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.

2. 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

3. 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.

4. 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.

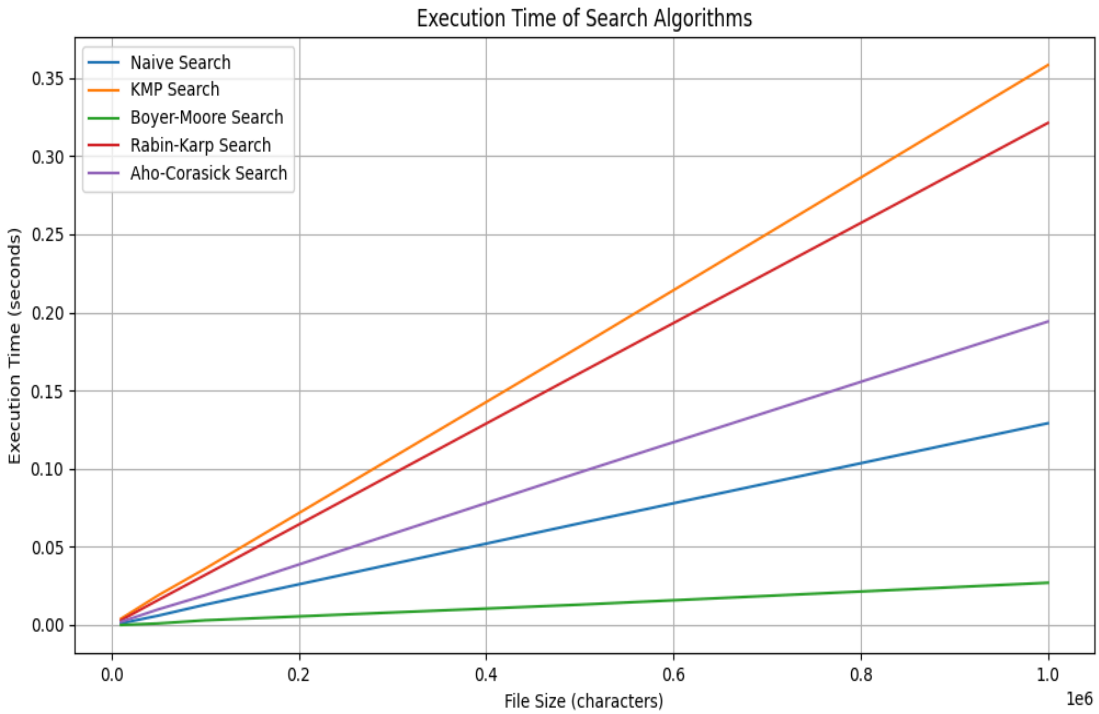
5. **Aho-Corasick Algorithm**

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.

TABLE: DIFFERENT SEARCH ALGORITHMS PERFORMANCE WHERE FILE SIZE IS TABULATED AGAINST THE TIME TAKEN TO COMPLETE THE SEARCH.

File Size (characters)	Naive Search	KMP Search	Boyer-Moore Search	Rabin-Karp Search	Aho-Corasick Search
10,000	0.00100088	0.00400352	0	0.00400352	0.00100112
50,000	0.00700617	0.0180078	0.00100136	0.017015	0.0100183
100,000	0.0130203	0.0390267	0.00300288	0.0320287	0.0190177
500,000	0.0670612	0.183167	0.0130119	0.166151	0.0970793
1,000,000	0.134122	0.377334	0.027025	0.330301	0.212202



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

ADDITIONAL SEARCH ALGORITHMS

1. Boyer-Moore-Horspool Search Algorithm

Description: This algorithm is a simplification of the Boyer-Moore algorithm. It uses only the bad character rule to skip sections of the text that cannot contain the pattern.

Complexity: The best case time complexity is O(n/m), The average case is O(n) and worst is O(nm)

2. Z Algorithm

Description: This algorithm constructs the Z-array for a string, which stores the lengths of the longest substrings starting from a given position that match the prefix of the string. It is useful for pattern matching.

Complexity: The time complexity is O(n + m), the space complexity is O(n + m)

3. **Finite Automata Algorithm**

Description: This algorithm constructs a finite state machine (FSM) for the pattern and processes the text through it. It uses states and transitions to match the pattern against the text.

Complexity: The time complexity is O(N) and the space complexity is O(m|x|), where |x| is the size of the alphabet

4. **Bitap Algorithm**

Description: This algorithm (also known as the Shift-Or or Shift-And algorithm) uses bitwise operations to perform exact or approximate string matching. It builds a bitmask for the pattern and processes the text using bitwise operations.

Complexity: The time complexity is O(n) and the space complexity is O(m + |x|), where |x| is the size of the alphabet

5. **Wu-Manber Algorithm**

Description: this algorithm is an extension of the Bitap algorithm that supports approximate string matching. It uses bitwise operations and bit masks for efficient pattern matching.

Complexity: The time complexity is O(n), the space complexity is O(m|x|), where |x| is the size of