CS 4501 - Executive Summary

Aleesha Khurram, Alex Suh, Tanush Siotia

April 2025

Introduction

For our final project, we will be discussing the Knuth-Morris-Pratt Algorithm (KMP), which is a fundamental idea used in string matching. String matching is a key operation in computer science wherein one wants to find occurrences of a given pattern within a larger text. This problem has applications in a wide variety of areas from search engines and text editors to network security amongst many others. As such, we must come up with an efficient algorithm to find all positions in a given text where a specified pattern occurs.

The naïve solution, as we will see below in more detail, runs in $O(n * m)$ where n is the length of the text we are sifting through and m is the length of the pattern we are looking for. Given the inefficiency of the brute force approach, other sophisticated algorithms have been developed for the problem of string matching such as Rabin-Karp and Boyer-Moore. Rabin-Karp, for example, uses hashing to check for potential matches quickly but could suffer from the possibility of a collision taking place.

With that in mind, KMP seems to be an attractive alternative to this problem with a fixed upper bound runtime of $O(n+m)$, where n represents the length of the text we receive, and m is the length of the pattern we are trying to match — a significant improvement to our brute force approach. KMP approaches the problem slightly differently by preprocessing the pattern to compute a partial match table. This then allows one to check how much of the pattern matches before a mismatch occurs to skip redundant comparisons. To build up to this, we will start with an investigation of the naïve approach.

Naïve Approach

The naive string matching algorithm works by attempting to match the pattern against the text starting from every possible position. For each position in the text, it compares the substring of the text with the pattern, character by character, until either a complete match is found or a mismatch occurs. If a mismatch happens, the algorithm shifts the pattern one character forward and repeats the process. The main issue with this approach is that it often performs redundant comparisons. For example, when a partial match occurs, the naive method does not take advantage of the information from previous comparisons and restarts from scratch, leading to a time complexity of $O(n * m)$, where n is the length of the text and m is the length of the pattern. Below is a pseudocode implementation of this idea.

```
Naïve − String − Searching(text = $s$, pattern = $p$)
$n$ = length of $s$
$m$ = length of $p$
let MATCHES be an empty array
let SUCCESS be a boolean variable
for $i = 0$, $i < (n - m)$ do
    SUCCESS = TRUE
    for $j = 0$, $j < (m - 1)$ do
        if $s[i + j] \neq p[j]$ then
            SUCCESS = FALSE
            break
        end if
        if SUCCESS == TRUE then
            append $i$ to MATCHES
        end if
    end for
end for
return MATCHES
```

As we will see, KMP improves upon this by using a preprocessing step to create a ‘partial match’ table, allowing the algorithm to skip over redundant checks and consequently improve the efficiency of solving this problem. We start by describing the prefix function to build up to the final KMP algorithm.

Prefix Function

Given a string s of length n, the *prefix function* of this string is an array π of length n, where $\pi[i]$ denotes the length of the longest *proper* prefix of the substring $s[0, \ldots, i]$ which is also a suffix of this substring. Symbolically, this can be expressed as

$$\pi[i] = \max_{k=0,\ldots,i} \{k : s[0, \ldots, k-1] = s\,[i-k+1, \ldots, i]\}$$

Note that the word proper indicates that the prefix cannot be the entire string itself. By definition, $\pi[0] = 0$. For example, given the string s = “abababaca”, $\textsc{PrefixFunction}(s) = [0, 0, 1, 2, 3, 0, 1]$.

The naïve approach for this algorithm consists of traversing the string, and for each index traversing the string once again from the beginning and checking the prefix condition. The pseudocode implementation is given below.

```
Naïve − Prefix − Function($s$)
    $n$ = length of $s$
    let $\pi[0 : n-1]$ be new array
    for $i = 0,\ i < n$ do
        for $k = 0,\ k < i$ do
            if SUBSTRING($s$, 0, $k$) = SUBSTRING($s$, $i - k + 1$, $k$) then
                $\pi[i] = k$
            end if
        end for
    end for
    return $\pi$
```

The time complexity of the naïve solution is $O\left(n^3\right)$. This is due to the two nested for-loops, and the SUBSTRING(s, i, j) method inherently running a for-loop. Thus, we have room for optimization. Knuth, Pratt, and Morris came up with two critical optimizations independently, and the resulting prefix function became the crux of the Knuth-Morris-Pratt string-searching algorithm.

Optimization 1: Consecutive Values of π Increase by at Most One

Proof. For the sake of contradiction, assume $\pi[i+1] > \pi[i] + 1$. Then, we can remove the last character from the suffix ending at index $i+1$ with length $\pi[i+1]$. The result is a suffix ending at index i with length $\pi[i+1] - 1$. However, this would imply that $\pi[i+1] - 1 > \pi[i]$, which is a contradiction. □

Given $\pi[i] = k$, we know that there are three possibilities for $\pi[i+1]$: $k - j$ $(j \leq k)$, k, or $k+1$. Since the prefix function can only increment by one in each step, given a string of length n, the function can grow (and decrease) at most n steps. This reduces the number of string comparisons and reduces the time complexity from $O\left(n^3\right)$ to $O\left(n^2\right)$.

Optimization 2: Reducing String Comparisons with Computed Values

Suppose we wish to find $\pi[i+1]$. At this state, we have already computed $\pi[0], \ldots, \pi[i]$. We begin by considering the case where $s[i+1] = s\left[\pi[i]\right]$. We know that the suffix at index i of length $\pi[i]$ is equal to the prefix of length $\pi[i]$, by definition of π. Using this, we can conclude that $\pi[i+1] = \pi[i] + 1$. This case can be illustrated as follows, wherein $\pi[i] = 2$.

$$\underbrace{\overbrace{a_0 a_1}^{\pi[i]}\ \overbrace{a_2}^{a_2 = a_{i+1}}}_{\pi[i+1]=\pi[i]+1} \cdots \underbrace{\overbrace{a_{i-1} a_i}^{\pi[i]}\ \overbrace{a_{i+1}}^{a_2 = a_{i+1}}}_{\pi[i+1]=\pi[i]+1}$$

If $s[i+1] \neq s\left[\pi[i]\right]$, we jump to the largest $j < \pi[i]$ such that $s[0, \ldots, j-1] = s[i - j + 1, \ldots, i]$. If such a j exists, if $s[i+1] = s[j]$, then $\pi[i+1] = j + 1$.

Else, we find the largest $k < j$ such that $s[0, \ldots, k-1] = s[i-k+1, \ldots, i]$. This process continues until we reach $j = 0$. From here, $s[i+1] = s[0]$ would mean $\pi[i+1] = 1$, and if they are unequal, then $\pi[i+1] = 0$. This optimization reduces whole string comparisons to individual character comparisons.

With these two optimizations, we present the final implementation of the prefix function.

```
Prefix − Function(s)
  $n$ = length of $s$
  let $\pi[0 : n-1]$ be new array initialised to 0
  for $i = 0$, $i < n$ do
      $j = \pi[i-1]$
      while $j > 0$, $s[i] \neq s[j]$ do
          $j = \pi[j-1]$
      end while
      if $s[i] = s[j]$ then
          $j = j + 1$
      end if
      $\pi[i] = j$
  end for
  return $\pi$
```

The final implementation runs in $O(n)$. Note that the nested while loop is rather deceptive as at a cursory glance, one might think the algorithm runs in $O\left(n^2\right)$. The key point is that the total number of iterations of the while loop over the *entire run* is *at most n*.

The last thing to note is that the prefix function is an *online* algorithm, meaning the prefix function is able to process the data as it arrives due to the optimization of only doing character comparisons instead of string comparisons. This feature is particularly notable in the Knuth-Morris-Pratt Algorithm, discussed below.

Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt Algorithm is an extension of the algorithm for identifying the *prefix function* of a string. It uses the fact that this prefix function compares the proper prefix of the string against suffixes of the string constructed by cutting the string at consecutive points along the string. This means that, assuming some substring *sub* is part of a larger string s, if *sub* is appended to the beginning of s, at some point, the prefix function will identify that the longest proper prefix (the substring) that is equal to the current suffix (the occurrence of the substring) is of at least length n and therefore that the substring is at the point in the string. By placing a separator between *sub* and s that occurs in neither string (commonly notated as #), we can limit the length of the longest possible proper prefix that matches the suffix to the length of *sub* (to account for the separator), and we only need O(m) space, where m is the length of *sub*, to store the string *sub* and its prefix function.

IPC Programming Challenge

Summary

This programming challenge aims to identify target substrings (”jewels”) inside a longer string (the set of ”mines”), while avoiding certain objects (the ”bombs”). Its overall learning objective is to gauge understanding of the prefix function and the Knuth-Morris-Pratt algorithm. It challenges students by incorporating three primary changes to the standard Knuth-Morris-Pratt algorithm:

1. The program must handle searching for multiple words simultaneously
2. The program must uniquely return the indices of tokens in the string instead of indices of characters in the string.
3. The program must handle recognizing which token indices must *not* be included

Solution

Our solution relies on a couple observations. The most straightforward is that mine indices can be tracked in a set and advanced for every new space-separated token that is encountered. The second is that Knuth-Morris-Pratt can be implemented as an online algorithm that processes characters one at a time. The third is that once a bomb is found, the rest of a mine can be skipped since it cannot be part of the solution. This means that processing the bombs simultaneously will optimize the search for a solution, even if the worst-case time complexity remains the same. The given solution keeps track of a data structure representing the prefix functions for each word, updating the data structure with each new letter and querying for the current status of identified matches. It is written in Python, Java, and C++.

This solution runs in O(k(m+n)) time and uses O(km) space.

Conclusion

Pattern recognition in strings is a foundational problem in computer science. As we have seen, traditional approaches to this problem are much to slow, as such we need to develop more efficient methods of doing so. The Knuth Morris Pratt algorithm improves upon this runtime improving it from a $O(m * n)$ to a runtime of $O(m + n)$ where m is the length of the pattern and n is the length of text we are searching in for the pattern.

Through this project we built up to the final implementation of the KMP algorithm starting off with the prefix function and introducing 2 optimizations that we can integrate into our solution to achieve the desired runtime.

Lastly, the programming challenge posed introduces some edge cases one might need to consider while implementing KMP and introduces the idea of when one needs to search for multiple patterns in a text.