lps values to skip over unnecessary comparisons, storing the first index of each match in a result list.

Rabin-Karp Algorithm Design

A string search algorithm in O(n + m) time where n = text length and m = pattern length.

This algorithm is a string-searching algorithm that uses hashing to find a pattern in a string. It checks each substring in the text and compares the hash value of the pattern to the hash value of the current substring. If there’s a hash match, it then compares the individual characters to confirm the match.

Rabin-Karp Development Strategy

We need to define a search function that takes in two parameters: the text and the pattern. The method will calculate initial hash values for the pattern and the first window of the text, along with a variable h to update the hash efficiently as the window slides across the text, reducing the need for recalculating from scratch. A prime modulus, which could be selected randomly, is used to reduce hash collisions; the hashing function is defined as:

hash(S) = (Σ((S[i] - 'a' + 1) * (P^(i)))) % mod

The algorithm then compares hash values of the pattern and the current text window. If they match, it performs a character-by-character comparison to confirm and stores the starting index if successful.

Boyer-Moore Algorithm Design

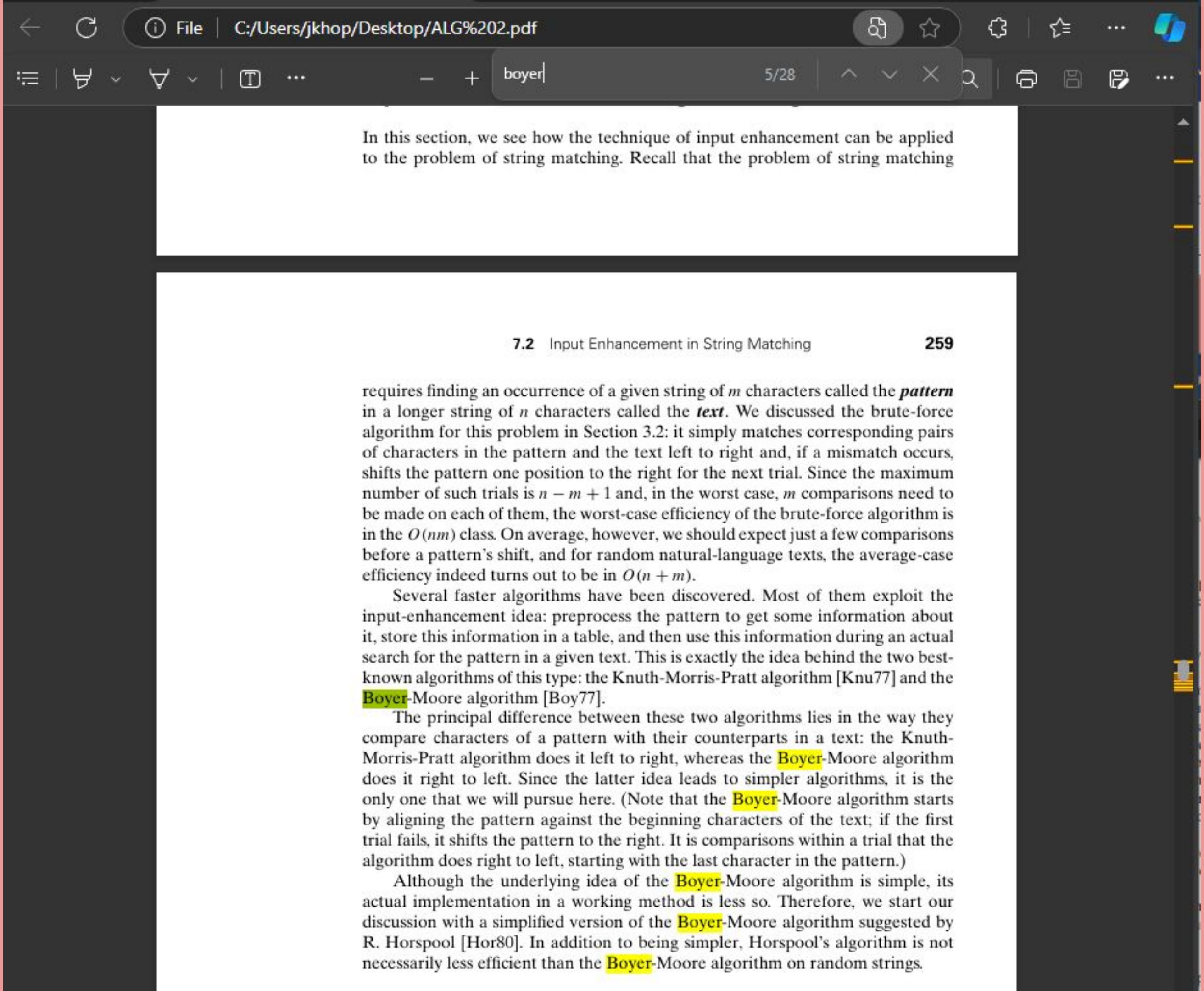
A string search algorithm in the best case $O(n / m)$ time and at worst $O(n * m)$ time where

n = text length and m = pattern length, average being $O(n)$.

- **Heuristics**
    - This algorithm commonly uses two different heuristics (shortcut techniques), to compare strings
        1. Bad character heuristic: If there is a mismatch, it skips to the next occurrence of the mismatched character in the pattern, and if it does not occur it moves on
        2. Good character heuristic: If a suffix matches but a mismatch follows, it shifts the pattern to align with the next matching suffix in the text

Boyer-Moore Development Strategy

We create heuristics to make the algorithm perform better (not optimally). For instance, a badCharHeuristic method that initializes a badChar array with the last occurrence indices of each character in the pattern, for preprocessing purposes. Then, for each alignment, check the pattern left to right and shift based on mismatches using badChar. When it matches, print the index of that substring and move on until the whole string has been processed.



In this section, we see how the technique of input enhancement can be applied to the problem of string matching. Recall that the problem of string matching

requires finding an occurrence of a given string of m characters called the ***pattern*** in a longer string of n characters called the ***text***. We discussed the brute-force algorithm for this problem in Section 3.2: it simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is $n - m + 1$ and, in the worst case, m comparisons need to be made on each of them, the worst-case efficiency of the brute-force algorithm is in the $O(nm)$ class. On average, however, we should expect just a few comparisons before a pattern's shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in $O(n + m)$.

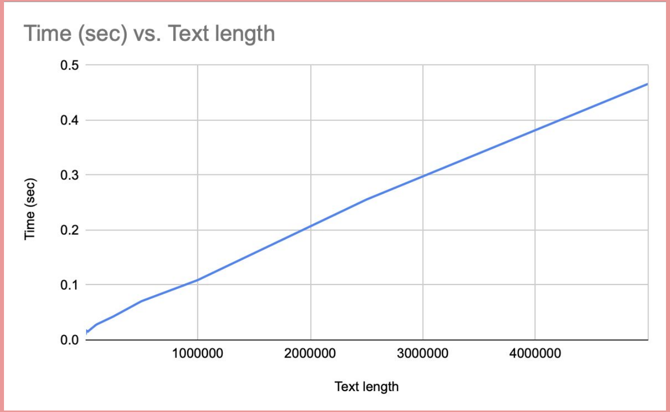
Several faster algorithms have been discovered. Most of them exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text. This is exactly the idea behind the two best-known algorithms of this type: the Knuth-Morris-Pratt algorithm [Knu77] and the Boyer-Moore algorithm [Boy77].

The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text: the Knuth-Morris-Pratt algorithm does it left to right, whereas the Boyer-Moore algorithm does it right to left. Since the latter idea leads to simpler algorithms, it is the only one that we will pursue here. (Note that the Boyer-Moore algorithm starts by aligning the pattern against the beginning characters of the text; if the first trial fails, it shifts the pattern to the right. It is comparisons within a trial that the algorithm does right to left, starting with the last character in the pattern.)

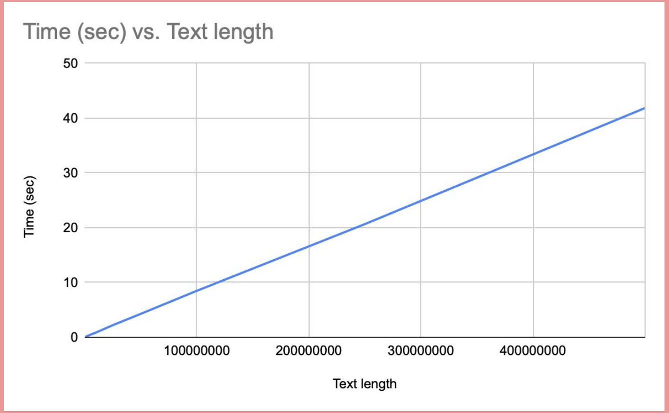
Although the underlying idea of the Boyer-Moore algorithm is simple, its actual implementation in a working method is less so. Therefore, we start our discussion with a simplified version of the Boyer-Moore algorithm suggested by R. Horspool [Hor80]. In addition to being simpler, Horspool's algorithm is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

Results (pattern “aaba” in text of randomly generated a’s and b’s)

KMP time complexity chart

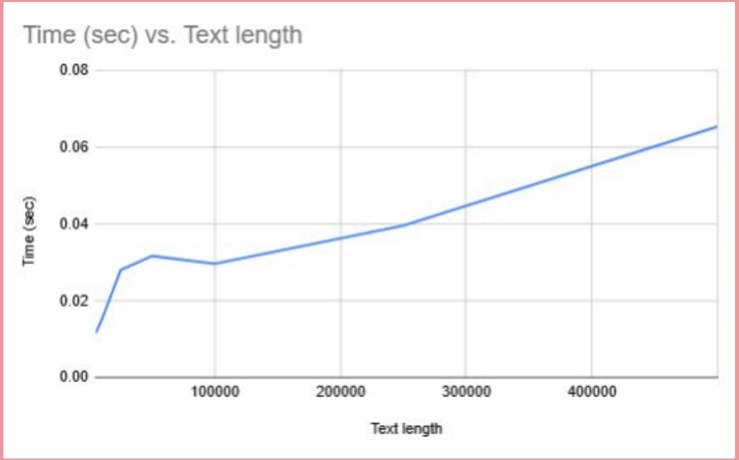


500k vs 500M chars

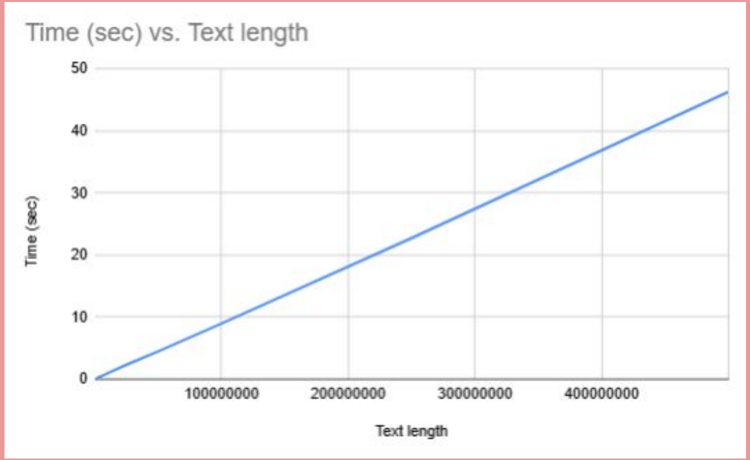


Results (pattern “aaba” in text of randomly generated a’s and b’s)

Boyer-Moore time complexity

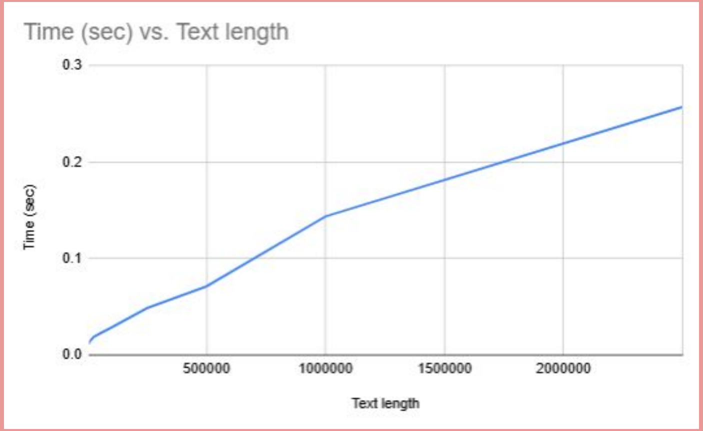


500k vs 500M chars

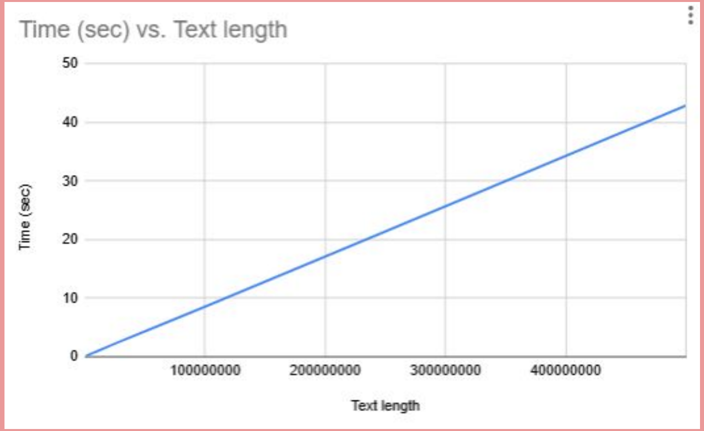


Results (pattern “aaba” in text of randomly generated a’s and b’s)

Rabin-Karp time complexity



2.5M vs 500M chars



Results (pattern is “aababbaabbaabababbaa”)

KMP: reduced to 1.45 secs (from 41.9 secs)

Boyer-Moore: reduced to 1.93 secs (from 46 secs)

Rabin-Karp: reduced to 3.36 secs (from 42.9 secs)

This demonstrates the idea that searching for less common patterns (instead of words like “the” and “of”) takes less time because of the way the algorithm works.

Optimization

Due to the very straightforward nature of string matching algorithms, one of the best ways to optimize when searching very large texts (many millions of characters) is to parallelize. One good way to do this would be to have a light algorithm that scans a file in and is able to divide and conquer by splitting the search in half with parallel processes. One would assign threads to chew through specified portions of the file, raising CPU utilization and decreasing time. Ctrl+F does not commonly use parallelism because the relative benefits are only seen when you are processing a file that's truly enormous.

References

- Geeks, G. F. (2022, November 12). KMP algorithm for pattern searching. GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
- Geeks, G. F. (2023, September 1). *Rabin-Karp algorithm for pattern searching*. GeeksforGeeks. https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/
- Siddharth. (2021, October 19). *The boyer moore string search algorithm*. Medium. https://medium.com/@siddharth.21/the-boyer-moore-string-search-algorithm-674906cab162
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. The MIT Press.