String Search Algorithms

Boyer Moore and KMP

Project Report

Arlen Dumas

Sabrina Johnson

Introduction

The Boyer Moore and Knuth-Morris-Pratt algorithms are both string search algorithms that search for an instance of a pattern “P” within a string “S”. Both of these algorithms rely on preprocessing the pattern being searched to determine how many characters can safely be skipped before a match is found. By skipping a portion of the comparisons between the pattern and the string, these algorithms both aim to offer better space complexities than a brute force search.

Boyer Moore works more efficiently with a suitably long pattern to search for due to the computational cost of preprocessing. The Boyer Moore algorithm has O(*nm)* time complexity when the pattern does appear in the text and O(*n+m)* when the pattern does not appear, where m is the length of the pattern and m is the length of the text [1].

Knuth-Morris-Pratt works best with a limited alphabet due to the need for sufficient information to be given in the pattern for a notable number of shifts to be made. The Knuth-Morris-Pratt algorithm has O(*n+k)* time complexity where k is the length of the pattern [2]. Unlike Boyer-Moore, this time complexity will not change if the pattern is not found within the text being searched.

Boyer Moore in particular has a great number of practical applications. It has been part of the C++ standard string search library since C++ 17, is included in the Boost algorithm library, and is used in GNU’s grep for standard string searches [1]. Knuth-Morris-Pratt has significantly less practical applications but is popular for searching in limited alphabets [2].

Methods

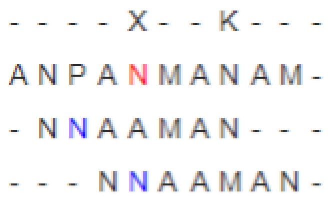
The methods for both Boyer-Moore and Knuth-Morris-Pratt rely heavily on preprocessing the string being searched for to make decisions on how many comparisons can safely be skipped.

Boyer Moore

Boyer Moore is the benchmark for practical string-search algorithms and is defined by using two different heuristics, or rules, to preprocess the pattern being searched for. By searching the string backwards and utilizing both of the heuristics, the shift amount for each can be compared and the algorithm is able to skip sections of text where no match would occur, resulting in a smaller time complexity than many other string search algorithms [3].

Bad Character Heuristic

The bad character heuristic considers a character that has been mismatched during the comparison process and applies one of two shift rules. If the mismatched character in the string appears to the left of the mismatched character in the pattern, a shift will be proposed to create a match. If the mismatched character in the string does not appear within the pattern, a shift that will move the pattern entirely past the mismatched character is proposed [4].



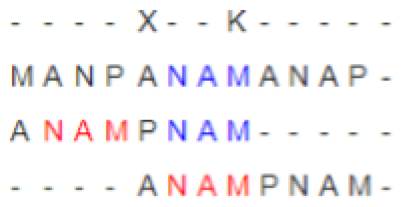
[1]

Preprocessing Bad Characters

By creating a table that is indexed by the last instance of a character within the pattern, we can return the index of a specific character in constant time. If there is no occurrence of a character with a pattern, the preprocessing table will return -1.

Good Suffix Heuristic

Like the bad character heuristic, the good suffix heuristic exploits preprocessing to compare substrings within the pattern being searched for to propose shifts to the next instance of a match [3].


- - - - X - - K - - - - -
M A N P A N A M A N A P -
A N A M P N A M - - - - -
- - - - A N A M P N A M -

Preprocessing Good Suffixes

[1]

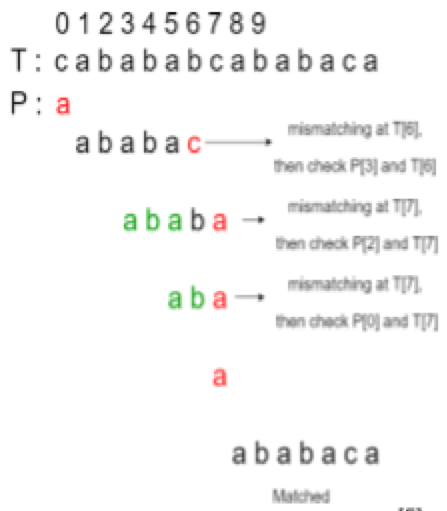
Preprocessing for the good suffix heuristic requires an array of beginning and ending indexes for borders are calculated, where a border is a substring is both a proper prefix and suffix for a particular word [4]. This table of borders will help propose a shift by calculating how many comparisons must be skipped to match prefixes or suffixes from the pattern to the string.

KMP

Knuth-Morris-Pratt focuses on preventing extraneous comparisons by skipping over previously matched characters using the longest proper suffix.

Longest Proper Suffix (LPS)

Knuth-Morris-Pratt also requires preprocessing to create a match table. By identifying the longest possible substring within a given pattern, mismatches with the text can be identified and skipped over before the comparison process.


0 1 2 3 4 5 6 7 8 9
T : c a b a b a b c a b a b a c a
P : a
a b a b a c
mismatching at T[6], then check P[3] and T[6]
a b a b a
mismatching at T[7], then check P[2] and T[7]
a b a
mismatching at T[7], then check P[0] and T[7]
a
a b a b a c a
Matched
[5]

An example of a longest proper suffix is taking the pattern bbxbbbx and creating the suffix vector 010223 that corresponds to matches and mismatches within the substring [6]. In this example, the value 2 is repeated when b is repeated. Following this logic, the longer pattern bbxbbbbbbbbx would result in the LPS vector 010122222223.

Knuth-Morris-Pratt works best when sufficient information is given during the preprocessing step. If a stronger correlation between the pattern and the alphabet is able to be determined, skips along the string can be made more frequently [8]. Because of this, KMP is often limited to use within small sets with little cardinality, such as DNA sequencing.

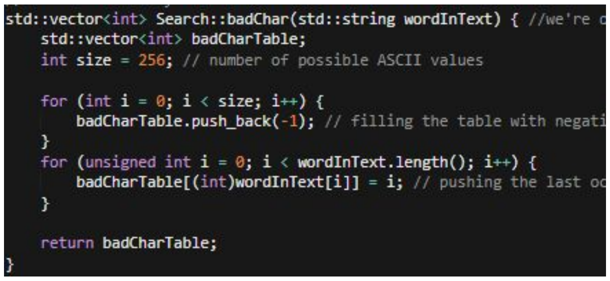
Implementation

The implementation can be found on our github https://github.com/a-dumas/CSC-212-Final

Handling Boyer Moore

Bad character

For the bad character heuristic, I implemented a private function that creates a vector populated with the value -1. The position of the last instance of any given character is then pushed to the vector before it is returned to the main search function.

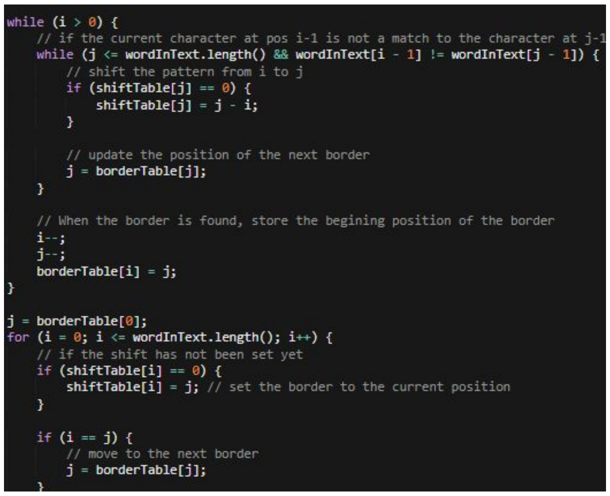


```
std::vector<int> Search::badChar(std::string wordInText) { //we're c
    std::vector<int> badCharTable;
    int size = 256; // number of possible ASCII values

    for (int i = 0; i < size; i++) {
        badCharTable.push_back(-1); // filling the table with negati
    }
    for (unsigned int i = 0; i < wordInText.length(); i++) {
        badCharTable[(int)wordInText[i]] = i; // pushing the last oc
    }

    return badCharTable;
}
```

Good suffix

The good suffix heuristic was a bit more complicated, but was able to be handled similarly to the bad character function. I implemented a private function that fills a vector of shift values with 0, and a vector of border positions with -1. The border table at the searched position is then initialized to be the end of the pattern before checking if there are any matches between the pattern and the string being searched.

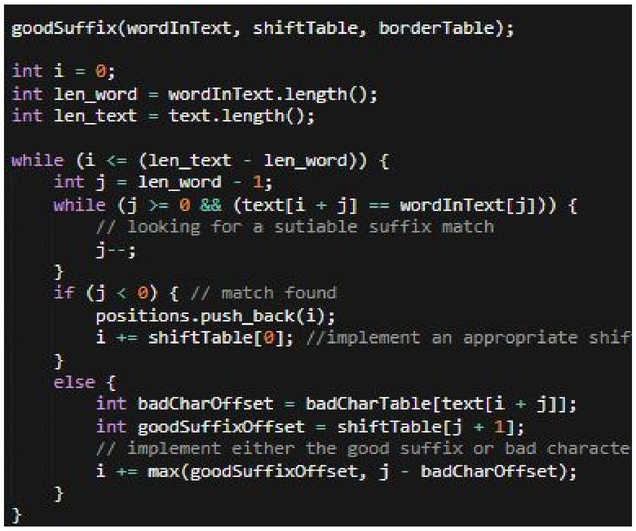
If there is a mismatch, the rightmost position of the border table is searched and the pattern is shifted from position i to position j before the border position is updated. When a proper border is found, the beginning position of that border is found. If there has been no shift set for a particular border, the border is set to the current position before moving to the next border. This function will not return anything but will be accessible to the main search function.



```
while (i > 0) {
    // if the current character at pos i-1 is not a match to the character at j-1
    while (j <= wordInText.length() && wordInText[i - 1] != wordInText[j - 1]) {
        // shift the pattern from i to j
        if (shiftTable[j] == 0) {
            shiftTable[j] = j - i;
        }

        // update the position of the next border
        j = borderTable[j];
    }

    // When the border is found, store the begining position of the border
    i--;
    j--;
    borderTable[i] = j;
}

j = borderTable[0];
for (i = 0; i <= wordInText.length(); i++) {
    // if the shift has not been set yet
    if (shiftTable[i] == 0) {
        shiftTable[i] = j; // set the border to the current position
    }

    if (i == j) {
        // move to the next border
        j = borderTable[j];
```

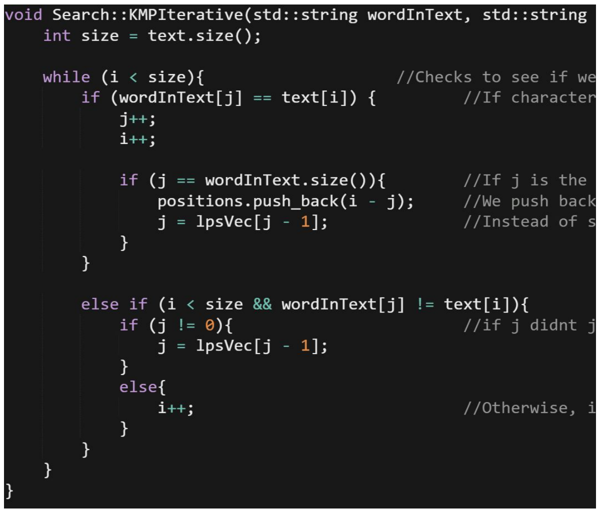
Searching using preprocessing

With the preprocessing complete, the function that found the positions of matches was simple to implement. As long as the pattern is shorter than the string being searched, the program loops through a string and searches for a suitable suffix match. If no match is found, it will compare the results given by the good suffix and bad character lookup tables and complete the largest possible shift.



```
goodSuffix(wordInText, shiftTable, borderTable);

int i = 0;
int len_word = wordInText.length();
int len_text = text.length();

while (i <= (len_text - len_word)) {
    int j = len_word - 1;
    while (j >= 0 && (text[i + j] == wordInText[j])) {
        // looking for a sutiable suffix match
        j--;
    }
    if (j < 0) { // match found
        positions.push_back(i);
        i += shiftTable[0]; //implement an appropriate shif
    }
    else {
        int badCharOffset = badCharTable[text[i + j]];
        int goodSuffixOffset = shiftTable[j + 1];
        // implement either the good suffix or bad characte
        i += max(goodSuffixOffset, j - badCharOffset);
    }
}
```

Handling KMP



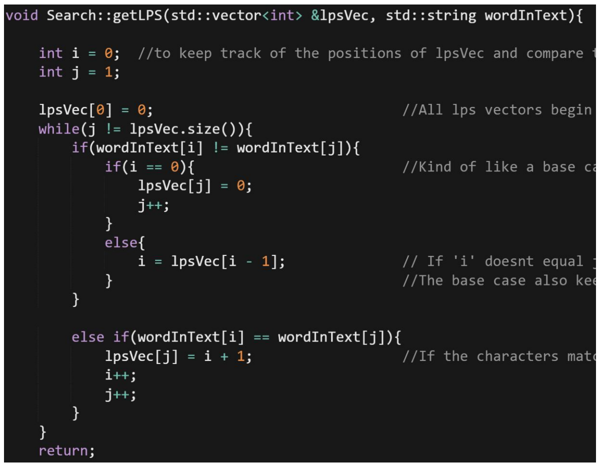
```
void Search::KMPIterative(std::string wordInText, std::string
    int size = text.size();

    while (i < size){                           //Checks to see if we
        if (wordInText[j] == text[i]) {         //If character
            j++;
            i++;

            if (j == wordInText.size()){        //If j is the
                positions.push_back(i - j);     //We push back
                j = lpsVec[j - 1];              //Instead of s
            }
        }

        else if (i < size && wordInText[j] != text[i]){
            if (j != 0){                        //if j didnt j
                j = lpsVec[j - 1];
            }
            else{
                i++;                            //Otherwise, i
            }
        }
    }
}
```

KMP Iterative and Recursive

Initially a recursive function for Knuth-Morris-Pratt was proposed and implemented due to the repetitive work shown upon initial research [6]. During the testing process it became evident that a recursive Knuth-Morris-Pratt searching algorithm is extremely inefficient and is killed by Linux after approximately 18,000 characters.

Due to the difficulty of optimizing this recursive function, I decided to move forward with an iterative implementation of the algorithm that used the same general methodology as the recursive implementation. Because the program was functional recursively, I renamed the function and kept it for testing purposes.

LPS

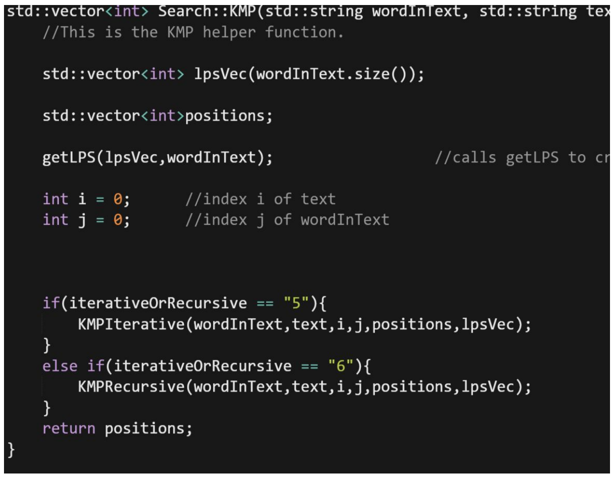


```
void Search::getLPS(std::vector<int> &lpsVec, std::string wordInText){

    int i = 0;  //to keep track of the positions of lpsVec and compare t
    int j = 1;

    lpsVec[0] = 0;                              //All lps vectors begin
    while(j != lpsVec.size()){
        if(wordInText[i] != wordInText[j]){
            if(i == 0){                         //Kind of like a base ca
                lpsVec[j] = 0;
                j++;
            }
            else{
                i = lpsVec[i - 1];              // If 'i' doesnt equal j
            }                                   //The base case also kee
        }

        else if(wordInText[i] == wordInText[j]){
            lpsVec[j] = i + 1;                  //If the characters matc
            i++;
            j++;
        }
    }
    return;
```

For the LPS, I created a private function that takes in a reference to a vector that keeps track of the numbers in the longest proper suffix, named ‘lpsVec’, and the pattern being searched for as parameters. This function then iterates through a while loop in linear time to create the actual lps vector. This function should not be returning the LPS vector, so it takes in a reference to the vector as a void function

Searching using LPS

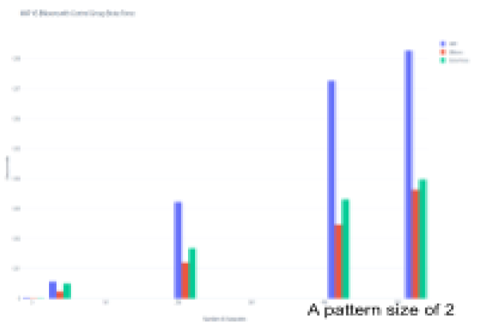
After completing the preprocessing, I implemented the main search function. The initial LPS vector is created within the search function and passed to the LPS function as a reference. Because the recursive and iterative functions are implemented, KMP acts as a helper function that is able to call either the iterative or recursive search function.



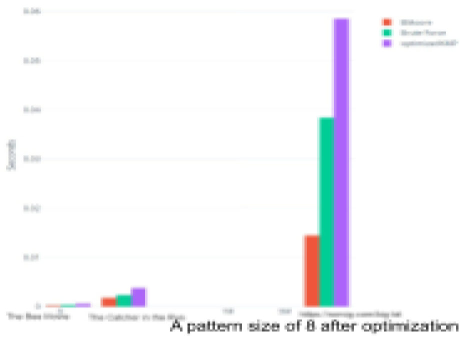
```
std::vector<int> Search::KMP(std::string wordInText, std::string tex
    //This is the KMP helper function.

    std::vector<int> lpsVec(wordInText.size());

    std::vector<int>positions;

    getLPS(lpsVec,wordInText);                  //calls getLPS to cr

    int i = 0;      //index i of text
    int j = 0;      //index j of wordInText



    if(iterativeOrRecursive == "5"){
        KMPIterative(wordInText,text,i,j,positions,lpsVec);
    }
    else if(iterativeOrRecursive == "6"){
        KMPRecursive(wordInText,text,i,j,positions,lpsVec);
    }
    return positions;
}
```

Comparisons

Using a brute force algorithm as a control group [7], we made several comparisons between the Boyer Moore and Knuth-Morris-Pratt algorithms. Using data sets of 130,000 characters, 350,000 characters, and 5 million characters, we used a pattern length of two and eight to see how the time complexity of these algorithms grew over time. Upon initial review, Boyer-Moore (red) appeared to run much more quickly with a larger pattern size but Knuth-Morris-Pratt (blue) appeared to be running extremely slowly in both cases. This caused us to research what data sets and pattern lengths these algorithms are intended to work with. Because we did not want to limit the brute force and Boyer Moore implementations with a limited alphabet, we tested for a second time with a pattern length of eight and did not see a notable difference. Because brute force (green) was still running significantly faster than Knuth-Morris-Pratt, we became aware that the program needed to be optimized.


A pattern size of 2

After reviewing the work we had done for Knuth-Morris-Pratt and re-referencing any algorithms we had looked at [6], we realized that a sizing method was being called repeatedly in a loop. By removing extraneous .size(), a large amount of computational space was freed and there was a performance increase of 28%.


A pattern size of 8 after optimization

After optimizing, we ran another test with the same 130,000, 350,000 and 5 million character sets with a pattern size of eight and noticed a significant increase in the runtime of Knuth-Morris-Pratt (purple), though it did not reduce the time to be under that of the brute force algorithm (green).

Contributions

Once we had chosen the content of our project, we worked together on the pseudocode for both Knuth-Morris-Pratt and Boyer Moore. Once we had gathered enough information and felt that we had a good understanding of each algorithm, we divided the C++ implementation up with Max working on KMP and Arlen working on Boyer Moore.

We maintained regular content throughout the five week duration of the project and had an open line of communication for thoughts, concerns, and issues that arose. We used Discord frequently to contact each other and share screens for pair programming debugging.

Anything that wasn't able to be resolved after we had both worked on the problem was brought to office hours. The header and main were worked on together. The project report was written together and edited by Arlen. The presentation was primarily written by Max and edited by Arlen.

The C++ brute force pattern searching algorithm that was used as a control was taken from GeeksforGeeks.com [7] and is documented in the readMe on the github.

Citations

[1]“Boyer–Moore string-search algorithm,” *Wikipedia*, 18-Nov-2020. [Online]. Available: https://en.wikipedia.org/wiki/Boyer–Moore_string-search_algorithm. [Accessed: 14-Dec-2020]

[2]“Knuth–Morris–Pratt algorithm,” *Wikipedia*, 10-Nov-2020. [Online]. Available: https://en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm. [Accessed: 14-Dec-2020].

[3] “Boyer Moore Algorithm: Good Suffix heuristic,” *GeeksforGeeks*, 31-Oct-2019. [Online]. Available: https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/. [Accessed: 14-Dec-2020].

[4]. P. Mittal, “Boyer Moore String Search Algorithm,” *OpenGenus IQ: Learn Computer Science*, 15-Oct-2019. [Online]. Available: https://iq.opengenus.org/boyer-moore-string-search-algorithm/. [Accessed: 14-Dec-2020].

[5]“KMP Algorithm for Pattern Searching,” *GeeksforGeeks*, 28-Sep-2020. [Online]. Available: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/. [Accessed: 14-Dec-2020].

[6] Rajinikanth, “Data Structures,” *Data Structures Tutorials - Knuth-Morris-Pratt Algorithm*. [Online]. Available: http://www.btechsmartclass.com/data_structures/knuth-morris-pratt-algorithm.html. [Accessed: 14-Dec-2020].

[7] “Naive algorithm for Pattern Searching,” *GeeksforGeeks*, 30-Oct-2020. [Online]. Available: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/. [Accessed: 14-Dec-2020].

[8] arcticriki (https://cstheory.stackexchange.com/users/32005/arcticriki), What is the applications of kmp algorithm?, URL (version: 2015-02-23): https://cstheory.stackexchange.com/q/30568