REPORT

Introduction to Focus Areas - Complex System Block

Group 3: Ibraim Ibraimi, Dennis Kragen, Melika Moradi

Full list of author information is available at the end of the article
*Equal contributor

Abstract

Goal of the Project: Our goal is to implement different search algorithms for large sequences, and finally compare the speed and memory consumption of these different search algorithms.

Methods and Datasets: To find the markers on the genome we implemented a naive algorith, a suffix array search and a FM-Index search. We used (parts of) the Humane Reference Genome "hg38" and a list of markers.

Estimated working hours: 130 hours

Scientific Background

Time and memory complexity are critical components of algorithms, and search algorithms are no exception. Choosing the right search method for your data can lead to significant savings in time and storage. Sorting has also attracted a lot of attention due to its crucial role in optimizing algorithms. Typically, sorting algorithms consist of two nested loops, but other factors such as the number and type of data points also play a role.

SeqAn is a C++ library that provides efficient algorithms and data structures for processing biological sequence data. Like any other C++ library, SeqAn seeks efficiency, consistency, and code reusability. SeqAn3 is a newer version of SeqAn. It provides high performance through efficient algorithms and appropriate data structures for sequence representation and transformation, full-text indexing, and effective search and sequence alignment.

In computational biology, string matching algorithms are used to find exact sequences and identify all occurrences of a particular pattern in a larger text, such as DNA sequences. The simplest idea is to use a naive search algorithm. The naive search is based on the idea of sliding the pattern sequentially over the text and looking for an exact match. Another idea is a suffix array-based search. A suffix array is a data structure that represents text and allows substring queries. Suffix arrays store all suffixes of a given string, in our case sequence readings, where a suffix represents a pointer to its first character. A suffix array is sorted (lexicographically). After the suffix array is created, binary search is applied to a given suffix array.

Goal

The goal was to evaluate different algorithms and figure out which one is the best to find certain markers in the human genome.

Data

We used the data from the project "ImplementingSearch" on GitHub (https://github.com/SGSSGene/ImplementingSearch). We had a reference text that contained parts of the first chromosome of the human genome, and we also had a list of markers that we needed to find out how many of them were in the reference.

Method

Naive Search Algorithm

Naive searching algorithm checks for all character of the reference to the query. To implement the Naive Search Algorithm, we used a nested loop to match the query sequence with subsequences of the same length as the query from the reference sequence. Each query moves over the reference. If an exact match is found in the, the position of the match is printed in the reference text and that means we have found an occurrence. We also checked the runtime and Memory-consumption for the different number of queries (100, 1000, 10000, 100000 and 1000000). Additionally, we explored the performance of the algorithm on the query data with different read length (40, 60, 80 and 100).

Suffix Array Search

Suffix array represents a compact data structure used in pattern search in order to find the coordinates as well as the counts of a pattern in a a given text. More precisely, after generating all suffices and sorting them based on a lexicographic order, the indices of the sorted suffices in the original text (in our case DNA), build the proper suffix array which is used for pattern search. In our case we implemented binary search to search for a pattern given a suffix array by determining the upper and lower bounds in which the pattern eventually appears in the suffix array. For the upper bound, Lp, we used the pseudocode from the slides. For the lower bound we propose the following pseudocode:

Data: Suffix array $Suftab$.
Result: Lower bound of the pattern appearance in the suffix array.

```
if L_p = n then
    R_p = L_p;
end
if p = Suftab_[n] then
    R_p = n;
end
else
    (L, R) = (L_p, n);
end
while R − L > 1 do
    M = ceil(L + R)/2; if p < Suftab_[M] then
        R = M;
    end
    else if p = Suftab_[M] AND p < Suftab[M + 1] then
        R = M;
    end
    else
        L = M;
    end
end
```

Algorithmus 1: Pseudocode for finding the lower bound R_p.

FM-Index

The FM-index search is a method which searches patterns in large texts like DNA sequences by transform the original text into a compact representation, the Burrows-Wheeler Transform. It provides a structure which is sorted in such a way that frequently occurring characters appear behind each other.
To implement the FM-Index search we used the construction function for the FM-index and the search function of sequan3.

Results

Naive Search Algorithm

As can be seen in Table 1, as the number of queries increases, the runtime of the search algorithms grows exponentially. This was expected because as the number of queries increases, more matches can come in the reference. However, the amount of memory used by the program remained the same, even with different numbers of queries.
Table 2 shows the naive algorithm's performance on files containing queries with different read lengths. For all files, the runtime and the maximum size of the resident set did not change with increasing read length. We obtained a runtime of about 49 seconds for all files for each read length, and the maximum size of the resident set did not change significantly either.

Number of queries	Run-time/Memory-consumption
100	49s/ 213600
1000	8min16s/ 212892
10000	1h36min/ 212396
100000	15h09min24sec/ 213808
1000000	-

Table 1 Run-time and Memory-consumption (maximum resident set size) for illumina_reads_40.fasta.gz.

Read Length	Run-time/Memory-consumption
40	49s/ 213600
60	49s/ 215836
80	49s/ 217696
100	49s/ 220420

Table 2 Run-time and Memory-consumption (maximum resident set size) of different read length in reference file for 100 queries.

Suffix Array Search

Unfortunately, based on the suffix array we used, we could not get any results. Either the process would end prematurely or it will last very long. We used both the cluster assigned to us and the local computer.

FM-Index Search

The run time of the FM-Index search is only 0.13 seconds with zero errors (table 3), 0.16 seconds with one error (table 4), 0.13 seconds with two errors (table 5) and 0.14 seconds with three errors (table 6) at a read length of 40. Furthermore its only 0.32 seconds with zero errors , 0.25 seconds with one error, 0.23 seconds with two errors and 0.23 seconds with three errors at a read length of 100. Also you can see that the run time of the FM-Index search grows linear when the read length increases and less if the error count is higher than zero. The memory consumption between the different error counts is nearly constant and only varies by a few bytes.

Read Length	Run-time/Memory-consumption
40	0.13s/ 76948
60	0.18s/ 80080
80	0.21s/ 81632
100	0.32s/ 83820

Table 3 Run-time and Memory-consumption (maximum resident set size) of different read length in reference file for 100 queries by error count 0.

Read Length	Run-time/Memory-consumption
40	0.16s/ 76940
60	0.16s/ 80012
80	0.20s/ 81586
100	0.25s/ 83160

Table 4 Run-time and Memory-consumption (maximum resident set size) of different read length in reference file for 100 queries by error count 1.

Read Length	Run-time/Memory-consumption
40	0.13s/ 77384
60	0.18s/ 80056
80	0.21s/ 81564
100	0.23s/ 83164

Table 5 Run-time and Memory-consumption (maximum resident set size) of different read length in reference file for 100 queries by error count 2.

Read Length	Run-time/Memory-consumption
40	0.14s/ 77044
60	0.18s/ 80132
80	0.22s/ 81600
100	0.23s/ 83192

Table 6 Run-time and Memory-consumption (maximum resident set size) of different read length in reference file for 100 queries by error count 3.

Comparison

The FM-Index needs only 0.32-0.23 seconds at a read length of 100 (table 3). Compared to 49 seconds for the same read length in the naive algorithm search (table 2) its nearly 154 times smaller. Also the memory consumption is 3 times lower. That shows the extreme superiority of the FM-Index over the naive Algorithm .

Discussion

We were challenged with many technical issues. Coupling the code that proved to be efficient on small texts locally, to the template in the server which used the libdivsufsort library, proved to be very difficult. The main issue consisted in adapting the c++ code with the c-like environment in the template provided in the server. Consequently, we weren't able to deploy the libdivsufsort algorithm in constructing the suffix array that was provided as default. Instead, we were led to use an algorithm which provided the suffix array in $O(n^2)$ running time. We tried many optimizations in building the suffix array without using libdivsufsort. For example, we cut the suffices at length 100 before sorting them out and building the suffix array. We first compared this algorithm as of its validity for small texts versus the usual one. They proved to provide the same results. Nevertheless, on the scale of large texts, the optimization didn't seem to be any better than the one used on the unchanged suffices. The next measure we undertook was to account for only a fraction of the length of the reference, 1k, 10k, 100k, and providing up to 1000 queries. Unfortunately, we could only get the count 0 and a large execution time. We thus learned how difficult it is to handle big data.

Acknowledgement

We thank Simon for the support that he provided along this block.

References

Lecture slides.