Custom Sequence Aligner

Introduction

Objective

The goal of the project is to provide an aligner of given reads to the given reference genome.

Input

1) .fasta file - provides information about reference genome.

2) .fastq file - information about reads.

Output

.sam alike file, tsv file with following fields:

1.	Read name - name from fastq file
2.	Flag - 0 - read, 1 - read complement, 4 - unmapped
3.	Reference name - from fasta file
4.	Position - starting position for alignment
5.	Mapping quality - value from dynamic programming matrix
6.	CIGAR string - from edit transcript
7.	Read sequence - from fastq file
8.	Sequence quality - from fastq file

Seed-and-extend Approach in Nutshell

In seed-and-extend approach, the alignment is done in two phases. First, seeding, where each read is split into small fragments (seeds). Those seeds are searched for in the reference genome using selected index structure for the reference genome (hash map, Burrows Wheeler transform, suffix array...). After the seeds are found, extend phase starts. For each read, the best seed is selected for performing dynamic programming and calculating edit distance between that read and reference genome at the corresponding positions. Info for all reads are saved in the .sam file, which is used for further processing.

Custom Approach

This project combines classic seed-and-extend approach, with some optimizations and modifications. The goal was to implement an accurate aligner, whose speed is at the reasonable rate. That partially succeeded, while there is still room for improvement.

One of the issues of the seed-and-extend algorithm is the repeated work in dynamic programming. In this project, we tried to reduce number of cases where dynamic programming actually needs to be performed. We tried to delay and avoid dynamic programming whenever possible, and essentially use classic seed-and-extend only when other options were not possible.

The results showed 99.85 % successfully aligned reads, with execution time of around 100 seconds. This will be examined in further detail on the following pages.

Implementation

Data

Circular DNA

We are analyzing genome of the human mitochondria. The genome is circular, which means there will be reads which are aligned partially at the end of genome and partially at the beginning. They will be referenced as "circular reads", or if we are talking about blocks (seeds) - "circular blocks".

General Algorithm

Those are the steps to perform alignment. Each of the steps will be further discussed.

1. Define parameters of the aligner (seed interval and seed length)
2. Parse .fasta file
3. Parse .fastq file
4. Seed
5. Extend
6. Save to .sam alike file

1. Define Seed Parameters

Parameters can be defined experimentally. Work on this project showed that seed length of 20 and seed interval of 10 work the best.

2. Parse .fasta File

Read data from. fasta file, such as name and sequence. It was assumed there was only one entry in .fasta file. That should be fixed for real life problems, but was considered acceptable for the experiment.

Besides reading fasta info, in this part we determine the window size for the seed ranking. We want seed windows to be around two or three times longer than size of the read. We would like these windows to be of the same size, so that there is no bias in number of seeds in a window. If this is not the case it will be probably happen that the last window gets smallest number of seeds. In order to prevent this, we search for a number which divides length of fasta sequence without remainder. This number then represents the window size.

3. Parse .fastq File

Save info about each read, such as name, strand and strand quality.

4. Seed

Make seeds

For efficient seed querying, we needed an appropriate index structure for the reference. Hash map was chosen for the index structure. Index value was instantiated from the reference strand.

For each read we calculated the reverse strand. Then, seeds were made for both strands. Index value was queried with seeds from both strands. The better value in terms of number of seeds is saved. This is how we decided if the strand is forward or reverse (comparing to the strand value from .fastq file).

In this phase seed ranking map was also populated.

Merge seeds

After previous phase, we had information about every seed - read it comes from, position in reference where it is aligned, start and end positions within the read. The idea is that for every read we merge seeds that can be merged. That way, instead of lots of small fragments, we get, depending on the sequencer quality and aligner parameters, small number of longer fragments which are aligned.

5. Extend

Proposed Algorithm

Merging the seeds gives us a chance to check if reads are maybe completely aligned, or read's first and last aligned block are within read's length. In these cases we can avoid searching for best candidates, since we already have the positions where the read needs to be aligned. If the read is completely aligned, dynamic programming is not needed at all. If the read's seeds fit into read's length, we need to perform dynamic programming exactly once. If a read contains circular blocks, dynamic programming is done for the first and last block. If these conditions are not fulfilled, we do classic seed-and-extend.

Proposed Algorithm Pseudo Code

```
for each read:
    if (read has no aligned blocks)
        save_info()
    else
        if (read has one block)
            if (block is completely aligned)
                save_info()
            else
                do_dynamic_programming(block_start, block_end)
                save_info()
        else
            if (difference between first and last block is less than read's length)
                do_dynamic_programming(first_block_start, last_block_end)
                save_info()
            else
                if (there are circular blocks)
                    do_dynamic_programming(first_block_start, first_block_end)
                    save_info()
                    do_dynamic_programming(last_block_start, last_block_end)
                    save_info()
                else
                    find_best_candidate()
                    do_dynamic_programming(best_cand_start, best_cand_end)
                    save_info()
```

6. Save to .sam Alike File

Save information about every read to a file.

Algorithm Choices

Index Structure

Hash map was chosen as an index structure. It is a relatively simple structure, efficient, and since we did not use extremely large datasets in the project, was also good in terms of memory. If we were using larger files, Burrows Wheeler transformation with FM index would have been a better option.

Dynamic Programming

Global alignment (Needleman-Wunsch) was chosen as the algorithm for inexact matching, because we were interested in global matching between read strings and a taken portion of the reference genome. The dynamic programming was done between read and the portion of the reference which has the same length as the read.

Best Candidate Finding

For each seed window we have number of seeds that appear in that window. If we have couple of candidates, first we need to map position of each of the candidates to the seed window they belong to. After that, all candidates are sorted by number of seeds in their respective windows. Three best candidates, or less, depending on number of candidates, are taken and done dynamic programming over. The info about candidate with best score is saved.

Results

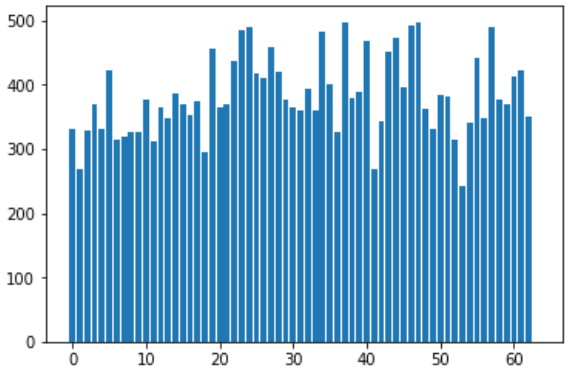

Figure 1 - Number of seeds in a seed window

Fastq file	Seed length	Seed interval	Result [%]	Execution time [s]
art.fq	20	10	100	1.68
test.fq	**20**	**10**	**99.85**	**112**
art.fq	10	6	98.31	128
test.fq	10	6	93.61	449
art.fq	**20**	**5**	**100**	**1.43**
test.fq	20	5	99.85	137
art.fq	30	15	100	106
test.fq	30	15	98.5	116
art.fq	20	15	100	129
test.fq	20	15	99.56	240

Table 1 - Results from various experiments

Discussion

Pros

If the quality of the reads is good, this approach should give good and efficient results. That happens because the program will avoid performing dynamic programming whenever possible. Good quality of reads allows that. Also, aligner parameters should be chosen well.

Cons

If the read quality is not good, there are lots of mismatches in seeding phase, resulting in merging only small fragments, increasing the necessity to perform dynamic programming repeatedly. This slows algorithm down and also reduces performance.

Improvements

A better strategy for ranking can be introduced. Program shows worse performance when finding candidates is done more often.

Local alignment might be used instead of global alignment.

Execution

Program can be run as a Python notebook.

Links

Project can be found at: github.com/dejangolubovic/genome _sequence _aligner