RESEARCH

Introduction to Focus Areas in Bioinformatics Project Week 12

Sina Glöckner[*], Christina Kirschbaum, Swetha Rose Maliyakal Sebastian and Gokul Thothathri

[*]Correspondence:
sina.gloeckner@fu-berlin.de
Institute for Informatics, Freie Universität Berlin, Takustr. 9, Berlin, DE
Full list of author information is available at the end of the article

Abstract

Goal of the project: The objective of this project is to apply and compare different search algorithms for reference text, a segment of the human genome, as well as the queries, reads from an Illumina sequencing machine.

Main results of the project: We were able to successfully compare the performance of both algorithms and benchmark run time and memory for different amounts and lengths of queries.

Personal key learnings:
Sina & Christina: Refresh C++ skills
Gokul & Swetha: Understood benchmarking, the naive search algorithm and suffix array based search.

Estimation of the time:
Sina: 8h
Christina: 7h
Gokul & Swetha: 5h

Project evaluation: 3

Number of words: 1142

1 Scientific Background

The problem of finding a query with m characters in a sequence of length n, both of them using the alphabet $\sum$ is a common issue in bioinformatics. In the following, we discuss two different approaches to this problem, a naive algorithm and an algorithm that utilizes suffix arrays.

Naive pattern searching is the most basic approach. It looks for all of the main string's characters in the pattern. For smaller texts, this approach is useful. There are no pre-processing steps required. By checking for the string once, we can find the query. It also does not require additional space to carry out the process. The Naive Pattern Search method has an $O(m * n)$ time complexity. [1, 2]

One approach to decrease the time-consuming nature of this process are suffix trees. A more efficient way to save suffix trees was introduced in 1990 by Manber and Myers in the form of suffix arrays, who also documented the first algorithms for building and using suffix arrays. Suffix array creation algorithms have multiplied in a baffling profusion since that time, particularly in the last few years. [3]

For many, if not all, of the string processing problems that suffix tree methods can solve, suffix arrays have become the data format of choice. Suffix trees are data structures that allow for quick string searches. In $0(n*log|\sum|)$ time and $0(n)$ space, a suffix tree can be created. [4]

A suffix array is a new data structure that is essentially a sorted list of all the suffixes of the sequence. String searches can be answered in $0(m * log(n))$ time with a conventional binary search. A few Augmentations to the binary search when combined with information about the longest common prefixes (lcps) of nearby elements in the suffix array, improve the run time further [4]. In the following, we applied the mlr-trick, that makes use of this concept.

2 Data

The searches to investigate the algorithms were performed with a reference text, a genomic sequence that is part of the human genome, and query reads from an Illumina sequencing machine, in fasta format.

3 Methods

3.1 Naive Search Algorithm

String matching algorithms are critical components in most operating systems' implementations of practical software applications and have an important role in bioinformatics. Any string matching method must be able to swiftly detect some or all instances of a user-specified reference text in a query. Among the several pattern search algorithms, naive pattern searching is the most fundamental.

Its methodology is simple: in case of success in matching the first element of the pattern, each element of the pattern is successively tested against the text until failure or success occurs at the last position. After each unsuccessful attempt, the pattern is shifted exactly one position to the right, and this procedure is repeated until the query is found or the end of the target is reached. [5]

3.2 Suffix Array Algorithm

Another approach to string matching are suffix arrays (SA). In this case, we built the suffix array using libdivsufsort [6]. Multiple algorithms utilize a sorted list of suffixes of the reference, one of them is the so called mlr algorithm.

The algorithm finds the first and the last occurrence of the query in the SA and returns all positions in between. Both borders are identified with a modified binary search algorithm. During the binary search, the lcps of the left and the right side are stored and their minimal value is denoted as mlr. Since the SA is sorted, only the positions between $mlr + 1$ and the query length need to be compared in the next search step. This decreases the search time in comparison to a simple binary search, that always compares every position of the query to the current suffix in SA.

3.3 Benchmarking

In computer science, comparative evaluations of computer systems, compilers, databases, and many other technologies are performed using standard benchmarks. The adaptation of a standard benchmark decreases testing variation and allows for a statistical, performed analysis of diverse techniques and methodologies. Benchmarking also has the benefit of including replication into the process. [7]

4 Results

First, the performance of both algorithms was compared with different numbers of queries (Table 1). The naive approach was applied to 100; 1,000; and 10,000 queries. Since the run with 10,000 queries was very time-consuming, no further tests could be conducted. The suffix array approach was computed for query numbers between 100 and 1,000,000 with a tenfold increase. In both cases, a tenfold increase in the number of queries results in a tenfold increase in the run time, while memory consumption stays constant overall runs.

Table 1: Results for the benchmarking with 1,000, 10,000, 100,000 and 1,000,000 queries of length 100.

Queries	Naive approach		Suffixarray approach	
	Runtime	Memory	Runtime	Memory
100	4m23s	217004 kB	2ms	607952 kB
1,000	41m36s	217000 kB	16ms	607952 kB
10,000	6h52m55s	217000 kB	159ms	607952 kB
100,000	-	-	1558ms	607948 kB
1,000,000	-	-	1655ms	631392 kB

Additionally, the two approaches were benchmarked for queries with a length of 40, 60, 80, and 100. For the naive approach, it was tested with only 100 queries because of the high run time that we encountered during the benchmarking for several amounts of queries. However, the suffix array approach was examined with 100,000 queries.

The results show that the runtime of the suffix array approach is with 1.4 to 1.5 seconds superior for all lengths, while the naive approach takes over 4 minutes in every case. The memory consumption is nearly constant again over the different runs. A full summary of the results of the runtime and the memory for the approaches can be found in Table 2.

Table 2: Results for the benchmarking of queries of the length 40, 60, 80, and 100 with 100 queries for naive approach and 100,000 queries for suffixarray approach.

Query-Length	Naive approach		Suffixarray approach	
	Runtime	Memory	Runtime	Memory
40	248s	210756 kB	1427ms	601704 kB
60	249s	213880 kB	1481ms	604824 kB
80	248s	215440 kB	1539ms	606392 kB
100	249s	217004 kB	1558ms	607948 kB

When comparing the two approaches, big differences can be seen. The naive approach is much slower than the mlr algorithm but takes up less space. A suffix array requires three times the space of the naive approach, which is still less than 1 GB in our example. Its construction is not included in the table and took about 30 seconds in every run. Even when the time for the construction is added, the suffix array is still much faster, especially for an increasing amount of queries.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
Gokul Thothathri and Swetha Rose Maliyakal Sebastian wrote the scientific background, data and the method of the report. Sina Glöckner and Christina Kirschbaum implemented both search algorithms, conducted benchmark procedures, and wrote the results in the report.

References
1. Nemytykh A. On Specialization of a Program Model of Naive Pattern Matching in Strings (Extended Abstract). 2021 08;.
2. Nemytykh AP. On Specialization of a Program Model of Naive Pattern Matching in Strings. arXiv preprint arXiv:210810865. 2021;.
3. Simon J Puglisi WFS, Turpin A. A taxonomy of suffix array construction algorithms, ACM Computing Surveys, Vol. 39, Issue 2,. 2007 01;p. 2141–2144.
4. Manber U, Myers E. Suffix Arrays: A New Method for On-Line String Searches. SIAM J Comput. 1993 01;22:935–948.
5. Lovis C, Baud RH. Fast exact string pattern-matching algorithms adapted to the characteristics of the medical language. Journal of the American Medical Informatics Association. 2000;7(4):378–391.
6. Mori Y. libdivsufsort; 2022. Original-date: 2015-03-17T15:30:25Z. Available from: https://github.com/y-256/libdivsufsort.
7. Aniba MR, Poch O, Thompson JD. Issues in bioinformatics benchmarking: the case study of multiple sequence alignment. Nucleic Acids Research. 2010 07;38(21):7353–7363.