# Knuth-Morris-Pratt and Boyer-Moore String Search Comparison

**Authors:** Cameron Brickett, Jay Gomes, Ishan Chadha

**Abstract:** String searches are some of the most commonly used algorithms in everyday use. Anytime you hit ctrl+f, you are using a string search algorithm to find the pattern you're searching for. Because sizes and complexities of the search and the source you're searching from can vary, it is important to use the most efficient algorithm for that situation. We looked at the Knuth-Morris-Pratt as well as the Boyer-Moore string search algorithms to find the pros and cons of each, and overall which is more efficient. We found that for shorter alphabets (patterns with lower variation i.e. AAA), that the Knuth-Morris-Pratt algorithm was faster. For longer alphabets (patterns with higher variation ABC), the Boyer-Moore algorithm was faster.

## Introduction:

For something as simple as looking for a word in a wall of text, or comparing two papers to look for plagiarism, string search algorithms are required to accomplish these tasks. Efficiency is the most important factor in deciding which string search algorithm to use. We chose to look at two algorithms, the Knuth-Morris-Pratt algorithm and the Boyer-Moore algorithm. These are both algorithms that take in a string or pattern that you want to search for, as well as a source to search for said pattern in, and returns where/how many times that pattern is in the source text. Their difference lies in how they go about searching for the pattern. Their speeds, time complexities, and efficiency differ enough to make an impact when brought to industrial scales. By comparing the two algorithms, we aim to find the most efficient of the two algorithms overall, and where one would beat out the other. The goal of this is to see which algorithm would serve better as a plagiarism checker. As the internet grows everyday, plagiarism becomes more and more of a problem. To check every piece of information that exists on the internet and compare it with a student's paper to look for plagiarism is not realistically possible, at least not in a timely manner. However, as search algorithms become faster and more efficient, the range that a plagiarism checker can search in a shorter amount of time grows. The string search algorithm is just one part of a plagiarism checker, and choosing the right string search algorithm can decide the range, accuracy, and efficiency of the whole program.

## Methods:

### Knuth-Morris-Pratt:

The Knuth-Morris-Pratt algorithm is a string search algorithm that searches for a pattern within a string. It was created by James H.Morris and Vaughan Pratt, and discovered a few weeks later by Donald Knuth using automata theory. By analyzing the Naive string search algorithm, they found that it threw away a lot of information. So when they tested an algorithm that could use some of that thrown away information, they found it was more efficient than the Naive string search algorithm. The time complexity of the Knuth-Morris-Pratt algorithm is O(n+m), where n is the size of the source text and n is the size of the pattern being searched.

The way the algorithm works is similar to the Naive algorithm for string search, except it keeps some of the information that is otherwise thrown out. It starts out like the naive algorithm, by comparing the first character in the pattern W[0] with characters in the source string S  until it gets a hit. Once it gets a hit, we move to the next character in the pattern, W[1], and the next character in string S. This continues until either the pattern is found (the algorithm is done here), or when a character in W does not match a character in S. When this happens, it checks to see if any characters before the non matching character can be used as a start for the pattern. For example, if the pattern is W=ABC, and the string is S=ABACBABC, the program will encounter the wrong character at S[2]. However, because S[2] =A, and W[0] =A, it will use that as the start. It won't have to check if S[2] is A, and will instead check if W[1] = S[3]. This is how the KMP algorithm searches more efficiently than the Naive program.

When implementing the program, the KMP algorithm requires a preprocessing step. This is how the algorithm can find starting points for the pattern in the source string that are already checked.

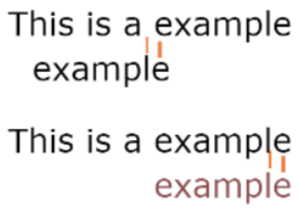
## Boyer-Moore:

The Boyer-Moore algorithm is a string search that was created by Robert Boyer and J Strother Moore in 1977. This algorithm works by checking the end of the pattern first and then making its may to the front of the pattern. The two rules it uses are the good suffix and the bad character heuristic rules. The time complexity of the Boyer-Moore algorithm is in the worst case it will be O(N*M) and O(N+M). The best time complexity will be O(N/M).

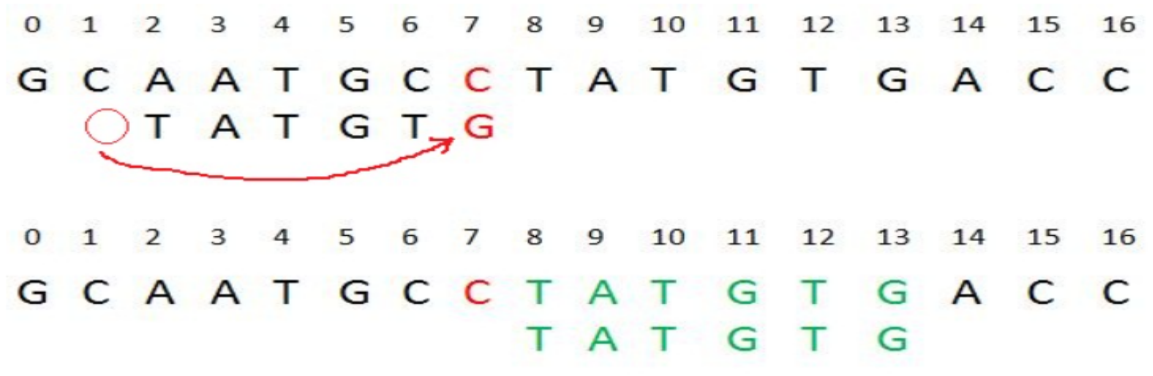
**Boyer-Moore Workings**

The algorithm starts by processing

The algorithm will search for a matching character in the pattern you are searching for in the certain document. It will look at the last letter of the pattern you are searching for and once it finds a matching letter, it will move one letter to the left on the pattern. It will keep doing this until a mismatch is found. Once that happens it will take the mismatch and align it to match up with the document you are looking in. It can look like this…



It will find a match for E and then see if pattern[pattern.length()-1] will have a match in the document you are searching in. Here there isn't a match so it will look for prefixes in the pattern that could possibly match up. In this example there is another E, so it would match the first E up to the E we have already found. Now the function checks from right to left again making sure that the pattern matches completely. The Bad Character Heuristic was when E was found at the end of the pattern "exampl**e**", but the L didn't match up to the corresponding letter in the Document. So it turned to the Good Suffix Rule where it found the first E and moved the pattern to match up with that of the first E. For the Bad Characteristic if there are no matching letters after the first comparison or when it only finds one match it will skip that section of the document.



## Implementations:

### Our Application

**Main.cpp**:

Our main.cpp file will first include our header files, knutt.h followed by booyer.h. The main file will also house our main function where our program will start. The main function will take in the parameters "int argc, char** argv" which will allow us to input command line inputs from the user. The function will then create two strings and store the two inputs expected from command lines to those strings. The first string will be the name of the file that will contain the original text. The second string will be the pattern that will be searched for in the original text. Then, the function will calculate the length of the pattern and save it into a newly created int called "len."

Now the function will create two objects of their respective classes with a custom constructor that will take in three parameters. Both constructors will take the parameters: the filename, the pattern, the length of the pattern. The function will then call the functions, "readfile" followed by "solvelps" which are respective to the knutt class. Finally, the calls to "readfileBoyer" and "searchBoyer" are made and are also respective to the boyer class.

**Knutt.h**

This header file will first include all the libraries our knutt class will require. The class called "Knutt" is created with three private strings and one private integer. The class also includes a public section with the parameterized constructor and the methods: readfile, solvelps, search with a single parameter an array.

**Knutt.cpp**

The knutt.cpp will house all the function definitions for the knutt class. The program will first include our knutt.h header file. Then our constructor will set our private variables equal to the parameters that were sent from the main. This is done so we can use the variables later in the code without having to pass them to a function every time.

The readfile function will initialize two strings, line and temp. That will be used to to fill our “originaltext” string. Ifstream is used to open the file. Then a while loop will run while there is still a line in the txt file. For every line the code uses stringstream to account for white spaces and adds the words to the string “originaltext.” Now the high resolution clock will start to time the solving of the lps array.

The solvelps function is part of the knuth-morris algorithm and will solve the lps array that will house the longest prefix that is also a suffix. The function will first initialize the pointers we will need to traverse the pattern and initialize our lps array. The first position is always set to 0 because no comparison can be made. A while loop will run for the length of the pattern. If a match is found from our position on the pattern with the previous character the lps array at the pattern current position will be equal to len+1.

In the case that the comparison does not match and len is equal to 0,  the lps array at current position is set to 0 and only our current position is advanced.

In the case we have advanced len and a mismatch is made we can not advance our current position and try to make new comparisons because previous made matches will be discounted. So, we will set len equal to the lps at len-1. This will allow us to account for previous made matches. The time for solving the lps array will be stopped and a call to search will be made with the lps array and the time elapsed as parameters.

Lastly, the actual search will be made. The pointers to the original text and the pattern are made. The clock will again start here. A while loop will run for the length of the original text. If a match is made, both pointers will advance and continue the search.

Else if a match is not made only the pointer to the original text is advanced. In the case that a mismatch is made and the pointer for the pattern is not 0 we will set the pattern pointer equal to lps[j-1]  which is the previous longest prefix that is also a suffix. This will allow us to make less comparisons while not missing the overlapping matches made. If the j is equal to the length of the pattern, this means we have reached the end of our pattern and the pattern has been found in the original text. A break will stop our while loop. The clock is then stopped and the final output of total elapsed time is printed.

**Booyer.h**

This header file will first include all the libraries our Boyer class will require. The class called “Boyer” is created with three private strings and one private integer. The class also includes a public section with the parameterized constructor and the methods: readfileBoyer, badcharHeuristic that takes in three parameters a string and two integers, searchBoyer.

**Boyer.cpp**

When the Boyer.cpp file is called from the Boyer.h it will first read the file that has been given.  It will assign the txt file name to be fname, the pattern we are searching for to pat, and the length of the pattern to len.  The header file will then call the searchBoyer function.  It will set a int called patLen equal to pattern.length(), and will set txtlen equal to originaltxt.length();.  It will next, set int badchar[NO_OF_CHARS].  Inside the brackets will be the number 256 to represent all of the possible characters in the database.  After the ints are set it will call the function badCharHeuristic giving this function the pattern, pattern length, and badchar.  The function badCharHeuristic will set i =0 to help make a for loop.  The for loop will go through all 256 characters until i is 1 less than badchar.  Within this for loop it is taking badchar[i] and subtracting that value by 1.  Once this for loop ends another is created but in this for loop it will go until 1 less than the pattern length.  This for loop will fill the actual value of the last occurrence of a certain character.  After this is complete it will go back to searchBoyer and set int s equal to 0 and int count to 0.  Count is going to be used to see how many comparisons are made to get a match.  The next two lines of code are creating a timer and starting it to see how long the program will take to run.  A while loop is created and will stop when s is less than or equal to txtlen - patlen.  Each time the while loop goes to the top a int called small will be set equal to patlen -1.  After this another while loop will be created and won't stop until small is greater than 0 and Pattern[j] is equal to originaltxt[s+j].  In this while loop it will make j smaller by 1 each time it is looped through and the counter will go up by 1.  After the second while loop finishes a if statement saying if small is less than 0, s should add (s + patLen < txtlen)? m-badchar[originaltxt[s + patLen]] : 1.  This line of code will shift the pattern so that the following character matches up with the last place where that character is. (s+patLen<txtlen) is used to find the pattern if it is the last few characters in the string. If this condition isn't met an else statement will shift the pattern to the next matching character, following the bad characteristic rule. The first while loop will close once the pattern is found or the entire string has been looked through. The program will then output how many comparisons were made. The timer will then end and will output how long the program took to run.

**Comparing the two algorithms:**

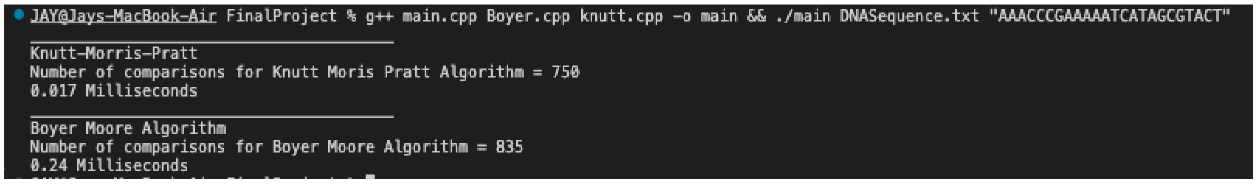
After comparing the two algorithms it is not so clear which algorithm is generally more efficient than the other. The performance will depend on the type of search we want to perform.

Time complexity refers to the amount of time taken by an algorithm to run, as a function of the length of the input. This means that time complexity is not the total time it takes a program to run because that can vary depending on many variables. The time complexity expresses the run time of an algorithm in terms of how quickly it grows relative to the input.

The KMP algorithm has a time complexity of O(m+n) which is a linear time complexity. In this scenario, m represents the size of the pattern and n is the length of the original text. The KMP algorithm requires a loop to run the length of the pattern(m) and also a loop to run the size of the original string(n). This is where the bigO notation O(m+n) comes from.

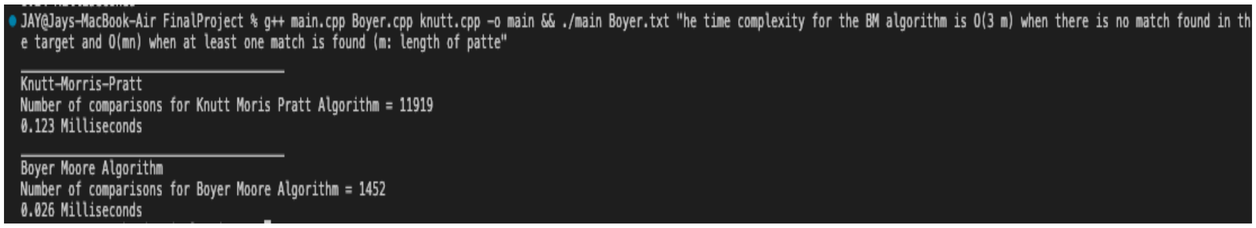
The BMH algorithm is a little different than KMP in the sense that the performance will depend drastically on the amount of character that can be skipped. In the worst-case the bigO notation of the BMH algorithm is O(mn), where m is the length of the pattern and n is the length of the original text.

BMH and KMP have their advantages and disadvantages and the right tool to use will depend on the kind of pattern you will search for. KMP works best when your alphabet is small. When searching a DNA sequence KMP would be the better choice as there is a higher probability that the pattern has reusable sub-patterns.

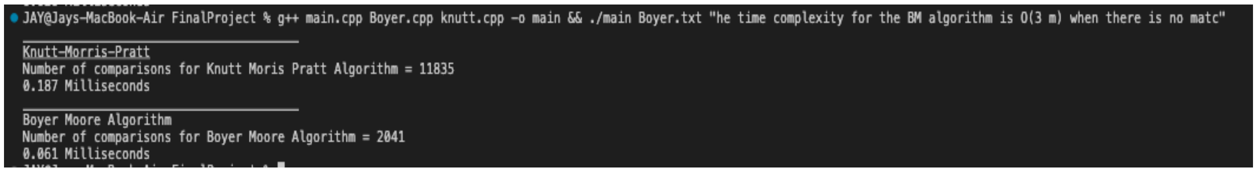


```
JAY@Jays-MacBook-Air FinalProject % g++ main.cpp Boyer.cpp knutt.cpp -o main && ./main DNASequence.txt "AAACCCGAAAAATCATAGCGTACT"
  ________________________________
  Knutt-Morris-Pratt
  Number of comparisons for Knutt Moris Pratt Algorithm = 750
  0.017 Milliseconds
  ________________________________
  Boyer Moore Algorithm
  Number of comparisons for Boyer Moore Algorithm = 835
  0.24 Milliseconds
```

In our first comparison we downloaded a txt file with a gene sequence to test our algorithms. The KMP algorithm is superior and is able to locate the pattern in a shorter elapsed time and with less comparisons made. In the figure above the DNA sequence search shows the elapsed time for KMP to be 0.017 milliseconds and 0.24 milliseconds for BMH.

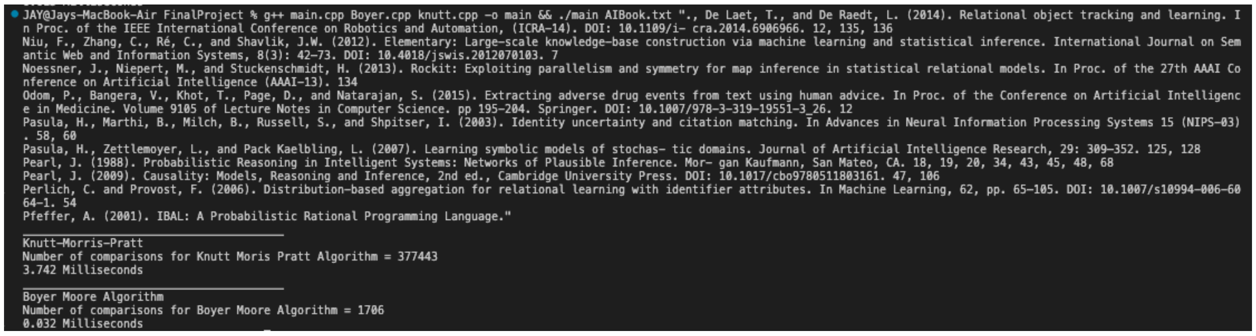


```
JAY@Jays-MacBook-Air FinalProject % g++ main.cpp Boyer.cpp knutt.cpp -o main && ./main Boyer.txt "he time complexity for the BM algorithm is O(3 m) when there is no match found in th
e target and O(mn) when at least one match is found (m: length of patte"
_______________________________
Knutt-Morris-Pratt
Number of comparisons for Knutt Moris Pratt Algorithm = 11919
0.123 Milliseconds
_______________________________
Boyer Moore Algorithm
Number of comparisons for Boyer Moore Algorithm = 1452
0.026 Milliseconds
```



```
JAY@Jays-MacBook-Air FinalProject % g++ main.cpp Boyer.cpp knutt.cpp -o main && ./main Boyer.txt "he time complexity for the BM algorithm is O(3 m) when there is no matc"
_______________________________
Knutt-Morris-Pratt
Number of comparisons for Knutt Moris Pratt Algorithm = 11835
0.187 Milliseconds
_______________________________
Boyer Moore Algorithm
Number of comparisons for Boyer Moore Algorithm = 2041
0.061 Milliseconds
```

One advantage of the BMH algorithm is that as the longer the pattern is, the more efficient the algorithm is because with each unsuccessful attempt to find a match the algorithm uses the bad match table to rule out positions where the pattern cannot match. In the figure above we downloaded a txt file and used 2 long patterns and a pattern that does not have a prefix that matches a postfix. The figure shows how BMH is much more efficient.



```
JAY@Jays-MacBook-Air FinalProject % g++ main.cpp Boyer.cpp knutt.cpp -o main && ./main AIBook.txt "., De Laet, T., and De Raedt, L. (2014). Relational object tracking and learning. I
n Proc. of the IEEE International Conference on Robotics and Automation, (ICRA-14). DOI: 10.1109/i- cra.2014.6906966. 12, 135, 136
Niu, F., Zhang, C., Ré, C., and Shavlik, J.W. (2012). Elementary: Large-scale knowledge-base construction via machine learning and statistical inference. International Journal on Sem
antic Web and Information Systems, 8(3): 42-73. DOI: 10.4018/jswis.2012070103. 7
Noessner, J., Niepert, M., and Stuckenschmidt, H. (2013). Rockit: Exploiting parallelism and symmetry for map inference in statistical relational models. In Proc. of the 27th AAAI Co
nference on Artificial Intelligence (AAAI-13). 134
Odom, P., Bangera, V., Khot, T., Page, D., and Natarajan, S. (2015). Extracting adverse drug events from text using human advice. In Proc. of the Conference on Artificial Intelligenc
e in Medicine. Volume 9105 of Lecture Notes in Computer Science. pp 195-204. Springer. DOI: 10.1007/978-3-319-19551-3_26. 12
Pasula, H., Marthi, B., Milch, B., Russell, S., and Shpitser, I. (2003). Identity uncertainty and citation matching. In Advances in Neural Information Processing Systems 15 (NIPS-03)
. 58, 60
Pasula, H., Zettlemoyer, L., and Pack Kaelbling, L. (2007). Learning symbolic models of stochas- tic domains. Journal of Artificial Intelligence Research, 29: 309-352. 125, 128
Pearl, J. (1988). Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Mor- gan Kaufmann, San Mateo, CA. 18, 19, 20, 34, 43, 45, 48, 68
Pearl, J. (2009). Causality: Models, Reasoning and Inference, 2nd ed., Cambridge University Press. DOI: 10.1017/cbo9780511803161. 47, 106
Perlich, C. and Provost, F. (2006). Distribution-based aggregation for relational learning with identifier attributes. In Machine Learning, 62, pp. 65-105. DOI: 10.1007/s10994-006-60
64-1. 54
Pfeffer, A. (2001). IBAL: A Probabilistic Rational Programming Language."
_______________________________
Knutt-Morris-Pratt
Number of comparisons for Knutt Moris Pratt Algorithm = 377443
3.742 Milliseconds
_______________________________
Boyer Moore Algorithm
Number of comparisons for Boyer Moore Algorithm = 1706
0.032 Milliseconds
```

In our final comparison we used a book sized txt file to check our comparison. In the figure above it is clear that with a long pattern or input the better choice of algorithm is Booyer-Moore. In the case you are working with text with small alphabets the better choice of algorithm is Knuth-Morris-Pratt.

## Contributions:

Ishan Chada:

- Slide Show
- Worked on Project Report
- Adapted code from both algorithms into one folder
- Debugging

Cameron Brickett:

- Boyer-Moore algorithm
- Slide Show
- Worked on project report
- Created GitHub repository

Jay Gomes:

- Knutt-Morris-Pratt
- Slide Show
- Worked on Project Report

## Bibliography

*Boyer Moore algorithm for Pattern Searching*. GeeksforGeeks. (2022, January 27). Retrieved April 23, 2022, from https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

Laud, Saharsh. “Knuth-Morris-Pratt Algorithm.” *CodesDope*, 24 Jan. 2021, https://www.codesdope.com/blog/article/kmp-algorithm/.

Encora. “Boyer-Moore-Horspool String Matching Algorithm.” *Encora*, Encora, 18 July 2022, https://www.encora.com/insights/the-boyer-moore-horspool-algorithm.

Yse, Diego Lopez. “Essential Programming: Time Complexity.” *Medium*, Towards Data Science, 22 Nov. 2020, https://towardsdatascience.com/essential-programming-time-complexity-a95bb2608cac.

Behera, Sweta. “Knuth-Morris-Pratt (KMP) vs Boyer Moore Pattern Searching Algorithm.” *OpenGenus IQ: Computing Expertise & Legacy*, OpenGenus IQ: Computing Expertise & Legacy, 24 Apr. 2020, https://iq.opengenus.org/kmp-vs-boyer-moore-algorithm/.

“Download Cottongen Marker Data.” *Cottongen*, https://www.cottongen.org/data/download/marker.