**Searching for patterns: a comparison between several pattern matching algorithms**

Siarhei Miachkou, Andrii Venher

April 17, 2023

1. **Introduction**

In this paper we approach the problem of exact pattern matching: that is finding all occurrences of the given pattern in a particular text. The final goal of this research is to find the best pattern matching algorithm by comparing the most popular options available.

We will investigate five of the well-known pattern matching algorithms: Sunday, FSM, KMP, Rabin-Karp, Gusfield Z and compare them to the Brute force method. In addition, we will also implement wildcard supportive versions of the Brute force method and the Sunday algorithm. As well as a modification of the Rabin-Karp that operates on two-dimensional (matrix-like) text to make our research even more exciting.

**1.1 Brute force**

As always, the simplest solution to any problem is the intuitive one, a so-called Brute force approach. Our case is no exception. The easiest way to find all occurrences of the pattern is to check every single character as many times as we might need. Thus, we go into our text and starting with every character we compare the ones that follow to our pattern, once there’s a mismatch, we stop this process, but if we got to the end of the pattern, then it means that we have found an occurrence of the pattern, now we go to the next character in the text to repeat the process.

This solution is extremely simple to implement, however it is very slow. While the best case (not a single character matches) is O(n), the worst case (every character of the text matches every character of the pattern) is O(mn) which can get to large numbers really fast. Average complexity: O(mn).

**1.2 Sunday**

Sunday is conceptually the closest algorithm to the Brute force implementation, however it manages to skip unnecessary parts of the text which allows it to be faster in most cases. But, for it to understand when to skip which characters we need to prepare some data in advance. We construct an array the size of the encoding system we use (in our case it’s ASCII) and go through the pattern to put the indexes of the last occurrence of each character in this array at the position of its code, this has O(m) complexity.

Now, we iterate through our text but not in a simple way. We start at the index one below the pattern size and go backwards trying to match the pattern. If we encounter a mismatch, we move forward, but not always by one character. We check the value of the failed character in our array and if it has no registered position there (we used –1) we jump over it, but if it has a record, we move our comparison window to match its occurrence in the pattern. The same way we act after confirming a match. This intricate technique allows us to go through the text faster.

Unfortunately, this does not help us with our worst case (all character of the text and of the pattern are the same), it is still O(mn), but the best case (none of the characters match) is now O(n/m) since we are able to skip so many of them. Do not forget that we also have a O(m) preprocessing stage. Average complexity: O(n).

**1.3 Rabin-Karp**

At this point it is clear that comparing strings is extremely time-consuming. But is there anything we can do to bypass this inconvenience? Actually, yes, we can convert our pattern to a number via a hash function typically used for encoding. This idea is the trick behind making Rabin-Karp algorithm fast.

Let us first prepare some extra values that we will need. We want the size of the encoding system N (again 256 in our case) to be our base and then we ought to calculate the highest power of it that we will have to use (N ^ (m – 1)). Now we can create two unsigned integers (unsigned to avoid negative numbers in case of the overflow) to represent our pattern’s and text’s hash values. First, we compute the hash value for the pattern, we start from the back, and multiply the code of our index by the base of our system raised to the power of its index (starting with 0) just like in any other numerical system. We are not worried about overflows yet, we will deal with them later. Then we take the part of our text of size m and hash it the same way. But we cannot do it every time we move forward, it takes O(m) to do this which will result in O(mn) in the end. Thus, we do not recompute our text hash value from scratch, we take the one we had before, subtract the code of the first character multiplied by the highest possible coefficient and simply add the code of the next character making the whole process much faster.

Now we can go through our text rather fast and compare the two hashes we have. However, equality of the hash values is not enough to guarantee the match due to the controlled overflow (or modulo division in some implementations) producing similar values for different patterns in rare cases. So, we brute force our way through each text part with a hash value matching our pattern’s one to make sure we make no mistakes. Unfortunately, this means that we came back to our worst case of O(mn) when all the characters are the same, but the best and average cases are just O(n+m) now, which is very close to being linear. Average complexity: O(n+m).

**1.4 Finite State Machine**

Just like Sunday, Finite State Machine algorithm requires some data to be prepared in advance, however its approach to the problem of pattern matching is completely different. This time we are going to work with states, every time we read a character, our algorithm gets into a new state based on the previous state and the character received. There is one state (equal to the size of the pattern) that indicates the match.

So, the transition function table is a two-dimensional array where for each of the m+1 (0, 1, 2, …, m) states we define a next state to enter when any of the characters is read. Since we used ASCII encoding system, we had m+1 rows and 256 columns in our transition function table. Now we need to fill the table. Each state represents the number of characters successfully matched prior to the current step. Thus, at each stage we increment the state by 1 for the next character of the pattern. But for the mismatches we need to calculate our new state, because the pattern may have been overlapped and we cannot just drop the state back to 0. Starting with the largest possible value we are looking for a prefix of our pattern which is also a suffix by decrementing the state value, trying to find our character at the position of the state and then checking if the prefix of it matches the end of the pattern. Eventually, the number at which we stop is our new state (can be 0 if no such prefix has been found).

Luckily, now when we have the transition function table, looking for pattern is extremely easy and amazingly fast. We just go through our text, updating our state every step and when it hits m+1 it is a match. This in fact has O(n) complexity and no worst or best cases. But our preprocessing is devastatingly slow. It has a complexity of O(m^3 \* N) in the worst case (has to try all possible prefixes every time) and O(m^2\*N) in the best case (matches all the prefixes right away), here N is the size of the encoding system used. However, for the small patterns and alphabets this will not be so much time, and the search itself is still O(n). This makes FSM algorithm appropriate in some scenarios, but I would not recommend using it as a regular solution. Average complexity: O(m^2\*N+n), but can be shortened to O(m^2)

**1.5 Knuth–Morris–Pratt**

This algorithm also preprocesses the pattern to create an auxiliary array, called LPS, which is used to avoid unnecessary comparisons of characters. The LPS array is precomputed in linear time O(m).

LPS array is defined as follows: for each position i in the pattern, it stores the length of the longest proper suffix of the substring pattern[0:i] that is also a prefix of pattern[0:i]. A proper suffix of a string is a suffix that is not equal to the string itself, and a prefix is a substring that starts at the beginning of the string.

The KMP algorithm uses the LPS array to determine the next position to start the comparison from in the pattern. Specifically, if a mismatch occurs at position i in the pattern while comparing with a character j in the text string, the algorithm moves to the next position in the pattern using the value of LPS[i-1], which is the length of the longest proper suffix of the substring pattern[0:i-1] that is also a prefix of pattern[0:i-1]. This ensures that the algorithm never backtracks in the text string, and therefore, the time complexity of the algorithm is linear in the length of the text string.

**1.6 Gusfield Z**

Not surprisingly, the Gusfield Z algorithm also makes use of pattern preprocessing. The algorithm works by computing the Z-values of a string, which are the lengths of the longest substrings starting from each position that matches the prefix of the string.

To compute the Z-values, the algorithm maintains a "window" of the rightmost substring that matches the prefix and computes the Z-values iteratively by comparing characters in the window to characters in the rest of the string.

When searching for a pattern, the algorithm concatenates the pattern and the text with a special separator character not present in either the pattern or the text and computes the Z-values for the resulting string. It then scans through the Z-values to find any positions where the Z-value equals the length of the pattern. These positions correspond to matches of the pattern in the text.

The overall time complexity is O(n + m).

1. **Edge cases**

As we mentioned earlier the same algorithms do not always perform the same way, there are best and worst cases. This is an important attribute to take into consideration. If you happen to choose a pattern matching algorithm for a nonstandard task you should look deeper than the average case complexities and investigate which algorithm suits your situation the best. Here are some examples of this.

**2.1 Binary Sunday vs Gusfield Z**

Binary Sunday (a simplified Sunday algorithm that only checks if a character is in the pattern or not, instead of getting its index) is a pretty simple algorithm, especially when compared to Gusfield Z. But, what if not a single character is shared between the text and the pattern? Well, Binary Sunday will fly through the text not spending any time trying to match the pattern and completely skipping a huge part of the text. It will complete the task in just O(n/m) time. While Gusfield Z is not able take advantage of the situation like this and will still require the standard O(n+m) time (Figure 1).

**2.2 KMP vs Rabin-Karp**

Rabin-Karp seems like a well-constructed algorithm, which allows to speed up the procedure of pattern matching while remaining fairly easy to understand and implement compared to such complex algorithms as KMP. However, we are not able to produce an infinite number of unique hash values due to the technical limitations. This creates a possibility for a false match, thus we are obliged to double check every match and it takes O(m). What if all the characters are the same and every step, we take is another match? The Rabin-Karp is no better than a Brute Force then, in fact due to the time required for preprocessing it might even be worse, it will take O(nm) to get through a situation like this. KMP, being a more reliable algorithm, does not have this issue and can easily deal with such extreme cases keeping it linear (Figure 2).

**2.3 Rabin-Karp vs Sunday**

This one might not seem obvious as we have determined that both Rabin-Karp and Sunday have the same worst/beast case scenarios, the less matches – the better. However, there is an interesting detail to the worst-case scenario. Changing the last character of the pattern helps Rabin-Karp to completely overcome the problem of constant matching as the hash values are always different now. But it does not help Sunday this much, it still needs to go extra up until the final character and cannot jump far ahead afterwards. This shows how it is important not only to know the worst cases but also to understand why they are the worst ones (Figure 3).

1. **Wildcards**

Now, when we have described all the typical algorithms used for pattern matching, we can start experimenting with them. First, let us imagine the situation in which you do not know the pattern you are looking for exactly or maybe it has a special symbol you cannot type. The solution is wildcards. So, we modified our Brute Force and Sunday algorithms to be able to recognize and apply ‘?’ and ‘\*’ not forgetting to include the backslash escape mechanic.

The main challenge when implementing a wildcard pattern matching algorithm is undoubtedly a star (‘\*’) pattern which matches any number of any character and brings plenty of rules to consider when implementing an algorithm.

Let me say a few words about the general approach we developed for wildcard pattern matching. The idea was to split the pattern into parts which do not contain a star wildcard and then subsequently match corresponding text fragments with these pattern parts using a slightly modified version of a corresponding pattern matching algorithm.

Therefore, both Brute force and Sunday algorithms share the same “wiring”, i.e., pattern validation and preprocessing, between-star-fragment matching and cursor movement. Then, this template method uses a pattern-specific version to match a fragment without a star wildcard.

**3.1 Wildcard Brute Force**

Wildcard Brute force acts pretty much alike as the normal version so we would not repeat the same description one more time. The only difference is that it matches ‘?’ wildcard with any character in the text and supports escape sequences (‘\\*’, ‘\?’ and ‘\\’).

**3.2 Wildcard Sunday**

Wildcard Sunday is again very similar to a non-wildcard version of the algorithm. However, Wildcard Sunday is reversed so it traverses the pattern from left to right compared to a classic version which goes from right to left. It is needed because we introduced a star wildcard which does not allow us to travers as usual. The key idea though is to fill the positions of characters that do not exist in the pattern with the last position of ‘?’ in the pattern if it exists because it matches any character and therefore can be used in the role of it. Of course, it also supports escape sequences (‘\\*’, ‘\?’ and ‘\\’).

1. **2D Rabin-Karp**

Up until this point we have been interpreting the text as a string of characters. A one-dimensional sequence. But what if we want to look at it from a different perspective. For instance, we want to see our text as a two-dimensional structure, much like a matrix or a table, where there are rows and columns of characters rather than a line of them. How do we do pattern matching in this case? Let us imagine we want to find out if the top left K by K square of symbols repeats anywhere else in the text. Brute Forcing it will take ages but modifying any of our algorithms to work with such a structure is not an easy task. The best fit would be the Rabin-Karp algorithm and here is how we have done it.

First, we need to calculate the hash value for our pattern. There are several ways to do this in the 2D space, but we have come up with the following approach: compute the hash values of each column of the K by K square, interpreting the columns as simple strings, then compute the hash of the whole square treating hash values of the columns as our characters’ codes. Unfortunately, there is no way to apply rolling to both rows and columns at the same time, so by going with columns’ hashes we must encode the entire first row column by column, saving their hash values in an array.

But when we have those computed, we can get going faster. After checking the first row, we go down and instead of computing the next one from scratch we just recompute the column values, the same way we did with simple strings: subtracting the first code multiplied by max coefficient and adding the code of next character. And as we now have all the columns’ hash values already computed we can apply the same idea going through our row. Now, we can top down until we reach the end comparing all the hash values, we compute to our pattern’s one.

When there is a matching hash value, we must make sure that it is indeed a match by comparing all the characters directly using Brute Force method, just as we did with usual Rabin-Karp. So again, the worst case is having a match at every step. But when we are lucky not to have so many matches, we end up with a nice linear time algorithm (Figure 4).

1. **Methodology**

All algorithms were implemented in C++. They were tested on texts of size 1000, 2000, 3000, …, 200000 characters. The small pattern was 10 characters long and the large pattern was 50 characters long. The text and the patterns were designed to have both matches and mismatches quite often. Each text size was tested 100 times. The result at each size is the average time it took the given algorithm to find all the matches in the text.

1. **Results**

Concerning the small pattern of size 10 characters: the difference between all the algorithms is not huge, even the Brute Force’s O(nm) is not that bad when the pattern is so small relative to the text. However, we can still clearly see that KMP and Rabin-Karp are two of the fastest algorithms (Figure 5). When we go the large pattern though, things change drastically, now there is a huge gap between the Brute Force and the others (Figure 6), so we will take it away to see the gaps better. The KMP algorithm has come closer, but Rabin-Karp remains the fastest algorithm. Also, we can observe how devastating the larger pattern size is for FSM algorithm whose preprocessing time is O(m^2) at best (Figure 7).

1. **Conclusions**

Taking everything into consideration, I would say that the best pattern matching algorithms are Sunday, Rabin-Karp and KMP. Sunday is simple to implement and extremely fast when the matches are unlikely, Rabin-Karp is still not too complex and is less sensitive to many incomplete pattern matches and KMP is a complex but very effective algorithm with no worst cases and reliable efficiency.

The choice of a particular algorithm depends on factors such as the size of the input, the characteristics of the pattern and text, and the desired trade-offs between time complexity, space complexity, and simplicity. In practice, it is common to use multiple algorithms and choose the one that performs best for a particular problem.

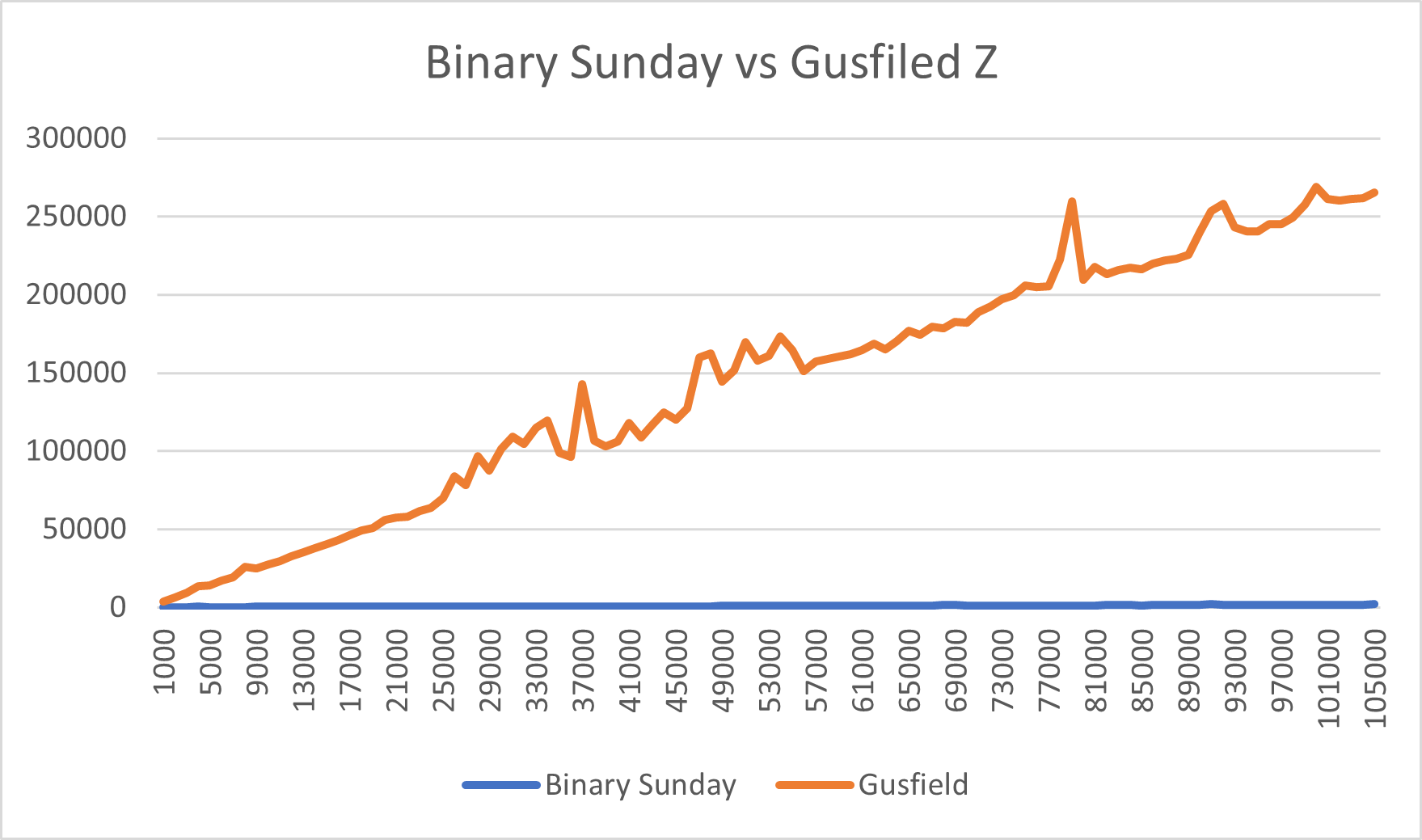


Figure 1: Binary Sunday vs Gusfield Z

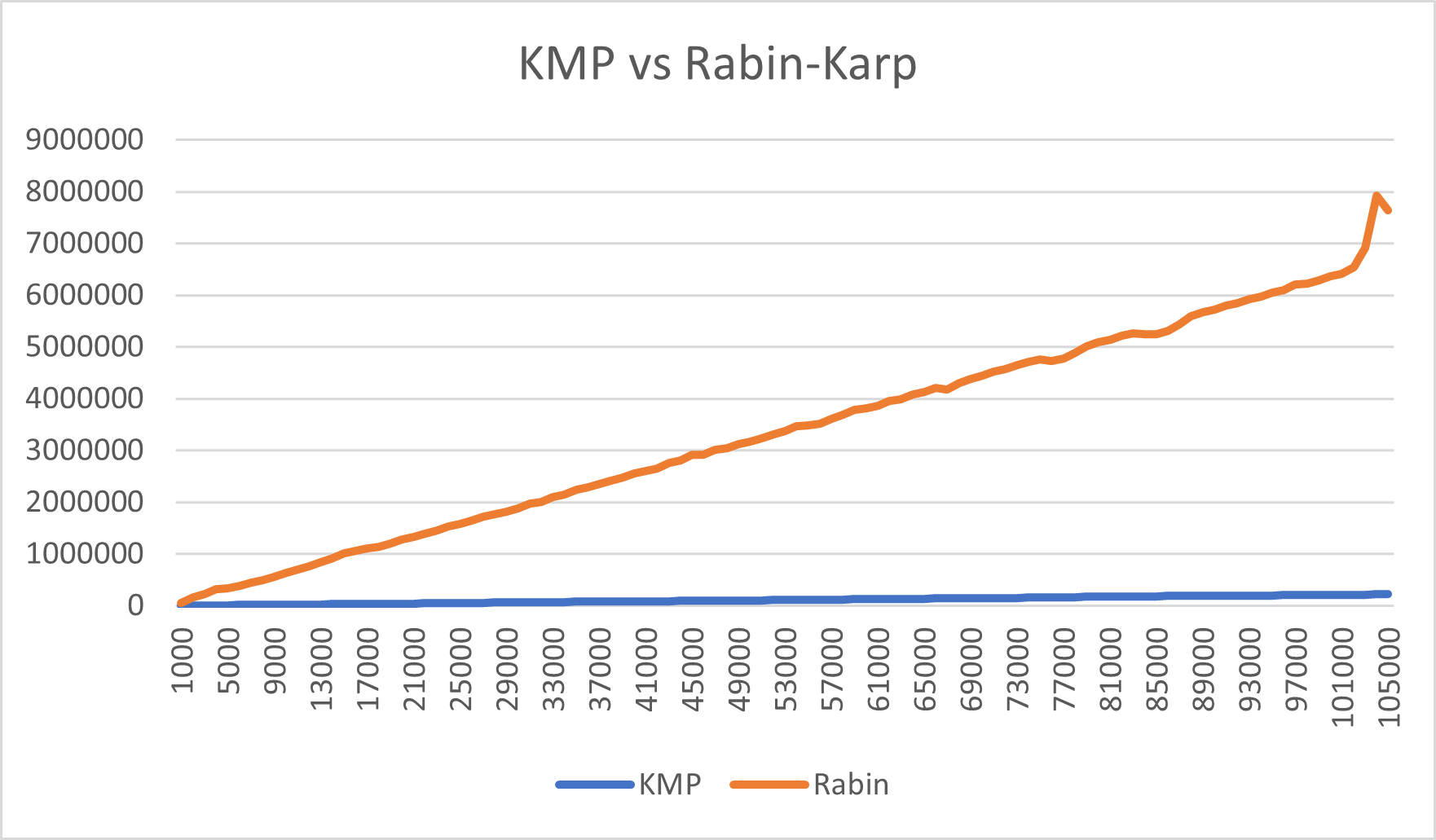


Figure 2: KMP vs Rabin-Karp

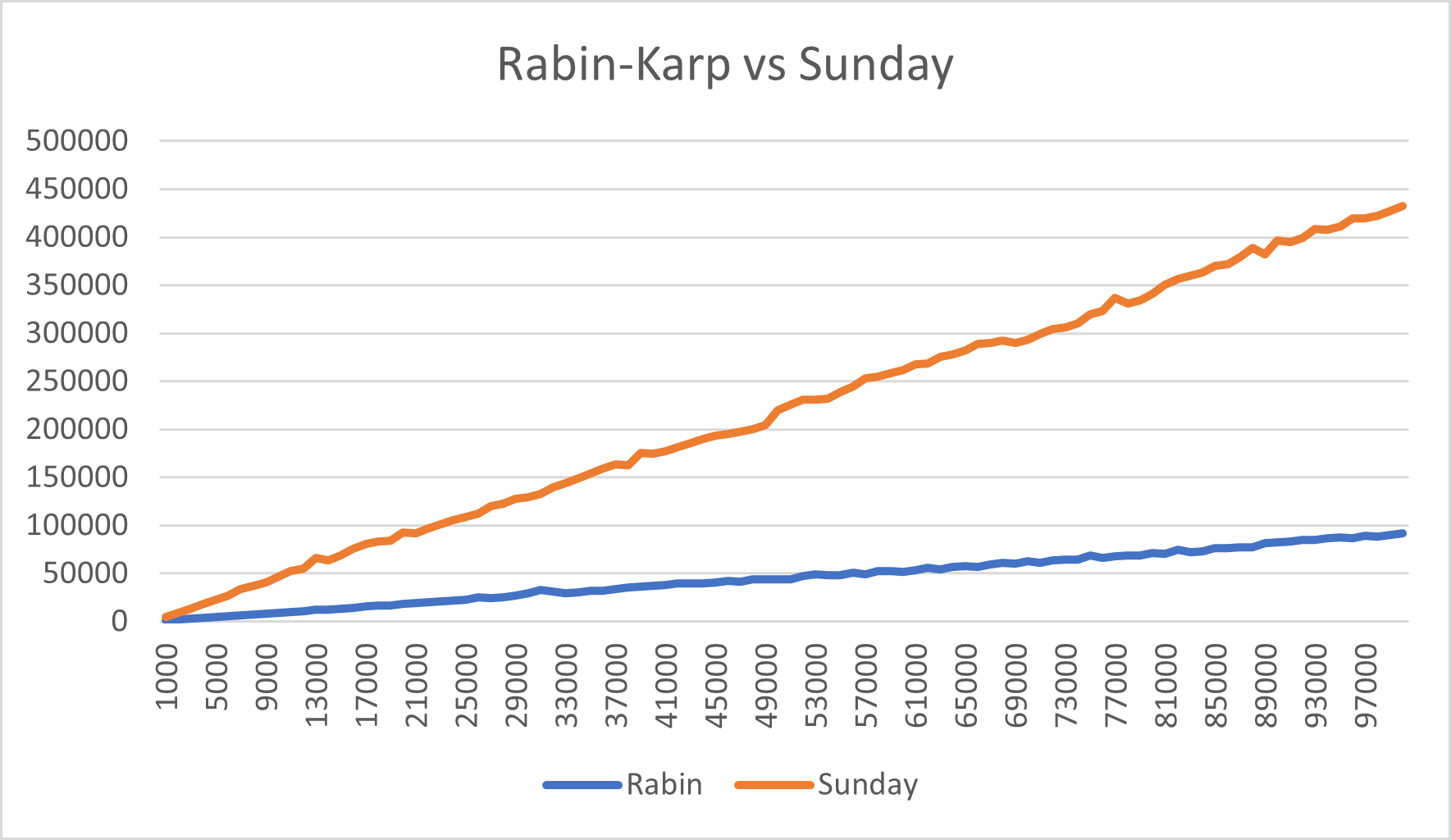


Figure 3: Rabin-Karp vs Sunday

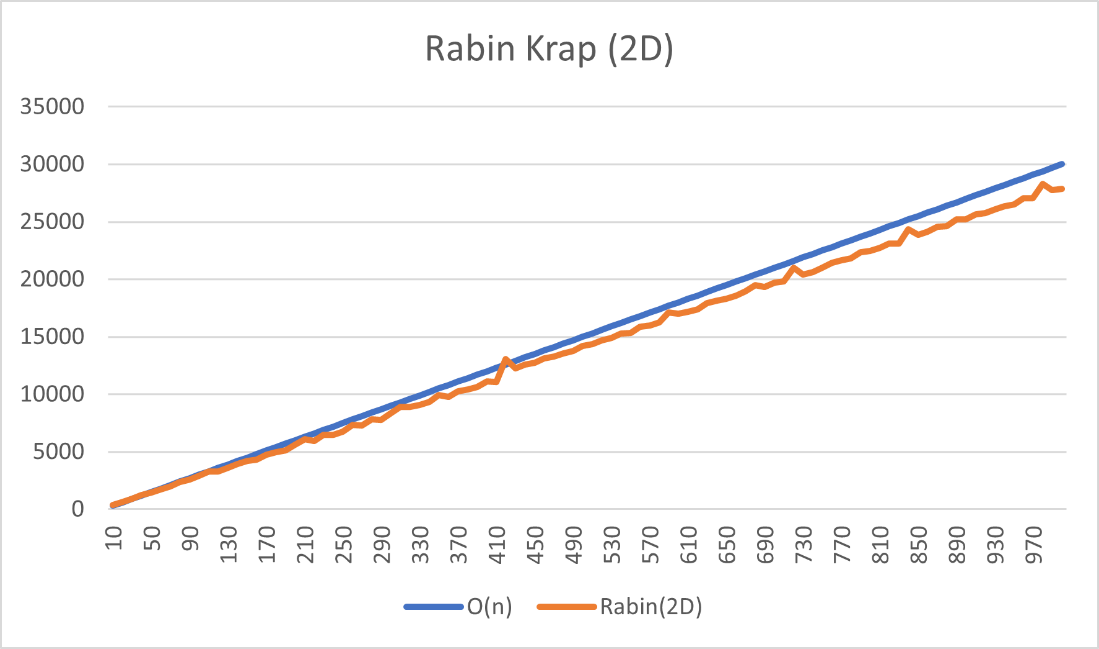


Figure 4: 2D Rabin-Karp

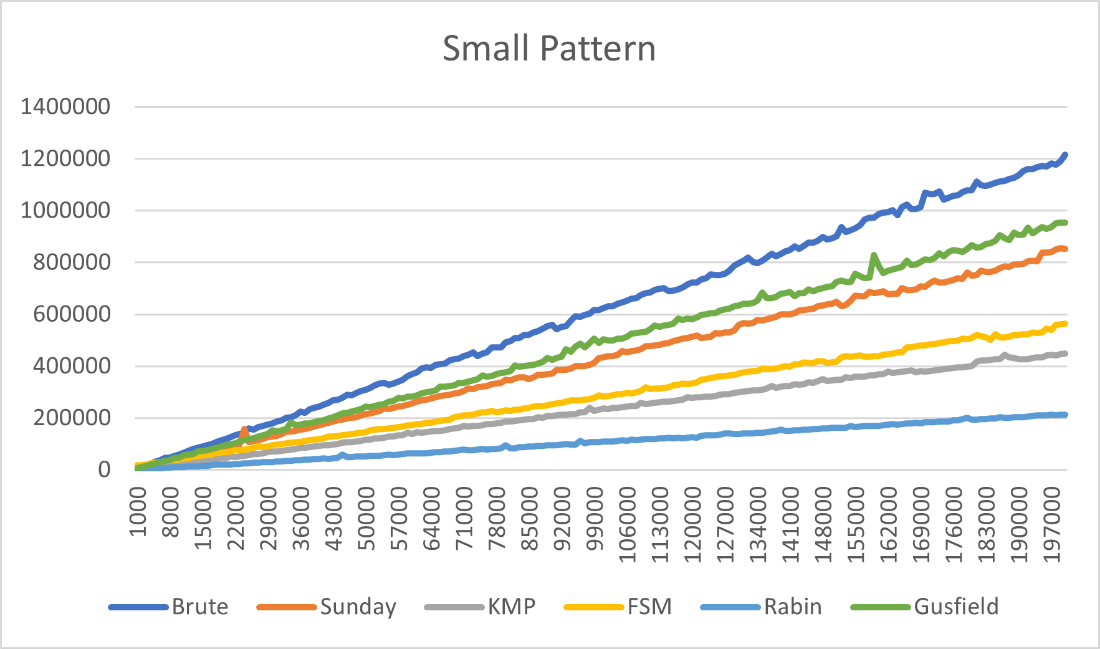


Figure 5: Small pattern matching comparison

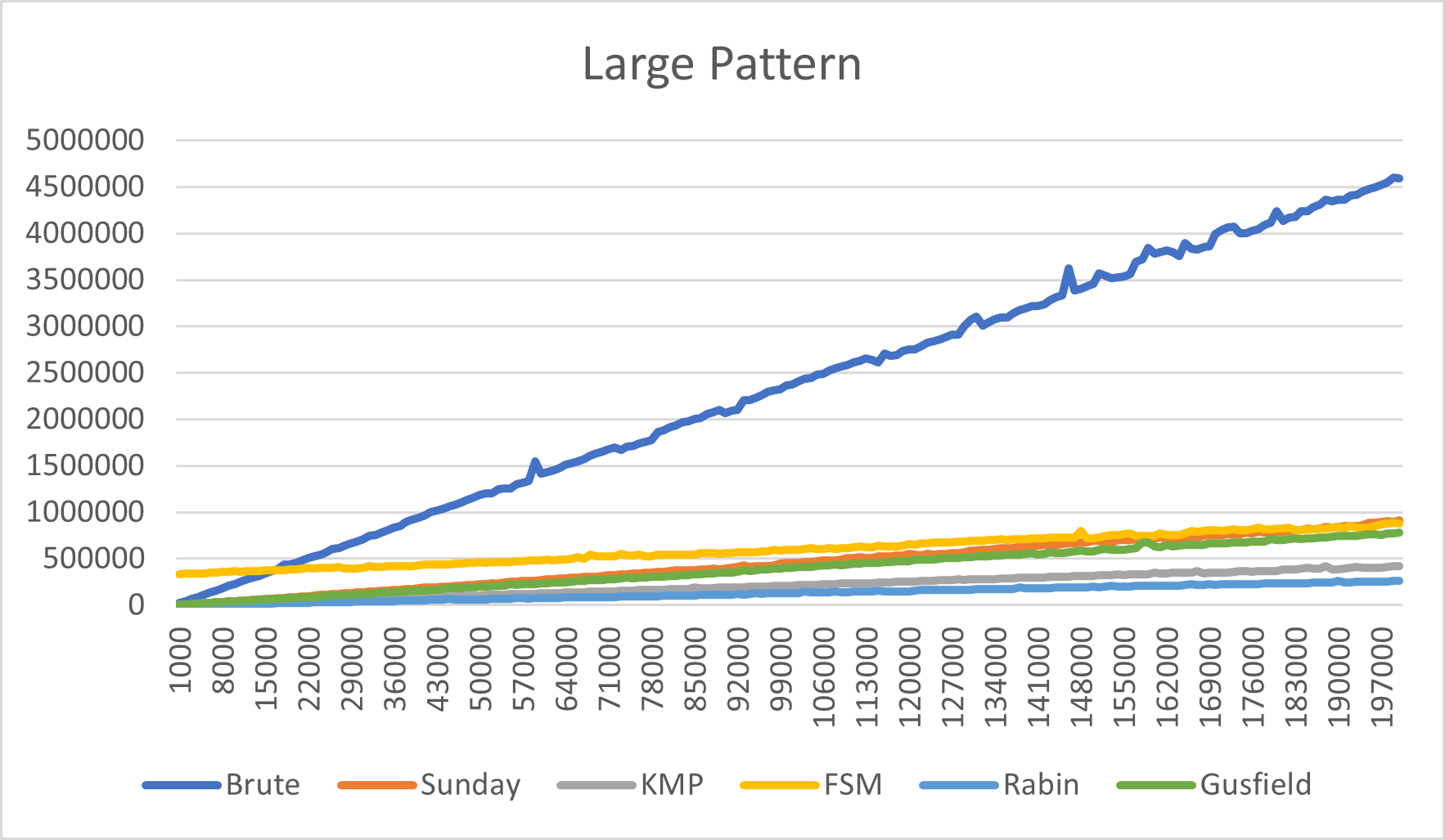


Figure 5: Large pattern matching comparison

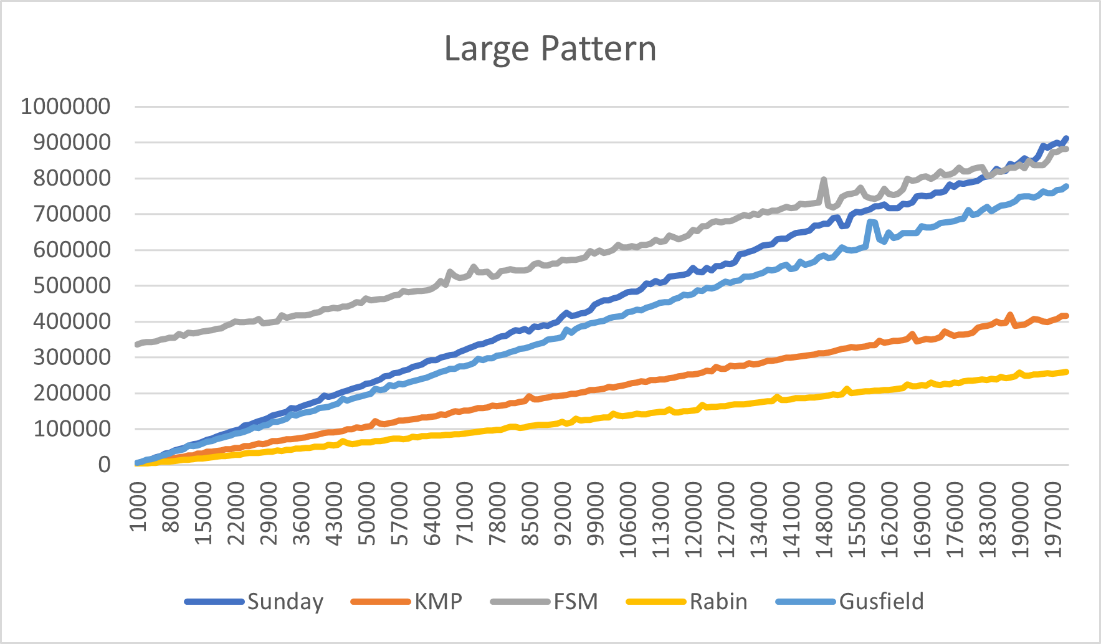


Figure 5: Large pattern matching comparison (no Brute Force)