HAL
open science

Approximate Hashing for Bioinformatics

Guy Arbitman, Shmuel Klein, Pierre Peterlongo, Dana Shapira

To cite this version:

Guy Arbitman, Shmuel Klein, Pierre Peterlongo, Dana Shapira. Approximate Hashing for Bioinformatics. CIAA 2021 - 25th International Conference on Implementation and Application of Automata, Jul 2021, Bremen, Germany. pp.1-12. hal-03219482

HAL Id: hal-03219482
https://hal.inria.fr/hal-03219482

Submitted on 6 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Approximate Hashing for Bioinformatics

Guy Arbitman* Shmuel T. Klein* Pierre Peterlongo** Dana Shapira‡

*Bar Ilan University
Dept. of Computer Science
Ramat-Gan 52900, Israel
guy20495@gmail.com
tomi@cs.biu.ac.il

**Inria, Univ Rennes,
CNRS, IRISA
F-35000 Rennes, France
pierre.peterlongo@inria.fr

‡Ariel University
Dept. of Computer Science
Ariel 40700, Israel
shapird@g.ariel.ac.il

Abstract. The paper extends ideas from data compression by deduplication to the Bioinformatic field. The specific problems on which we show our approach to be useful are the clustering of a large set of DNA strings and the search for approximate matches of long substrings, both based on the design of what we call an approximate hashing function. The outcome of the new procedure is very similar to the clustering and search results obtained by accurate tools, but in much less time and with less required memory.

1 Introduction

A particular form of lossless data compression is known as *deduplication*, which is often applied in a scenario in which a large data repository is given and we wish to store a new, updated, version of it. A case in point would be a backup system, which regularly saves the entire content of the digital storage of some company, even though the changes account only for a tiny fraction of the accumulated information. The idea is then to find duplicated parts and store only one copy P of them; the second and subsequent occurrences of these parts can then be replaced by pointers to P. The problem is of course how to define these parts in a useful way, and then how to locate them efficiently.

One of the approaches to solve the problem is based on hashing and can be schematically described as follows. The available data is partitioned into parts called *chunks*; a cryptographically strong hash function h is applied to these chunks, and the set S of different hash values, along with pointers to the corresponding chunks, is kept in a data structure D allowing fast access. These hash values act as *signatures* of the chunks, uniquely representing them, but requiring orders of magnitude less space than the original data. For each new chunk to be treated, its hash value is searched for in D, and if it appears there, we know that the given chunk is a duplicate and may be replaced by a pointer to its earlier occurrence. If the hash value is not in D, the given chunk is considered new, so it is stored and its hash value is adjoined to the set S [14].

An alternative has been suggested in [2] and is implemented in the IBM ProtecTIER Product [7]. The main idea there is to look for *similar*, rather than identical chunks and if such a chunk is located, only the difference is recorded,

which is generally much smaller than a full chunk. This allows the use of significantly larger chunks than in identity based systems. However, for similarity, a classical hashing function cannot be used to produce the signature, since one of the properties of hashing is yielding uniformly distributed values, regardless of regularities in the input, so that when changing even a single bit of the file, the resulting hash value should be completely different.

This lead to the design of what could be called an *Approximate Hash* (AH) function, a notion which seems bearing an internal contradiction, since unlike standard hash functions, their approximate variants should not be sensitive to "small" changes within the chunk, and yet behave like other hash functions as far as the close to uniform distribution of their values is concerned. The idea of AH functions is an extension of the notion of locality-sensitive hashing introduced in [9]. The approach of using similarity instead of identity has been adapted in [3] to applications in which the data is more fine grained, such as backup systems. The current paper is an extension, which applies similar techniques to string processing problems arising in Bioinformatics.

We concentrate in this paper on the following two problems, clustering and substring search, though similar ideas can be applied to a wide variety of other bioinformatic challenges. The first problem is that of *clustering* a large collection of DNA strings into sub-collections forming clusters, in the sense that strings assigned to the same cluster may be considered as similar for practical biological purposes (e.g., one may be obtained from the other by a limited number of mutations), whereas strings of different clusters are different enough to be judged not originating from the same source. Many clustering methods have been suggested, such as CD-HIT (CD) [12], or MeShClust^2 (MC) [10].

The second problem is that of locating a single string within a large collection on the basis of one of its fragments, or rather, one of its fragments that has undergone some limited number of mutations. We show how our notion of an approximate hash may be adapted to these and similar problems and report on the experimental setup and its results in the following sections.

2 Design of an approximate hash function

Before trying to cluster a set of strings, one first needs some measure for the *distance* $d(\omega_1, \omega_2)$ between two given strings ω_1 and ω_2. If they were of equal length n, the *Hamming distance*, counting the number of corresponding positions in which the strings differ, would be a plausible candidate, and can be computed in $O(n)$. However, the Hamming distance is biased when insertions and deletions are allowed and is a reasonable choice only when ω_2 can be obtained from ω_1 by a series of substitutions. Therefore, in a general setting, one should rather use the *edit distance*, defined as the minimal number of single character insertions, deletions or substitutions necessary to transform one string into the other. Using dynamic programming, it takes quadratic time $O(nm)$ to compute the edit distance between strings of lengths n and m. The clustering problem is thus a difficult one: if a million (p) strings are given, each of length

about one million (q), the time to evaluate the edit distance between all pairs of strings would be $O(p^2q^2) = O(2^{80})$, which is still too much for our current technology. We therefore suggest a more practical solution as follows.

2.1 Definition of the signature

The idea is, given a collection $\mathcal{C}$ of DNA strings ω, to produce a signature encapsulating the main features of the strings in as few as possible bits. A first approach could be to devise what could be called an *occurrence map* of the various substrings of length k, called k-mers, for $k \geq 1$, of all strings ω in $\mathcal{C}$. Since our alphabet consists of just four nucleotides represented by the 4 letters, $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$, there are 4 1-mers, 16 2-mers, 64 3-mers and generally 4^k different k-mers. Depending on the available space, a general approach to devise a signature could include the following steps:

1. Fix lower and upper limits ℓ and u for the values of k we wish to include in the definition of the signatures, each of which will consist of a bitstring of length $4^\ell + 4^{\ell+1} + \cdots + 4^u$;
2. iterate over all the DNA strings ω in the given set $\mathcal{C}$ and perform for each string:
   (a) Choose a threshold t_k for each of the values of k, depending only on k and the lengths of the given DNA string ω;
   (b) sort, separately for each $\ell \leq k \leq u$, the 4^k k-mers according to some predefined order, e.g., lexicographically;
   (c) for all k in $[\ell, u]$, the bit indexed $i + \sum_{j=\ell}^{k-1} 4^j$, $0 \leq i < 4^k$, corresponding to the i-th ordered k-mer, will be set to 1 if and only if the number of occurrences of this i-th k-mer within the given string ω is at least t_k. For example, AAAA is the first 4-mer in lexicographic order, so if $\ell = 2$, then the bit indexed $4^2 + 4^3 + 0 = 80$ will be set if the number of occurrences of AAAA in the string ω is at least t_4.

A reasonable choice for the thresholds t_k would be the median of the number of occurrences of the 4^k k-mers within the given string ω, for each k, which would yield signature strings in which the probability of a 1-bit is about $\frac{1}{2}$. Since this is only a heuristic, the median can be approximated by setting t_k as the expected number of occurrences, that is, their average, which is easier to evaluate.

As example, consider the input string

ACCTTGAAGTTGGGCCAACTGTTGCCC

of length $n = 27$ and set $\ell = u = 2$. The number of occurrences of the 16 possible pairs are:

AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
2	2	1	0	1	4	0	2	1	2	2	2	0	0	4	3

There are $n-k+1$ overlapping k-mers in a string of length n, so the average number of occurrences for each specific k-mer is $(n-k+1)/4^k = 1.63$ on our small example. One could thus set the threshold to $t_2 = 1$, for which the resulting signature would be 1110 1101 1111 0011, where spaces have been included for readability. For $t_2 = 2$, one would get 1100 0101 0111 0011.

By concentrating on the distribution of the different k-mers within a string we try to catch underlying similarities, since DNA strings that are essentially different not just because of a limited number of mutations, will not tend to exhibit matching occurrence distributions. On the other hand, the proposed measure is flexible enough to allow some fluctuations, because the exact number of occurrences of a given k-mer is not given importance, only the fact whether or not this number exceeds the given threshold.

The idea of using k-mers to derive features of entire DNA strings is not new to Bioinformatics, and has been used in [8, 16, 5], to cite just a few, though, our approach is different.

2.2 Clustering

To extend the approach used for the deduplication of chunks, we shall apply here the clustering on the signatures rather than on the corresponding DNA strings, in order to obtain clusters from which the partition of the original set of strings can be deduced. There is obviously a significant reduction of the required time complexity, turning the clustering attempt into a feasible one. In particular, instead of using the edit distance between two strings, the appropriate choice for the distance between their signatures is the Hamming distance, as the signatures are of the same length and bits at the same index correspond to identical k-mers.

To check whether one can indeed identify clusters on the basis of using just the much smaller signatures, we report here on the details of a series of tests we have performed, first on artificially constructed sets, then real-life data. Even for the first set, we started with real DNA strings, downloaded from the website of the *National Center for Biotechnology Information*[1], and only the modified strings simulating data after mutations, were artificially generated. A sample of 50 different DNA sequences of various lengths and origins was randomly chosen, with lengths between thousands and millions of nucleotides. For each of the chosen strings, 15 variants, partitioned into three groups of 5, were generated, simulating various mutations. The first group consisted of strings derived from the given one by deleting some of their characters. More precisely, the heuristic used to produce the strings was:

1. Choose randomly an integer r between 1 and 50;
2. choose randomly a position t within the given string;
3. delete r consecutive characters starting from position t;
4. if the cumulative number of deleted characters does not exceed 7% of the length of the original string, repeat the process from step 1.

[1] https://www.ncbi.nlm.nih.gov/nuccore

For the second group, a similar heuristic was applied, but instead of deleting, substrings of length r chosen randomly from the alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$ were inserted at position t. The five strings in the third group were obtained by allowing both deletions and insertions, more precisely, applying first the heuristic for the deletion and then that of the insertion on the output of the first, until the cumulative change reaches about 14%. The total number of strings in our set $\mathcal{S}$ was therefore 50 originals plus 50×15 variants $= 800$.

Generating artificial DNA strings by inserting or deleting elements is often used for simulations in Bioinformatics, as, e.g., in [13]. We wish to emphasize that there is no claim, nor does it really matter, that the 50 chosen elements be representive of the entire NCBI database of more than 200 million sequences. We could have just as well started with arbitrarily produced strings.

The aim of the test was to check whether after applying our approximate hash function ah, the generated signatures $ah(\omega)$ for $\omega \in \mathcal{S}$ carry enough information of the DNA strings ω they were produced from to identify the natural clusters, each consisting of one of the 50 randomly chosen strings and its 15 variants. We thus took the 800 signatures and calculated the Hamming distance between each of the $\frac{1}{2}800 \times 799 = 319,600$ pairs. We used $\ell = 2$ and $u = 4$ on our tests, yielding signatures of length $4^2 + 4^3 + 4^4 = 336$ bits. Choosing $u = 5$ would already require 1024 more bits for each signature, a significant increase, for getting only moderately better results. As mentioned above, the thresholds t_k were chosen as the expected number of occurrences of a specific k-mer, which is $(n - k + 1)/4^k$, where n is the lengths of the string.

	A	A.i.1	A.i.2	A.d.1	A.d.2	A.di.1	A.di.2	B	B.i.1	B.i.2	B.d.1	B.d.2	B.di.1	B.di.2	C	C.i.1	C.i.2	C.d.1	C.d.2	C.di.1	C.di.2	D	D.i.1	D.i.2	D.d.1	D.d.2	D.di.1	D.di.2	E	E.i.1	E.i.2	E.d.1	E.d.2	E.di.1	E.di.2
A		0.5	0.2	0.2	0.8	0.0	1.7	19.6	19.0	19.0	19.0	20.5	19.3	22.0	12.7	12.7	13.0	13.0	13.3	12.7	12.2	13.6	14.2	12.7	13.6	13.3	13.9	13.0	13.0	13.0	13.6	13.0	14.5	13.9	13.9
A.i.1			0.8	0.2	1.4	0.5	2.3	19.6	19.0	19.0	19.0	20.5	19.3	22.0	12.2	12.2	12.5	12.5	12.7	12.2	11.6	13.0	13.6	12.7	13.0	12.7	13.3	12.5	13.0	13.0	13.6	13.0	14.5	13.9	13.9
A.i.2				0.5	1.1	0.2	2.0	19.9	19.3	19.3	19.3	20.2	19.6	21.7	12.5	12.5	12.7	12.7	13.0	12.5	11.9	13.3	13.9	12.5	13.3	13.0	13.6	12.7	13.3	13.3	13.9	13.3	14.8	14.2	14.2
A.d.1					1.1	0.2	2.0	19.3	18.7	18.7	18.7	20.2	19.0	21.7	12.5	12.5	12.7	12.7	13.0	12.5	11.9	13.3	13.9	12.5	13.3	13.0	13.6	12.7	13.3	13.3	13.9	13.3	14.8	14.2	14.2
A.d.2						0.8	0.8	19.9	19.3	19.3	19.3	20.8	19.6	22.3	13.0	13.0	13.3	13.3	13.6	13.0	12.5	13.9	14.5	13.0	13.9	13.6	14.2	13.3	13.9	13.9	14.5	13.9	15.4	14.8	14.8
A.di.1							1.7	19.6	19.0	19.0	19.0	20.5	19.3	22.0	12.7	12.7	13.0	13.0	13.3	12.7	12.2	13.6	14.2	12.7	13.6	13.3	13.9	13.0	13.0	13.0	13.6	13.0	14.5	13.9	13.9
A.di.2								20.8	20.2	19.6	19.6	21.1	20.5	22.0	12.7	12.7	13.0	13.0	13.9	12.7	12.2	14.8	15.4	13.9	14.8	14.5	15.1	13.6	14.2	14.2	14.2	13.6	15.7	14.5	15.1
B	2.88%								1.1	1.7	1.1	2.6	2.0	3.5	19.3	19.9	19.6	20.2	19.9	19.9	20.5	20.2	20.2	19.9	20.2	19.9	20.5	20.8	23.8	23.2	23.2	24.4	23.5	24.1	24.7
B.i.1										1.7	1.1	2.6	1.4	3.5	18.7	19.3	19.0	19.6	19.3	19.3	19.9	20.2	20.2	19.9	20.2	19.9	20.5	20.8	23.2	22.6	22.6	23.8	22.9	23.5	24.1
B.i.2											1.1	2.0	2.0	2.9	19.3	19.9	19.6	20.2	19.9	19.9	20.5	20.2	20.2	19.9	20.2	19.9	20.5	20.2	23.2	22.6	22.6	23.8	22.3	23.5	23.5
B.d.1												2.0	2.0	2.9	18.7	19.3	19.0	19.6	19.3	19.3	19.9	20.2	20.2	19.9	20.2	19.9	20.5	20.2	22.6	22.0	22.0	23.2	22.3	22.9	23.5
B.d.2													2.9	1.4	19.6	20.2	19.9	20.5	20.2	20.2	20.8	20.5	20.5	20.2	20.5	20.2	20.8	20.5	24.1	23.5	23.5	24.7	23.2	24.4	24.4
B.di.1														3.2	19.0	19.6	19.3	19.9	19.6	19.6	20.2	20.5	20.5	20.2	20.5	20.2	20.8	21.1	23.5	23.5	22.9	24.1	23.2	24.4	24.4
B.di.2															20.5	21.1	20.8	21.4	21.1	21.1	21.7	22.0	22.0	21.7	22.0	21.7	22.3	22.0	25.5	25.0	25.0	26.1	24.7	25.8	25.8
C	3.41%							3.31%								1.1	0.2	0.8	1.1	0.5	1.1	8.0	8.6	8.3	8.0	7.7	8.3	8.0	12.2	12.7	11.6	12.2	13.6	13.0	13.6
C.i.1																	1.4	0.8	1.1	1.1	2.3	8.0	8.6	7.7	8.0	7.7	8.3	7.4	11.6	12.2	11.0	11.6	13.0	12.5	13.0
C.i.2																		1.1	0.8	0.8	1.4	8.3	8.9	8.6	8.3	8.0	8.6	8.3	11.9	12.5	11.3	11.9	13.3	12.7	13.3
C.d.1																			0.8	0.2	1.4	7.1	7.7	7.4	7.1	6.8	7.4	7.7	12.5	13.0	11.9	12.5	13.9	13.3	13.9
C.d.2																				1.1	2.3	7.4	8.0	7.7	7.4	7.1	7.7	7.4	11.6	12.2	11.6	12.2	13.0	13.0	13.0
C.di.1																					1.1	7.4	8.0	7.7	7.4	7.1	7.7	8.0	12.7	13.3	12.2	12.7	14.2	13.6	14.2
C.di.2																						8.0	8.6	8.3	8.0	7.7	8.3	8.6	12.2	12.7	12.2	12.2	13.6	13.0	13.6
D	3.94%							3.47%							4.81%								0.5	1.4	0.0	0.2	0.2	1.7	13.6	13.6	13.6	14.2	14.5	14.5	15.1
D.i.1																								1.4	0.5	0.8	0.2	2.3	14.2	14.2	14.2	14.8	15.1	15.1	15.7
D.i.2																									1.4	1.7	1.7	1.4	13.3	13.3	13.3	13.9	14.2	14.2	14.8
D.d.1																										0.2	0.2	1.7	13.6	13.6	13.6	14.2	14.5	14.5	15.1
D.d.2																											0.5	2.0	13.9	13.9	13.9	14.5	14.8	14.8	15.4
D.di.1																												2.0	13.9	13.9	13.9	14.5	14.8	14.8	15.4
D.di.2																													12.5	12.5	12.5	13.0	13.3	13.3	13.9
E	3.25%							2.72%							3.50%							4.15%								1.1	1.1	0.5	1.4	1.4	2.0
E.i.1																															1.7	1.1	2.0	1.4	2.0
E.i.2																																1.1	2.6	2.0	3.2
E.d.1																																	2.0	0.8	2.0
E.d.2																																		1.7	1.7
E.di.1																																			2.3
E.di.2																																			

Table 1. Sample of normalized Hamming distances between signatures.

Table 1 displays a sample of these distances, showing, in the upper right triangle, only the results for the original strings indexed $j \in \{A, B, C, D, E\}$[2], and for each of these, two variants of each of the 3 groups, identified by $j.d.r$, $j.i.r$ and $j.di.r$ for the strings obtained by deletion, insertion and both, respectively, with $r \in \{1, 2\}$ giving the index of the variant within its group. For a pair $\omega_1, \omega_2 \in \mathcal{S}$, the displayed value is the normalized Hamming distance between $ah(\omega_1)$ and $ah(\omega_2)$, that is, the number of 1-bits in $ah(\omega_1)$ XOR $ah(\omega_2)$ divided by 336, expressed as percentage. For visibility, cells containing values below 10% have been shaded in light green and the others in red. The lower left triangle contains, for each pair, a measure for the similarity of the original DNA strings. We chose the number of shared canonical 11-mers, as percentage of their total number, averaged for the two members of the pair.

One can see on this sample, which is representative for the entire 800×800 matrix, that while the distances between elements within the set of variants of the same original string are all small, all inter-set distances are much larger, so that one may conclude that using the signatures instead of the much longer original strings to perform the clustering process may be justified. A noteworthy exception are the sets produced by the strings indexed C and D, for which the pair-wise distances are only about 7–8%. This is in accordance with the corresponding similarity measure of 4.81%, the only one exceeding a threshold of 4.5% (in green), and may be explained by the fact that the DNA strings were similar to begin with, being related to the same parasite.

As a control experiment, we also applied a real hash function instead of our approximate one. The choice was MD5 [15], for which all the values of the matrix were between 0.37 and 0.62, so that, as expected from a hash function, MD5 did not detect any of the clusters and would thus not be useful in this context.

To enable a fair comparison with alternative clustering methods, we took the same test collection as the one used for MC by [10] as second set of DNA strings: the top-level FASTA sequences containing one chromosome from Ensembl Genomes release 35 [6], a set of 3670 bacteria genomes taken from a collection of about 42,000. The size of the sequences varied from 114KB to 15MB, with an average of 3.5MB.

We need a measure to compare the outcome of different clusterings A and B. Note that this measure is not symmetric: A is considered to be the base scenario, and we shall use MC as defining it, and B is a suggested new clustering method, one derived from ah in our case, and we wish to assess how much B deviates from A. It is acceptable that A should be a refinement of B, that is, every cluster in A is included in one of the clusters of B, but if a cluster of A is split over several different ones in B, we consider this as an error. Iterating over all the clusters c of A, we accumulate the *error counts* of c, defined as the difference of the size of cluster c with that of the largest intersection of c with one of the

[2] the names of these 5 strings in the database are:
A - KV453883.1, B - NZ_DS996920.1, C - UPTC01000856.1, D - UPTC01000985.1, E - VAHF01000278.1

clusters of B. Finally, we define the normalized *error rate* by dividing the sum of the error counts by the number of sequences.

Fig. 1. Comparing two different clusterings A and B

For example, consider 6 sequentially indexed strings. Figure 1 shows the clustering performed by a clustering A into clusters x, y and z, and by B into clusters a, b and c. We see that B has merged the clusters x and z into a, but split the cluster y into b and c. Thus for clusters x, z, which are subsets of a, there is no error, but cluster y is not a subset of any cluster of B, and the largest intersection is with cluster b of B. This yields an error rate of $\frac{1}{6}$.

The reason for preferring such an asymmetric measure is that we intend using the clustering derived from our *ah* function in a preliminary filtering stage, on the outcome of which some other clustering can then be applied, with significantly reduced complexity. If several clusters of A are entirely included in a single cluster c of B, this is acceptable because the A clustering will anyway be applied on c after the filtering stage. Table 2 brings the comparative results. All tests were run on a Dell XPS 15 7590 with 32GB RAM i9-9980HK @ 5.0GHz, running Ubuntu 18.04.

Method	Running time (mm:ss)	Memory (MB)	Error rate	Number of clusters	Max size of cluster
MC	16:25	31744	–	1861	176
ah – CC	1:08	107	0.35%	861	1053
MC after *ah*	14:12	8200	2.02%	1862	177

Table 2. Comparison of clustering on 3670 strings from a Bacteria database

We see that while there is a significant reduction in both time and required RAM when replacing MC by a simple Connected Component (CC) clustering[3] based on our *ah* signatures rather than the original DNA strings, this comes at a price of only marginally hurting the resulting clusters themselves, with an error rate of less than 1% of falsely assigned sequences. Even if we use *ah* only as a preliminary filter, the processing time is improved and the memory consumption

[3] each string is a vertex, and vertices are connected by an edge if the distance between them is smaller than some threshold

is cut to a quarter, whereas the error rate is just 2%, and 96% of the clusters match those produced by MC alone. The table shows also that while most clusters are small, there are also some larger ones. The large difference in the number of clusters in spite of a low error rate implies that most MC clusters are entirely included in *ah* ones. We tried also to apply CD as alternative clustering, but had to abort its run after 24 hours without results.

To enable also a comparison with CD to run in reasonable time, we limited the lengths of the strings in our third test set to be between 50 and 100K and retrieved 4523 strings of viruses from the GenBank database[4]. The results appear in Table 3.

Method	Running time (mm:ss)	Memory (MB)	Error vs CD	Error vs MC	Number of clusters	Max size of cluster
CD	38:44	1500	–	–	3774	67
MC	00:54	2200	–	–	1934	208
ah – CC	00:04	160	0.71%	0.08%	1158	794
CD after *ah*	22:04	1115	0.75%	–	3804	67
MC after *ah*	00:45	265	–	0.80%	1945	208

Table 3. Comparison of clustering on 4523 strings of a Virus database

The conclusions are similar to those for the set of bacteria DNA strings. CD is much slower than MC but requires only 1.5GB of RAM instead of 2.2GB and produces about twice as many clusters. If *ah* is used as a preliminary clustering, 99% and 98% of the original clusters are recovered for CD and MC, respectively.

2.3 Searching for a string including some read

In the problem we consider here, a large collection $\mathcal{C}$ of strings is given, where both the size of $\mathcal{C}$ and that each of its individual elements may be of the order of millions and more. In addition, we are given a read R whose length could be in the thousands, and we wish to retrieve the subset of elements $C_i \in \mathcal{C}$ for which R is a substring of C_i. Actually, the notion of being a substring has to be understood in a broader sense, as we allow a limited number ℓ of mismatches. If $\ell = 0$, this is the *exact matching* problem that has been thoroughly investigated. For general ℓ, the problem is much more difficult; the best deterministic algorithm has a complexity proportional to $n\sqrt{\ell \log \ell}$ [1], which is not reasonable for large values of ℓ. A faster probabilistic algorithm, running in time $O(n \log n)$ can be found in [4], where n is the total length of the strings.

Our approach here is similar to the Karp-Rabin probabilistic algorithm for string matching [11], but using our approximate hash function instead of simple

[4] `https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/`

hashing modulo a large prime number. A brute force approach would be to compare R with the substrings $C_i^j = C_i[j]C_i[j+1]\cdots C_i[j+m-1]$ of length $m = |R|$ of C_i starting at position j, for all $C_i \in \mathcal{C}$ and all possible values of j, $1 \le j < n_i - m$, where $n_i = |C_i|$ is the length of C_i. This yields a complexity of mn, with n the total length of all the strings in $\mathcal{C}$, which may be prohibitive for the intended application. Instead, we suggest applying the approximate hash function ah to both R and the substrings C_i and compare the results. A Hamming distance *above* some threshold is a clear indication that the pattern R does not occur at the given position in C_i, yet being *below* does not guarantee that it does appear there. Nevertheless, the function ah can serve as a filter, allowing us to restrict a full comparison of R with substrings of C_i only to indices at which a match has been declared.

At first sight, for calculating $ah(C_i^j)$ for all i and j, one needs $O(nm)$ operations, so there seems to be no gain by applying the approximate hash. Note, however, that the value of $ah(\omega)$ for a string ω is defined as a function of the statistics of occurrences of the different k-mers forming the string, and it does not matter where exactly they occur in ω. One can thus easily evaluate $ah(C_i^{j+1})$ as a function of $ah(C_i^j)$ in constant time, because of the large overlap of size $m-1$ they share, just as in the Karp-Rabin algorithm. The global complexity may therefore be reduced to $O(n+m)$.

The strategy of replacing comparisons between long reads by comparisons of the much shorter ah signatures will only be useful if, for DNA string fragments ω_1 and ω_2, there is a strong enough correlation between the edit distance $d(\omega_1, \omega_2)$ and the corresponding Hamming distance $\mathsf{HD}(ah(\omega_1), ah(\omega_2))$. Note that we do obviously not expect a perfect match and that the edit distance between strings could be replaced by the HD between their signatures, so that

$$d(\omega_1,\omega_2) < d(\omega_3,\omega_4) \Longleftrightarrow \mathsf{HD}(ah(\omega_1), ah(\omega_2)) < \mathsf{HD}(ah(\omega_3), ah(\omega_4)).$$

This is theoretically impossible because the signatures are shorter and thus cannot carry the same amount of information content. Even requesting just a weak inequality on the right hand side would not be realistic, and for a fixed edit distance, the corresponding HD values might fluctuate. We do, however, expect, that in spite of these fluctuations, the results may be partitioned into regions allowing to derive some cut-off points, that is, that d is small if and only if the corresponding HD is small, for some reasonable definition of smallness. The following experiment illustrates the validity of this assumption.

A sample of $s = 200$ strings has arbitrarily been chosen from the bacteria database, and the normalized edit distance has been evaluated for each of the $\binom{s}{2}$ pairs. The increasing purple line in Figure 2 shows these values as function of their rank, after having sorted them in non-decreasing order. The blue line plots the corresponding Hamming distances between the ah values for the same pairs. Though these values are strongly fluctuating, one can still identify a clear cutoff point at about 3200, separating the plot into two regions with distinct and different extreme values for the Hamming distance. This fact enables the

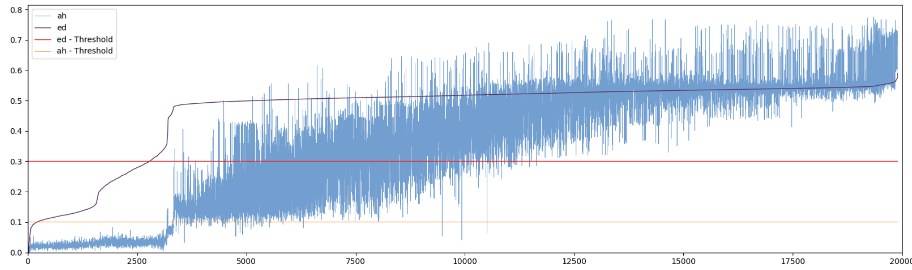

Fig. 2. Comparing edit distance between strings with Hamming distance on corresponding signatures. The x-axis is the index in the sorted sequence of 19900 pairs of strings.

definition of thresholds for both edit and Hamming distances, and we chose 0.3 (in red) for the former, and 0.1 (in yellow) for the latter.

It will be convenient to describe our experiment borrowing the vocabulary of the Information Retrieval field. We are looking for *relevant* pairs, defined here as those for which their edit distance is below the chosen threshold, but we choose them by means of the Hamming distance between their signatures, so the *retrieved* pairs are those for which the HD is below the threshold. The outcome is color coded in Figure 3 showing the matrix of all the pairs. True positive results are those for which both distances are below their thresholds and are shown in light green; true negatives (both distances above their thresholds) appear in blue. Erroneous outcomes are false positives, shown in red, with a low HD in spite of a large edit distance, and false negatives (HD above in spite of edit distance below), which is not shown in this example — not a single pair fell into this category.

ah threshold	edit dist threshold	Precision	Recall
0.125	0.275	0.61	1
0.1	0.3	0.82	1
0.075	0.35	0.97	0.99

Table 4: Recall and Precision for various threshold settings.

Table 4 displays the Recall / Precision values obtained for various settings of the two chosen thresholds. Recall is the fraction of the relevant items that have actually been retrieved, precision is the fraction of the retrieved items that are indeed relevant. We see that recall is very close to 1, as there are very few false negative results, and precision can also be very high for well chosen thresholds.

Our last experiment directly checked the applicability of the approximate hash approach to searching a long read in a DNA string. An arbitrary string C

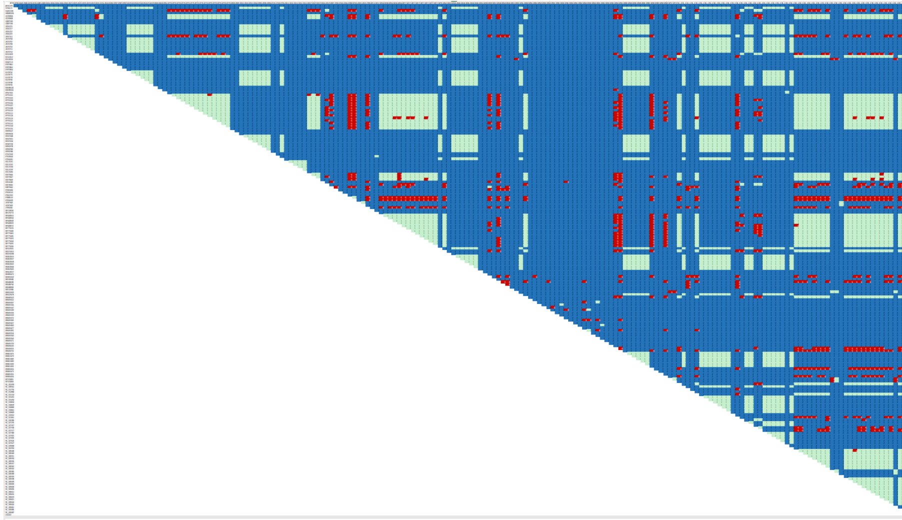

Fig. 3. Comparing pair distances: Blue – true positive; green – true negative; red – false positive; no false negative

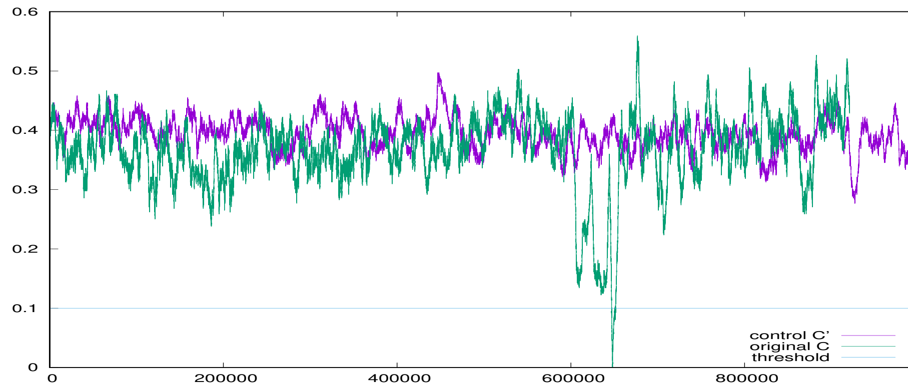

Fig. 4. Searching for a substring by comparing signatures.

of length about 900K was chosen from the bacteria set, as well as a substring R of length $m = 5000$ starting at an arbitrary position (at about 650K), serving as pattern to be located. Figure 4 plots in green, as function of i, the normalized Hamming distance between $ah(R)$ and $ah(C_i)$, where C_i is the substring of length m of C starting at i. We see that there is only a very narrow region for which the distance is practically zero. As a control experiment, the search for the same pattern was repeated with a different DNA string C' of length about 1M, yielding the purple curve with not a single value even approaching zero.

The search procedure will thus declare a match if the HD is below some threshold, symbolized in Figure 4 by the blue line. It should be emphasized that if there are indeed matches, the procedure will find them all, and the possible errors are only to declare non-existing matches. However, the validity of the match can be verified, since we know where to check. We conclude that using the *ah* is a powerful tool significantly reducing the amount of work while only marginally affecting the quality of the results.

References

1. Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.
2. Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR*, page 6, 2009.
3. Lior Aronovich, Ron Asher, Danny Harnik, Michael Hirsch, Shmuel T. Klein, and Yair Toaff. Similarity based deduplication with small data chunks. *Discrete Applied Mathematics*, 212:10–22, 2016.
4. Mikhail J. Atallah, Frédéric Chyzak, and Philippe Dumas. A randomized algorithm for approximate string matching. *Algorithmica*, 29(3):468–486, 2001.
5. V.B. Dubinkina, D.S. Ischenko, V.I. Ulyantsev, A.V. Tyakht, and D.G. Alexeev. Assessment of k-mer spectrum applicability for metagenomic dissimilarity analysis. *BMC Bioinformatics*, 17,38, 2016.
6. Paul Julian Kersey et al. Ensembl Genomes 2018: an integrated omics infrastructure for non-vertebrate species. *Nucleic Acids Research*, 46(D1):D802–D808, October 2017.
7. Michael Hirsch, Haim Bitner, Lior Aronovich, Ron Asher, Eitan Bachmat, and Shmuel T. Klein. Systems and methods for efficient data searching, storage and reduction, U.S. Patent 7,523,098, issued April 21, 2009.
8. Michael Höhl, Isidore Rigoutsos, and Mark Ragan. Pattern-based phylogenetic distance estimation and tree reconstruction. *Evol. Bioinform. Online*, 2:359–75, 2006.
9. Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th STOC*, pages 604–613, 1998.
10. Benjamin T. James and Hani Z. Girgis. MeShClust^2: Application of alignment-free identity scores in clustering long DNA sequences. *bioRxiv*, 451278, 2018.
11. Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
12. Weizhong Li and Adam Godzik. cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658–1659, 2006.
13. Burkhard Morgenstern, Binyao Zhu, Sebastian Horwege, and Chris Leimeister. Estimating evolutionary distances between genomic sequences from spaced-word matches. *Algorithms Mol. Biol.*, 10:5, 2015.
14. Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proc. FAST '02 Conference on File and Storage Technologies*, pages 89–101, 2002.
15. Ronald L. Rivest. The MD5 message-digest algorithm. *RFC*, 1321:1–21, 1992.
16. Zhong Wang et al. A new method for rapid genome classification, clustering, visualization, and novel taxa discovery from metagenome. *BioRxiv*, 812917, 2019.