BIN hw02: BLAST algorithm implementation

Matouš Soldát, 5. 4. 2023

1 BLAST Implementation

The basic gapless version of the BLAST algorithm was implemented. It consists of three steps: generating seeds from the query sequence, finding seed hits in the database and extending found hits into longer gapless matches.

1.1 Seed generation

In the first step of the algorithm, seeds are generated from the query sequence. Seeds are words of predetermined length k which are either substrings of the query sequence (I denote these as *pure* seeds) or are similar to a substring of the query sequence (*impure* seeds). The algorithm generates the seeds in two steps.

First, pure seeds are generated. Each substring of length k of the query sequence is scored against itself using the score matrix. The following equation shows the computation of the score of string a and b of the same length k using score matrix S:

$$score(a,b) = \sum_{i=1}^{k} S\big(a(i), b(i)\big)$$

All substrings with score greater or equal to a predetermined seed-score threshold are accepted as pure seeds. In the second step, all possible words of length k (using keys of scoring matrix S as the alphabet) are scored against each of the pure seeds and are accepted as impure seeds if they score above the seed-score threshold with at least one of the pure seeds.

1.2 Hit search

To find all seed hits in one passage of the database, a finite state automaton is constructed. The states of the automaton correspond with the last 0 to k potentially useful symbols found in the database. For example, if the only seed was "ACT", the states of the automaton would be *empty string*, "A", "AC" and "ACT". Whenever the automaton visits a state denoted by a string of length k, the position of the first symbol of the state string in the database is saved as a hit.

1.3 Gapless extension of hits

Each hit is expanded to the left and to the right separately until the score of the extension decreases below the best score seen during this particular one-sided extension minus a score decrease threshold. This threshold sdc is computed from the scoring matrix as follows:

$$sdc = \max\big[0, -3 \cdot \mathrm{mean}_{i \neq j}\big(S(i,j)\big)\big]$$

The meaning of the *sdc* formula is that it should allow the gapless extension to overcome three average substitutions of symbols. The max with zero is just a safeguard against some weird scoring matrices where the mean of non-diagonal elements would be positive.

1.4 Output

After finding the matches, the algorithm checks for match-duplicates (which can happen since one extended match can contain multiple seed hits), sorts the matches according to their total score (left-extension score + right-extension score + seed score) and prints found matches to the standard output along with the total number of found (unique) matches and run time. Matches with score less than the best score divided by 10 are not printed in standard output individually but they are counted towards the total number of found matches. All matches regardless of score are printed into file *blast_output.txt* (this can be disabled by commenting the last line of *main.py*).

2 Technical details

2.1 Used programming language and libraries

The algorithm is implemented in Python3. It makes use of the following standard libraries:
time, functools
and the following additional libraries:
argparse, Bio (biopython), csv.

2.4 Implementation architecture

The */lib* folder is dedicated to code, the */data* folder is dedicated to input data and the */stuff* folder is dedicated to additional junk.

The file *main.py* in the root folder calls the functions *load_input*, *blast* and *print_results*, *print_into_file*. The function *laod_input* located in */lib/init/input.py* parses the arguments, loads all the inputs and preprocesses the inputs for smooth run of the BLAST itself. The function *blast* located in */lib/blast/blast.py* runs the BLAST algorithm itself. Lastly, the functions *print_results* and *print_into_file,* located in */lib/output/print_XXX.py*, output the results to standard output and to a *blast_output.txt* file, respectively.

2.3 Usage

The algorithm can be run with the included bash script *blast.sh* as well as by calling "python3 main.py" directly. The script accepts the following arguments:

argument	meaning
-e, --scoring_matrix	path to the score matrix in CSV format
-d, --database	list of database files in fasta format
-k, --word_length	length of the word
-t, --threshold	threshold on the seed score
-q, --sequence	Path to the query sequence
--delimiter	Specifies the delimiter used in the score matrix CSV file, ',' (comma) is the default

The *–sequence* and *–delimiter* arguments are not mandatory. If the *–sequence* argument is omitted, the user will be asked to input the query sequence in the standard input.

Two working examples of the call of the bash script are in *run_example.txt*. Apart from the attached files, the second example call also requires human chromosome 16 file, which is mentioned in the assignment, in path *"./data/database/"*.

3 Insights

3.1 Testing

The second of the attached example calls was used to test the algorithm on finding the human hemoglobin in the chromosome 16. The algorithm successfully matched the full length of the hemoglobin to the database file in approximately 8.5 minutes. The algorithm is obviously much slower than the BLASTn server, which can find it in about 8 seconds in much larger database (the human genome), but my algorithm runs in a reasonable time (not hours or days) and works as intended. I was actually pleasantly surprised that an implementation this basic does not run for ages. The algorithm was able to find the hemoglobin in multiple (five) chromosome files too, the run time was about 19 minutes (third example call).

I also tested the algorithm on the small protein alignment example we tried on paper in the labs as part of debugging (first of the example calls). For protein alignment purposes I used the blosum62 matrix. The algorithm is universally usable for both Nucleotied BLAST and Protein BLAST as long as the correct score matrix is provided (nucleotide -> protein BLAST and vice versa would require a bit of extra implementation).

3.2 Difficulties and recommendations

I found this assignment to be a very fun one. The freedom to experiment with different versions and implementations of BLAST is intriguing. I did manage to re-implement some steps of the algorithm to be a bit more efficient or robust (i.e. the one-pass automaton) and I managed to implement a basis for one of the possible improvements (gap extension with linear cost in */stuff/gap_extension.py*), which I sadly did not have time to get to a reasonable state (the dynamic search termination is missing, so the algorithm always searches through the whole sequence for every match).

This exploration process was great and I would definitely keep the freedom in chosen BLAST variant for future runs of the course. However, I would maybe suggest providing the students with implemented functions for the loading of the data as well as argument parsing. I must admit that it took me several hours just to get the inputs working before I was even able to start working on BLAST itself. In hindsight, the time was not wasted, since learning to use those Python libraries will surely come in handy, but I still believe that saving the students a few more hours to experiment with BLAST itself will be more beneficial than the experience with the libraries. Providing the initial functions would obviously mean limiting the freedom in chosen programming language, but I believe that being forced to implement it in Python (or C+), would not be an issue for anyone.