**STRING SEARCH ALGORITHMS**

There are different search algorithms which can be implemented although depending on the use case, only a few will be practical. Search algorithms such as linear search, binary search, depth first search, breadth first search and A\* Search are not very applicable in your scenario since we are searching for a word in a document or in a long string that might contain up to 250, 000 words.

In our case we implement word search algorithms that are more suited to finding words in documents. Searching for words in documents involves algorithms designed for text retrieval, string matching, and pattern recognition. Here are a few common algorithms used for searching words within documents:

1. **Naïve String Matching**
2. **Knuth-Morris-Pratt (KMP) Algorithm**
3. **Boyer-Moore Algorithm**
4. **Rabin-Karp Algorithm**
5. **Aho-Corasick Algorithm**

In this brief article I will explain the above algorithms and add a few bonuses at the end. The code for the implementation of the various algorithms will be found within the project’s repository in a file called searchalgorithms.py. I will also attach a graph showing the performance of each algorithm plotted against each other as a function of time and the size of the document where the string is being searched.

1. **Naïve String Matching**

**Description:** This basic algorithm checks for the presence of a substring (pattern) within a string (text) by comparing it character by character in a linear way.

**Complexity:** O(n\*m), where n is the length of the text and m is the length of the pattern.

1. **Knuth-Morris-Pratt (KMP) Algorithm**

**Description:** This algorithm improves the efficiency of string matching by preprocessing the pattern to create a partial match table (prefix table) that allows the search to skip characters in the text.

**Complexity:** O(n + m), where n is the length of the text and m is the length of the pattern

1. **Boyer-Moore Algorithm**

**Description:** This algorithm preprocesses the pattern to create two tables (bad character and good suffix) that guide the search, allowing it to skip sections of the text, making it efficient for longer patterns.

**Complexity:** O(n/m), where n is the length of the text and m is the length of the pattern, in the best case scenario.

1. **Rabin-Karp Algorithm**

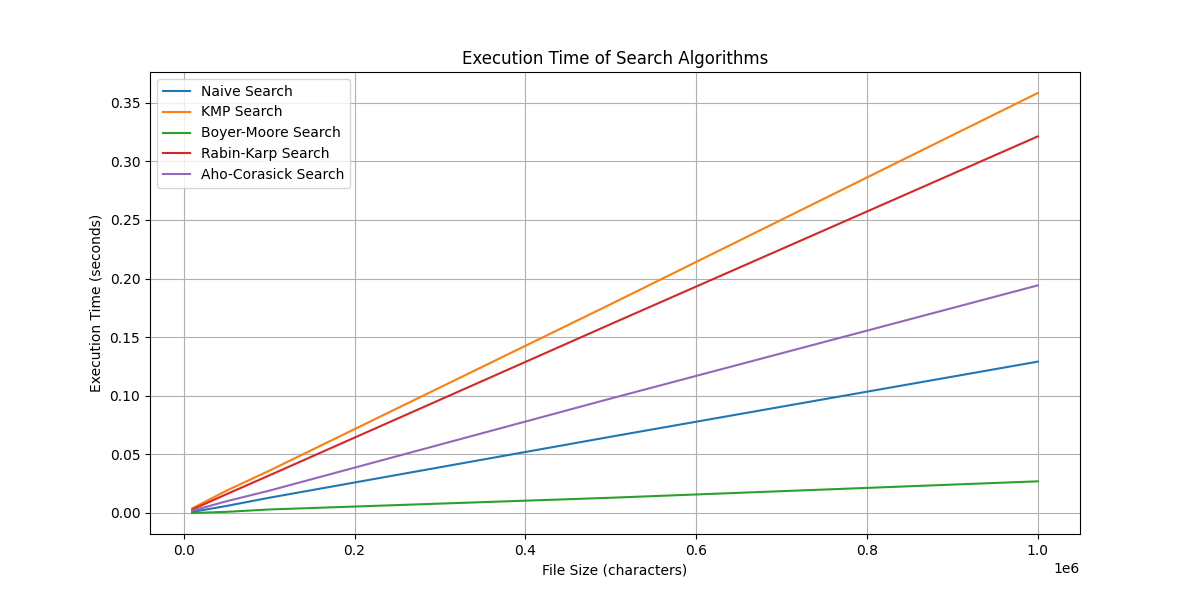
**Description:** This algorithm uses hashing to find a substring within a string. It calculates the hash of the pattern and compares it with the hash of substring in the text.

**Complexity:** O(n + m) on average, where n is the length of the text and m is the length of the pattern.

1. **Aho-Corasick Algorithm**

**Description:** This algorithm builds a finite state machine from a set of patterns and uses it to search for all occurrences of the pattern simultaneously in a text.

**Complexity:** O(n + m + z), where n is the length of the text, m is the total length of all patterns, and z is the number of pattern occurrences found.



The algorithms vary in complexity and suitability depending on the specific requirements and characteristics of the text and patterns being searched.