INTRODUCTION TO FOCUS AREAS WS 22/23 — GROUP: SEQSTYLES

Report: Implementing a search

Abhinav Mishra, Jule Brenningmeyer, Maike Herkenrath*

Correspondence:
abhinav.mishra@fu-berlin.de
Department of Mathematics and Computer Science, Freie Universität-Institute of Computer Science, Takustraße 9, 14195 Berlin, Germany
Full list of author information is available at the end of the article
*Group 7

Abstract

Imagine a fictional world in the future. A Virus has been spreading the world. You are working together with Virologists on a vaccine. To check if this vaccine is working the virologists need to find certain markers in the human genome.

Keywords: Naive; suffix-array; fmindex; C++; perl; R; runtime; memory; query

1 Introduction

For answering of various biological questions it is necessary to find a specific pattern in a given text such as a sequenced DNA or RNA. Therefore, different approaches with different advantages and disadvantages can be used. Before applying an algorithm to find the pattern, the text can be pre-processed, to create the static data structure of a suffix array. Suffix arrays, introduced by Manber and Myers, contain the indexes of all suffixes of a given text, which are sorted lexicographical. By transforming the given text to a suffix array the runtime of searches in the text can be reduced [1] [2].

The two search types exact and approximated string matching can be applied to find a pattern in a text. Exact string matching finds all occurrences of the pattern in the text, while approximated string matching finds additionally the pattern with a defined number of errors. Exact string matching can be performed by a binary search, which has a runtime of $O(m \cdot log(n))$. To improve the runtime of the search to $O(m + log(n))$ the mlr trick can be used. The longest common prefix of the left and the right border are saved during the search, whereby the intervening suffixes no longer need to be compared at these positions. However, if no common prefix exist the worst case runtime is still $O(m \cdot log(n))$. Another time reducing approach is to save the length of the longest common prefix (lcp) of the suffixes. To determine the left border, of the interval containing the pattern, the lcp-value of the left border and the pattern and the lcp-value of the left border and the middle are compared to decide if the middle will be set to the new left or right border. The procedure for the right border is performed accordingly. Through the use of the lcp-values the runtime can be reduced to $O(m + log(n))$ [2] [3] .

An alternative to the use of suffix arrays is the FM index, which uses less storage space. The FM index uses the data structures Occ, C and L. The data structure L is also called the Burrows-Wheeler transform and is obtained from the conceptual matrix M. The text T and all cyclic shifts of the text are saved in a lexicorgaphically order in the conceptual matrix M. The last column of the matrix corresponds to the data structure L. Occ is a matrix which contains how often each character occurs for each line of L. C is an array that indicates for each character in the alphabet how often a lexicographically smaller character occurs in the text T. To find the

pattern in the text with these data structures a backward search is performed. The backward search returns an interval with the length of number of occurrences of the pattern. These matches are located in the text. [4]

To lower the runtime of the FM Index search we make use of the pigeonhole principle. If a query matches a reference with 2 errors and we then split the query in 3 parts, at least one of the parts will not include an error. This can be implemented by searching for parts of a query without allowing an errors and then checking the previous and following sections to see whether the whole query matches (this time allowing errors).

2 Goal

Given a reference text, which has parts of the first chromosome of the human genome with a list of markers 'illumina_reads_XYZ.fasta.gz'. We have to find out how many of these markers appear in the reference, and which algorithm is best suited:

1 Implement a naive search algorithm.
2 Implement a suffix-array-based search.
3 Implement an fmindex-based search.
4 Implement an fmindex-based search that exploits the pigeonhole principle
5 Benchmark (runtime and memory) your solutions for 1000, 10,000, 100,000 1,000,000 queries of length 100.
6 Benchmark (runtime) queries of the length 40, 60, 80, and 100 with number of queries = 1000.
7 Download the Humane Reference Genome here.
8 Benchmark (runtime) of queries with $k = 0, 1, 2$ errors of length $= 40, 60, 80, 100$ with number of queries = 1000.

3 Methods: Implementation

The human reference genome file (size $\approx$ 900MB) was downloaded using a simple `curl` command via RefSeq ftp:

$$curl \ * ftplink * \ -O$$

3.1 C++

For the **naive approach** we did a very simple algorithm that first loops over the entries in the reference and for each of them loops over all entries in the query to check whether they are equal. If a different character is found it stops the loop and continues with the next entry in the reference.

The **suffix array search** used the provided function `libdivsubsort` to create the suffix array. Then a basic binary search is used to find a match in the reference. If such a match is found consecutive entries of the suffix array are also checked for a match. This continues until a non-matching suffix is found.

For the search with the **FM-Index** we constructed the FM-Index by using the provided code from the lecture. To find the queries the function `seqan3::search` was used.

The last task of using the pigeonhole principle in addition to the **FM-Index** search was not completed. The queries are split into 3 equal sized parts and for

each of the parts the **FM-Index** is searched. We know that of the 3 parts only 2 can include an error, so we search for perfect matches for each part. To complete the program the reference should be searched, surrounding the indices where parts of the queries were found, while allowing errors.

3.2 perl

The program matches marker sequences from illumina reads to the reference genome file. The input files are of *.fasta.gz* format. Usage of standard modules: `stricts`, `warnings`, and `diagnostics` facilitated the creation of a clean code with good coding practices. There are check points in the script to make sure error handling, and lexical scope is maintained [5].

Only core modules of `perl` were used [5]. As most `perl` versions have standard modules already installed, making it executable on unix-based servers, and local systems. Perl has a strong regular expression matching, so a *regex* form was used to search, and display number of occurrences for each sequence in the reference genome, globally [5].

As we know, *.fasta.gz* is a gzipped version of *.fasta* so file reading was done such that it doesn't has to decompress but directly read, in turn, saving memory, and time. A subroutine `read_fasta()` was written to read and parse the files, removing new line characters, *FASTA* header, and returning only the sequence.

A short walk through the program `read_pattern.pl` would be helpful:

1 Reading the reference file in one slurp using a written subroutine `read_fasta()`.
2 Asking user which read file to use for search/matching ($length = 40, 60, 80, 100$) using `STDIN` I/O file handling operator, and saving the header (ID) and sequence in the hash table as *key:value* pair.
3 Asking user the number of queries to perform for the selected read length, and correspondingly storing the sequences (values) in an *array*.
4 Asking the user for performing *regex* pattern matching or index-based matching.
5 Displaying the matched sequences, and the total counts out of numeber of queries.
6 Closing the file handlers, and exit the program.

3.3 R

The program reads one reference genome file, one read sequences file, number of queries for read file, and creating then searching using *fmindex* [6] [7]. The output displays the matched read sequences after the positions of reads in reference sequence. It has dependencies/required packages `seqinr`, `fm.index`, `optparse` were used [7] [8] [9].

`fm.index` wraps the `C++` library `SDSL v3` and uses a Compressed Suffix Array based on a **Wavelet Tree of the Burrow-Wheeler Transform** of the given string library [7] [10].

Generally speaking, fmindex consists of:

- a Burrows-Wheeler transform (BWT) of the text string represented with wavelet matrix.
- an array of size $O(\sigma)$ (σ: number of characters) which stores the number of characters smaller than a given character.

- a (sampled) suffix array.

A walkthrough the program `fmindex.R`[6]:

1 Passing file names, and query arguments using python-like `optparse` package.
2 Reading the input gzipped *.fasta* file for sequence retrieval using `seqinr` package.
3 Creating fmindex (1-based indices) using package `fm.index`.
4 Searching the index for sequence matching.
5 Displaying the matches with three columns:
   (a) `pattern_index` is the index of the read sequence.
   (b) `corpus_index` is the index of the matching string in the reference.
   (c) `position` is the starting position of the match within the reference sequence.
6 Displaying each matched sequence by retracing the index to the list of read sequences.

4 Results

The following tables shows the memory usage and runtime for the `C++` programs `naive_search.cpp`, `suffixarray_search.cpp` and `fmindex_search.cpp` including `perl` and `R` implementations as computed by `/usr/bin/time -v` and using timestamps within the program.

	Number of Queries			
	1000	10000	100000	1000000
Naive Search	528.52s	NA	NA	NA
Suffix Array Search	0.577083s	3.73734s	42.3837s	49.3992s
fmindex	0.414965s	0.773419s	4.47966s	5.69738s
Naive(perl)	1.24min	10min	2hrs	NA
fmindex(R)	1.20min	1.17min	1.24min	NA

Table 1 Runtime for queries of length 100

In fig.1 the runtime for queries varying numbers of queries of length 100 bases are shown. The time it took the naive search algorithm with 1000 queries was already over 8 minutes. If we extrapolate from that the time it would take for a million queries we could not hand in the assignment in time. Therefore we did not run the benchmark tests for over 1000 queries.

The Suffix Array search was clearly a lot faster than the naive approach with only 0.58 seconds for 1000 queries. With larger query sets the running time also increased to almost 50 seconds, which is still an order of magnitude faster than the naive search.

The search with the FM-Index was even faster then the suffix array search, especially for higher number of queries. The FM-Index search needed only 5.7 seconds for 1000000 queries, which is 10 times faster then the suffix array search.

Note that the query set does not have a million entries, so when it is resized to a million the given 100000 queries are reused.

For a fixed size set of 1000 queries and varying length of $40, 60, 80, 100$ bases the naive search takes about the same time, as can be seen in fig.2.

The suffix array search is slower for the queries of length 40. This is probably because there are many more occurrences for shorter queries and the way the algorithms was implemented.

	Number of Queries: 1000			
Query length :	40	60	80	100
Naive Search	542.596s	544.459s	542.447s	543.330s
Suffix Array Search	0.322s	0.216s	0.259s	0.437s
SAS, computation of SA included	34.560s	31.130s	31.720s	31.140s
FM Index	0.625168s	0.397651s	0.397606s	0.451831s
fmindex(R)	1.18min	1.27min	1.2min	1.16min
Naive(perl)	3.45min	3.6min	4min	4.09min

Table 2 Runtime for varying query lengths

The binary search for the suffix array will find an occurrence of the query if one exists. If there is more than one occurrence then there is an interval in the suffix array where all entries match the query. The first one that is compared to the query by the binary search can be any of them. The way the algorithm was implemented only the following suffixes are checked, not the previous ones. Thus, the more a query appears in the reference, the less efficient it gets because more suffixes are one after the other compared to the query. And also the less correct this algorithm gets, as more occurrences can be overlooked. In future implementations a more sophisticated approach should be chosen, that finds the upper and lower bound in the suffix array where the reference matches the query.

The search with the FM-Index has the slowest search time for a query length of 40, probably because there are more occurrences for short queries.

	Number of Queries			
	1000	10000	100000	1000000
Naive Search	221056 kB	-	-	-
Suffix Array Search	1000760 kB	1001392 kB	1001580 kB	1023772 kB
fmindex	83224 kB	83620 kB	83668 kB	126748 kB
Naive(perl)	420936 kB	419732 kB	422712 kB	NA
fmindex(R)	1422420 kB	1412300 kB	1427116 kB	NA

Table 3 Memory usage for queries of length 100

The used memory times are shown in fig.3. In comparison, the fm index search needed less memory then the other to searches.

In a further task, the queries were allowed to have 0 till 2 errors. The search was performed with the fm index for different number of queries. The runtimes are shown in fig. 4. The runtimes are increasing when more errors are allowed and when more queries are used. The same can be seen when searching for the queries on the GRCh38.fna.gz human genome assembly with a query lenngth of 40 (fig. 5) or 100 (fig. 6).

	Number of Queries			
	1000	10000	100000	1000000
FM Index (k=0)	0.414965s	0.773419s	4.47966s	5.69738s
FM Index (k=1)	0.919887s	4.61965s	45.1023s	147.019s
FM Index (k=2)	6.99858s	67.6588s	683.868s	-

Table 4 Runtime for queries with $k = 0, 1, 2$ errors with number of queries = 1000, 10000, 100000, 1000000 on h38_partial.fna.gz human genome assembly. The queries have a length of 100.

Fig.7 shows the runtimes of queries with k = 0, k = 1 and k = 2 errors of length 40, 60, 80, and 100. We used 10000 queries for each search to so that the overall runtime is not too long but a clear difference between the times is to be seen. Similar to fig.4, fig.5 and fig.6 the runtime is increasing for more allowed errors. If zero or one error is aloud queries with a length of 60 show the fastes runtimes, while for two

	Number of Queries			
	1000	10000	100000	1000000
FM Index (k=0)	1.38533s	2.2359s	17.0616s	56.755s
FM Index (k=1)	2.11465s	13.3906s	124.626s	434.756s
FM Index (k=2)	14.644s	145.249s	1442.37s	-

Table 5 Runtime for queries with $k = 0, 1, 2$ errors with number of queries = 1000, 10000, 100000, 1000000 on GRCh38.fna.gz human genome assembly. The queries have a length of 40.

	Number of Queries			
	1000	10000	100000	1000000
FM Index (k=0)	5.29037s	4.81658s	13.6813s	11.8815s
FM Index (k=1)	9.6846s	17.6205s	88.1585s	267.553s
FM Index (k=2)	22.8547s	206.266s	2157.6s	16061.4s

Table 6 Runtime for queries with $k = 0, 1, 2$ errors with number of queries = 1000, 10000, 100000, 1000000 on GRCh38.fna.gz human genome assembly. The queries have a length of 100.

errors a query length of 100 seems to be the fastes. The same search was performed on the GRCh38.fna.gz human genome assembly. In general, the running times are longer due to the longer sequence, but the ratio of the different times is similar.

	Number of Queries: 10000			
Query length :	40	60	80	100
FM Index (k=0)	0.625168s	0.542299s	0.674553s	0.849774s
FM Index (k=1)	5.46307s	4.30991s	4.38631s	5.02307s
FM Index (k=2)	78.0268s	73.6055s	75.0378s	68.0739s

Table 7 Runtime for queries with $k = 0, 1, 2$ errors of length $= 40, 60, 80, 100$ with number of queries = 10000 on h38_partial.fna.gz human genome assembly

In Table 1, 2, and 3, the memory and runtime for the `R` program `fmindex.R` is including the both fmindex creation and fmindex searching for different query lengths, and number of queries. It is established that creation of fmindex takes more time than searching it.

In the last task was to implement the pigeon hole principle. We did not manage to implement the verification of the rest of our query inside the text. Our main problem was that we did not know how to access the surrounding parts of the found query in the FM index.

5 Discussion

Suffix array was initially planned to write on `perl` using a `CPAN` module `Tree::Suffix`, but time-exhaustion coupled with problems in symlinking with `C++ libstree` library resulted in no further development. Later, it was found that the library is no longer maintained by the developer.

The *naive search* algorithm (with and without `index`), and *fmindex* were implemented in `perl`, and `R`, respectively.

The `perl` script can be easily modified to suit the needs for short or long sequence matching according to user's requirements. Calling another script or passing argument variables is also manageable to make it a standalone sequence matcher. Some modifications might be done for contigs, motifs, etc to account for biological intricacies. For that scenarios, using `BioPerl` is recommended.

The `R` script can be changed in a way that it saves the index using `fm_index_save()` and load using `fm_index_load()` from `fm.index` package for it doesn't need to create a new index every time, the user exits the R session. While reading the new task 6 .fna.gz file, the program `fmindex.R` gives an `basic_string::_M_create` error and exited at a non-zero status.

	Number of Queries: 1000			
Query length :	40	60	80	100
FM Index (k=0)	9.82509s	5.98889s	5.30293s	8.9126s
FM Index (k=1)	26.6287s	19.1427s	16.4632s	16.5994s
FM Index (k=2)	282.644s	217.802s	199.969s	187.066s

Table 8 Runtime for queries with $k = 0, 1, 2$ errors of length $= 40, 60, 80, 100$ with number of queries $= 10000$ on GRCh38.fna.gz human genome assembly

Considering the memory, and runtime for each benchmark, **fmindex search is the best suited algorithm.**

Abbreviations
fmindex: Full-text index in Minute space

Competing interests
Authors declare that they have no competing interests.

Author's Contributions (Group 7)

- **Abhinav Mishra** wrote the programs `read_pattern.pl` & `fmindex.R` for implementing & benchmarking naive search in perl, fmindex in R, including only benchmarking for `fmindex_search.cpp`. He wrote the corresponding methods sub-sections, and discussion, seperately.
- **Jule Brenningmeyer** wrote the section scientific background and partly on further parts of the review. She worked on the code for the naive search, the fm-index search and tried to implement the pigeon hole principle. She ran the benchmark tests for the fm index task. She familiarized herself with working on the servers via ssh, using vim and C++.
- **Maike Herkenrath** wrote the naive search, suffix array search and the partial pigeonhole search implementations, ran the benchmark tests for task 1 and worked on the report. She also did a lot of the research for getting all the tools to work since we all had no prior experience with working on a server, programming in cpp and vim and also had some issues with git.

References

1. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. acm Computing Surveys (CSUR) **39**(2), 4 (2007)
2. Reinert, K.: L1,2 suffixarrays-slides (2023)
3. Mitani, Y., Ino, F., Hagihara, K.: Parallelizing exact and approximate string matching via inclusive scan on a gpu. IEEE Transactions on Parallel and Distributed Systems **28**(7), 1989–2002 (2016)
4. Reinert, K.: L4-7-fm$_index(2023)$
5. Christiansen, T., Wall, L., Orwant, J., *et al.*: Programming Perl: Unmatched Power for Text Processing and Scripting. " O'Reilly Media, Inc.", ??? (2012)
6. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2022). R Foundation for Statistical Computing. https://www.R-project.org/
7. Hug, C.: Fm.index: Fast String Searching. (2022). R package version 0.1.1. https://CRAN.R-project.org/package=fm.index
8. Davis, T.L.: Optparse: Command Line Option Parser. (2022). R package version 1.7.3. https://CRAN.R-project.org/package=optparse
9. Charif, D., Lobry, J.R.: SeqinR 1.0-2: a contributed package to the R project for statistical computing devoted to biological sequences retrieval and analysis. In: Bastolla, U., Porto, M., Roman, H.E., Vendruscolo, M. (eds.) Structural Approaches to Sequence Evolution: Molecules, Networks, Populations. Biological and Medical Physics, Biomedical Engineering, pp. 207–232. Springer, New York (2007). ISBN : 978-3-540-35305-8
10. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: 13th International Symposium on Experimental Algorithms, (SEA 2014), pp. 326–337 (2014)