CE2001/CZ2001 ALGORITHMS

Project 1 - Searching Algorithm

Done by:

<Tutorial Group SE1 Group 6>

Woon Yoke Min

Tan Yap Siang

Tan Leng Hwee Gordon

Tey Yi Heng

Tan Jun Hong

Date:

14th September 2020

Design of Algorithm:

For the purpose of describing the algorithm design, we will refer to the pattern to be searched as "query" and the string to be searched from as "sequence".

Naive:

Naive algorithm will try to use a substring of sequence at index 0 and compare with the query, one by one. If it does not match, we go to index 1, check again and so on.

KMP Algorithm:

KMP algorithm is anchored in the concept of finding query strings that appear multiple times in the sequence string. The problem with the brute force(naive) algorithm is that it searches every element and undergoes repeated comparisons. KMP algorithm aims to avoid these comparisons and streamline the comparison process by recording indexes to skip the repeated comparisons. **The algorithm can be decomposed into two steps: 1. Preprocessing and 2. Searching.**

Preprocessing

Preprocessing step creates an array failureFunction[] with the size of query length, which is used to skip unnecessary comparisons in searching. To create the array, KMP implements the concept of finding substrings from the query that is the longest proper prefix which is also a suffix. A proper prefix cannot be the whole string.

Searching

Firstly, it will do comparisons for indexes that we have not processed before. We will implement the concept of finding substrings that are both a prefix and suffix. After which, we continually compare new indexes and identify substrings to match previous substrings. If we identify substrings that match, the indexes for the previously found substrings become the starting point of comparison for future comparisons. This helps to skip past repeated comparisons that have already been done before. The algorithm will start to store index values and recognise substrings that will lead to a match. The algorithm will process new indexes and if there are any recurring patterns proven to lead to a match, we will skip those indexes.

Optimized KMP Algorithm:

The optimized KMP algorithm is an improvement to the existing KMP algorithm. It retains the concept of shifting patterns but it also avoids the necessity of checking a mismatched character again.

Preprocessing

We will first obtain a list of unique characters that exist in the query. After which we will introduce a failure table that consists of a failure function that corresponds to every unique character. However, the failure function here differs from the one in KMP. The general idea is we perform an KMP search on the failure function for each character. Iterating through with index i, if the character matches, there's no need to double check that character again, so we use i + 1 and input it to the table, to compare the next character. If the character mismatches, it means that the longest suffix ends at the previous character. Similar to computing the KMP's failure function, we use the value of the failure function at that index (Let's call it k). We use k - 1 as an index to check the value in the failure table, and input it to the table.

Searching

Generally, the optimized KMP algorithm works the same way as the KMP algorithm when there is no mismatch. However, when a mismatch occurs, the algorithm considers the character being checked at the sequence. When the searched character from the sequence is not in the query, the optimized KMP algorithm will not conduct search for that character but instead skip past that character. When the searched character exists in the query, instead of having to go through multiple indexes, it will be directed immediately to the correct index based on the optimized failure function as mentioned above.

Analysis of Algorithm:

For analysis of both time and space complexity, we can split the analysis into two parts, the actual search and the preprocessing. Let length of query be M and length of sequence be N.

#1: Time complexity

Naive:

Best case of Naive algorithm is O(N), when the first character of the query does not match. The worst case, however, is O(N*M), when only the last character query does not match, so it will check through the query about N times to only find out the last character does not match.

KMP Algorithm:

The preprocessing includes the creation of the lps/failure function. The actual search uses the failure function to determine the next characters in the query to be checked.

```
# Pseudocode for KMP
i is for query index count
j is for sequence index count

search () {
    buildFailureFunction();                          ---- preprocessing step
    i = 0;
    for (j = 1 to length of sequence) {              ---- loops N times
        # If character matches, increment i and j
        if (char of query at i == char of sequence at j) {
            i++;
            # Check if no. of matching chars = pattern length
            if (i == length of query) {
                return position of occurrence;
            }
        }
        else {
            if (i > 0) {
                don't increment j;]*
            }
        }
    }
}
```

Let this if-else loop take constant B time

* no increment to j happens when chars don't match and not at start of query string

Let y be the no. of times this part of the if-else loop is run

```
matchLength is the value to be stored in failureFunction of an index

buildFailureFunction() {
    set first element of failure function as 0;      ---- let this be A0
    matchLength = 0;
    for (i = 1 : end of query) {                     ---- loops (M - 1) times
        if (char of suffix at i matches) {
            increment matchLength;
            update failureFunction[i] = matchLength;
        } else {
            if (matchLength == 0) {
                failureFunction[i] = 0;
            } else {
                matchLength = failureFunction[matchLength-1];
                don't increment i;]*
            }
        }
    }
}
```

Let this if-else loop take constant A_1 time, each time 1 comparison is made.

* happens when matchLength != 0 and char of suffix at i does not match

Let x be the no. of times this happens.

Best Case: Assuming *ed part of the if-else loop is NEVER ran.

For preprocessing step:

Total no. of operations = $A_0 + A_1 \times (M-1)$

$= A_0 + A_1(M-1)$

For searching step:

Total no. of operations = $B \times N$

$= BN$

$\therefore$ Total no. of operations = $A_0 + A_1(M-1) + BN$, where A_0, A_1, B are constants

Hence, time complexity for best case is $O(M+N)$

Worst Case: *ed parts of if-else loops are ran for the maximum no. of times.

$\Rightarrow$ where the query string is a long repetition of the same character (e.g. AAAA...AA - hundreds of A)

For preprocessing step:

Total no. of operations = $A_0 + A_1 \times (M-1+x)$

$= A_0 + A_1(M-1+x)$

For searching step:

Total no. of operations = $B(N+y)$

$\therefore$ Total no. of operations = $A_0 + A_1(M-1+x) + B(N+y)$

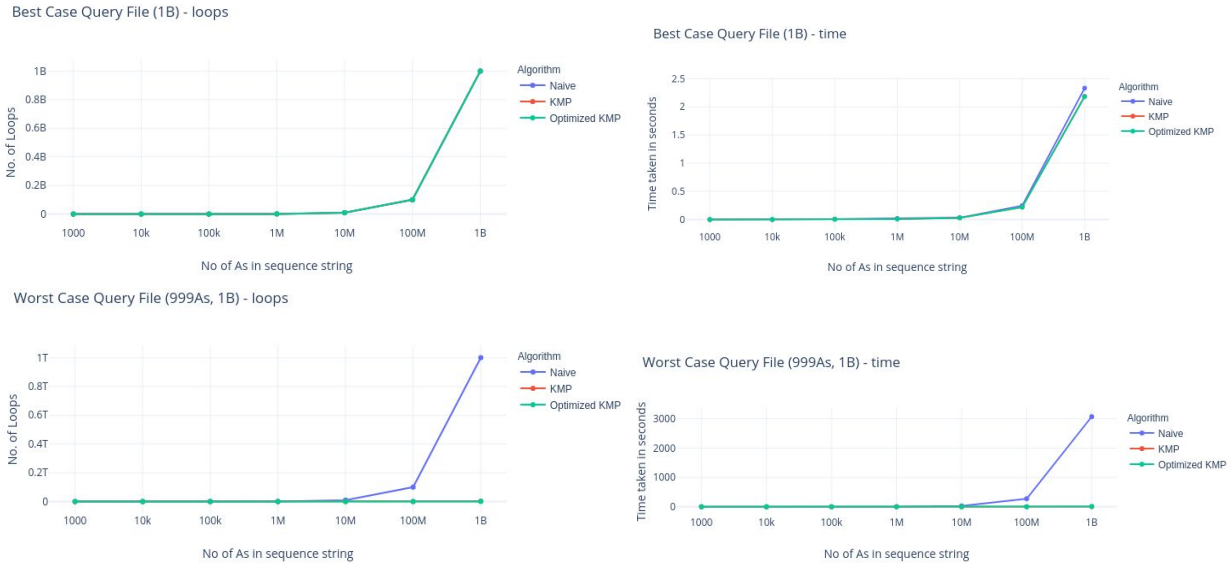
Hence, time complexity for worst case is $O(M+N)$.

Given that time complexity of worst case ≤ time complexity of average case ≤ time complexity of best case, and both the time complexities of best and worst case = O(M+N), we can conclude that average time complexity = O(M+N).

Why are x and y not considered in the computation of Big-O? When the special case of the if-else loop happens in either the search or preprocessing, an additional loop will be run. You will not increment j when there is a prefix/suffix in the array. However, the maximum number of times this can occur will not be greater than N. Similarly, the part of the loop where we do not increment i will not occur more than M times. Furthermore, these additional loops only contribute to the time complexity as an additive function. Hence, it will not have much impact on the computation of the Big-O time complexity.

Optimized KMP Algorithm:
The main difference between KMP and optimized KMP algorithms is in the preparation of the failure function. With a specialised failure function for each unique character in the query string, we can reduce the no. of comparisons in searching. However, such differences are minimal and barely affects time complexity. Hence, the time complexity for optimized KMP is also O(M+N).

Experiment: (Refer to the ExperimentalResults.pdf in zip file for table of full results)
Below are graphs which show the time taken / number of loops against different query strings.
Best Case Query String = “B”; Worst Case Query String = “A” * 999 + “B”
X-axis: Number of “A”s in sequence string; Y-axis: No. of loops / Time taken in seconds



The KMP and optimized KMP plots overlap each other as they have similar time complexity.

#2 Space Complexity

Naive:
Naive uses constant space as there are no pre-processed arrays or tables. As such, it is O(1).

KMP Algorithm:
For the preprocessing, a failure function is created and the size of the array is dependent on the length of the query, M.

For the actual search, if we store the indexes of occurrences in an array, then space complexity will be O(L), where L is the number of occurrences. However, we can "return" these indexes with a print function. Then, no extra memory space is used and the space complexity is O(M).

Optimized KMP Algorithm:
In preprocessing, optimized KMP creates a 2 dimensional array, FT[t][M], where t is a character. Hence, the space complexity for optimized KMP = M * no. of unique characters. If the characters are alphabets, it would be 26M. Otherwise, the worst case for space complexity would be $O(M^2)$, whereby every character in the query string is unique.

In the case of DNA sequences, there are at most 4 unique characters (G, A, C, T). Thus, the space complexity is 4M. Since 4 is a constant, it is considered as insignificant in the calculation of time complexity. Hence, big-O notation would also be O(M).

Implementation/Limitations

The implementation was done in Java. We removed some conditions to check, so the code for the KMP algorithm is slightly different to most that are found online, but it is slightly faster because of the reduced conditions to check.

We used Instant class to time the searching process, since it is thread-safe, meaning the timing would not change due to CPU usage. StringBuilder was used to append String as it significantly increased performance, and BufferedReader was also used as it significantly improved the reading of the file.

Due to the limitations of the String length of Java, the maximum sequence and query size we can use is 2,147,483,647, which is the 32-bit max signed integer value.

Heap size is also another issue. When reading in a file that is more than 1GB, most computers will throw this error: "java.lang.OutOfMemoryError: Java heap space". However, this could be resolved by giving Java more heap space, which means it is rather a limitation in the computer, rather than the code. 64-Bit Java may be required for large file sizes.

We could further optimize the code, such as rewriting it with more time-efficient data structures, however we have limited understanding of how it would affect the performance. We tried as much as possible to optimize it such that it will run as fast as possible, to our knowledge.

References

- KMP Algorithm for Pattern Searching. (2019, May 20). Retrieved from https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
- Online Graph Maker · Plotly Chart Studio. (n.d.). Retrieved from https://chart-studio.plotly.com/create/
- Real time optimized KMP Algorithm for Pattern Searching. (2020, May 23). Retrieved from https://www.geeksforgeeks.org/real-time-optimized-kmp-algorithm-for-pattern-searching/

Statement of Contribution:

All team members contributed equally.

- Implementation of Algorithm (done in Java): Jun Hong
- Analysis of Time and Space Complexity (Report): Yoke Min
- Design of KMP Algorithm (Report): Yap Siang
- Design of Optimized KMP Algorithm (Report): Gordon
- Implementation / Limitations (Report): Jun Hong
- Presentation slides: Everyone
- Presentation: Yi Heng