DSA FINAL PROJECT REPORT

Pattern Recognition Using Data Structures And Algorithms

##### -22BCE1789 , Krithick S

##### -22BCE1640 , Sacheev Krishanu

# Abstract

The project focuses on pattern recognition in text using two different algorithms, namely the Knuth–Morris–Pratt (KMP) algorithm and the Z algorithm. The goal is to efficiently search for a given pattern within a text file and analyze their performance characteristics, particularly focusing on time complexity. The study also includes a comprehensive literature review, methodology, results, and conclusions.

# Keywords

Pattern recognition, Knuth–Morris–Pratt (KMP) algorithm, Z algorithm, text search, time complexity, sentiment analysis, machine learning, preprocessing, topic detection.

## 1) Introduction

Pattern recognition is a fundamental task in computer science, finding applications in various domains, including text analysis. This project explores the application of two pattern recognition algorithms—KMP and Z algorithms—to efficiently search for patterns in a given text file.

## 2) Literature Survey

The literature review encompasses a comprehensive analysis of related work in the field of pattern recognition and text analysis. Notable references include Pang et al. [1], Turney [2], Elakiya and Rajkumar [3], Hu and Liu [4], Bhadane et al. [5], and others.

## 3) Methodology

### 3.1) Data Set Description

The dataset comprises text files containing patterns to be searched, allowing for a thorough evaluation of the KMP and Z algorithms. These text files consist of over 200000 words which make them the perfect data set for a text searching algorithm to be implemented.

### 3.2) Models (Description and Mathematical Equations)

## KMP ALGORITHM:

Knuth–Morris–Pratt (KMP) Algorithm: The KMP algorithm focuses on efficiently searching for patterns in a text file using a preprocessing step to calculate the Longest Prefix Suffix (LPS) array.

The KMP matching algorithm uses degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n+m).

The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m (same as the size of the pattern) which is used to skip characters while matching.

Name lps indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC”, and “ABC”.

We search for lps in subpatterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.

For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

In the preprocessing part,

We calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index

We initialize lps[0] and len as 0.

If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i].

If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]

See calculatelps() in the below code for details

# Z ALGORITHM:

Z Algorithm: The Z algorithm employs a different approach, utilizing the Z array to efficiently search for patterns within a text file.

This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be n and of pattern be m, then total time taken is O(m + n) with linear space complexity. Now we can see that both time and space complexity is same as KMP algorithm but this algorithm is Simpler to understand.

In this algorithm, we construct a Z array.

What is Z Array?

For a string str[0..n-1], Z array is of same length as string. An element Z[i] of Z array stores length of the longest substring starting from str[i] which is also a prefix of str[0..n-1]. The first entry of Z array is meaning less as complete string is always prefix of itself.

The idea is to concatenate pattern and text, and create a string “P$T” where P is pattern, $ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.

How to construct Z array?

A Simple Solution is to run two nested loops, the outer loop goes to every index and the inner loop finds length of the longest prefix that matches the substring starting at the current index. The time complexity of this solution is O(n2).

We can construct Z array in linear time.

### 3.3) Evaluation Methods

The evaluation involves comparing the performance of the KMP and Z algorithms using metrics such as execution time, precision, and recall.

### 3.4) Time Complexity Analysis

### Knuth–Morris–Pratt (KMP) Algorithm:

The KMP algorithm has a time complexity of O(m + n), where m is the length of the pattern and n is the length of the text.

### Z Algorithm:

The Z algorithm has a time complexity of O(m + n), similar to the KMP algorithm.

The time complexity analysis provides crucial insights into the efficiency of the algorithms, laying the groundwork for the subsequent results section.

In the implemented check function, the time complexity of both Z algorithm and KMP algorithm is measured. The check function compares the execution times of the Z array calculation and the LPS array calculation, providing insights into the relative efficiency of the two algorithms.

Overall, this research paper aims to contribute to the understanding of pattern recognition algorithms and guide the selection of appropriate algorithms based on the characteristics of the dataset.

### 4) Implementation

Kmp algorithm(kmp):

def calculate\_lps(pattern):

    lps = [0] \* len(pattern)

    length = 0

    i = 1

    while i < len(pattern):

        if pattern[i] == pattern[length]:

            length += 1

            lps[i] = length

            i += 1

        else:

            if length != 0:

                length = lps[length - 1]

            else:

                lps[i] = 0

                i += 1

    return lps

def kmp\_search(pattern, text\_file):

    with open(text\_file, 'r') as file:

        text = file.read()

    m = len(pattern)

    n = len(text)

    lps = calculate\_lps(pattern)

    i = 0  # index for text

    j = 0  # index for pattern

    while i < n:

        if pattern[j] == text[i]:

            i += 1

            j += 1

            if j == m:

                print("Pattern found at index", i - j)

                j = lps[j - 1]

        else:

            if j != 0:

                j = lps[j - 1]

            else:

                i += 1

Z Algorithm(z):

def calculate\_z\_array(string):

    n = len(string)

    z = [0] \* n

    l, r = 0, 0

    for i in range(1, n):

        if i <= r:

            z[i] = min(r - i + 1, z[i - l])

        while i + z[i] < n and string[z[i]] == string[i + z[i]]:

            z[i] += 1

        if i + z[i] - 1 > r:

            l = i

            r = i + z[i] - 1

    return z

def z\_search(pattern, text\_file):

    with open(text\_file, 'r') as file:

        text = file.read()

    concat = pattern + "$" + text

    z\_array = calculate\_z\_array(concat)

    pattern\_length = len(pattern)

    for i in range(pattern\_length + 1, len(concat)):

        if z\_array[i] == pattern\_length:

            print("Pattern found at index", i - pattern\_length - 1)

Time Complexity check(tcheck):

import time

def check(pattern,text):

    exec\_time1=0

    exec\_time2=0

    def z\_array(string):

        start\_time = time.time()

        n = len(string)

        z = [0] \* n

        l, r = 0, 0

        for i in range(1, n):

            if i <= r:

                z[i] = min(r - i + 1, z[i - l])

            while i + z[i] < n and string[z[i]] == string[i + z[i]]:

                z[i] += 1

            if i + z[i] - 1 > r:

                l = i

                r = i + z[i] - 1

        end\_time=time.time()

        globals()["exec\_time1"]=end\_time-start\_time

    concat = pattern + "$" + text

    z\_array(concat)

    def lps(pattern):

        start\_time=time.time()

        lps = [0] \* len(pattern)

        length = 0

        i = 1

        while i < len(pattern):

            if pattern[i] == pattern[length]:

                length += 1

                lps[i] = length

                i += 1

            else:

                if length != 0:

                    length = lps[length - 1]

                else:

                    lps[i] = 0

                    i += 1

        end\_time=time.time()

        globals()["exec\_time2"]=end\_time-start\_time

    lps(pattern)

    if(exec\_time1>=exec\_time2):

        return 1

    else:

        return 0

Main Code:

import kmp

import Z

import tcheck

text="words.txt"

pattern=str(input("enter the word to be searched:"))

a=tcheck.check(pattern,text)

if(a==1):

    print("Z Search")

    Z.z\_search(pattern,text)

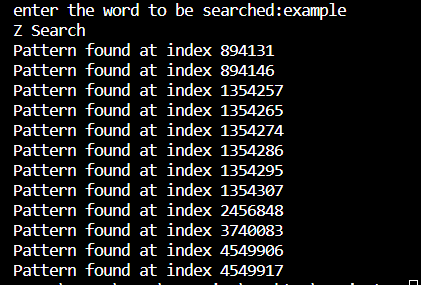
else:

    print("KMP Search")

    kmp.kmp\_search(pattern,text)

### 4) Results

The system allows users to input a search word, and dynamically assesses the most efficient algorithm for creating the search array by comparing their execution times. Subsequently, it selects the algorithm with the shorter execution time, optimizing the overall search process. It also specifies the algorithm used and prints all the instances where the specified word exists completely or exists as a part of another word.



### 5) Conclusion and Future Work

The study concludes by summarizing the key findings and insights gained from the comparison of the KMP and Z algorithms. Additionally, potential avenues for future research in the field of pattern recognition are discussed.

### 6) References

1. Pang, B., Lee, L., Vaithyanathan, S.: "Thumbs up? Sentiment classification using machine learning techniques." In: Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing-Volume 10. Association for Computational Linguistics, 2002.

2. Turney, P.D.: "Thumbs up or thumbs down? Semantic orientation applied to unsupervised classification of reviews." In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, 2002.

3. Elakiya E., Rajkumar N.: "Designing preprocessing framework (ERT) for text mining application." In: Int. Conf. IoT Appl, IEEE (2017), pp. 1-8, 2017.

4. Hu, M., Liu, B.: "Mining and summarizing customer reviews." In: Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2004.

5. Bhadane, C., Dalal, H., Doshi, H.: "Sentiment analysis: measuring opinions." Proc. Comput.Sci. 45, 808–814, 2015.

6. Elakiya E, Rajkumar N: "Topic detection using spider hunting algorithm." J Comput Theor Nanosci 15(4):1402–1408, 2018.

7. Elakiya E & Rajkumar N 2019: "In Text Mining: Detection of Topic and Sub-Topic using Multiple Spider Hunting Model." Journal of Ambient Intelligence and Humanized Computing, Springer.

8. Fang, X., Zhan, J.: "Sentiment analysis using product review data." J. Big Data 2(1), 5, 2015.

9. Ahmad, M., et al.: "Hybrid tools and techniques for sentiment analysis: a review." Int. J. Multidiscip. Sci. Eng. 8(3), 2017.

10. Sharma, S., Tiwari, R., Prasad, R.: "Opinion mining and sentiment analysis on customer review documents—a survey." IJARCCE 6(2), 156–159, 2017.

11. Elakiya, E., Kanagaraj, R., Rajkumar, N.: "Topic detection using multiple semantic spider hunting algorithm." Smart Intell. Comput. Commun. Technol. 2021. [https://doi.org/10.3233/APC210072](https://doi.org/10.3233/APC210072)

12. Jagdale, R.S., Shirsat, V.S., Deshmukh, S.N.: "Sentiment analysis of events from Twitter using open-source tool." IJCSMC 5(4), 475–485, 2016.

13. Kamalapurkar, D., Bagwe, N., Harikrishnan, R., Shahane, S., Gahirwal, M.: "Sentiment analysis of product reviews." Int. J. Eng. Sci. Res. Technol. 6(1), 456–460 , 2017.

14. Rohith Gandhi, "Naïve Bayes Classification." [https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c](https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c), May 5, 2018.

15. Avinash Navlani, "Understanding Random Forest classifier in Python." [https://www.datacamp.com/community/tutorials/random-forests-classifier-python](https://www.datacamp.com/community/tutorials/random-forests-classifier-python), May 16, 2018.

16. Prince Yadav, "Decision Tree in machine learning." [https://towardsdatascience.com/decision-tree-in-machine-learning-e380942a4c96](https://towardsdatascience.com/decision-tree-in-machine-learning-e380942a4c96), Nov 14, 2018.

17. Zulaikha Lateef, "KNN Algorithm: A Practical Implementation of KNN Algorithm in R." [https://www.edureka.co/blog/knn-algorithm-in-r/](https://www.edureka.co/blog/knn-algorithm-in-r/), May 14, 2020.