Final Project for CS 372

Keiran Berry, Robert Book

Algorithm, Application, Language Choice

- Knuth-Morris-Pratt
- Finding Words in Book Titles
- Python

Program usage or README

Running the program without any command line arguments results in a default run of the program. It will prompt the user for the text file containing the book titles to be searched. Each line of the text file should contain a book title. By using command line argument “-c”, the correctness tests will be run, and their results will be output to the console. Similarly, command line argument “-s” is used for speed tests, and “-r” is used for single string tests.

Where It Is Used

The Knuth-Morris-Pratt algorithm is a string-matching algorithm. In most implementations of this algorithm, it returns an integer array of the beginning indices of each occurrence of the substring within the string.

Other applications

- Can be used in genetic data, searching for specific DNA sequences
- Can be used for image and speech recognition to look for patterns within them
- Can be used in anti-virus software, searching for specific patterns to detect malware
- Can be used by compilers to match patterns in source code

Alternative algorithms

- *Boyer Moore algorithm*
- *Rabin Karp algorithm*
- *Zhu-Takaoka algorithm*

Reason for choice

We chose this algorithm because it seemed interesting at first glance, and because string matching has so many uses in the real world. Beyond the obvious uses of finding words in files or strings, it can also be used in numerous other applications. These include, but are not limited to, text editors, genetic data, security, image and speech recognition, file management, antivirus software, and data compression. This algorithm seemed like something that made conceptual sense to us, and was not too different from algorithms we had implemented in the past for string matching.

The Knuth-Morris-Pratt algorithm wins over its competitors in the sense that there are no worst-case inputs. The algorithm is O(n + m), where n is the length of the string being searched and m is the length of the pattern being searched for. The preprocessing will always take O(m) time, and the actual search will always take O(n) time. This linear runtime means that the KMP algorithm could be a better choice than competing algorithms such as the Rabin-Karp or Boyer-Moore, which both have runtimes of O(nm), as does the naive solution. In cases in which the string which is being searched is very large, the KMP algorithm would consistently be faster. This is due to the fact that a larger string to search means a larger n, so the linear runtime would obviously beat out the competing algorithms which have the O(n + m) runtime.

Our application of this algorithm, finding words in book titles, is actually marginally slower than the naive solution. This is because book titles are short strings, so the strength of the algorithm in dealing with very long strings (large n) does not quite shine through. This time is also reflected in our speed tests. While our application of the algorithm may not showcase where it shines in runtime, we were excited to implement the algorithm in an enjoyable application that has real-world applications. While it may not be the most abstract implementation, such as searching DNA, search functionality is used everywhere these days and it was interesting for us to get even more hands-on experience with it.

How Your Project Works

This project works by prompting the user to provide an input file, which is a list of book titles, and a string to search for in the book titles. It then searches for the string in all of the book titles and returns the number of titles that it found that contained the string.

The way that it searches for the string is that it takes each title in as a string and the string to search for as a substring. It applies the Knuth-Morris-Pratt algorithm to determine if the substring is contained in the larger string.

The way that the Knuth-Morris-Pratt algorithm works is that it first does some preprocessing on the substring to find smaller repeated substrings that are contained in the larger substring (e.g. abcaabc contains abc twice) and marks these substrings. It then goes into the main loop where it searches for the substring in the larger search string. The loop iterates through both strings and finds matching consecutive characters. The area where this algorithm optimizes from the naive solution is if there are multiple of the smaller substrings in the larger substring (abc in the above example). If the user is searching for a substring where the last few characters are the same as the first few characters, instead of having to go back to the start of where the last substring was found, the algorithm already knows that the last few characters of the substring are the same as the first few so it can immediately start looking at the next index in the larger string. In our implementation, the algorithm returns a boolean if the substring was found in the larger book title string to save some time so that if the substring appears at the beginning of the title, it does not have to look through the entire larger string.

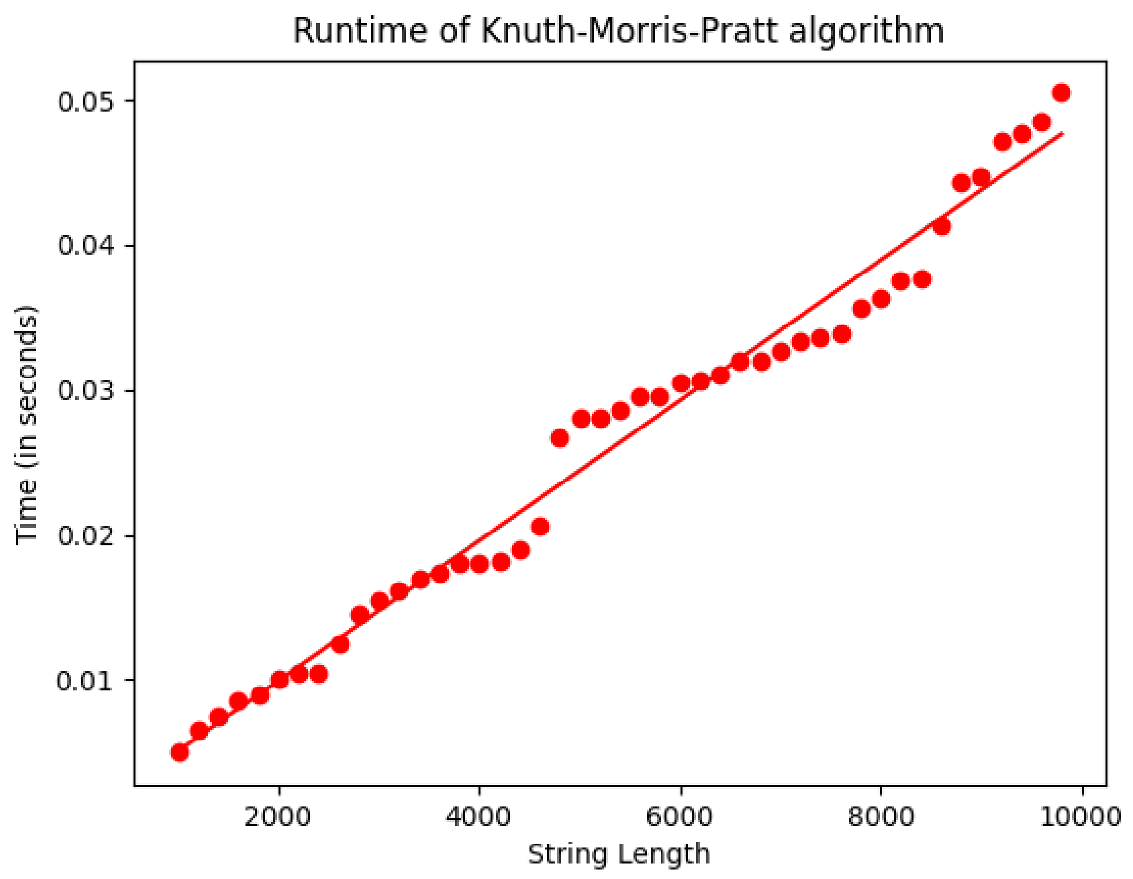
Correctness (Loop Invariant, Pre and Post Conditions)

The loop invariant in this algorithm is that j, the index of the character currently being compared, is equal to the current number of matching characters found. At the beginning of the run, j is equal to 0, because nothing has been searched or found yet. This is the initialization step. At the end of each iteration of the loop, j is equal to the current number of matching characters found, which holds true for the entire run of the algorithm. This is the maintenance step. The loop terminates either when there is no longer enough string to search for the substring, and therefore the pattern does not exist in the string, or j is equal to the length of the pattern. In this case, the pattern has been found in the string, as the number of consecutive matching characters (j) is equal to the full length of the substring. This is the termination step.

Preconditions of the running of the algorithm are that both the string and pattern are non-null and non-empty strings. The post condition of the algorithm is that “true” will be returned if there is at least one occurrence of the substring in the string, and “false” will be returned otherwise. Also, the original text and pattern strings remain unchanged at the end of the loop.

Run time

The Knuth-Morris-Pratt algorithm has runtime O(n + m), where n is the length of the string being searched and m is the length of the string that is being searched for. This holds true in our implementation, as well. Making the LPS array takes O(m) time, as our “while” (highlighted in yellow in the pseudocode below) loops through the length of the substring. The actual string search takes O(n) time, as our “while” in the kmp function (highlighted in yellow) loops a maximum of n times. Our implementation of the algorithm actually quits after it finds a match, as our goal is to simply determine whether or not the title contains the word in question. For this reason, we don’t have to keep looking for multiple matches after we reach our first match, and we can just quit then. Due to this, our application does not need the algorithm to loop all n times if a match has already been found, but the worst-case remains the same.



Formal Proof of Run Time Using Instruction Counting

Pseudocode courtesy of scaler.com, annotated for instruction counting:

LPS function:

```
LPS ← array [size = pattern length]
LPS[0] ← 0  {LPS value of the first element is always 0}
len ← 0  {length of previous longest proper prefix that is
also a suffix}
i ← 1
m ← length of pattern
while i < m do
    if pattern[i] is equal to pattern[len] then
        len ← len + 1
        LPS[i] ← len
```

```
                i ← i + 1
            else  {pattern[i] is not equal to pattern[len]}
                if len is not equal to 0 then
                    len ← LPS[len - 1]
                else  {if len is 0}
                    LPS[i] ← 0
                    i ← i + 1
        return LPS
```

The while loop, highlighted in yellow, runs m times. Each of the if and else statements in this pseudocode run in constant time. These are highlighted in green. Due to this, the runtime of this function is O(m), where m is the length of the pattern (the substring passed in).

KMP function:

```
        LPS ← ComputeLPS(Pattern) {build LPS table function}
        i ← 0
        j ← 0
        n ← string length
        m ← pattern length
        while i < n do
            if pattern[j] = string[i] then  {if the characters are
        a match}
                i ← i + 1
                j ← j + 1
              if j = m then  {j pointer has reached end of pattern}
                return i - j {index of the match}
                    j ← LPS[j - 1]

            else if i<n && pattern[j] != string[i] then {no match}
                    if j > 0
                    j ← LPS[j - 1]
                  else
                i ← i + 1
        return -1 {no match}
```

The while loop, highlighted in yellow, runs n times. Each of the if and else statements, highlighted in green, run in constant time. Therefore, the runtime of this function after filling the LPS array is O(n). Since filling the LPS array, highlighted in blue, is a part of the Knuth-Morris-Pratt algorithm, the overall runtime for this algorithm is O(n + m), where n is the length of the entire string and m is the length of the pattern/substring. The O(m) comes from the LPS function, and the O(n) comes from the rest of the KMP function. Therefore, this algorithm runs in linear time, O(n + m).

Code Correctness Tests

Test 1: no substring

There is no appearance of the substring in the Book Titles

Input:

file to search: 'fictionAntiWarBooks.txt'

substring: 'life'

Expected output: 0

Actual output: 0

Test 2: single space

Testing if a space gets correctly found

Input:

file to search: 'nonFictionAntiWarBooks.txt'

substring: ' '

Expected output: 70

Actual output: 70

Test 3: single word

Testing if a common word is found correctly

Input:

file to search: 'allAntiWarBooks.txt'

substring: 'new'

Expected output: 5

Actual output: 5

Test 4: capitalization

Testing if capitalization has any affect (it should not)

Input:

file to search: 'fictionAntiWarBooks.txt'

substring: 'wAr'

Expected output: 9

Actual output: 9

Test 5: repeated words in one line

Testing that the algorithm will only return the number of lines that a word appears rather than how many times it appears

Input:

file to search: 'allAntiWarBooks.txt'

substring: 'war'

Expected output: 34

Actual output: 34

Test 6: single, non alphanumeric character

Testing that the algorithm works with non alphanumeric characters

Input:

file to search: 'allAntiWarBooks.txt'

substring: ':"'

Expected output: 34

Actual output: 34

References

GeeksforGeeks. (2023, November 16). *KMP algorithm for pattern searching*. GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/#

Wikimedia Foundation. (2023, October 31). *Knuth–Morris–Pratt algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

Quora. (2023, Feb 8). *What are some applications of the Knuth-Morris-Pratt Algorithm in real life?* https://www.quora.com/What-are-some-applications-of-the-Knuth-Morris-Pratt-Algorithm-in-real-life

Khumaidi, A., Ronisah, Y. A., & Putro, H. P. (2020, January). Comparison of Knuth Morris Pratt and Boyer Moore algorithms for a web-based dictionary of computer terms.

Mathur, T. (2021, November 15). *KMP algorithm: Knuth Morris Pratt algorithm*. Scaler Topics. https://www.scaler.com/topics/data-structures/kmp-algorithm/