ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

RAPPORT DE PROJET INFORMATIQUE POUR SV

PROJECT NO 3: MOTIFS GÉNOMIQUES

Groupe 8:

Raphaëlle Hartwig, Marie Jaillot, Elsa Mauguin, Estelle Pfitzer, Alexander Popescu, Clémentine Semanaz

18th December 2019

Contents

CHAPTER 1

PROGRAM PRESENTATION

This program intents to represent the binding of transcription factors on specific sequences in a given segment of DNA. A transcription factor can bind to multiple sequences (those with which the affinity is high enough). In order to represent affinities computationally we used a simplified model[1].

CHAPTER 2

OBJECTS

THE POSITION-WEIGHT MATRIX (PMW)

The Position-Weight Matrix (PWM) is a Nx4 matrix that represents a motif, a short sequence of nucleotides, of length N. The number of lines of this matrix corresponds to the length of the motif and its four columns to the 4 different nucleotides (A-C-G-T). The M(i, j) value in the PWM represents the probability of finding given nucleotide j at position i in the motif.
format: .mat file

THE GENOME

The genome is a list of genes composed of many nucleotides in which we are able to look for a given motif[2].
format: .fasta file

POSITIONS IN THE GENOME

Contains 5 columns: the first one represents the chromosome of the sequence, the second column represents the position on this chromosome, the third one the region on the chromosome, the fourth one the score associated to the sequence and the last column shows if the sequence is on the positive or negative strand.
format: .bed file

CHAPTER 3

PROGRAM'S MODES

This program can be used in two different yet complementary ways.

- Mode 1, Motif: The user inputs the PWM, a sequence threshold and the genome sequence. As an output, the user will get the positions in the genome where the score is above the threshold.
- Mode 2, PWM: The user inputs a serie of positions in the genome, the genome and the length of the motif. As an output, he will get the PWM that represent the frequency of each nucleotide at these positions.
- Mode 3, Extension: The inputs are the same as for PWM, in addition there is a convergence threshold.

The user does not have to specify which mode he wants to use. Depending on the input, the program will switch to one or the other mode.

Examples on how to run the program:

- Mode Motif command: ./Team8 -m ../tests/files/DBP.mat -f ../tests/files/promoters.fasta -s 1
- Mode PWM command: ./Team8 -f ../tests/files/mm9_fasta/chr7.fa -B ../tests/files/BMAL1_chr7.bed -l 8
- Mode Extansion command: ./Team8 -f ../tests/files/mm9_fasta/chr2.fa -B ../tests/files/BMAL1_chr7.bed -l 8 -d 0.01

CHAPTER 4

IMPLEMENTATION

MOTIF

This class is used to find the motifs with highest affinity using a PWM matrix and the genome. We use a structure named *found_motif* which contains: a string representing the chromosome the motif it belongs to, the position of the fourth nucleotide of the motif in the genome, a character showing if the motif is found on the coding or non-coding sequence, a string representing the motif and finally the motif's score. Once a motif is found it will be saved as *found_motif* to then be printed out.
The update functions are repeatedly called by the reader class. Update constantly updates the current read nucleotide and its position. Update-sequences updates the buffer sequence and the complementary buffer sequence so we can find a match and create a *found_motif*. Updates-score updates the score for each buffer sequence so we can know what sequence to save. We can then call decision-maker to check if the score is high enough to save the motif as a *found-motif*.
Once a motif is found, we call *transform_bufferseq_to_string* and then print-found-motifs which prints the motif on a new tabulated file. Then we call reset_bufferseq to reset the attribute and therefore be able to compute the affinity of another sequence.
Transform_to_complementary is used to transform a given sequence to its complementary. This function enables us to check for the presence of the motif on the opposite strand.
The multiple getter and setter functions are mostly used by the reader's class and by our tests.

READER

This class contains all the functions needed to read the genome. The functions in this class are constantly calling the update functions of motifs . Those repetitive calls enable the program to read the genome only once while storing as little information as possible.
Reader has three attributes. *Motifs_vec* is a vector of motifs of which each motif contains a pwm and and the location where the motifs were found. As a given fasta file is read, the corresponding motif is updated nucleotide by nucleotide. In case the file names are not given by the motifs or if the user specifies to read from different fasta files than the chromosomal ones, the attribute fasta_files stores the filenames. The third attribute is *fut-vec* which is a vector of future and used for *multi_threading*.
Read initiates the reading of one file and sends the information of the corresponding Motif. Each motif

contains the name of the file where the motifs were found and the current sequence name. *Init_reading* initiates the reading of all the fasta files in *multi_threaded* manner and the search for motifs.

PWM

This class contains all the functions and attributes which are directly related to the second mode of use. It will therefore serve to compute the matrix of a given motif.
To describe the information given by a bed file line, we used a *gen-region* which is a structure containing the sequence name, where it starts and ends as well as on which strand it can be found.
First, *read_bed_line* extracts all the information we need from the bed file. Then, start-searching uses *find_sequences_on_chromosome* to find all the sequences on a given chromosome/fasta file and updates the vector with those sequences.
Compute-matrix computes the PWM from the genome. Sequences from which we want to compute the matrix come as a vector of int. While compute-matrix is being called with all the needed sequences, *matrix-counter* adds nucleotides to the count with its specific weight in the corresponding sequence. Once all the sequences have been considered for the calculations, we call *divide_matrix* with as a parameter the total number of sequences.
Divide_matrix gives us the final matrix which doesn't contain the count for each nucleotide anymore but rather the probability of finding each nucleotide at a given spot.
At the end, *matrix_output* produces the output matrix as a pwm file.

SIMULATION

This is the main class. It manages the user's inputs, defines the simulation parameters, then runs the simulation and prints the results to the output streams. The simulation parameters are the genome, a PWM and a threshold score if the program is used in the first mode. Or, a genome and a .bed file with positions if the second mode is being used. The initialisation method, init uses TCLAP to parse users' inputs.

MAIN

This class runs the simulation and catches the potential errors.

GENERAL ERROR HANDLING

In general, the program will try to continue at all costs and will only throw an error in extreme cases or during the parsing of the user inputs. Invalid sequences in the bed file or invalid letters in the fasta files will be ignored. In case one of the files could not open or was not found the program will continue its

operations with the files it has and print an error message to the terminal with the name of the file that was not found.

JUSTIFICATION OF DESIGN

The character by character lecture and the design of buffer sequences enables us to read the fasta file only once and with an acceptable speed, seeing that the characters are read directly from the stream buffer. The size of the stream buffer could also be adjusted manually but after some trials it was concluded that the default size of about 150kb RAM was the most optimal, since the increase would not improve the performance by a lot.
For both modules the coding of the nucleotides and therefore the sequences corresponded to their index in the position weight matrix (A = 0, C = 1, G = 2, T = 3, unknown/invalid character = 4). This coding was chosen to maximize the performance and avoid unnecessary multiple translations from a character to its index.
In order to avoid excessive information exchange between the classes the fasta lecture mechanism for the second module (Calculating the PWM) was implemented in the class PWM.
In Addition, for each fasta file lecture, one thread is created in order to allow for parallel processing. This further increases the speed of the program, for the reason that in both modules the lecture of the fasta files is the time determining operation.

CHAPTER 5

EXTENSION

We chose to add an expectation-maximization algorithm. This algorithm is an iterative method to find the maximum likelihood estimation. In our case, it means finding the optimal matrix created by the optimal sequences.
We added a Simulation class, which is a subclass of PWM. Iterate computes new matrices and uses *find_best_score_sequences* to find the sequences with the best score to recalculate the matrix.
Equal_matrix checks if the last two matrices are equal or close to a certain threshold. This function is very useful because their might be two different optimised matrix. If we do not check the similarities between our matrices, we will do way more iterations than necessary.

CHAPTER 6

TESTS

This program is provided with a "tests" folder containing a full set of tests which ensure the program behaves as expected. We use Google Test (gtest), a unit testing library for the C++ programming language, based on the xUnit architecture[3]. The tests are organized in different documents which are named according to the class they are testing. In the terminal, we can run all the tests with the command make test.
An interesting test is the one of *decision_maker*, in testMotifs. This test consists of three parts, first we call reinitialize which resets the motif's sequence, score and complementary arguments to base values. We then call the function *decision_maker*, which will compare the actual motif's score and complementary score to the threshold. Finally it will decide whether to store the motif and/or the complementary motif. Using the gtest *EXPECT_EQ()* function, the test takes different scores and verifies that the motifs are stored in the right cases.
Note also that the test for the class Reader includes the prototype classes of Motifs and Reader (simplified versions of the final classes called TReader and TMotifs) that were only created to test the reading mechanism.

BIBLIOGRAPHY

[1] Stormo, G. D. *DNA Binding Sites: Representation and Discovery.*. Bioinformatics, vol. 16, no. 1, 2000, pp. 16–23.

[2] Zhang Lab *What Is Fasta Format*.
`zhanglab.ccmb.med.umich.edu/FASTA/`

[3] Arpan Sen *A quick introduction to the Google C++ Testing Framework.*. IBM DeveloperWorks, 2010-05-11, retrieved 2016-04-12