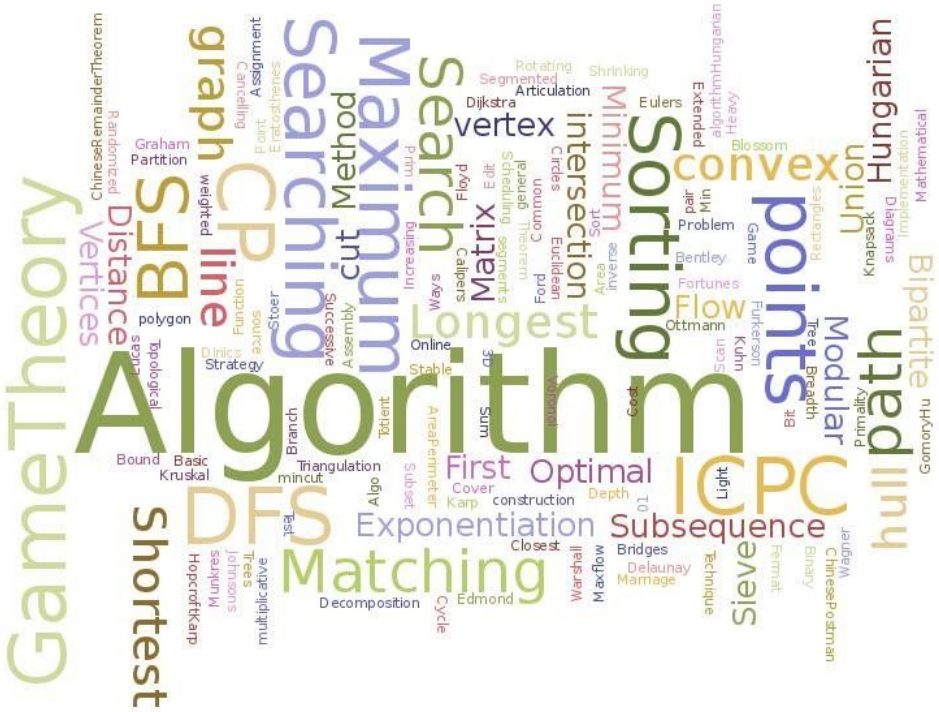

String Matching Algorithms

-By Ruchir Mehta (180001044) and Namani Sreeharsh (180001032)

—CS254 Design and Analysis of Algorithms

Ruchir Mehta & Namani Sreeharsh

180001044 & 180001032

Computer Science and Engineering

Abstract

We have chosen this theme to ensure us of the correctness of these algorithms and to learn deep mathematics behind these algorithms. We are desirous of learning how the text searching in text editors work, how are spell errors checked, etc. We will be analyzing the time and space complexities of the algorithms used in these practical cases and also implement them for our own satisfaction.

Algorithms to be Analyzed

1. Brute force algorithm
2. Knuth-Morris-Pratt
3. Karp-Rabin Algorithm
4. Automaton Algorithm
5. Bitap-Algorithm (Spell Check Algorithm)
6. Z-Algorithm()
7. Manacher's algorithm(longest palindromic substring)
8. Dictionary Implementation Using Trie

Brute Force Algorithm:

This is the most basic and naive pattern searching algorithm. In this approach, we simply search every window whose size is equal to that of the pattern and this the easiest algorithm to implement and the time complexity of the algorithm is O(m*n) where m is the size of the pattern and n is the size of the entire text tape. Though this is not advised, this stands as a base for analysing and inventing other efficient algorithms for solving the problem of pattern matching.

Knuth Morris Pratt (K.M.P.) String matching Algorithm:

This algorithm is the most important algorithm for solving the problem of string matching. It solves this problem in linear time and space complexity. This algorithm improves the basic naive algorithm by not just going to the next window blindly and redoing the same work. It skips a few windows selectively and chooses the next window to check for the pattern by the concept of Longest Proper Prefixes and Suffixes (L.P.S.) This involves greedy and dynamic programming, programming paradigms. This algorithm cleverly uses the properties of the longest proper prefix of a string which is also a proper suffix for choosing the next window after a deviation is detected in the current processing window.

Automation Algorithm:

This algorithm is a top-up on the above K.M.P. algorithm. The finite state transition machine which is built for the pattern as a part of the pattern pre-processing, needs the pre-processing done in K.M.P. algorithm. The only difference between these 2 algorithms is that we need not backtrack in a non-deterministic fashion, unlike K.M.P. for finding the next suitable window for checking if the current one has got deviation in it. We can immediately go to the next suitable window in O(1) time in a deterministic way. Time complexity is linear and space complexity is also linear.

Karp-Rabin Algorithm:

This algorithm is entirely different from the ones discussed ones. This algorithm works on the concept of role hashing, which calculates the hash of one window from the previous window in the order of constant time. Ideally or theoretically, this hash function is collision-free as it works like a new number system having 256 digits and a number can be represented only in one way in a particular number system. But practically, a few collisions may

occur by applying a modulus on the hash value for preventing arithmetic and space overflows. This method uses the hash of the windows as the primary filter for detecting pattern thus reducing the need of checking every window, unlike the naive algorithm.

Manacher's Algorithm

There are times when we need to find the longest substring in a given string that reads the same backwards and forwards. Here we will discuss Manacher's Algorithm, one of the classic **dynamic programming** algorithms that search for the **longest palindromic substring**. There do exist algorithms which use dynamic programming to solve this problem in $O(n^2)$ however this algorithm **reduces the complexity to O(n)** using some clever insights and properties of a palindrome.

Bitap Algorithm

It is an **approximate string matching** algorithm which compares two strings and finds out the **Levenshtein distance or Edit Distance** between them which can be broadly defined as *the minimum number of edits (transition, addition, deletions) in one string to make it equally as the other* using a very common programming paradigm generally known as **Dynamic Programming in $O(n^2)$**. There is some cost attached to each edit. In this way, we can compare how dissimilar two strings are.

It has a lot of applications in the field of bioinformatics and natural language processing.

Dictionary Implementation

We all have used a dictionary at least once in our life. But do we know how a computer lookup for the meaning of a word and that too instantly? Deep

inside it uses a data structure known as **Trie**. We will try to implement the same in this project. Using trie we can retrieve meaning in **O(|s|) time complexity.** It kinda uses a programming paradigm known as **dynamic programming.**

Z Algorithm

It is used to find the **pattern** as a substring in a string in **O(n)**. KMP Algorithm also does the same task at the same time complexity but Z algorithm is quite more **intuitive and easy to understand**. It is commonly applied in many applications like finding a phrase in a pdf or a virtual dictionary. To accomplish this task a preprocessing stage occurs where we trade space for time. We create a **Z Array** where Z[i] represents the longest length of a substring starting at Z[i] which is also the prefix. To accomplish this is the main aim to understand in this algorithm. Rest follows quite easily as we will see in this project. This approach also follows **Dynamic Programming** as we are reusing the previously calculated results.