REPORT 2

Introduction to Focus Areas - Advanced Algorithms - Group 8 "C run ch $time$: $O(T - deadline)$"

Maximilian Otto*, Dominik Bannwitz and Florian Herzler

*Correspondence:
maximilian.otto@fu-berlin.de
Department of Mathematics and Computer Science, Free University of Berlin, Takustraße 9, 14195 Berlin, Germany
Full list of author information is available at the end of the article
†Equal contributor

Abstract

Goal of the project: Implement and compare different methods for finding patterns within a reference. Analysis of runtime and memory consumption.

Methods used in the project: Implemented a naive search, binary search, mlr-heuristic approach, a *longest common prefix*-based approach, and the FM-Index in C++20.

Main results of the project: A huge improvement in runtime comes with a relatively small cost of memory consumption when choosing a suffix-array based approach over a naive search [$O(m * n)$ vs. $O(log_2(n * m))$].

Possible improvements: Overload the '$< / >$' operators for customized comparisons of dna5-types. Use a more efficient or better sorted map, so the lookup time remains constant. Find the left and right borders in parallel threads, or restrict the search for the right borders to the intervall from the previous found left border to references' end.

Keywords: C++; Exact matching; MLR; LCP; Benchmark; Weekly homework

1 Introduction

In the field of bioinformatics, pattern matching algorithms are frequently used to search for specific sequences of nucleotides or amino acids within DNA or protein sequences. It is often referred to as matching a query or pattern to a reference. Optimizing these searches is a crucial task to reduce time consumption, required computational power and memory consumption. Unlike methods and approaches that are capable of matching multiple patterns at once, like the data structure of an Aho-Corasick-Tree would allow it, we are focusing on exact, single pattern matching algorithms. One of the most often used approaches is based on a suffix-array or suffix-tree. This specific data structure requires a preparation step to store all possible suffixes of the reference with different length in a lexicographical order and at which positions they appear in the original reference. Even though this takes some time, it allows querying the data structure in a very efficient way.

1.1 Goal

With this report, we wanted to explore the different suffix-based approaches to match single queries of varying length to a reference DNA. To further explore the difference in runtime and memory consumption, we wanted to compare them to a naive and brute-force approach.

2 Methods

2.1 Benchmarks

For measuring the runtime the google benchmark library was used [1]. All benchmarks were performed on an Intel 12700KF with 8 performance (3.60 GHz) and 4 efficiency cores (2.70 GHz) resulting in 20 threads. As RAM two DDR-4 16 GB 3200 CL 16 sticks were used.

2.2 Data

We were given a file in the compressed `fasta.gz` format which contained $\sim$ 101.250.000 nucleotides of a reference chromosome. We were also given multiple `fasta.gz` files, each containing 100.000 reads with a different length (40, 60, 80, 100 bp). For a second analysis that allows mismatches, we used the whole Humane Reference Genome. To load the files directly as vector of `dna5` data, which occupies 3 bits per nucleotide, we used the `sequence_file_input` function of SeqAn3 [2]. All benchmarks were performed on the same hardware for all different query lengths and several amounts of queries.

2.3 Naive Approach

The brute force approach to find queries within the reference is a straight-forward implementation. Searching for the whole query with exact matches from the beginning of the reference allowed us to search multiple occurances by using the same function with a continuously adjusted start-parameter, until it reaches the end of the reference. Every position of a hit gets stored in a vector.

2.4 Suffix array based approaches

To construct and sort the suffix array, we used `divsufsort` by Yuta Mori, which is concisely described in [3]. The suffix array contains the positions of all lexicographically ordered suffixes in regards to the reference. Because of this, all matching hits of a query should be next to each other in the suffix array, pointing to their positions in the reference.

2.4.1 Binary Search

To speed up the search for a query on the reference, a binary search was implemented. It needs an ordered data structure, so the classical binary search is conducted on the previously created and sorted suffix array. The resulting left and right borders of all hits are stored in a struct. Every stored position of the beginning of a suffix in the referenc, that lays within the range of the borders found by the binary search on the suffix array can be easily accessed and printed as all exact matches.

2.4.2 MLR-Heuristic

To speed up the search on the suffix array, the MLR-trick got applied to the binary search. This trick is used to reduce the amount of comparisons of the query with a suffix, by quickly checking, whether the query is a prefix of the current suffix and how long it is, or not. This allows to skip the comparisons for the length of the prefix. With this, the binary search is improved.

2.4.3 Longest Common Prefix

To improve the previous method even further, the longest common prefixes (LCP) of all suffixes can be previously computed. In our case, the LCPs are stored in a map, representing a tree. The LCP-tree stores the length of the LCPs between each pair of consecutive suffixes in the suffix array. This will lead to a higher main memory consumption, but should save a lot of comparisons due to the sorted suffix array. This allows to quickly narrow down the search space and return the left and right borders of all hits. The LCP-tree gets computed and stored as a file, so it can be loaded anytime.

2.4.4 FM-Index

The FM-Index, first published in [4, 5] by Paolo Ferragina and Giovanni Manzini, is a data structure based on the Burrows-Wheeler-Transform (BWT) [6] which is closely related to the structure of a suffix array, but can be traversed as a prefix-tree. The FM-Index combines fast searching with good compressibility.

3 Results

3.1 Runtime Benchmarks

For benchmarking the runtime of our different methods, we performed each method multiple times to be able to compare their average execution times. No additional compiler based optimization-flags were used. Construction of the suffix arrays, LCP-trees and FM-Index are not included in this benchmarking.

Because of the high linear runtime of the naive search, we did not perform benchmarking with more than 10 queries of different lengths, but the difference in speed for all methods is already visible when only searching 10 queries.

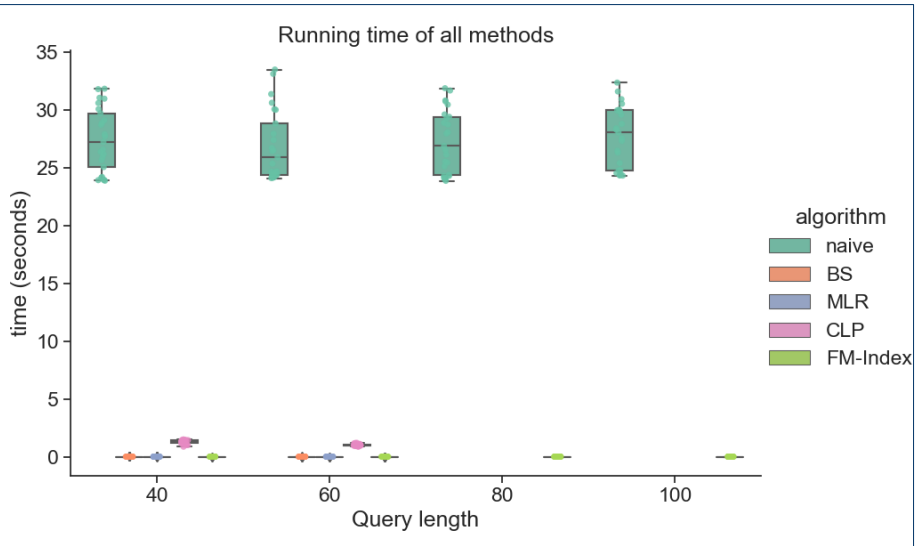

Figure 1 Runtime in seconds for all implemented methods, tested on multiple runs with different lengths of requested queries to find. Every time, 10 queries were searched.

We only benchmarked the naive and FM-Index searches on all different query lengths in this setting. As expected, the naive approach is the slowest when compared with the other implementations. The difference in time consumption is only barely noticeable for longer queries, but this might be due to the nature of the runtime of $O(m * n)$, since m is relatively small in regards to a reference length of 100.000.000.

To compare the binary search with the MLR-Heuristic and FM-Index, we analyzed the performance of our implementations with queries of $length = 100$ for a variety of different amounts of queries to be searched for.

It is shown in figure 2 that the MLR algorithm generally performs better than the more simple binary search. It seems like having a runtime of logarithmic nature. However, the FM-Index search is surprisingly the slowest.

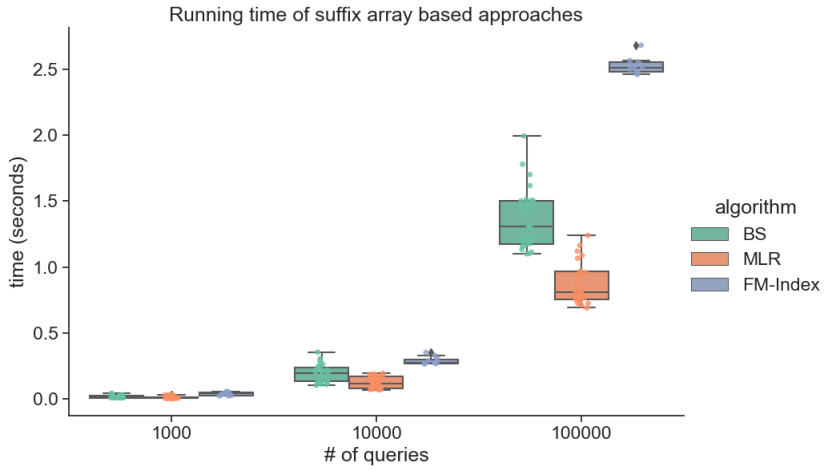

Figure 2 Runtime in seconds for Binary Search, MLR and FM-Index with different amounts of queries with length 100. 25 runs were performed for every method.

When comparing the runtimes depicted in figure 2 to the results of the LCP-approach we implemented (fig. 3), it is clear that there must be a flaw or mistake overseen in our implementation of the search with the LCP-tree.

3.2 FM-Index - Search with mismatches

Searching with the FM-Index allows mismatches, controlled by a parameter, it should be able to find much more queries in the reference. It is expected, that it drastically changes the overall runtime, so we searched different query lengths with up to 2 mismatches in the whole *Humane Reference Genome*. The results of this are shown in figure 4 and it clearly outperforms our LCP-based approach from the previous experiments.

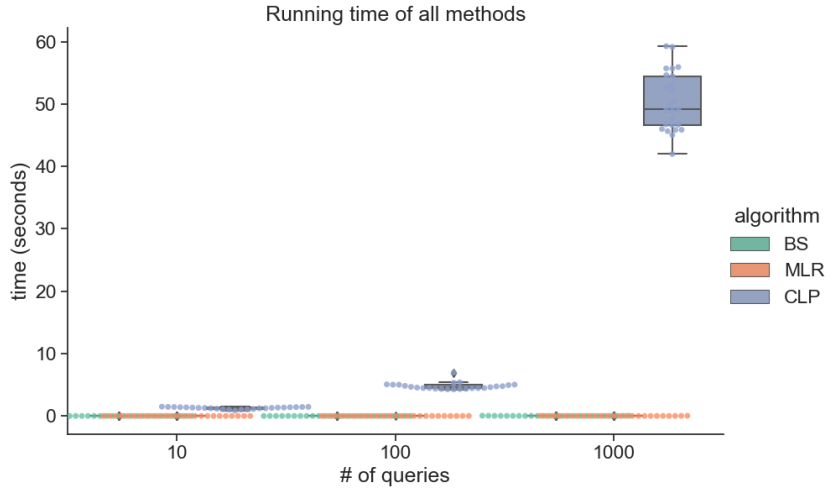

Figure 3 Runtime in seconds for three suffix array based implementations with different amounts of queries with length 40. 25 runs were performed for every method.

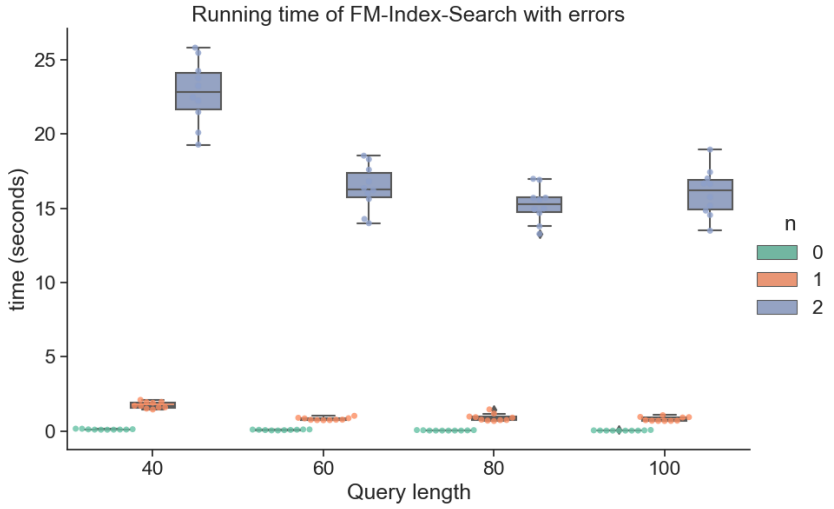

Figure 4 Runtime in seconds for searching 1.000 queries of different length in the *Humane Reference Genome*. 10 runs were performed on every configuration with up to two allowed mismatches (n).

3.3 Memory Consumption

The difference in memory consumption for all these methods is perceived as very straight-forward and can be measured but is also heavily dependent on the implementation.

3.3.1 Naive Search

While the naive search only requires the query and reference in memory, the space consumption can be expressed as $O(m + n)$, where m and n are the pattern and reference length.

3.3.2 Binary Search

The binary search of our implementation is performed on a suffix array. Hence, the memory consumption is $O(m + 2 * n)$. This works well for all our use cases.

3.3.3 MLR-Heuristic

Since the MLR-based approach uses the same underlying functionality as our implementation of the binary search but implements a trick for less comparisons, the overall memory consumption stays the same.

3.3.4 Longest Common Prefix

The approach with the LCP-tree uses an additional data structure to store the prefix-length of two consecutive entries in the suffix array, which leads to an additional memory consumption of $130 * n$, because the data type (dna5) used by SeqAn3 loads every nucleotide with only 3bit into main memory, but our `size_t` uses 64bit and we need multiple values to store this additional information. This is due to the data structure of the LCPs, consisting of a map with pairs, i.e. `std::map<std::pair<size_t, size_t>, size_t>`. Consequently, we are given a consumption of $O(m + 134 * n)$. This leads to an impressive consumption of over 12GB in main memory, but storing the LCPs in a file on a hard drive leads to a consumption of 4.5GB.

4 Discussion

While the naive search is the easiest of the proposed methods to understand and implement, it is also the slowest. The savings of memory consumption do not make up for the linear runtime. Our supposedly fastest implementation is by far not our most optimized version and therefore lacks a bit of speed. Also changing the structure of our LCPs could result in a way lower memory consumption. Also, beginning to search for the right boundary from the left boundary on, instead of index zero, could drastically reduce the overall runtime. The Binary Search on the other hand occupies a reasonable amount of space, but is a little bit slower than the MLR-based approach, which uses the same amount of space.

The FM-Index works great on large references data, and while perform well finding exact mathces or those with just one mismatch, it is weekend by finding shorter queries more frequently. This could be due to the implementation of the lookup, where exactly the found hits are positioned within the reference, because only every thenth suffix is stored. Having a higher number of allowed mismatches leads leads to an inclusion of this problem, so we would not recommend it for heaving approximate matching, but large reference data bases.

The code and data is available under:
https://git.imp.fu-berlin.de/herzlef98/ifa-2022/-/tree/main/advanced_algorithms/week1
The explanation of how to run the code can be found in a $README.md$ file in the same repository.

References

1. Google Benchmark. https://github.com/google/benchmark Accessed 2021-07-26
2. Reinert, K., Dadi, T.H., Ehrhardt, M., Hauswedell, H., Mehringer, S., Rahn, R., Kim, J., Pockrandt, C., Winkler, J., Siragusa, E., Urgese, G., Weese, D.: The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. Journal of Biotechnology **261**, 157–168 (2017). doi:10.1016/j.jbiotec.2017.07.017
3. Fischer, J., Kurpicz, F.: Dismantling divsufsort. CoRR **abs/1710.01896** (2017). 1710.01896
4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 390–398 (2000). IEEE
5. Ferragina, P., Manzini, G.: An experimental study of a compressed index. Information Sciences **135**(1-2), 13–28 (2001)
6. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. In: Digital SRC Research Report (1994). Citeseer