UMEÅ UNIVERSITY

Evaluating Performance of Pattern Searching Algorithms on Wildcard Patterns

Gustav Lindblad

Bachelor Thesis, 15 credits

Department of Computing Science

2023

Abstract

The pattern matching problem is the problem of finding a set of sequential characters in a text of equal amount of characters or more. There are many applications for pattern matching algorithms e.g. search engines and databases. Some algorithms allow for multiple patterns to be searched for at the same time. One such algorithm is the Aho-Corasick (AC) algorithm, it is an extension of the single-pattern matching algorithm called Knuth-Morris-Pratt (KMP). The aim of this thesis is to find out if multiple executions of KMP can match multi-patterns faster than AC in regards to wall-clock time. The results show that the time it takes for KMP to execute is barely affected by pattern length, while AC gets slower when the pattern length increases. KMP is faster than AC when the pattern length is long in relation to the texts length, and parallel executions of KMP is faster than sequential when there are more than 2 patterns.

Contents

1 Introduction

The pattern matching problem is the problem of finding a set of sequential characters in a text of equal amount of characters or more. Pattern matching have many useful applications e.g. search engines, databases and computational biology. Some of these systems may require for multiple patterns to be searched for at once, this is called multi-pattern matching. One case were multi-pattern matching is useful is when dealing with 'wildcards', characters in patterns which can be any character from the alphabet in use. Search engines use these so users can get suggestions while writing searches, another use could be in DNA sequencing were some character might be unknown.

It is important that these systems have efficient and fast algorithms for finding pattern matches. There are a lot of different algorithms that have been created for this purpose, this paper will study two of them. The algorithms studied in this paper are the Knuth-Morris-Pratt algorithm (KMP) and the Aho-Corasick algorithm (AC). KMP is an algorithm for single-pattern matching and AC is an extension of KMP which can be used for multi-pattern matching.

1.1 Thesis Aims

The aim of this thesis is to determine in which cases the KMP-algorithm might outperform the AC-algorithm on multi-patterns in regards to time of execution. Both of these algorithms will be implemented and tested to answer the question:

• Can multiple executions of the KMP-algorithm match multi-patterns in less time

with regards to wall-clock time than the AC-algorithm on genetic-data and binary texts?

1.2 Limitations

Genetic-data and binary text was chosen due to their small alphabet sizes, large alphabets make the amount of permutations of patterns with many wildcard symbols too large.

2 Theoretical Background

This section explains the theory needed to understand the results and discussion of this paper. Some terminology used in the paper will also be explained here. An *alphabet* is a set of characters which can be used to create texts. An alphabet of size 4 has 4 unique characters in it. A *text* is a string which will be searched in and a *pattern* is a smaller string which will be searched for in the text. *Pattern matching* is done by searching through the text and comparing the pattern to the current location in the text, if they are the same the pattern is matched.

2.1 Wildcards

Wildcards are a powerful tool in pattern matching, allowing matches where certain characters in the pattern may be unknown. One wildcard symbol is '*', which stands for zero or more characters. An example of how this symbol is used can be seen below in Figure 1 where the wildcard pattern could produce any of the example patterns.

abcd*efgh		
abcdefgh	**abcddiuhcj123voiefgh**	**abcd!efgh**

Figure 1: Different patterns constructed from the wildcard pattern

In addition to the '*' wildcard, there is also the wildcard symbol displayed as '?'. This symbol stands for a single character and is useful for matching patterns where only single characters may be unknown. An example of how this symbol is used can be seen below in Figure 2 where the wildcard pattern could produce any of the example patterns.

abcd?efgh		
abcd2efgh	**abcdaefgh**	**abcd!efgh**

Figure 2: Different patterns constructed from the wildcard pattern

Due to the nature of the '*' symbol it is impossible to produce all possible patterns since there are an infinite amount. The '?' symbol on the other hand is limited by the alphabet in use. If the alphabet in use is a set of 4 characters then a pattern with one '?' symbol will produce 4 patterns and a pattern with two '?' symbols will produce 16 patterns and so on. Therefore only the '?' symbol will be used in this study.

2.2 The Knuth-Morris-Pratt Algorithm

Knuth et al. (1977) created a pattern matching algorithm used for finding a single pattern in a text by making use of a prefix-table, also known as LPS. LPS stands for 'Longest proper Prefix which is also Suffix'. An example of how the prefix-table may look on the pattern "abcabacabc" can be seen below in Table 1.

Table 1: Visualization of the prefix-table for the pattern "abcabacabc"

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	a	c	a	b	c
skip	0	0	0	1	2	1	0	1	2	3

In this example the longest prefix which is also a suffix is 'abc', and the bottom row in the table shows is how many characters in the patterns can be skipped when trying to match a new pattern. In the Naive version of the pattern matching algorithm the pattern is matched from the start at every character in the text but by using a prefix-table some characters can be skipped since it is known what they contribute to the pattern. The longest prefix which also a suffix is "abc" here so if the whole pattern has been matched, then 'abc' has been read as the last characters in the pattern, now when trying to find a new pattern it will start matching at pattern index 3. Whenever a sub-string of the prefix is encountered in the text it will not need to reread it due to this table, this can save a lot of operations when comparing to the naive pattern-matching solution, especially when the prefix and pattern is long and many skips can be done.

To create the prefix-table two numbers are used, one for indexing the pattern and the other for keeping track of the longest prefix suffix value. A check is done to see if the pattern indexed by both of them match, if they do then both are incremented and the prefix suffix value is added to the table. If they do not match and the value is higher than 0 another check is done with the previous value, if the value is 0 then 0 is entered into the table and only the pattern index is incremented. Pseudo-code for the creation of the prefix-table can be seen below in Algorithm 1.

Algorithm 1: Implementation of prefix-table

```
CalculateLPS(pattern, pattern-length, lps)
 1 length ← 0  // Length of the previous longest prefix suffix
 2 lps[0] ← 0
 3 i ← 1
 4 while i < pattern-length do
 5     if pattern[i] = pattern[length] then
 6         length++
 7         lps[i] ← length
 8         i++
 9     else
10         if length ≠ 0 then
11             length ← lps[length - 1]
12         else
13             lps[i] ← 0
14             i++
```

The KMP-algorithm reads the text from left to right and when a mismatch occurs it shifts to the right and skips all positions where no matches can be made. It knows where no matches can be made by making use of the precomputed prefix-table. The algorithm uses two indexes when reading the text, one to go trough the text (i) and one to go through the pattern (j). When a mismatch occurs on $text[i] \neq pattern[j]$ a new pattern index is fetched from the prefix table by $j \leftarrow table[j-1]$ as long as j is not 0. The text index always increments by 1 each time and does not skip.

As explained by Knuth et al. (1977) the time complexity for the algorithm is $O(m+n)$ where m is the length of the pattern and n is the length of the text. It is $O(m)$ for creating the prefix-table and $O(n)$ for searching through the text.

2.2.1 Parallelization

The KMP-algorithm implemented will be tested in two different ways in this study. The first is running the algorithm in sequential order for each pattern, which means searching for one pattern at a time in this case. The other is running the algorithm in parallel for each pattern, this way multiple patterns can be searched for at the same time.

This can be accomplished by making use of threads. Threads are a sequence of instructions which can be processed by the CPU. They share memory with each other and many threads can exist in a process at once. They allow different parts of a program to execute at the same time. When deciding if it worth to create a multi-threaded version of a program it is important to look at how easy the work can be divided between the threads. Since each pattern is independent from the other patterns when running the KMP-algorithm it should be possible to achieve a linear speedup. This is when a program with p threads gets p times faster than the non-threaded version.

A simple solution would be to create one thread for each pattern but that might not turn out to be that effective. To find the optimal amount of threads for this program a test was conducted with several different pattern- and text sizes on different amount of threads, a graph of the results can be seen below in Figure 3.

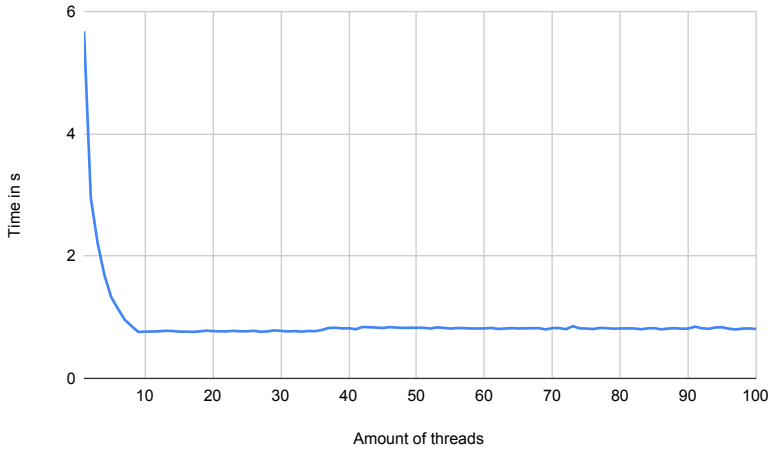


Figure 3: Running time for different amount of threads on a general test

The tests in this paper are all on a CPU with 8 cores. Concurrent execution of threads is limited by the amount of cores in the CPU of the machine. For example a CPU with 8 cores can only run 8 threads simultaneously. The tests results shown in Figure 3 show that 9 threads is optimal in regards to execution time. This is the amount of threads which will be used in all further tests of the threaded version of KMP.

All 9 threads are created for each run regardless of how many patterns there. The threads are created and sent to a 'thread pool', which is a loop where the threads can pick up work until there is no more work to be done. Each thread takes a pattern from a queue and starts searching for it in the text using the KMP-algorithm. If the queue is empty when a thread looks at it then that thread will end. This is to ensure there is no busy-waiting and uneeded usage of the CPU.

2.3 The Aho-Corasick Algorithm

Aho & Corasick (1975) created a pattern matching algorithm used for finding multi-patterns in a text by making use of an automaton. It works similarly to the KMP-algorithm in that it reads the text from left to right and shifts when a mismatch occurs. But the AC-algorithm uses an automaton instead of a prefix-table in order to make its shifts. To create the automaton a trie, also known as a prefix tree, is used. An example of how the automaton may look for the patterns 'abba, baab, aba' can be seen below in Figure 4.

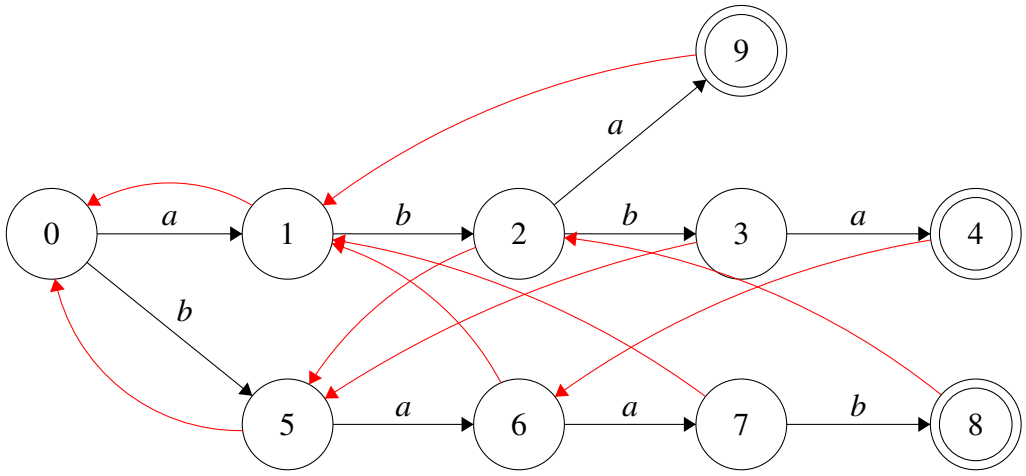


Figure 4: Visualization of an automaton for the patterns 'abba, baab, aba'

The black arrows in Figure 4 indicate which state the automaton changes to when that character is read, the red arrows indicate which state the automaton changes to when a mismatch occurs or if there are no further states on that branch. The red arrow points to the longest prefix that is also a suffix, like in the KMP-algorithm. When a state with rings in it has been reached a pattern is matched. When a character is read it is sent in to the automaton and it returns a new state, that new state can be used to see if a pattern has been found in the text.

To create the automaton three main functions are needed, 'Go To', 'Output' and 'Failure'. 'Go To' stores all states as indexes to the next state. It is created by simply looping through all characters of all patterns and adding them as states to a table. 'Output' stores indexes of all words that end at the current state. The last state in the loop from the 'Go To' function loop is added to a table and is used to check if a pattern is found. The 'Failure' function is used when a pattern is found or a mismatch occurs. The algorithm checks this function to see which state the automaton should change to. A queue is used to create the failure function, first all states which are of depth 1 are added to the queue. Then the deepest prefix suffix value from the root of the trie to the current state found in the queue is added to the failure function, then the state in the go to function is added to the queue. This keeps going until the queue is empty. Pseudo-code for the creation of the failure function can be found below in Algorithm 2

Algorithm 2: Implementation of failure function

```
1  failure-table[] ← -1
2  q ← create-queue
3  for every character in alphabet do
4      if trie node of depth 1 then
5          failure-table[node] ← 0
6          q.push(node)
7  while q is not empty do
8      state ← pop-queue
9      for every character in alphabet do
10         if go-to is defined for current character and state then
11             f ← failure-table[state]
12             while go-to is defined for current character and f do
13                 f ← failure-table[f]
14             f ← go-to[f][character]
15             failure-table[go-to[state][character]] ← f
16             q.push(g[state][character])
```

The time complexity for the algorithm is $O(n+m+o)$ where n is the length of the text, m is the length of the patterns and o is the number of occurrences of the patterns in the text. $O(m)$ for the precomputation of the automaton and $O(n+o)$ for searching through the text.

2.4 Related Works

Comparing different pattern matching algorithms with each other is not a new idea for a study, in fact a popular tool has been developed for just this purpose. Faro et al. (2016) presented an updated tool developed for comparing single pattern matching algorithms in their research paper. The open source software they

developed is called 'The String Matching Algorithm Research Tool' (or, for short, 'SMART'). It is a new version of a framework for testing string matching algorithms developed by Hume & Sunday (1991). SMART has over 120 different algorithms implemented and ready for testing. The tests settings are completely customizable with some examples of parameters being text-type and pattern length. The main draw of the software is that algorithms can be tested in the same environment and thus minimize discrepancies. SMART was not used in this study due to it only working with single-pattern matching algorithms.

Buluş et al. (2017) conducted a study to find out which algorithm is fastest at finding html-tags in web documents. It is important to have fast searches for content extraction from web documents, choosing a good algorithm is key for speedier searches. 31 different pattern matching algorithms were tested where one of them was the Knuth-Morris-Pratt algorithm. The experiments used data from 20 different online newspapers and tags of varying sizes was searched for in the documents. The preprocessing time and searching time were documented and compared in order to find the fastest algorithm. Their conclusion was that the Skip Search algorithm was the overall fastest. KMP had a faster preprocessing time than Skip Search but only a average search time making it just an average overall speed.

Automatons can get get quite cluttered in some cases, it would be nice to have a method of reducing the amount of transitions in them. Björklund et al. (2014) researched a method of reducing the amount of transitions by replacing some with catch-all failure transitions. An example of an algorithm which makes of failure transitions is the Aho-Corasick algorithm. In the case of the AC-algorithm the failure transitions are constructed by pointing to the longest prefix that has

been read. The transition-reduction problem is a NP-complete problem, but it was found that via approximation a solution that is within 2/3 of an optimal solution can be achieved in polynomial time.

3 Method

The Knuth-Morris-Pratt and the Aho-Corasick algorithms were built and tested in C. To measure the time each function-call takes to execute the 'clock_gettime' function from the time.h library was used. All the tests were executed 1000 times to reduce the volatility of the OS. Because the tests were run several times each the results are more accurate than if they were only run once. The data used for the tests is randomly generated text in 2 different sizes. The shortest text is 10000 characters long and is the length of an average message. The longest text is 100000 characters long to simulate genomes and longer messages. There are 2 different alphabets used in the tests, one with a size of 2 which represents binary data. The other alphabet is of size 4 and represents genome data. Patterns are taken from the text at random positions and modified with wildcard symbols at random positions in the pattern. A pattern size of 100-3000 which increments 100 each time was used for the tests. The text files were randomly filled with characters from their respective alphabets. A last test was done to see the difference between large and small pattern sizes on different amount of wildcards. Each test of the threaded version of KMP uses 9 threads regardless of the amount of patterns. All tests were run on a Windows machine, full specification can be seen below in Table 2.

Table 2: Specifications of the computer running the tests.

Operative System	Windows 10 Home V.21H2
CPU	Intel i7-11700F @ 2.50GHz
Memory	16GB
CPU-cores	8

4 Results

Figure 5, 6, 7, 8 and 9 show a comparison of how the two algorithms are affected by the parameters discussed in the method section. The tests these five figures show to results of all have identical parameters except the amount of wildcards in the pattern. Each figure shows an unique amount of wildcards. Each Figure is divided into 4 graphs, for simplicity the first Figure will be explained. The explanation is the same for the other Figures mentioned. In (a) binary text of 10000 characters is displayed. In (b) genetic data in the form of a 10000 character long text is displayed. In (c) binary text of 100000 characters is displayed. In (d) genetic data of in the form of a 100000 character long text is displayed.

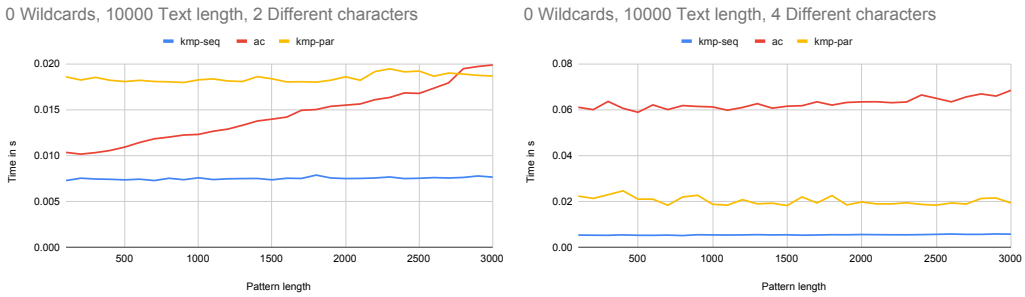
Figure 5 shows that sequential execution of the KMP-algorithm is always faster than the other alternatives when there are no wildcards. Parallel KMP is faster than AC on the larger alphabet.

Figure 6 shows in (c) and (d) that sequential execution of KMP is slower than the other alternatives on the large text file, parallel KMP is faster than AC on the larger alphabet but has the same run-time on the smaller alphabet. In (a) AC is faster than the sequential KMP in the beginning but quickly becomes slower than it. Parallel KMP seems strictly slower than the sequential version here. In (b) AC is slower than the two KMP versions which seems to have the same run-time.

In Figure 7 the sequential KMP version is always slower than the parallel version. In (a) and (b) parallel KMP and AC are have similar run-time on smaller patterns but AC becomes slower on larger patterns while the parallel KMP version keeps the same run-time. In (c) and (d) the results are similar to the previous but AC slows down less and its run-time seems closer to the parallel KMP version even on larger patterns.

Figure 8 and Figure 9 continue the trends shown in the previous graphs. The sequential KMP version is always slower than the parallel version. AC starts out faster than both versions in (a) and (b) but becomes slower than both when there running on larger patterns. In (c) and (d) AC and the parallel version seems to have very similar run-times. AC is slightly better on smaller patterns while the parallel KMP version is slightly better on large patterns.

Figure 10 shows a comparison of the algorithms on the amount of wildcards present in the pattern. Both (a) and (b) have 100000 character long texts and use the binary alphabet. (a) is on patterns that are 3000 characters long and shows that the sequential KMP version is fastest on 0 wildcards but becomes the slowest one all other amounts. AC and the parallel version are have similar run-time but AC is slightly slower. (b) is on patterns that are 30000 characters long and shows that AC is always the slowest and the sequential KMP version is only faster than the parallel version when there are no wildcards.



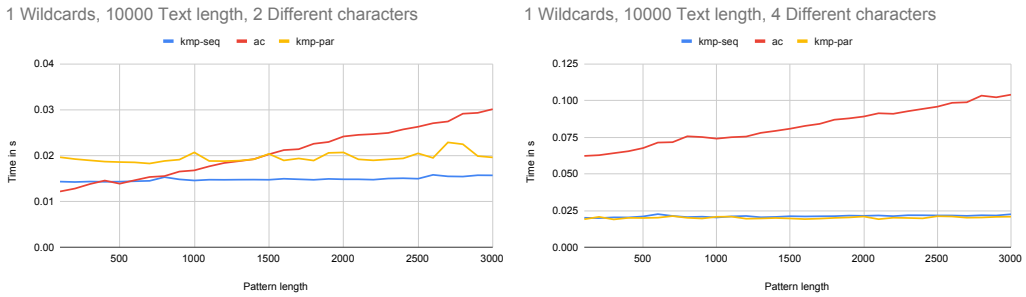
(a) 10000/2

(b) 10000/4

(c) 100000/2

(d) 100000/4

Figure 5: Graphs of KMP and AC-algorithms execution time when there are 0 wildcards



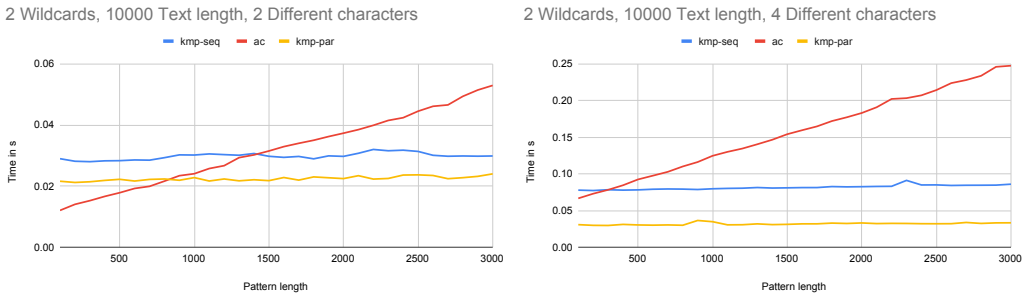
(a) 10000/2

(b) 10000/4

(c) 100000/2

(d) 100000/4

Figure 6: Graphs of KMP and AC-algorithms execution time when there is 1 wildcard



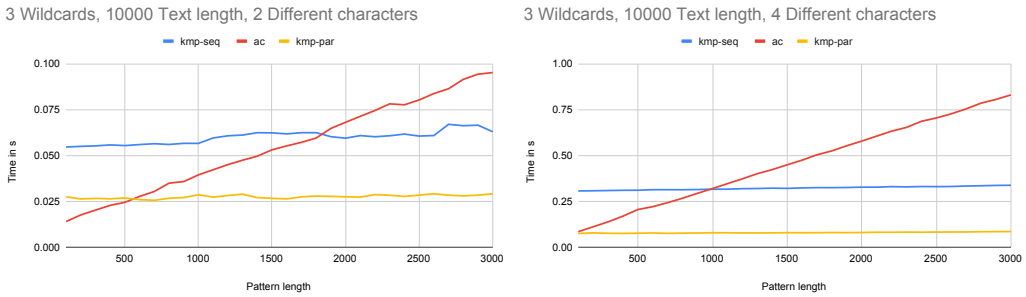
(a) 10000/2

(b) 10000/4

(c) 100000/2

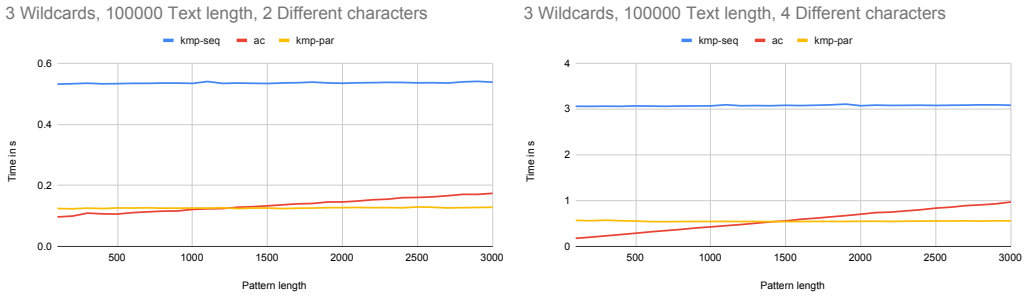
(d) 100000/4

Figure 7: Graphs of KMP and AC-algorithms execution time when there are 2 wildcards



(a) 10000/2

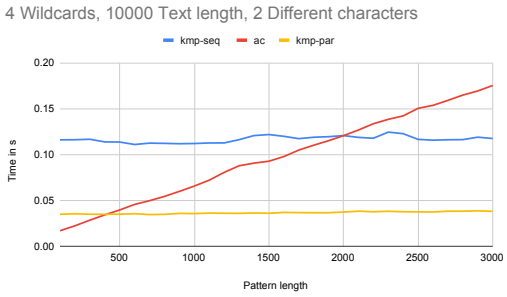
(b) 10000/4



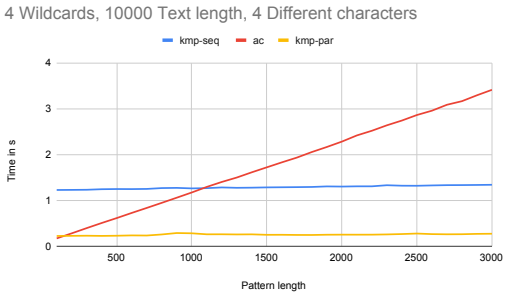
(c) 100000/2

(d) 100000/4

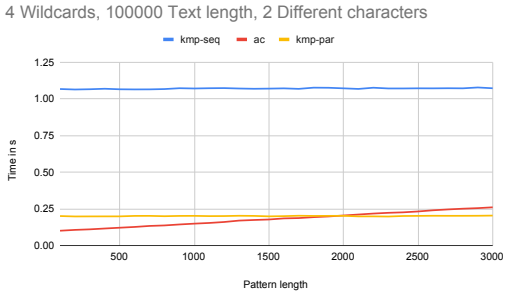
Figure 8: Graphs of KMP and AC-algorithms execution time when there are 3 wildcards



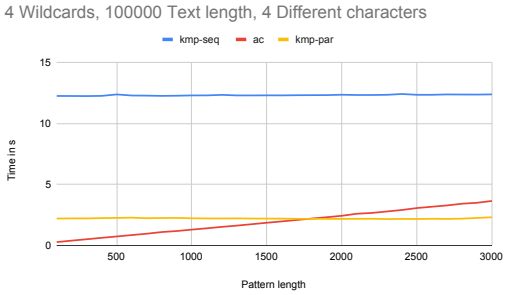
(a) 10000/2



(b) 10000/4

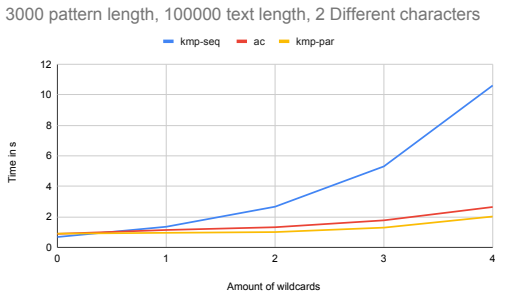


(c) 100000/2

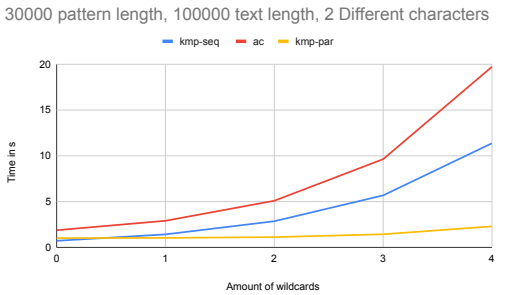


(d) 100000/4

Figure 9: Graphs of KMP and AC-algorithms execution time when there are 4 wildcards



(a) 3000 pattern length



(b) 30000 pattern length

Figure 10: Graphs of KMP and AC-algorithms execution time on different amount of wildcards

5 Discussion

When comparing Figure 5(c) and (d) to any of the other graphs on the large text it is clear that patterns with length 3000 and under makes the sequential KMP slower than AC and the parallel version when there is more than 1 pattern. This is most likely due to the fact that the sequential KMP version need to read through the whole text up to 256 times while AC only needs to do it once, the parallel version is also at an advantage since it can read 8 patterns at once. In Figure 10 a comparison between a pattern of size 3000 and 30000 can be seen. The graphs indicate that if the pattern is longer then KMP will perform better. This behavior can also be seen in others graphs such as in Figure 8(b), where the longer patterns increases the execution time for AC while KMP is almost a constant. Both algorithms are affected by pattern length but it seems the processing time increase for the prefix-table in the KMP-algorithm is negligible. The AC-algorithm on the other hand increases its computation time noticeably alongside the pattern length. This is most likely due to how the automaton is built, it is built with 3 arrays and a queue. All of these structures also grow with the pattern size, so the preprocessing will be slower. Accessing states when mismatches occurs is also harder when there are longer patterns due to the failure function also growing in size.

The sequential KMP version is faster than the parallel version when there is only 1 pattern that need to be searched for. This is due to the fact that it takes time to create the threads in the parallel version, since they are always created even if they are not needed. when there are two patterns the sequential version is only better when the text is small. It can probably read the text fast enough for a multi-threaded version not to be more effective. On the rest of the tests the parallel version was better. This is not surprising since being able to search for multiple patterns simultaneously should be faster than searching for them in order. The

overhead of creating threads seems to not be an issue when looking for more than 2 patterns.

In conclusion sequential executions of KMP is faster than AC when the pattern length is long in relation to the texts overall length. Parallel executions of KMP is faster than sequential executions when there are more than 2 patterns. AC was faster than the parallel version when there were smaller patterns, but as the patterns grew AC got slower than the parallel version. The KMP-algorithm could match multiple large patterns in less time when compared to the AC-algorithm on Genetic data and binary texts.

6 Future Work

Future work on this study could entail looking into the placement of the wildcards on the patterns. In this study the wildcard placement was random, it could be interesting to see if there is a difference when the wildcard symbols are in the front of the pattern or the back for example. This could perhaps impact the performance of the AC automaton.

Bibliography

Aho, A. V. & Corasick, M. J. (1975), 'Efficient string matching: an aid to bibliographic search', http://doi.acm.org/10.1145/360825.360855.

Björklund, H., Björklund, J. & Zechner, N. (2014), 'Compression of finite-state automata through failure transitions', *Theoretical Computer Science* **557**, 87–100.

Buluş, H. N., Uzun, E. & Doruk, A. (2017), Comparison of string matching algorithms in web documents, *in* 'Int. Sci. Conf', Vol. 2, pp. 279–282.

Faro, S., Lecroq, T., Borzi, S., Di Mauro, S. & Maggio, A. (2016), The string matching algorithms research tool., *in* 'Stringology', pp. 99–111.

Hume, A. & Sunday, D. (1991), 'Fast string searching', *Software: Practice and Experience* **21**(11), 1221–1248.

Knuth, D. E., Morris, J. H. & Pratt, V. R. (1977), 'Fast pattern matching in strings', https://www.cs.jhu.edu/~misha/ReadingSeminar/Papers/Knuth77.pdf.