

# MBI Homework 1 for CS students (academic year 2023/24, winter semester)

In this assignment, your task is to implement and evaluate several strategies for comparing two genomes based on their sets of  $k$ -mers. Tasks 1, 2 and 3 ask you to implement several techniques covered in Tutorial 5 for CS students. The remaining tasks ask you to observe the behavior of these techniques in terms of accuracy, running time and memory on real and simulated data.

If you want to read more about the concepts from this homework, check these sources

- Jaccard index: [https://en.wikipedia.org/wiki/Jaccard\\_index](https://en.wikipedia.org/wiki/Jaccard_index), <https://www.statology.org/jaccard-similarity/>
- Minimizers: <https://homolog.us/blogs/bioinfo/2017/10/25/intro-minimizer/>
- MinHash (with a single hash function): [https://en.wikipedia.org/wiki/MinHash#Variant\\_with\\_a\\_single\\_hash\\_function](https://en.wikipedia.org/wiki/MinHash#Variant_with_a_single_hash_function)

## Submission format

Submit three files to Moodle:

1. A written report in PDF format, named `report_<your AIS username>.pdf`. The report should contain all tables and plots with discussion, required by the individual tasks. You can write it in English or Slovak. The individual plots and tables should be properly annotated (axis labels, legends and titles or captions).
2. Your source code in a single Jupyter notebook named `code_<your AIS username>.ipynb`. Please start from the provided template and follow the requirements and recommendations stated below.
3. Output files required by the tasks placed to a single zip file named `outputs_<your AIS username>.zip`.

## Template

The template notebook already contains the following functions:

- A function `read_fasta` which gets a URL or filename of a file in the FASTA format, reads DNA sequences from this file and returns them as a list of strings consisting of letters A,C,G,T. It also returns a list of descriptions of these sequences.
- Function `canonical` returns the *canonical* version of an input  $k$ -mer defined as follows. For a  $k$ -mer  $w$ , let  $w'$  be its reverse complement (obtained by reversing the string and exchanging complementary bases A $\leftrightarrow$ T, C $\leftrightarrow$ G). The canonical  $k$ -mer  $w$  is then the smaller of  $w$  and  $w'$  in the lexicographic (alphabetical) order. For example, 4-mer CACT has reverse complement AGTG, which happens to be lexicographically smaller than CACT, and thus it is its canonical version.
- Function `full_kmer_set` gets a string consisting of letters A,C,G,T representing a

single genome and integer  $k$  and returns a Python set of all canonical  $k$ -mers of the input string. Canonical  $k$ -mers help us to represent both strands of the DNA molecule (the genome and its reverse complement will produce the same set of canonical  $k$ -mers).

- Function `mutate` gets a string consisting of letters A,C,G,T representing a single genome and a mutation probability  $p$  (a real number between 0 and 1). For each base of the input string it mutates it with probability  $p$  and leaves it as it was with probability  $1-p$ . If the base is mutated, it is replaced by a randomly chosen base from the remaining three.
- Function `shash` computes a hash value for a  $k$ -mer. Use it for minimizer and Minhash computations.

The top part of the template also loads some real genomes and creates mutated genomes as follows.

- It uses `read_fasta` on the FASTA file at <http://compbio.fmph.uniba.sk/vyuka/mbi-data/du1/bacteria.fasta> to variables `real_genomes` and `real_genome_descriptions`. This file contains several bacterial genomes. Their descriptions include an identifier from the RefSeq database. We will refer to these genomes by numbers 0,1,... based on their index in the list. Some of these genomes contain unknown bases denoted as N; these are for simplicity omitted.
- It uses real genome 0 as a starting point and creates artificially mutated versions with mutation probabilities  $p=0.01, 0.05, 0.1, 0.2, 0.3, 0.4$  (using the provided `mutate` function). As a result you have 6 new genomes stored in list `mutated_genomes`. The corresponding values of  $p$  are in list `mutation_probabilities`. To make the results reproducible, we set random seed to 47, do not change this value.

## Technical requirements and recommendations

- Please start from the template provided by us, write your code to designated places, and if possible, do not modify the provided code.
- Your notebook should be executable on Google Colab. If you use any libraries not available on Colab by default, install them using `pip install` commands included in the notebook. If you use a different environment to develop your code, test it in Colab before submitting.
- In your code for tasks 1-3, use only standard Python libraries and functions provided by us, such as `canonical` and `'shash'`.
- In parts 4-7 you can use any libraries. Measure all required quantities in the notebook and either visualize them directly there, or if you prefer, you can create final visualizations in another system.
- To avoid running out of memory, we recommend deleting variables which are no longer needed using `del` and occasionally calling `gc.collect()` to run garbage collection and free unused memory.
- Some computations may take a long time to run. Leave sufficient time for finish the assignment. If the computation takes too long, you may try to improve your implementation.

## Task 1: Jaccard similarity

- Implement function `jaccard` which gets one set and a list of  $n$  sets and computes a vector of  $n$  Jaccard similarities of the first set compared each of

the sets in the list.

- Run `full_kmer_set` on all real genomes for  $k=13$  and run Jaccard similarity on the real genome 0 compared to the list of all real genomes (including itself). Write the results to file `jaccard.txt`, each value on one line. The value on the second line of the file should be approximately 0.64.

## Task 2: Minimizers

- Implement function `minimizer_set` which gets a string consisting of letters A,C,G,T and two parameters  $k$  and  $w$ . For each sliding window of  $w$  consecutive  $k$ -mers, it computes the canonical versions of these  $k$ -mers, applies `shash()` function on each canonical  $k$ -mer, chooses the canonical  $k$ -mer with the smallest hash value and stores this  $k$ -mer in the set.
- Run `minimizer_set` on the first 1000 bases of genome 0 for  $k=9$  and  $w=5$  and print the resulting  $k$ -mer set to file `minimizers.txt` in lexicographic order, one  $k$ -mer per line. The file should contain 320  $k$ -mers.

## Task 3: MinHash

- Implement function `minhash_set` which gets a string consisting of letters A,C,G,T and two parameters  $k$  and  $m$ . It applies `shash()` function on each canonical  $k$ -mer of the input string and returns the set of  $m$  canonical  $k$ -mers with the smallest hash values. You can assume that the input string has at least  $m$  different canonical  $k$ -mers. If this is not the case, raise an exception.
- For partial points, you can build the set of all canonical  $k$ -mers (e.g. by `full_kmer_set`) and choose ones with the smallest `shash()` values. For full points, create an implementation such that all memory used by your function, except for the input string itself, has size  $O(km)$ . Consider using `heapq` data structure from the standard Python library `heapq`.
- Run `minhash_set` on the first 1000 bases of genome 0 for  $k=9$  and  $m=30$  and print the resulting  $k$ -mer set to file `minhash.txt` in lexicographic order, one  $k$ -mer per line. The file should contain 30  $k$ -mers, the first of which is AATATCGGC.
- Create function `minhash_jaccard` which has the same interface as function `jaccard` from Task 1, but it gets two MinHash sets instead of full sets of  $k$ -mers. Therefore, in order to obtain unbiased estimate of the Jaccard similarity, it divides the size of the intersection by the size of one set instead of the union. You can assume that all input sets have the same size; raise an exception if this is not the case.
- Run `minhash_set` on all real genomes for  $k=13$  and  $m=100$  and run `minhash_jaccard` on the real genome 0 compared to the list of all real genomes (including itself). Write the results to file `minhash_jaccard.txt`, each value on one line.

## Task 4: Accuracy of $k$ -mer Jaccard similarity

In this task we want to evaluate if Jaccard similarity on a full set of  $k$ -mers is a suitable measure of sequence similarity. To this end, we will use randomly mutated genomes for different values of mutation probability  $p$ . We will consider  $1-p$  as the true genome similarity (the percentage of bases that were conserved) and see if the Jaccard similarity is in some way related to this value.

1. Write a mathematical formula in your report giving the probability that a

- $k$ -mer at a fixed position in the original genome does not contain any mutation in the mutated version for some value of  $p$  (the formula will use parameters  $k$  and  $p$ ). Calculate this probability for values of  $p$  from `mutation_probabilities` list and values of  $k=9,13,17$ . Include a table of the results in your report.
2. Compute Jaccard similarities of  $k$ -mer sets between genome 0 and all 6 mutated genomes for values of  $k$  listed in tasks 4.1. Include a table of the results in your report.
  3. Create a line plot showing  $1-p$  on the x axis (true genome similarity) and the Jaccard similarity (genome similarity estimate) on the y axis. Use one line for each value of  $k$ ; the line connects values for mutated genomes for increasing value of  $1-p$ . You can also add other plots to support your discussion.
  4. Discuss observations one can make from your tables and plots. For which values of  $k$  do you see the best correspondence between the mutation rate and the Jaccard similarity? Which values of  $k$  are not suitable? Why is that? How is this related to probabilities in Task 4.1 or some other calculations you can make?

## Task 5: Jaccard similarity on real genomes

- Compare genome 0 to all real genomes using Jaccard similarity on a full set of  $k$ -mers for  $k=13$  and include the results in your report.
- Discuss the results. Which other genomes are most similar to genome 0? Can you explain this by looking at their descriptions? Can you make some estimate of the percentage of mutated bases between these pairs of genomes based on your results in Task 4?

## Task 6: Accuracy of minimizers and MinHash

Instead of computing Jaccard similarity for the full set of  $k$ -mers, we can also run it on the set of minimizers, or we can use `minhash_jaccard` on the sets produced by MinHash. Both these methods can be understood as estimates of the Jaccard similarity on the full set of  $k$ -mers using smaller memory. In this task we will compare their accuracy to the Jaccard similarity computed for the full set of  $k$ -mers, which we consider here as “the correct answer”.

1. Compute the similarity of genome 0 to all mutated genomes as in task 4, using  $k=13$  and values of  $w=2,5,10$  for minimizers and values of  $m=10,100,1000$  for MinHash. Include the results as tables in your report.
2. Compute the difference between the Jaccard similarity values for  $k=13$  from task 4 (using all kmers) and the values from task 6.1 and print these differences to a table in your report.
3. Optionally visualize the values using appropriate plots.
4. Discuss your findings. What trends do you see with increasing  $w$  and  $m$ ? What causes these trends? (Explain them based on your understanding of the used algorithms.)

## Task 7: Running time and memory

1. Measure the running time of computing the  $k$ -mer sets on genome 0 using functions `full_kmer_set`, `minimizer_set` and `minhash_set` for parameters  $k$ ,  $w$  and  $m$  as in Task 6.

2. As a proxy for memory consumption, report the size of the sets produced in Task 7.1.
3. List the times and set sizes as tables in your report.
4. Optionally visualize the values using appropriate plots.
5. Discuss your findings. How do the parameter values and the choice of algorithm influence the running time and memory? Can you make some recommendations on which method to choose considering all aspects (accuracy, running time and memory)?