Summary

We developed two algorithms for keyword replacement in a corpus. The first algorithm used a tree to compare and replace keywords, while the second used regular expressions and string-matching. We followed the standard four-step process, formulating the problem, designing two algorithms - algorithm A, which was the straightforward solution to the problem, and algorithm B, which was a more efficient solution, proving the correctness of B, comparing their time complexities, implementing both algorithms, and conducting simulations to compare their running times.

Comparison of Algorithm 1 and Algorithm 2

Problem Formulation

The first algorithm involves saving the keywords in a tree structure, where each node contains two parts: the abbreviation and its corresponding phrase/keyword. The tree is sorted alphabetically, and the algorithm compares each word in the text with the words in the tree to find any matches. If a match is found, the algorithm replaces the abbreviation with its corresponding phrase/keyword.

The second algorithm involves using regular expressions to search for patterns in the text, and a string-matching algorithm to find the closest matches to an abbreviation in a list. The list of abbreviations is preprocessed by sorting them in order of length, and an index of the corpus of tweets is created. The algorithm then searches through the index for each abbreviation in the list and uses the string-matching algorithm to find the closest match. If a match is found, the abbreviation is replaced with its corresponding phrase/keyword.

Pseudo Code

The first pseudo-code uses a tree-based approach for abbreviation expansion. It saves words in a tree structure with each node containing the abbreviation and its corresponding expansion. The algorithm performs a sequential traversal of the tree to compare the input word to the abbreviation stored at each node until it finds a match and replaces the abbreviation with its expansion.

The second pseudo-code uses an index-based approach. It preprocesses and sorts abbreviations by length and creates an index of the corpus. For each abbreviation, the algorithm searches the index to find occurrences in the text and applies a string-matching algorithm to replace it with the closest match.

Algorithm Analysis

Algorithm 1 found that storing words in a tree structure improves efficiency and accuracy when searching for and replacing abbreviations in large volumes of text. Sorting the words alphabetically and using parallel traversal quickly narrows down the search space and identifies the correct expansion for an abbreviation. The time complexity is O(n) due to the for loop in the

parallelTraverse function. However, the Insert method has a time complexity of O(1), which does not significantly affect the overall time complexity. The algorithm's space complexity is also O(n), using a linked list to store nodes in the branch. Overall, this algorithm provides a fast and accurate solution for replacing abbreviations in large volumes of text.

Algorithm 2's correctness can be proven by replacing all the abbreviations in the corpus of text with their respective words or phrases. Its time complexity is O(nm), where n is the number of abbreviations and m is the length of the corpus. The algorithm iterates through each abbreviation in the list and searches for it in the corpus, requiring O(nm) operations. Its space complexity is O(n+m) as it requires space to store the list of abbreviations, the index, and the corpus. The algorithm provides a fast and efficient way to replace abbreviations in a corpus of text. Analyzing correctness and efficiency is crucial for both algorithms (1 & 2). Correctness is ensured by producing the right output for all inputs. The sorting algorithm's efficiency is determined by time complexity, while the abbreviation replacement algorithm's efficiency is based on both time and space complexity. The running time of the sorting algorithm is determined by the number of operations required to sort n elements, while the abbreviation replacement algorithm's time complexity is established by the number of operations to process n abbreviations and m words, and space complexity is also analyzed. Overall, choosing the right algorithm for a task based on its correctness and efficiency is essential.

Unexpected Cases/Difficulties

Both algorithms face similar challenges, mainly dealing with multiple meanings of abbreviations and homophones, which require context-aware techniques and linguistic data to determine the correct interpretation. These challenges reflect the complexity and ambiguity of natural language, making algorithm development challenging. Therefore, careful handling of such cases is necessary to obtain accurate and reliable results for both algorithms.