Keyword Replacement in a Corpus

A Summary and Comparison of Two Algorithms

Introduction

- Two algorithms were developed to solve the problem.
- Algorithm A involved saving keywords in a tree and replacing abbreviations with proper phrases.
- Algorithm B involved regular expressions and string matching to find closest matches to abbreviations.
- Standard four-step process followed; problem formulated, two algorithms designed (A and B), correctness of algorithm B proved, time complexities compared using asymptotic notation, both algorithms implemented and compared through simulations.

Problem Formulation

Algorithm 1: keywords saved in a tree structure with abbreviation and corresponding phrase/keyword in each node. Sorted alphabetically, algorithm compares each word in text with tree nodes to find matches and replace abbreviation with corresponding phrase/keyword.

Algorithm 2: uses regular expressions to search for patterns in the text and a string-matching algorithm to find the closest match for an abbreviation from a pre-sorted list. It creates an index of the corpus and replaces the abbreviation with its corresponding phrase/keyword if a match is found.

Pseudo-code

The first algorithm uses a tree-based approach, while the second algorithm uses an index-based approach. The first algorithm sequentially traverses the tree structure, and the second algorithm searches the index for any occurrences of the abbreviation and uses a string-matching algorithm to find the closest match.

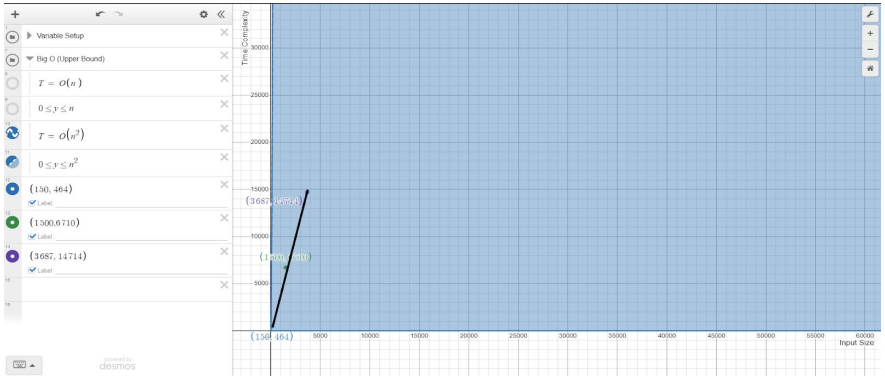
Algorithm 1

```
class Node<T> {
T abbreviations;
T expansion;
Node<T> next;
Node<T> tail;
Node(T abbreviations, T expansion) {
this.abbreviations = abbreviations;
this.expansion = expansion;
}
}
Class AbbTree {
Node root;
Node tail;
int maxBranchSize = n;
AbbTree(root) {
root.abbreviations = null;
root.expansion = null;
root.next = null;
}
Insert(String abbreviations, String expansion) {
If(branchSize < maxBranchSize) {
Node newNode = new Node(abbreviations,expansion);
tail.next = newNode;
tail = newNode;
}
Else {
Node newNode = new Node(abbreviations,expansion);
root.next = newNode;
tail = newNode;
}
}
//all getters and setters
}
//each branch is going to send from cpu to gpu for
parallel computing
Function parallelTraverse(String word,Branch b1) {
Node current = b1.root;
For(i; i < branch.len; i++) {
If(word.equals(node.abbreviations)) {
Replace(node.expansion);
}
Else {
current = current.next;
}
}
}
```

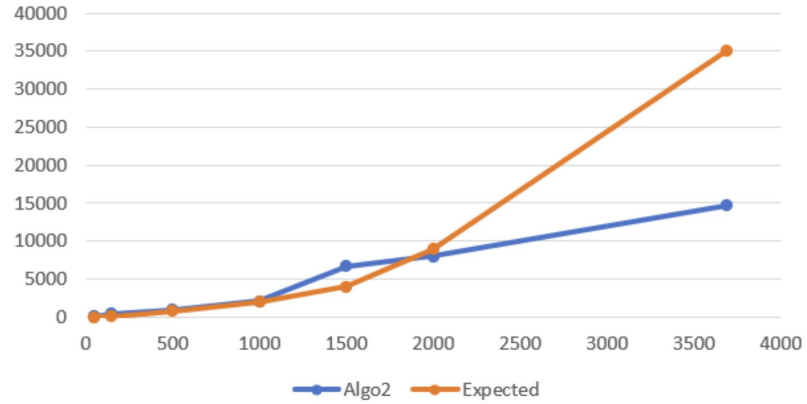
Algorithm 2

```
ist<String>                abbreviations
sortByLength(listOfAbbreviations);
// Create index of corpus
Map<String,         List<Integer>>         index         =
createIndex(corpusOfText);
// Iterate through abbreviations
for (String abbreviation : abbreviations){
// Search index for abbreviation List<Integer> matches =
searchIndex(abbreviation, index);
// If abbreviation found, use string matching algorithm
if (matches.size() > 0)
{
String                    closestMatch                    =
stringMatchingAlgorithm(abbreviation,
matches);
// Replace abbreviation with closestMatch
corpusOfText      =      replaceAbbreviation(abbreviation,
closestMatch,
corpusOfText);
}
}
// Output corpus with abbreviations replaced
outputCorpus(corpusOfText);
```

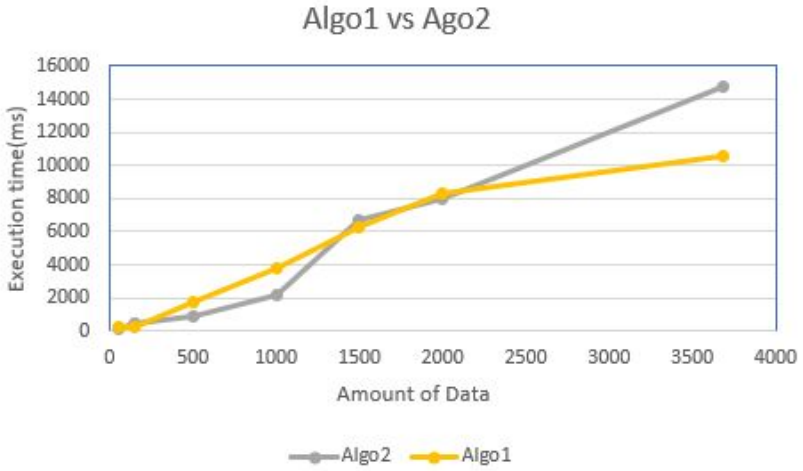
Algorithm analysis



Algorithm 1



Algorithm 2



Unexpected Cases and Difficulties

1. The difficulties for both algorithms are similar.
2. One of the main challenges is dealing with abbreviations that have multiple meanings.
3. Another challenge is dealing with homophones.
4. These difficulties are related to the ambiguity and complexity of human language, which can pose significant challenges in the development of algorithms that process natural language.

Thankyou!

Members-

1. Anna Panakkal
2. Erfan Kazemi
3. Sahil Chawla