Improving Exact Search of Multiple Patterns From a Compressed Suffix Array

Kalle Karhu

Department of Computer Science and Engineering
Aalto University
kalle.karhu@aalto.fi

Abstract. Self-indexes are largely studied and widely applied structures in string matching. However, the exact matching of multiple patterns using self-indexes is a topic that has not been the subject of concentrated study although it is an area that may have direct and indirect applications and uses in fields such as bioinformatics. This paper presents a method of improving the exact search of multiple patterns from a compressed suffix array. The proposed method is able to cut down run-times for the handled patterns by as much as 71.6 %. A set of 1000 patterns of length 1000 nucleotides each is found from a text of 50 MB in size 14.0 % faster than by searching the patterns using the locate functionality of the compressed suffix array.

Keywords: self-indexes, pattern matching, indexing, text algorithms

1 Introduction

Self-indexes have been an expanding area of research in recent years. As one leading example solution, FM-index [1] is widely used in many approaches and tools, large portions of which are closely related to bioinformatics [5,3]. In the problem frames of bioinformatics, it is not uncommon to search multiple sequences successively from the same text database. However, the possible improvements related to searching multiple patterns at once have not been studied very broadly to date, when considering the cases of searching text from an index structure.

The focus of this work is to seek possible improvements in one case of searching multiple patterns from an index structure. The index structure that is being considered is a self-index, the compressed suffix array (CSA) [6]. More specifically, this work focuses on the cases where one or more preprocessed sets of patterns are being searched from multiple preprocessed text databases. In such a problem frame, the preprocessing of a set of patterns needs to be done only once per set, but as the single set will be searched multiple times, the cost of the preprocessing is spread over multiple searches. Because of this, it is not reasonable to take the preprocessing times directly into account when looking at the run-time of a single search.

Moreover, the focus of this work is on exact matching which can be seen as a starting point for more practical implementations, including approximate matching. Even in bioinformatics, where exact matching is rarely sought after, it is noteworthy that a large number of successful tools use exact matching as part of a seed-and-extend methodology.

2 Methods

The workings of the proposed method are divided into three work phases: preprocessing of the text, preprocessing of the set of patterns, and searching the set of patterns

Kalle Karhu: Improving Exact Search of Multiple Patterns From a Compressed Suffix Array, pp. 226–231.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2

from the text. The two preprocessing steps need to be done only once for each set of patterns and each text. The search phase uses both of these preprocessing steps to improve speed in the search.

2.1 Preprocessing of text

The text is preprocessed by making a compressed suffix array [6] of it. The implementation provided in the Pizza&Chili corpus [2] is used to produce this index.

The most important functionality for the searches that are the focus of interest of this work is the *locate* function. Locate function allows location of the *occ* occurrences of a query of length m from a text of length n in $O(m \log(n) + occ \cdot \log^{\epsilon}(n))$ time. Here ϵ belongs to $(0, 1)$, depending on the chosen space/time trade-off.

2.2 Preprocessing of patterns

The set of patterns is preprocessed in order to find a certain set of substrings of the patterns. The goal is to find a collection of substrings, where each substring would occur in a large number of patterns, while still occurring comparatively rarely in the text.

To achieve this goal, the proposed method uses a compression tool named Re-Pair [4] to find suitable substrings from the set of patterns. Re-Pair recursively replaces the most frequent pair of symbols in a text by a new symbol, reevaluating the frequencies of all of the symbol pairs with respect to the now-extended alphabet, repeating this process until there is no pair of adjacent symbols that occur twice. These new symbols correspond to phrases, which are substrings occurring more than once in the processed text. These phrases comprise the pool of potential substrings to be used in our method.

In order to limit the number of occurrences of the substrings in the text, a threshold value is set to limit the minimum length of a substring. After applying this threshold, the remaining substrings are searched from a CSA made from the set of patterns, to retrieve the number of patterns the substrings occur in and the offsets of the substring occurrences from the start of the patterns. After this search, the substrings are sorted in descending order, by the number of patterns in which the substring occurs. The patterns in which each phrase occurs and the respective offsets from the start of the pattern are saved, as this information is needed in the search phase.

2.3 Searching a set of patterns from text

In the search phase, the preprocessed set of patterns is searched from the preprocessed text. The substrings obtained during the preprocessing are searched from the text in descending order by the number of patterns in which they occur, using the locate functionality of the CSA. For each occurrence of a substring, possible occurrences of the patterns that include the substring are checked by character comparison. First the pattern is compared, character by character, with the text, starting from the beginning of the pattern, continuing up to the occurrence of the substring in the pattern. This is followed by comparing the characters of the pattern and the text, starting from the end of the pattern, moving towards the occurrence of the substring. If any mismatch is found during the exact match or if the whole pattern matches

the text, the search continues with processing the next pattern where the substring occurs.

When all occurrences of a substring have been checked with all of the patterns corresponding to the substring, all of these patterns are marked as *treated*. As all occurrences of a pattern are found when checking all occurrences of a substring of the pattern, the patterns that are marked as treated need not to be checked when handling later substrings.

After all of the substrings obtained from the preprocessing have been handled, the remaining patterns that are not yet marked as treated are searched using the locate functionality of the compressed suffix array for the full pattern. Alternatively, the search using the substrings can be terminated after a certain set amount of patterns have been marked as treated, finishing the remaining patterns with the locate functionality.

3 Results

3.1 Data

The text used was a DNA text of 50 MB in size, obtained from the Pizza&Chili corpus [2]. From this text, 1000 substrings of length 1000 nucleotides each were randomly picked, comprising the set of patterns. It was noticed that each of these patterns occurred exactly once in the text.

3.2 Experiments

All the following runs were done using a single Intel®Core™i7 CPU 860 @ 2.80 GHz on a PC with 16 GB RAM. The implementations were done with C++.

The pattern set described above was preprocessed as described in Section 2.2. The creation of the compressed suffix array for the patterns took 0.23 seconds, resulting in an index totaling 743.5 KB in size, using parameters samplerate = 16 and samplepsi = 128. The Re-Pair compression tool was run on the set of patterns to retrieve the full list of substrings occurring more than once as the side product of compressing the set of patterns, which took 0.44 seconds. Lastly, the occurrences of the substrings in the patterns were searched, varying the threshold determining the minimum substring lengths. This parameter was given values of 25, 28, 30, 33 and 35. Resulting run-times for this third step of preprocessing the pattern set are shown in Table 1, together with the total times taken by the preprocessing for each minimum substring length. The substring search times are averages over five repeats of searches.

Minimum substring length	25	28	30	33	35
Searching substring occurrences from patterns (s)	0.166	0.150	0.142	0.132	0.130
Total preprocessing time (s)	0.836	0.820	0.812	0.802	0.800

Table 1. The times taken by searching the substrings from the set of patterns as the function of minimum substring length, together with the total preprocessing times.

After this preprocessing of the pattern set, text was preprocessed by creating a compressed suffix array of it, using parameters samplerate = 16 and samplepsi = 128. The creation of the index took 22.69 seconds and the total size of the resulting index was 36.8 MB.

Figure 1. Time taken for searching the 1000 patterns from the text as the function of patterns handled by the proposed method, with varying minimum substring lengths. Dashed light gray line marks the average run-time for the pattern when using locate of CSA for the full patterns.

The preprocessing steps were followed by searching the set of patterns from the text. Searches were done separately for each of the minimum substring lengths. Additionally, the number of patterns allowed to be searched with the proposed method varied from 100 to 500. However, the actual number of patterns that had common substrings of required length within them was in some cases less than this, resulting in a smaller number of patterns being handled with the proposed method. The run-times of the proposed method were compared to searching all of the patterns with the locate functionality of the compressed suffix array implementation. Each of the runs were repeated 50 times. The average run times for each of the parameter sets and the traditional CSA are shown in Figure 1.

The average times for a pattern to be found by searching a substring and then extending are shown in Figure 2. For comparison, the average time for searching a pattern using the locate functionality for the full pattern is also shown in the figure.

Looking at the full runs of 1000 patterns, the best results were retrieved when using a minimum substring length of 30, resulting in 14.0 % saving in run-times, when 238 patterns were found by using the proposed method. When considering the average time for a single pattern to be found by searching the substring and then checking the exact match, the best results were retrieved when using a minimum substring length of 35, resulting in 71.6 % saving in run-times per pattern, when 155 patterns were found by using the proposed method.

Lastly, when a pattern was handled by searching the substring and then checking the exact match, the time the exact matching took was compared to the total time of this process. The portion of the time taken by the exact matching per pattern as the function of minimum sequence length is shown in Table 2.

Figure 2. Average search time of a pattern when using the proposed method as the function of patterns handled by the proposed method, with varying minimum substring lengths. Dashed light gray line marks the average search time of a pattern when using locate of CSA for the full pattern.

Minimum substring length	25	28	30	33	35
Portion of time spent in exact matching (%)	14.9	14.5	14.4	8.1	7.9

Table 2. The portion of run-time taken by exact matching, when using the proposed method for a pattern, as the function of minimum substring length.

4 Conclusions

The results show two clear trends as far as run-times are concerned as a function of minimum substring length and the number of patterns handled by the proposed method. Firstly, it is clear that as the minimum substring length is raised, the average time it takes for a pattern to be found by the proposed method decreases. Secondly, as more patterns are handled by the proposed method, again the average time it takes for a pattern to be found by the proposed method decreases.

The first of these trends suggests that longer substrings are better in improving the run-times of the searches than short ones. This is very sensible, as longer substrings are expected to occur in the text less commonly, on average, resulting in fewer occurrences to be checked by the exact method. The latter trend suggests that the first substrings on the list, which occur in the largest number of patterns, are not optimal in the sense of decreasing the run-times per pattern. This is probably because the substrings that occur commonly in the set of patterns also occur commonly in the text, resulting in a large number of excess occurrences to be checked with the exact matching. However, regardless of the mentioned flaw, the proposed method was able to improve the run-times of the searches remarkably by reducing the total sum of the lengths of the patterns and substrings to be searched with the locate functionality of the CSA.

4.1 Discussion and Future Work

The current methods for choosing and sorting the substrings to be used are relatively straight-forward. As the substrings that head the list currently in use are clearly less than perfect, it is also clear there is room for improvement. This may be because the data is not independently and identically distributed, which means that longer sequences are not necessarily occurring more rarely in the text, especially so if they occur frequently in the set of patterns.

One approach, that will be studied in the future, is to consider the k-mer distributions of the substrings and compare them to the k-mer distributions of samples of sequences similar to the text to be. This would most likely give better estimates of the probability of a substring to occur in the text.

The scoring and sorting of the substrings that overcome a set threshold is another area of future improvements. Instead of using a simple threshold, it would probably be profitable to take into account both the expected number of occurrences in the text and the number of occurrences in the pattern set and give a score to each substring.

Lastly, when the substrings are being sorted, it should be dynamically taken into account which patterns are already being taken care of by a substring with a higher score.

Acknowledgements

I would like to thank Travis Gagie for comments and suggestions that helped arrive at the ideas presented in this work. The work was supported by the Academy of Finland (grant 134287).

References

1. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in IEEE Symposium on Foundations of Computer Science (FOCS 2000), 2000, pp. 390–398.
2. P. Ferragina and G. Navarro: *Pizza&Chili – compressed indexes and their testbeds*, May 2011.
3. B. Langmead, C. Trapnell, M. Pop, and S. Salzberg: *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.* Genome Biol., 10(3) 2009, p. R25.
4. N. J. Larsson and A. Moffat: *Off-line dictionary-based compression.* Proceedings of the IEEE, 88(11) 2000.
5. H. Li and R. Durbin: *Fast and accurate short read alignment with Burrows-Wheeler transform.* Bioinformatics, 25(14) 2009, pp. 1754–1760.
6. K. Sadakane: *New text indexing functionalities of the compressed suffix arrays.* Journal of Algorithms, 48(2) 2003, pp. 294–313.