Supplementary Data

I. How iDoComp handles the different chromosomes that compose the genome

As stated in the manuscript, iDoComp is composed of three stages: the mapping generation, the post-processing of the mapping, and the entropy encoder. In the first step, given a target sequence T and a reference sequence S, the algorithm constructs the sequence of matches $\mathcal{M}$, using for that a previously computed suffix array of the reference. Note that the sequence of matches $\mathcal{M}$ suffices to reconstruct the target sequence T, as long as the sequence S is available for decompression. In the second step a post-processing step is performed in $\mathcal{M}$, and as a result, the size of $\mathcal{M}$ is reduced and two new sets $\mathcal{S}$ and $\mathcal{I}$ are created. Finally, the third step compresses $\mathcal{M}$, $\mathcal{S}$ and $\mathcal{I}$ with an entropy encoder.

However, a question that remains to be answered is how iDoComp handles the different chromosomes throughout the different steps. In iDoComp, we generate a sequence of matches $\mathcal{M}$ (and subsequently the sets $\mathcal{S}$ and $\mathcal{I}$) for each chromosome of the target, given the corresponding chromosome of the reference. We assume the correct pairing of chromosomes is provided, i.e., chromosome 1 of the reference is used to compress chromosome 1 of the target, and so on. We also compute off-line a suffix array for each of the chromosomes that compose the reference genome. Thus, the algorithm generates the triplet $\mathcal{M}$, $\mathcal{S}$, $\mathcal{I}$ for each of the chromosomes, i.e., steps one and two are performed sequentially for each of the chromosomes. Finally, all the instructions generated for the different chromosomes are fed to the entropy encoder (third step), which generates a unique binary file for all of them.

II. Generation of the sequence of matches $\mathcal{M}$

Given a target sequence T and a reference sequence S, the first step of the algorithm is the mapping generation, i.e., the construction of the sequence of matches $\mathcal{M}$. For that, the algorithm makes use of a suffix array previously computed for the sequence S. In this section we describe this step, also denoted in the manuscript as a "greedy mapping", in more detail.

Recall that the sequence of matches $\mathcal{M}$ should suffice to reconstruct the sequence T given the reference sequence S. Specifically, concatenating the elements of $\mathcal{M}$ we should be able to generate T perfectly. Recall that a match m of $\mathcal{M}$ is composed of the triplet (p, l, c), where p indicates the position of the reference where the match starts, l denotes the length of the match, and c the character that appears in the target right after the match.

Assume the reference sequence S and the target sequence T are given by

$$S = \texttt{ACCTGCCATTAGCCTAGGCAATGCCGTATGGTGGGGCCAAAT}$$
$$T = \texttt{ACCAGCCATTGGCCGAGGCAATATGGTAAAAGGGGCCC}$$

The algorithm starts at position 1 of T, and finds the longest match that appears in the reference S. If more than two places in the reference contain the longest match, the algorithm picks one of them at random. The suffix array is a powerful tool that helps the algorithm find the longest match in a reasonable time. In the above example, the algorithm will find that the longest match is `ACC`, as `ACCC` is not found in the reference. After this string in the reference, which starts at position 1, there is a `T`, whereas in the target there is an `A`. Thus the first match in $\mathcal{M}$ is $m_1 = (1, 3, \mathrm{A})$. Note that with m_1 we can reconstruct T up to position 4. Thus we repeat the previous step but now from position 5 in the target, and find that the longest match is `GCCATT`. So we have $m_2 = (5, 6, \mathrm{G})$. If we continue this process, we will find that

the sequence of matches $\mathcal{M}$ is given by

$$
\begin{aligned}
m_1 &= (1, 3, \texttt{A}) \\
m_2 &= (5, 6, \texttt{G}) \\
m_3 &= (23, 4, \texttt{A}) \\
m_4 &= (17, 6, \texttt{A}) \\
m_5 &= (29, 4, \texttt{A}) \\
m_6 &= (39, 3, \texttt{G}) \\
m_7 &= (34, 5, \texttt{C})
\end{aligned}
$$

III. Post-Processing step of iDoComp

In this section we show in more detail the post-processing step done by iDoComp to create the set $\mathcal{S}$. The flowchart is depicted in Figure 1.

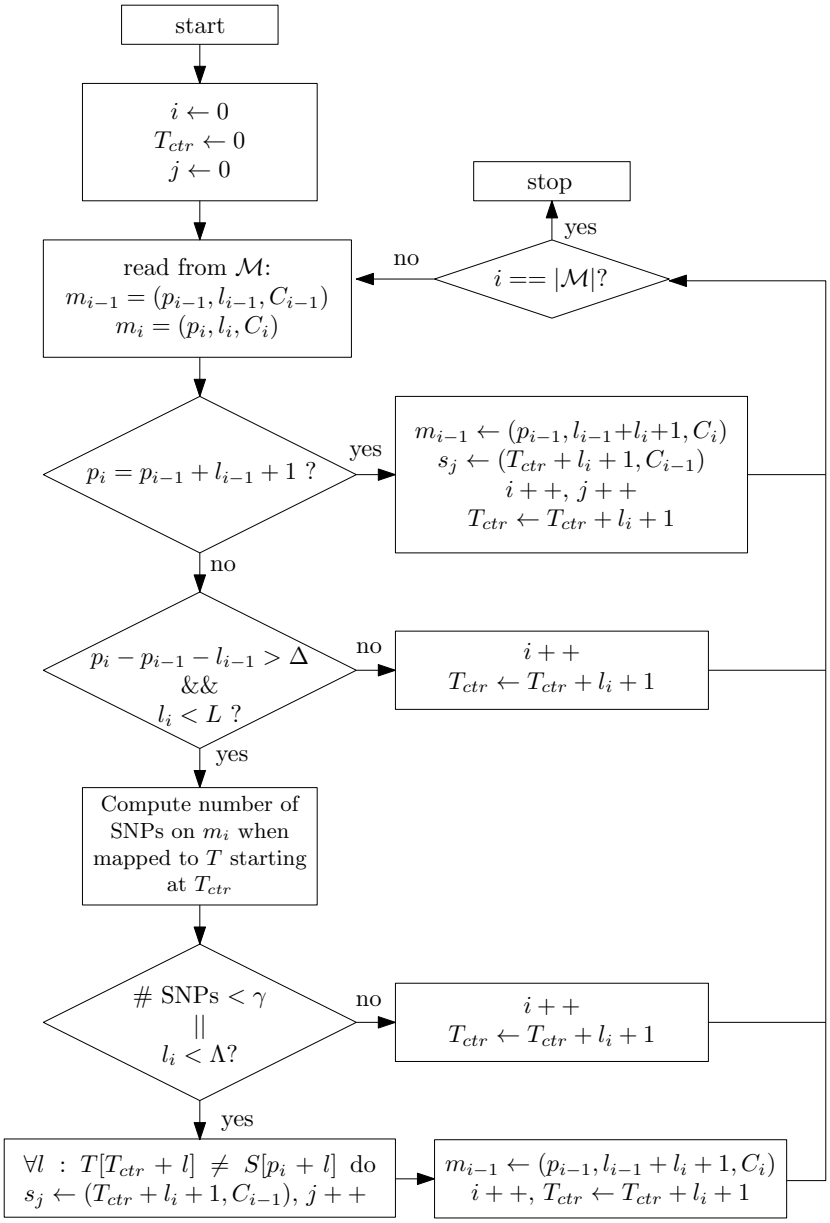

Fig. 1. Flowchart of the post-processing step of the proposed algorithm that generates the set $\mathcal{S}$.

IV. Program parameters used for the simulations

In this section we provide more details regarding the default parameters used in iDoComp in all the simulations presented in the manuscript. We also specify the versions of the programs used to compare the performance of the proposed algorithm iDoComp.

1) iDoComp:
   As explained in the paper, we used the default parameters for all the simulations. Specifically, the following configuration is hard-coded in the code:
   a) $\Delta = 10$
   b) $L = 400$
   c) $\rho = 25$
   d) $\Lambda = 20$
2) GDC:

Variant	Not H. Sapiens datasets	H. Sapiens datasets
Normal	-ma1000000000 -rn1	-ma1000000000 -mp20,4,20 -rn1 -hs8
Advanced	-rn1	-mp20,4,20 -rn1 -hs8
Ultra	-ma1000000000 -rn40	-ma1000000000 -mp20,4,20 -rn3 -hs8

TABLE I
PARAMETERS USED BY THE DIFFERENT VARIANTS OF GDC.

We used the version v.0.2 in all the simulations. The commands for the variants *normal*, *advanced* and *ultra* are specified in Table I (the user can also refer to the Supplementary Data provided in [Deorowicz *et al*., 2011]).

3) GReEn:
We used the version of the algorithm provided in *ftp://ftp.ieeta.pt/∼ap/codecs/GReEn1.tar.gz*, with the default options.
4) GRS:
We used the version v.1.0 in all the simulations, with the default options.
5) Algorithm presented in [Chern *et al*., 2012]:
We used the version provided by the authors, with the default options.

V. TIME AND MEMORY USAGE TO GENERATE THE SUFFIX ARRAYS

In this section we give more details regarding the time and memory needed to generate the suffix arrays of those datasets considered in the paper as references. To generate the suffix array, we used package *sais-lite-2.4.1*, which can be found in [https://sites.google.com/site/yuta256/sais/]. The results regarding the running time are shown in Table II. Note that the suffix array was used solely as a tool for compression, and it is not the main contribution of our work. Thus, we did not optimize its generation in terms of running time.

Species	Chr.	Assembly	Size [MB]	Suffix Array Time [sec]
L. pneumohilia	1	NC_017525.1	2.7	1
E. coli	1	NC_017651.1	5.1	1
S. cerevisiae	17	sacCer2	12.4	3
C. elegans	7	ce6	102.3	34
A. thaliana	7	TAIR9	121.2	46
Oryza sativa	12	TIGR5.0	378.3	145
D. melanogaster	6	dmelr31	119.1	44
H. sapiens	25	hg18	3,100	1,335
H. sapiens	25	hg19	3,100	1,238
H. sapiens	25	KOREF_20090131	3,100	1,252
H. sapiens	25	YH	3,100	1,271

TABLE II
TIME NEEDED TO GENERATE THE DIFFERENT SUFFIX ARRAYS.

Regarding the memory consumption of the suffix array generation, we give numbers for the *H. Sapiens* datasets, as they are the biggest ones and therefore the more representative. Specifically, the memory needed to generate the suffix array of each of the chromosomes of the H. sapiens datasets is at most 1.1 GB (this is the case of chromosome 1, which is the largest).

Species	Raw Size	[Chern *et al*., 2012]			iDoComp			Gain
	[MB]	size [KB]	C. time	D. time	size [KB]	C. time	D. time	
L. pneumohilia	2.7	0.062	1	1	**0.084**	0.1	0.1	-35%
E. coli	5.1	0.075	1	1	**0.086**	0.2	0.2	-14%
S. cerevisiae	12.4	5.30	2	1	**2.53**	0.4	0.4	52%
C. elegans	102.3	-	-	-	**13.3**	3	4	-
A. thaliana	121.2	3.49	5	6	**2.09**	4	5	40%
Oryza sativa	378.5	$30,679$	7170	230	**105.4**	11	15	99%
D. melanogaster	120.7	$8,226$	959	75	**364.4**	4	4	96%
Homo Sapiens 1	3,100	-	-	-	$\mathbf{1,025}$	95	130	-
Homo Sapiens 2	3,100	-	-	-	$\mathbf{7,247}$	120	126	-
Homo Sapiens 3	3,100	$8,912$	5028	271	$\mathbf{6,290}$	118	125	29%
Homo Sapiens 4	3,100	$7,913$	3905	272	$\mathbf{5,767}$	115	130	27%
Homo Sapiens 5	3,100	-	-	-	$\mathbf{11,559}$	122	130	-
Homo Sapiens 6	3,100	-	-	-	$\mathbf{5,241}$	100	120	-

TABLE III
COMPRESSION RESULTS FOR THE PAIRWISE COMPRESSION. C. TIME AND D. TIME STAND FOR COMPRESSION AND DECOMPRESSION TIME, RESPECTIVELY. THE RESULTS IN BOLD CORRESPOND TO THE BEST PERFORMANCE AMONG THE DIFFERENT ALGORITHMS. WE USE THE INTERNATIONAL SYSTEM OF UNITS FOR THE PREFIXES, THAT IS, 1MB AND 1KB STANDS FOR 10^6 AND 10^3 BYTES, RESPECTIVELY.

VI. PERFORMANCE COMPARISON OF IDOCOMP AND THE ALGORITHM PROPOSED IN [CHERN *et al*., 2012]

In this section we compare the performance of iDoComp with that of the algorithm presented in [Chern *et al*., 2012]. We apply both algorithms to the datasets considered in the paper. However, there are some datasets for which no performance of the algorithm presented in [Chern *et al*., 2012] is shown. These datasets either contain sequences with lower case characters (either the target or the reference), which are not accepted by [Chern *et al*., 2012], or the algorithm was not able to compress them. The results are summarized in Table III.

VII. MORE SIMULATION RESULTS ON *H. Sapiens* DATASETS WITH IDOCOMP AND GDC

The goal of this section is to provide more results on the performance of both iDoComp and GDC-*Normal*. To that end, we consider the compression of several combinations of reference-target genomes. Specifically, we use the *H. Sapiens* genomes introduced in the main manuscript, i.e., *hg18, hg19, YH, KOREF_20090131* and *KOREF_20090224*, and consider all the twenty possible combinations of reference-target. We modified the two korean genomes to contain only upper case letters, except when the two genomes are compressed together. The performance results of both iDoComp and GDC-*Normal* are shown in Fig. 2. The results are sorted as follows: first we show the results when *hg18* is used as a reference, and the target genomes are *hg19, YH, KOREF_20090131* and *KOREF_20090224*, then when *hg19* is used as a reference, with target genomes *hg18, YH, KOREF_20090131* and *KOREF_20090224*, and so on. Besides, Fig. 3 shows the gain of iDoComp with respect to GDC for the same combinations.

The following statistics can be inferred from the figures. The mean and the standard deviation of the compressed size achieved by iDoComp are $7,676$ KB and $2,603$ KB, respectively, whereas for GDC-*Normal* the mean is 8.431 KB and the standard deviation $2,514$ KB. Regarding the gain of iDoComp with respect to GDC, we get on average a 9.92% gain, with a standard deviation of 18.03%. For the cases where iDoComp shows a positive gain, the mean and standard deviation are 16% and 20%, respectively. On the other hand, for the cases of negative gain, iDoComp shows on average -1.45% gain with a standard

deviation of 0.8%, i.e., in those cases where GDC achieves better compression rates, the difference with iDoComp is very small. Thus we can conclude that iDoComp offers better compression results in general.

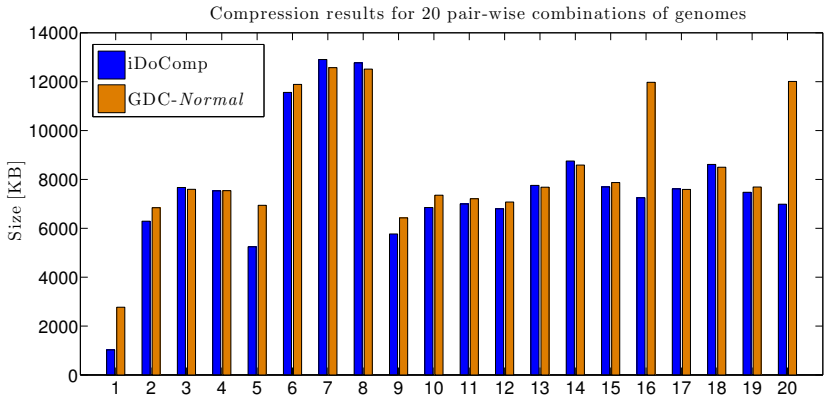

Fig. 2. Compression size of iDoComp and GDC-*Normal* for the considered 20 pair-wise combinations.

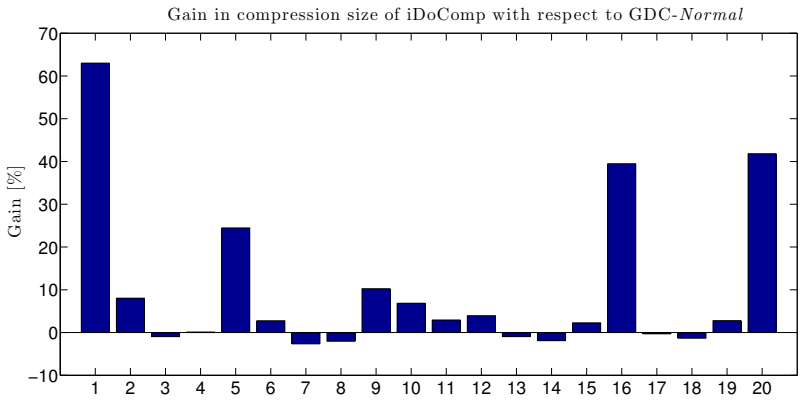

Fig. 3. Gain on compression size of iDoComp with respect to GDC-*Normal* for the considered 20 pair-wise combinations.

Finally, regarding the running time, iDoComp employs on average 116 seconds to compress, with a standard deviation of 6 seconds. For decompression, the average is 133 seconds, with a standard deviation of 4 seconds. On the other hand, GDC takes on average 454 seconds to compress, with a standard deviation of 703 seconds. For the decompression, the mean and standard deviation of GDC are 71 seconds and 5 seconds, respectively. Thus, the compression time of iDoComp is significantly less variable than that of GDC, which exhibits a standard deviation of 703 seconds. On the other hand, the time employed for decompression by GDC is less than that of iDoComp, with a difference of 62 seconds on average. Fig. 4 and Fig. 5 show the running time of both algorithms for compression and decompression of the twenty considered pair-wise combinations, respectively.

VIII. Influence of the choice of reference for compression

As stated in the main manuscript, in pair-wise compression, the choice of reference can be important in terms of compression results. In this section we provide some numbers and examples that corroborate this fact. Specifically, we look closely at the compression results presented in Section VII (20 pair-wise compression). For example, iDoComp compresses the genome *hg19* to $1,024$ KB when *hg18* is used as the reference, whereas it compresses it to $8,752$ KB when *KOREF_20090224* is selected as the reference. Similarly, iDoComp compresses the genome *YH* to $6,290$ KB when *hg18* is used as reference, and to $11,559$ KB when *hg19* is used as the reference instead. Similar results are observed when GDC-*Normal*

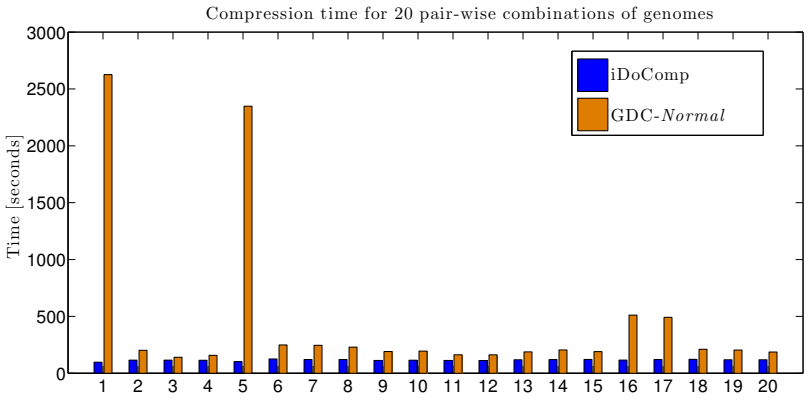

Fig. 4. Compression time employed by iDoComp and GDC-*Normal* for the considered 20 pair-wise combinations.

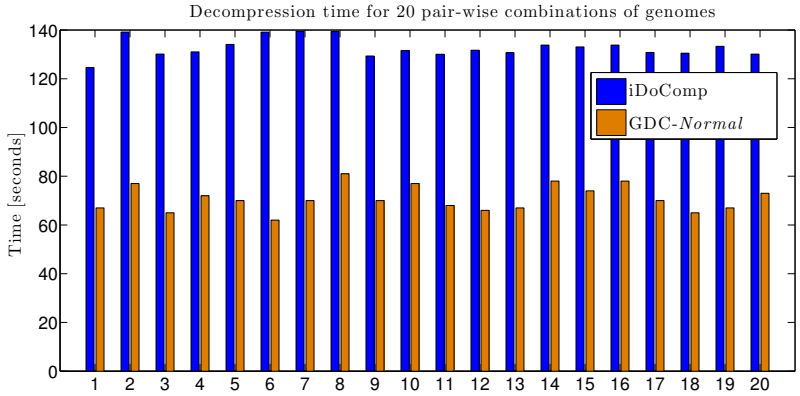

Fig. 5. Decompression time employed by iDoComp and GDC-*Normal* for the considered 20 pair-wise combinations.

is used for compression. For example, the compression of *YH* varies from $6,840$ KB if *hg18* is used as reference, to $11,888$ KB if *hg19* is selected as the reference instead.

IX. Compression of sets

The purpose of this section is to show the boost in compression ratio that can be achieved when compressing a collection of genomes if an algorithm designed for this purpose is used instead of an algorithm designed for pair-wise compression. To that end, we consider the compression of a collection of ten genomes downloaded from the *1000 genomes project* [http://www.1000genomes.org/]. The genomes are available in VCF format, i.e., as differences with respect to a reference genome, given by the *H. Sapiens* genome build 37.1. Thus we reconstructed the genomes and generated the corresponding fasta files. Specifically, the ten genomes that compose the set are *NA18508, HG00320, NA18499, HG00278, HG00275, NA07056, NA06986, HG00324, NA07347* and *NA06984*.

We compress the set with GDC-*Ultra*, designed specifically to compress collection of genomes, and compare the performance with that of iDoComp. Recall that the proposed algorithm is not designed to compress a collection of genomes, but to perform pair-wise compression. Thus we selected the genome *NA18508* as the reference, and compress the remaining nine genomes using that reference for compression. As mentioned in the main manuscript, the algorithm FRESCO [Wandelt *et al*., 2013] is also designed as a tool for compression of a collection of genomes. However, we do not provide results for this algorithm as we were unable to run it.

The result of iDoComp is a compressed file of 52.6 MB, i.e., the nine genomes can be compressed up to that using *NA18508* as the reference. If we consider the compression of the whole set, we should add to

this number the compressed size of the reference. We do not include it in the final size as iDoComp is not designed to compress a genome without a reference. On the other hand, the performance of GDC-*Ultra* is better, achieving a compressed size of 31.7 MB for the same nine genomes.

REFERENCES

[Chern *et al*., 2012] Chern, B. G., Ochoa, I., Manolakos, A., No, A., Venkat, K., and Weissman, T. (2012, September). Reference based genome compression. In Information Theory Workshop (ITW), 2012 IEEE (pp. 427-431). IEEE.

[Deorowicz *et al*., 2011] Deorowicz, S. and Grabowski, S. (2011) Robust relative compression of genomes with random access, *Bioinformatics*, **21**, 2979-2986.

[http://www.1000genomes.org/] 1000 genomes project.

[https://sites.google.com/site/yuta256/sais/] Suffix Array package.

[Wandelt *et al*., 2013] Wandelt, S., and Ulf, L., (2013) FRESCO: Referential Compression of Highly Similar Sequences, IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB).