Memory-Efficient Assembly using Flye

Borja Freire, Susana Ladra, and José R. Paramá

Abstract—In the past decade, next-generation sequencing (NGS) enabled the generation of genomic data in a cost-effective, high-throughput manner. The most recent third-generation sequencing technologies produce longer reads; however, their error rates are much higher, which complicates the assembly process. This generates time- and space- demanding long-read assemblers. Moreover, the advances in these technologies have allowed portable and real-time DNA sequencing, enabling in-field analysis. In these scenarios, it becomes crucial to have more efficient solutions that can be executed in computers or mobile devices with minimum hardware requirements. We re-implemented an existing assembler devoted for long reads, more concretely Flye, using compressed data structures. We then compare our version with the original software using real datasets, and evaluate their performance in terms of memory requirements, execution speed, and energy consumption. The assembly results are not affected, as the core of the algorithm is maintained, but the usage of advanced compact data structures leads to improvements in memory consumption that range from 22% to 47% less space, and in the processing time, which range from being on a par up to decreases of 25%. These improvements also cause reductions in energy consumption of around 3–8%, with some datasets obtaining decreases up to 26%.

Index Terms—compact data structures, genome assembly, long-reads assembly, memory efficiency, third-generation DNA sequencing

1 Introduction

Since the 50s, we are attending to a rapid increase in the scale of the treated bioinformatics datasets [1]–[3]. Current databases, such as the Sequence Read Archive [4], contain a large number of datasets; moreover, they are also growing in size, making old techniques unable to process them. Genomics poses unique challenges in terms of data acquisition, storage, distribution, and analysis [5], which require new innovative approaches.

Nowadays, the most common approach for facing computationally very expensive processes is to use some sort of parallel computing [6]–[10]. This is due to the availability of techniques, useful tools, and cheap hardware. However, another, less frequent, way to obtain scalable systems is to use more efficient methods or data structures in order to reduce the memory consumption and/or time [11]–[13]. Despite now being an unusual approach, in the early days of computer science, it was common to spend considerable effort to obtain more efficient software, as the hardware was expensive and had low computational power. In addition to their intrinsic benefits, these techniques are completely compatible with parallel computing strategies, thus, we can join the advantages of both approaches.

In-memory databases [14] constitute an example of this. Instead of using the traditional setup, where data reside in disk and portions are translated to main memory when needed, these database management systems keep all data in main memory all the time. Obviously, this is a challenge that requires complex procedures and data structures, including compression techniques. Another example is the so-called compact data structures [15], [16]. The idea is basically the same as in in-memory databases, data are stored in the upper levels of the memory hierarchy by using compression. The difference is that this field faces all types of data.

Many compact data structures use bitmaps as the main basic block to build complex data structures. Given a bitmap $B[1 \dots n]$ storing a sequence of n bits, there are three basic operations: *access(B,i)*, which obtains the bit at position i of B; $rank_a(B, i)$, which counts the occurrences of bit $a \in \{0, 1\}$ in $B[1 \dots i]$; and $select_a(B, i)$ locates the position for the i^{th} occurrence of $a \in \{0, 1\}$ in B.

Several data structures (see [17] for example) allow solving these operations in constant time and using $n+o(n)$ bits of total space. There exist implementations that enable fast $rank$ and $access$ operations in only 5% extra space over the original bit array.

By using bitmaps, we present a modification of the well-known assembler Flye [18], aimed at decreasing the amount of main memory used when creating the draft genome assembly and the subsequent assemblies during the rest of phases of the process. Furthermore, the execution time is not affected, but improved in most cases.

Flye was recently compared with five state-of-the-art assemblers, obtaining better or comparable assemblies, while it is an order of magnitude faster [19]. Moreover, Flye obtains longer contigs as it doubles the NGA50 metric. Therefore, taking as a starting point such an efficient assembler in terms of speed as Flye, we decided to face the other relevant parameter for an efficient implementation, its memory consumption.

Decreasing the memory footprint of assembly processes is crucial for new DNA sequencing technologies that aim at offering portable and real-time genome sequencing [20]. In-field analyses are now possible thanks to the development of mobile and affordable devices, such as the pocket-sized Oxford Nanopore Technologies' (ONT) MinION. The main advantage of these mobile genomic labs, which can be deployed for in situ DNA extraction and sequencing, is the possibility of shortening the time from the collection of the

• *B. Freire, S. Ladra, and J.R. Paramá are with Universidade da Coruña, Centro de investigación CITIC, 15071, A Coruña, Spain. E-mail: {borja.freire1,susana.ladra,jose.parama}@udc.es*

Manuscript received xxxxx xx, 2020; revised xxxxx xx, xxxx.

sample to the data analysis. This approach has been recently used, for instance, during the Nigeria 2018 Lassa fever outbreak, where real-time analysis allowed a better understanding of its molecular epidemiology [21]. Currently, most bioinformatics pipelines are executed on high-performance computing cluster or powerful computers, most of the time on the cloud. Thus, data must be transferred and queued into those systems, which slows down the whole process. Moreover, it is possible that these analyses involve personal and sensitive data, such as genetic data; therefore transferring the data to external facilities can be problematic. Thus, due to privacy issues or immediateness, there exist scenarios where the analysis of genomic data in the same place where the data is extracted, is of vital importance. In these scenarios, it is necessary that these real-time analyses can be done not only using portable DNA sequencers but also portable computing devices, such as laptops or smartphones, which have memory limitations.

1.1 Long error-prone reads assemblers

The expected outcome of an assembly process is a set of safe contigs that belong to the genome of interest, hopefully covering as much of it as possible. The traditional way of doing this is to build a de Bruijn graph [22] from k-mers[1] or an Overlap-Layout-Consensus (OLC) graph [23] and then look for safe paths within these graphs [24]. Nowadays, there are several algorithmic ways to discover these paths, and even graph transformations that lead to longer paths like Y-to-V transformation [25], EULER [26], or using omnitigs [27].

Although all these techniques are well-known and they are widely used in practice when working with short reads obtained from second-generation sequencing[2] platforms, they are not as useful with third-generation sequencing reads, i.e. PacBio SMRT, or Oxford NanoPore sequencing. The reason is that, due to the high error rate of the reads, the de Bruijn graphs built with a standard k-mer size (between 25–30 bp) are extremely tangled and the OLC graphs need an extremely large coverage to work properly.

The first attempts of long error-prone assemblers were based on Overlap-Layout-Consensus or on similar string graph approaches [28], but these methods have quadratic complexity. Another way to face the problem is to return to the de Bruijn graph, or more precisely, to a variation called the A-Bruijn graph, which was originally designed to assemble a rather long Sanger reads [29].

Based on the A-Bruijn graph, Lin et al. presented Flye [18], which is able to obtain good results with a rather efficient process. Our work is based on this approach, which is better explained at Section 2.2.

2 Background

2.1 Compact data structures

Compact data structures have been extensively used in bioinformatics. The best example is the FM-Index [30], which is able to store a text using roughly the space required for representing that text in compressed form and, at the same time, is able to locate any substring in sublinear time. It is the main data structure of the majority of short-read aligners including Bowtie [31], BWA [32], and SOAP2 [33]. More specific tasks, such as k-mer counting and k-mer indexation, have also been addressed by using compact data structures. Välimäki and Rivals [34] used a FM-Index-like structure, called compressed suffix array [35] for this. Claude et al. [36] used techniques coming from the field inverted indexes. There are many succinct versions for de Bruijn graphs [37]–[40] that use different compact data structures techniques, among others, the FM-index. An important recent research line in compact data structures is to represent and index genomes of different individuals in very little space [41]–[43]. There is a large list of works in this field, just to cite a few [44]–[48].

As explained, many compact data structures use bitmaps combined with fast rank and select operations. Jacobson [49] proposed a solution able to compute rank in constant time. Given a bitmap B of size n, it uses a two-level directory structure. The first-level directory stores $rank_1(B, p)$ for every p multiple of $s = \lfloor \log n \rfloor \lfloor (\log n)/2 \rfloor$. For every p multiple of $b = \lfloor (\log n)/2 \rfloor$, the second-level directory keeps the relative rank value from the previous multiple of s. By using these data structures, $rank_1(B, i)$ can be computed in constant time by accumulating the values from both directories. The first-level directory returns the rank value until the previous multiple of s. The second-level directory gives the value of rank from that position until the previous multiple of p. Finally, the number of 1s from that position until position i is computed using a precomputed table that stores the rank values for all possible byte values. The sizes s and p are carefully chosen so that the auxiliary dictionary structures use $o(n)$ additional space.

Although $n + o(n)$ representations are asymptotically optimal for incompressible binary sequences, it is possible to obtain better space when the binary sequence is compressible, for example when the number of 1s (alternatively the 0s) is small. In that case, the bitmaps are usually called *sparse bitmaps*. For instance, Raman et al. [50] presented two representations for sparse bitmaps with $nH_0(B) + o(\ell) + O(\log \log n)$ and $nH_0(B) + O(n \log \log n / \log n)$ bits, where $H_0(B)$ is the zeroth-order entropy of B and ℓ is the number of 1-bits in B.

The constant time solution for select is significantly more complex than that of rank. Clark [51] presented a solution based on a three-level directory that requires $3n/\lceil \log \log n \rceil + O(\sqrt{n} \log n \log \log n)$ bits of extra space. For example, in case $n = 2^{30}$, the additional data structures occupy 60% of the original bitmap. Practical implementations of select [17] reuse the same directories used for rank, although this yields $O(\log n)$ time. The simple solution is to binary search in B a position i such that $rank_1(B, i) = j$ and $rank_1(B, i-1) = j-1$. Real implementations, instead of using the rank operation as a black box, binary search the directories D_s and D_b to speed up the query, yielding a $O(\log \log n)$ cost.

1. k-mers are subsequences of length k contained within a biological sequence. In our work, they are sequences of k nucleotides (i.e. A, T, G, and C).

2. Second-generation sequencing is also known as next-generation sequencing (NGS).

2.2 Flye assembler

Most single-molecule or third-generation sequencing assemblers spend much time making sure that the created contigs are correctly assembled. Flye, in contrast, does not waste time on making sure that the created contigs are correct. Actually, Flye intentionally builds misassembled contigs and, from them, it builds accurate and longer contigs. Briefly, Flye constructs an assembly graph and, starting from a given read, it creates random walks on that graph. Once a random walk is built, all reads that map with that walk are found, and a consensus contig is built from the whole set of reads. Obviously, this new consensus contig is legit because it is supported by several reads. Once all contigs have been created, they are glued into an accurate assembly graph which is untangled, and unbridged repetitions are solved by using the number of reads that traversed consecutive edges in the accurate assembly graph. More precisely:

1) ***Building a draft genome assembly and generating consensus contigs***. To deal with the drawbacks of long reads, Flye corrects them before building the assembly. This stage computes an A-Bruijn graph that uses only solid k-mers[3] instead of all the k-mers from the reads. Furthermore, the value stored in an edge between nodes X and Y of the graph corresponds to the distance between them in a given read Z, unlike traditional de Bruijn graphs, which store the char following node X.
   Once the graph is built, Flye generates arbitrary paths in the graph[4] creating inaccurate contigs. Then, a consensus process is carried out using all the reads that contribute to the contig. Finally, these new "accurate" contigs are used to build an accurate assembly graph, which can be traversed like a regular de Bruijn graph.
2) ***Treating repetitive regions***. One of the biggest challenges when assembling NGS data is to deal with genome repetitive regions, which are the result of recombination, transposons and/or mini/microsatellites. The success of the de Bruijn graphs in the genome assembly field is mainly due to their ability to represent repeat families as mosaics of ideally error-free[5], sub-repeats [29].
   Furthermore, overlap graphs scale quadratically with the number of reads, thus, they become unfeasible for NGS. The direct application of the classic de Bruijn graph does not obtain accurate assembles from long reads, and thus OLC graphs with high-coverage reads are the usual choice to obtain an accurate assembly. However, with OLC, it is not possible to define repeat families, thus much research efforts have been devoted to adapt de Bruijn graphs to be useful for long reads.
   Flye is able to treat repetitive regions using a de Bruijn graph because the reads have been corrected in the previous step. Therefore, using previous consensual reads, an unravelled *repeat graph*[6] is built. Since the idea behind repeat graphs is actually the same as that of the de Bruijn graph, Flye is able to transfer the power of de Bruijn graphs to long reads without losing accuracy or demanding extremely high levels of coverage throughout *repeat graphs*.
3) ***Polishing the final genome***. Once the contigs are obtained, most NGS assemblers are capable of increasing their length by using paired-end information to add topological knowledge. These extended contigs are then referred to as scaffolds. Furthermore, this step also allows polishing the obtained contigs removing those which are misassemblies, namely chimaeras, and thus increasing the veracity of the assembly.

All of these three steps are critical and need to be treated carefully. Even though all the steps are equally relevant, the most memory demanding phase is the first one, since it has to quickly process all k-mers in reads and build consensus contigs, which requires having all overlaps between reads. Therefore, since our goal is to reduce memory consumption, we have focused our improvements on this module.

Next, we will introduce how Flye builds the assembly draft. It is important to remark that this work does not introduce any changes to the Flye's assembly algorithm. Our goal is to improve Flye's memory efficiency, and therefore its scalability, by using new data structures to store and manipulate data, but always using Flye's assembly procedure.

The traditional genome assembly process is based on creating a graph representation from the reads using k-mers and then looking for safe solutions/paths inside the graph. Afterwards, multiple heuristic steps are applied to extend these solutions, and once the contigs have been stretched, the final stages of the process are scaffolding and gap filling.

However, this process suffers from problems when dealing with long reads due to, as explained, their high error ratio, which is around 15% for long reads compared to 0.1–1% for NGS. To overcome this, the Flye's assembly process consists of three different phases: (i) approximate k-mer counting, (ii) selection and indexing of solid k-mers, and (iii) draft genome assembly. The first two steps focus on filtering the k-mers in reads and selecting those k-mers that can be considered genomic. Once Flye has selected the legit genomic k-mers, it assembles the reads looking for those that overlap each other and also fulfil a set of restrictions. As a final step, Flye generates consensus contigs to build *long error-free reads* and, from them, it obtains an accurate genome assembly.

3. A solid k-mer is a k-mer that appears at least t times in the reads, being t a threshold.

4. Instead of looking for the best one, Flye extends the paths selecting an arbitrary read. Therefore, it does not lose time assaying every single overlapping read and selecting the most promising one.

5. Note that de Bruijn graphs work for "perfect" repetitions but unfortunately DNA repeat families are usually full of gaps and mismatches. Therefore, even with de Bruijn graphs, treating repeat regions remains messy.

6. A repeat graph is an alternative graph representation that compactly represents all repeats in a genome and reveals their mosaic structure [18], [29]. Furthermore, repeat graphs can deal with mismatches and gaps inside repetitions, offering a better way of treating and detecting repetitive regions.

2.2.1 Counting k-mers

The high error rate of the reads produces a large number of false or non-genomic k-mers.[7] In order to overcome this problem, Flye separates the genomic and non-genomic k-mers by counting the appearances of the k-mers in the reads and selecting only those whose frequency is above a threshold t. While this might sound simple, it actually becomes a hard problem when working with long reads, due to the huge amount of k-mers present in the reads, which makes their indexation extremely difficult in a reasonable time and space. In fact, this problem is the origin of a research line by itself [52]–[54].

The threshold used by Flye in this phase is automatically selected as a value between 2 and 5, depending on the coverage of the input dataset, which is estimated by Flye from the genome length (provided as a parameter) and the sum of the read lengths.

The solution of Flye requires two counts: (i) approximate counting, and (ii) exact counting. The approximate counting is carried out by using a Bloom filter [55], a hash table that does not solve collisions. Therefore, one hash entry can accumulate appearances of different k-mers. This step removes k-mers with a very low frequency.

We illustrate an example of the use of the hash table for the approximate counting in the left part of Figure 1. In the upper part, we include two reads (read_1 and read_2), and, for each of their positions, the k-mer that starts at that position. As explained, in this approximate counting, collisions are not solved. Observe that entry 4 accumulates six appearances, which correspond to four appearances of k_{13} and two appearances of k_{14}, since the hash function $hash_1$ maps both k-mers to the same entry.

Being L_r the average length read, N_r the number of reads, and $GenomeSizeFactor$ a value between 1 and 16 that Flye calculates based on the estimated size of the genome provided by the user, this phase costs $\theta(L_r * N_r)$ time and uses $GenomeSizeFactor * 4^{15} * 8$ bits.

In a second pass of the reads, Flye counts the exact occurrences of the k-mers whose entries reached the threshold t in the first pass. For this exact counting, Flye uses the well-known data structure Cuckoo hash [56]. It is a variation of hashing that assures a small and constant number of iterations for indexing and accessing operations. This is achieved by using several hash functions, usually two is enough, and a collision handler. When a collision is produced, the corresponding key already stored at the entry is moved to one of its other possible positions, defined by an alternative hash function. This can cause another collision, which is solved in the same way.

We illustrate the exact counting in the right part of Figure 1, including different states of the hash table for different instants of the process. Now, the entries of the hash table contain the key (a k-mer), in addition to the counter. In reality, a Cuckoo hash usually uses two tables of the same size, and, at each entry, several keys can be stored (collisions). For a given key, if $hash(key) = i$, that key can be in the entry i of either table. However, in order to simplify the figure, we use just one table with just one key per entry.

In the hash table under the (a) label, we show the state of the table after traversing read_1 until position 6. As seen, k_{13} and k_{54} appeared twice and k_{32} once. These keys are placed in the position given by the hash function $hash_1$. Under the label (b), we show the state of the hash table after processing position 7 of read_1. The k-mer in that position (k_{44}) is mapped to the entry 5 according to $hash_1$, which is already occupied by k_{32}. Therefore, k_{32} is moved to its alternative position given by the hash function $hash_2$, that is, to position 15. Once position 5 is released, k_{44} is placed there.

Under label (c), we show the state of the hash table when processing position 4 of read_2. In that position, the k-mer k_{14} begins, and $hash_1$ maps it to entry 4, which is already occupied by k_{13}. Then, k_{13} should be moved to its alternative position given by $hash_2$, which is position 5, thus releasing position 4 for k_{14}. However, position 5 is also occupied by k_{44}. Then, k_{44} is also moved to its alternative position defined by $hash_2$, which is position 11. Again, position 11 is occupied, in this case by k_{54}, but now, we enter in a cycle, because the alternative position of k_{54} is 4. To avoid this situation, there is a limit in the number of handled collisions (logarithmic in the table size). If that value is reached, this requires the change of the hash functions $hash_1$ and $hash_2$, and all the k-mers should be reallocated accordingly.

This method guarantees constant time access as long as the table is not occupied more than 50% of its capacity. To avoid this, when the number of collisions triggers the change of the hash functions, the table size is doubled as well.

The worst-case time complexity of the exact count is $O(L_r * N_r + S^2)$ time, being S the number of k-mers that passed the approximate counting. Recall that all k-mers must be processed, checking if they have passed the first threshold, and then inserted in the Cuckoo table. These insertions may cause duplication of the hash table and the reallocation of most entries. Observe that each duplication implies $O(S)$ reallocations, and, in the worst-case scenario, this can happen for $O(S)$ keys.

Regarding space, the memory required at the beginning is $O(S)$ bits, but this is doubled when the number of collisions triggers a reallocation, therefore it can reach $O(S^2)$ bits in the worst-case scenario. In practice, this easily reaches 20–30 GB.

Here we can see one of the weaknesses of Flye that our work addresses. In order to ensure an efficient use of the Cuckoo hash, and therefore to allow fast access to the information of any k-mer, Flye needs big amounts of memory. We aim at replacing the Cuckoo hash table by a more compact data structure without losing speed.

2.2.2 Selecting/Indexing k-mers

Although the previous phase significantly reduces the amount of non-genomic k-mers, some non-genomic or low frequent k-mers remain. Therefore, a second filtering step is necessary to minimize those non-genomic k-mers. Furthermore, more information about k-mers is needed for the next steps, such as the reads in which each k-mer appears,

7. The maximum number of non-genomic k-mers is $(L_r * Fr) * Cov * N_r$, where: L_r is the average length read, Fr is the failure ratio, Cov is the coverage, and N_r is the number of reads.

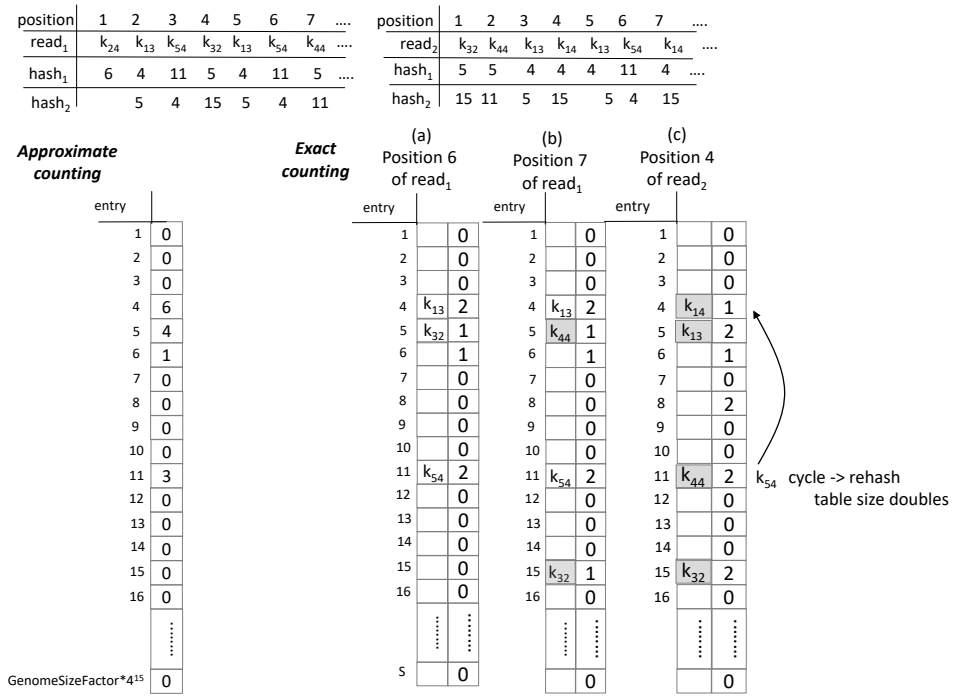


Fig. 1: Counting k-mers in Flye.

and the position inside these reads. This second filter uses a different threshold, which depends on the results of the first filtering step. Actually, the second threshold is equal to the highest frequency that ensures that the number of selected k-mers is greater than the length of the genome.

This step requires another traversal of the reads, and the k-mers that pass this second threshold are indexed into a Cuckoo hash again, but now, they are stored with the positions within the reads where they appear,[8] that is, a list of pairs $(read, positions\_inside\_read)$, thus creating an index of the appearances of the solid k-mers.

This index is the key element for the subsequent step of Flye, that is, the assembly phase. To perform the assembly, Flye uses a variation of the de Bruijn graph called *A-Bruijn graph*. However, the graph is never created and Flye implements it "virtually" by means of this index. The nodes are the reads and the edges are simulated by searching "on the fly" overlaps between reads. Therefore, since that operation is very frequent, this operation must be fast. The overlaps are computed by taking all the k-mers of a read, and, for each one, this index gives all the reads where that k-mer is also found. With this information, Flye computes the overlapping reads.

The creation of this index consumes $O(L_R * N_r)$ time. In space, we need $O(S^2)$ bits for the hash table, plus $O(S * Cov)$ bits for the appearances of the k-mers[9], being Cov the average coverage of the dataset. Thus, the worst-case space complexity is $O(S^2 + S * Cov)$.

8. Each k-mer can occur an arbitrary number of times within a read.

9. There is another filter in this phase, and the number of k-mers considered decreases, but we keep S as the number of solid k-mers.

Algorithm 1: FlyeWalk (*AllReads*, *MinOvelap*, *HashTable*)

```
1  Contigs ← empty set of contigs
2  UnprocessedReads ← AllReads
3  IndexOfReads ← BuildIndex(AllReads, HashTable)
4  for each Read in UnprocessedReads do
5      ChainOfReads ← ExtendRead (UnprocessedReads,
         Read, IndexOfReads, MinOverlap)
6      ContigSequence ← Consensus(ChainOfReads,
         AllReads, MinOverlap, IndexOfReads)
7      add ContigSequence to Contigs
8      remove Overlap(ChainOfReads) from
         UnprocessedReads
9  end
10 return Contigs
```

2.2.3 Assembling contigs

As previously explained, Flye does not use a real de Bruijn graph that is traversed seeking for contigs, but simulates this graph navigation by obtaining all the possible overlaps between reads. More precisely, the construction of contigs is done by checking the overlaps between a processed read and others, while considering several restrictions.

Algorithm 1 shows this process. The algorithm starts by obtaining for each read all its overlapping reads. The process uses the index described in Section 2.2.2. The procedure *BuildIndex* processes all reads, each one is traversed position by position. The k-mer starting at the processed position is used to query the index and, if it is solid, it gives all the overlapping reads.

Once the overlaps have been computed, the process continues with the *for* in Line 4, which processes the reads in

Algorithm 2: ExtendRead(*UnprocessedReads*, *Read*, *MinOverlap*, *IndexOfReads*)

```
1  ChainOfReads ← empty sequence of reads
2  while true do
3      NextRead ← FindNextRead(UnprocessedReads,
         Read, MinOverlap,IndexOfReads)
4      if NextRead = empty string then
5          return ChainOfReads
6      else
7          add NextRead to ChainOfReads
8          Read ← NextRead
9          remove Read from UnprocessedReads
10     end
11 end
```

random order. Then, in Line 5, each read is extended using the function *ExtendRead*, which is shown in Algorithm 2. Given a read *Read*, *ExtendRead* finds an unprocessed read that overlaps with *Read* by at least *MinOverlap* base pairs. This is done during the call of the function *FindNextRead*. Flye does not waste much time checking if the next read fits well in the current path; it just chooses one that overlaps *MinOverlap* base pairs with the current read and fulfils other simple conditions. This process continues until it cannot find another read overlapping the last processed read.

Finally, *Consensus* constructs the consensus of all reads that contribute to the processed *ContigReads*. The process is as follows. Let $Read_1$, $Read_2$, ..., $Read_n$ be the reads in *ContigReads*. Let $prefix(Read_i)$ be the overlapping region between consecutive reads $Read_{i-1}$ and $Read_i$, let $suffix(Read_i)$ be the suffix of $Read_i$ after the removal of $prefix(Read_i)$, and let $concatenate(ChainOfReads)$ be the concatenation $suffix(Read_1)||suffix(Read_2)||\ldots||suffix(Read_n)$.
Then, all reads from the dataset are aligned to $concatenate(ChainOfReads)$ using the method minimap2 [57]. The consensus is taking by the majority vote. Finally, the reads considered in the consensus step are removed from the *UnprocessedReads* (Line 8), and therefore they are not considered anymore.

In this process, with $O(N_r^2)$ time, the *Consensus* is the dominant cost.

3 Our proposal: Compact Flye

In this section, we describe our memory-efficient variant of Flye assembler, where we use compact data structures. We detail how our proposal addresses each of the phases of the method.

3.1 Counting k-mers

3.1.1 Approximate counting

Our aim is to perform the same filtering, but without using such a large amount of memory. Moreover, we do not want to penalise the temporal efficiency; thus, our target is also to maintain the execution times in the same order of magnitude or even improve them.

The original method uses 8-bit counters for this phase, allocating a hash table of size $GenomeSizeFactor * 4^{15} * 8$ bits, where $GenomeSizeFactor$ is set to 1 or 16 depending on the size of the genome. This large space hardly provokes collisions. For example, in the case of k-mers of length 15, those 8-bit counters for all the 4^{15} possible combinations would require 8 GB.

Instead, we propose an adaptive solution that is able to store the same number of entries as the hash tables of Flye (both, of 1 GB and 16 GB), but using much less space. For the approximate counting, instead of allocating a complete byte for the counters of k-mers, we use just t bits, being t the given threshold, which is a number between 2 and 5. When a new appearance occurs, the first bit set to 0 is changed from 0 to 1. For instance, if the threshold is 5, then each entry initially has the value 00000. The first appearance changes the entry to 10000, the second to 11000, the third one to 11100, and so on. With this approach, to perform the update, it is only necessary to access the memory location and flip just one bit, rather than adding 1 to the memory location, which requires moving the data to the Arithmetic Logical Unit of the processor and executing an operation of addition.

With this approach, we do not have a real count, that is, we do not know how many times an entry has been processed. This is not a problem, as at this step of the process, we are only interested in those entries that have been found t times or more, and thus, their corresponding k-mer passes this filtering step. Recall that in Flye, this is an approximate counting, as each entry can accumulate appearances of different k-mers.

Our approach is able to filter as many k-mers as Flye does, but only requiring $GenomeSizeFactor * 4^{15} * t$ bits. Of course, the key of a low space requirement is that we are assuming that the threshold value is small, between 2 and 5, otherwise valuable k-mers would be removed. Therefore, in the worst case, our structure is always cheaper than 1 GB or 16 GB (the value of the hash table of Flye), independently of the genome size.

Figure 2 shows an example of this process under the brace labelled "approximate counting". More concretely, we have two reads composed of different k-mers, and we will filter those that have more than 3 appearances ($t = 3$). Thus, our data structure contains 3 bits per entry, and only those entries having their third bit set to 1 pass this filter. We denote b the bitmap composed of the last bit of all entries, as indicated in the figure. Observe that entry 5 has reached the threshold, due to the occurrences of two different k-mers that have the same hash value, namely two appearances of k_{32} and two appearances of k_{14}. After the third appearance, additional occurrences are no longer recorded.

Although the conceptual description is that shown in Figure 2, we implemented it in a faster way, to avoid the sequential search of the first bit set to 0. Specifically, the first $t-1$ bits of each entry are joined in a unique bitmap. In addition, the bits are completely flipped so the first 1 is set in the least significant position. Then, given the $t-1$ bits of an entry, when a new occurrence should be registered, we simply shift one bit to the left and we introduce a 1-bit on the right. The leftmost bit, which is lost, is used to update the corresponding entry of the t^{th} bit, which is stored separately from the others in bitmap b. For example, if the entry is 001, first it is shifted one bit to left, and the leftmost bit (0), is used

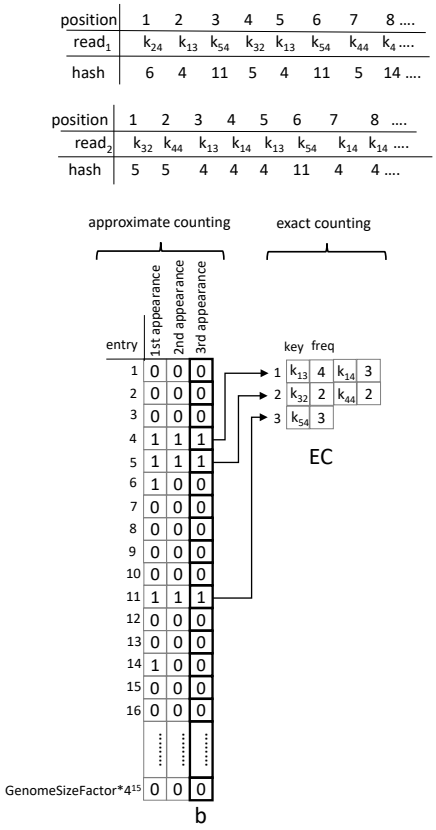


Fig. 2: Example of the k-mer counting.

to update the corresponding entry at b, and we introduce a 1 on the right, obtaining 011. Therefore this method has a cost of $O(1)$, regardless of t.

An alternative implementation is using a counter of $\log\lceil(t)\rceil$ bits, which would save some space. In this case, the bitmap corresponding to the first $t-1$ bits is replaced by an array of counters of $\lceil\log(t-1)\rceil$ bits, while bitmap b is kept. When a new occurrence of a k-mer appears and the corresponding value at the counter is already $t-1$, we set the 1-bit of the corresponding entry of bitmap b. In practice, we will use the method shown in Figure 2 when $k \leq 15$, and the counter of $\lceil\log(t-1)\rceil$ bits when $k > 15$.

The time cost for this step is the same of Flye, that is, $\theta(L_r * N_r)$. The space consumption is $GenomeSizeFactor * 4^{15} * t$ bits if $k \leq 15$ and $GenomeSizeFactor * 4^{15} * (\log\lceil(t-1)\rceil + 1)$ when $k > 15$.

3.1.2 Exact counting

Once the approximate counting has been done, we create a new data structure, denoted EC, to support the exact counting. This data structure has as many entries as positions set to 1 in the k-mer bitmap b, that is, $rank_1(b, size(b))$ entries. At each entry, we store a pointer to a list of pairs, each pair containing a k-mer that has passed the first filtering and an integer counter for storing the exact number of occurrences. Then, in a second traversal of the reads, for each read k-mer k_r that satisfies $b[hash(k_r)] = 1$, we compute its position $pos = rank_1(b, hash(k_r))$. In case k_r is already stored at $EC[pos]$, we increase its counter by one. Otherwise, a new pair $(k_r, 1)$ is added to the list pointed to by the pointer in $EC[pos]$.

In our example of Figure 2, when the traversal reaches position 4 of $read_1$, which is the first appearance of k_{32}, $EC[2]$ is empty[10], therefore we write $(k_{32}, 1)$ in the list pointed to by $EC[2]$. When the traversal of $read_1$ reaches position 7, k-mer k_{44} is also hashed to position 5, but the list pointed to by $EC[2]$ has only one pair, with key k_{32}, therefore a new pair $(k_{44}, 1)$ is added to the list pointed to by the pointer in $EC[2]$. It is important to remark that the use of a list of pairs at each entry does not damage the execution times significantly, as these lists are generally short.[11]

Here we can see one of the main differences between our method and Flye. If we compare Figures 1 and 2, we can see that compact Flye uses a simple bitmap of $GenomeSizeFactor * 4^{15}$ bits to index the k-mers, and the collisions, such as in the case of k_{13} and k_{14}, are stored in a list pointed to by an entry of EC. However, as explained, the lists of collisions are short. Instead, in Figure 1, Flye uses the Cuckoo hash table, which gives constant time access but requiring a big amount of memory and also time, due to reallocations. We can see how the entries of k_{13} and k_{14} are separated. Thus, compact Flye reduces the memory usage in exchange of probably increasing time processing when accessing to those counters in next steps of the process.

The worst-case time complexity of our approach is $O(L_r * N_r + S^2)$, since this phase processes the $L_r * N_r$ input k-mers, and, in the worst-case scenario, all the solid k-mers are mapped to the same entry, and thus, the S k-mers are added to just one list pointed to by one entry of EC. The data structure requires $GenomeSizeFactor * 4^{15}$ bits of space for bitmap b, which is constant, plus $O(S)$ counters, therefore the worst-case space complexity is $O(S)$.

3.2 Selecting/Indexing k-mers

Analogously to original Flye, reads are processed again to index only the k-mers with a number of occurrences higher than the second threshold.

Figure 3 shows the result of this step with our running example, assuming that the new threshold is 3 again. As in the case of the exact count, compact Flye relies on bitmap b, of $GenomeSizeFactor * 4^{15}$ bits, to index the k-mers, and now the collisions, such as in the case of k_{13} and k_{14}, are stored in an array, ordered by frequency of appearance. In our example, k_{13} is the first entry of the list corresponding to the hash entry 4, since it is more frequent than k_{14}.

Therefore, the worst-case time complexity of our method is $O(L_r * N_r + S^2)$, which includes both indexing and sorting the arrays. In space, we need the $GenomeSizeFactor * 4^{15}$ bits of bitmap b (constant), and $O(S * Cov)$ for storing the appearances of the solid k-mers.

3.3 Contigs Assembly

As the main goal of this research is to prove the good properties of compact data structures for the implementation of

10. Since $hash(k_{32})$=5, and $rank_1(b, 5) = 2$, then k_{32} must be included in the *second* entry of EC.

11. For instance, in our experiments for the *E. coli* datasets, the average number of elements at each list of EC was lower than 2, and the maximum number of elements was not higher than 5.

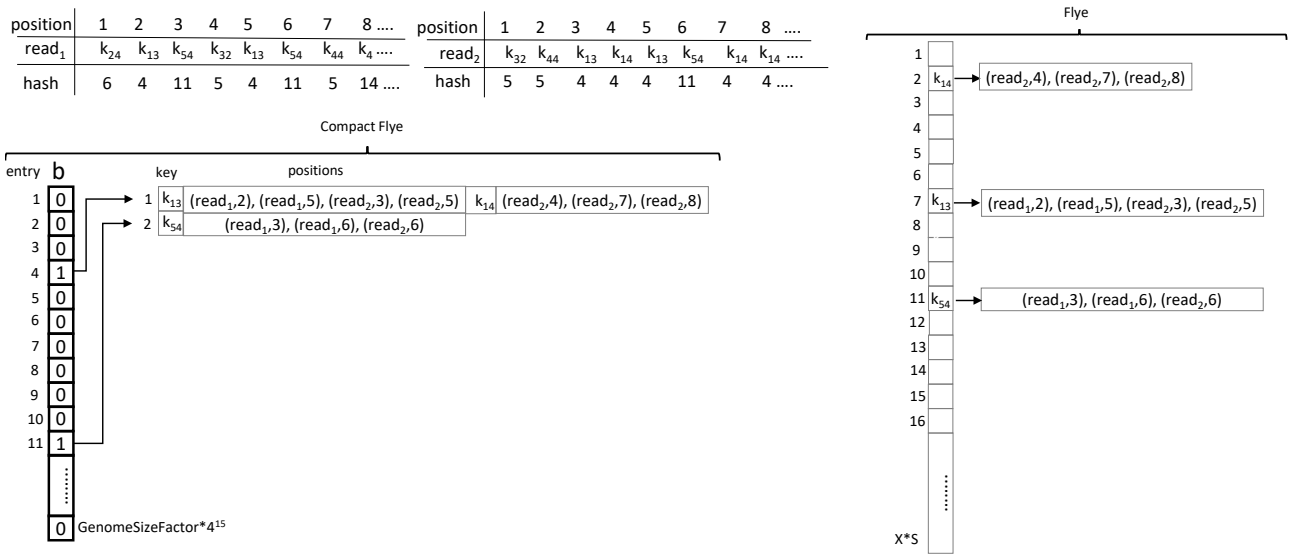


Fig. 3: Example of the k-mer indexing.

TABLE 1: Time complexities. L_r average length of reads, N_r number of reads, S the number of solid k-mers.

	Compact Flye	Original Flye
Approx Count	$\theta(L_r * N_r)$	$\theta(L_r * N_r)$
Exact Count	$O(L_r * N_r + S^2)$	$O(L_r * N_r + S^2)$
Indexing	$O(L_r * N_r + S^2)$	$O(L_r * N_r)$
Assembly	$O(L_r * N_r + S^2 + N_r^2)$	$O(L_r * N_r + N_r^2)$

bioinformatics tools, we wanted to compare time and space results of our proposal with those obtained by the original software, but without significantly changing the underlying algorithm. Thus, we maintain the original A-Bruijn algorithm for genome assembly, as changes on the algorithm would lead to different results with a higher/lower number of contigs and/or longer/shorter contigs.

3.4 Comparison of time and space

We include a summary of the time complexities for our proposal and the original method in Table 1. Worst-case time complexities of the approximate and the exact counting phases are the same for both approaches. However, these worst-case scenarios are too pessimistic. For instance, in the case of compact Flye, the worst-case scenario occurs when all the keys are mapped to the same entry of the hash table, and thus it degenerates to a linked list. This is extremely rare in practice when using well-designed hash functions. In the case of the original Flye, collisions may cause resizing and rehashing the table. This may not cause a quadratic time in the number of solid k-mers, but it actually has a great impact, and it will be reflected in the experimental section.

In the indexing phase, original Flye has a cost of $O(1)$ per processed k-mer, whereas the compact version has to deal with collisions plus the sorting of the arrays. However, collisions only occur when $k > 15$, and arrays are short, therefore, as we will see in the experimental section, only when $k > 15$, compact Flye pays a price in time. However, the use of the index during the assembly phase (employed to search for overlaps between reads) does not introduce significant changes in times in any case, since the dominant cost is the quadratic cost in the number of reads, which is the same for both approaches.

TABLE 2: Space complexities. GSF denotes $GenomeSizeFactor$, S the number of solid k-mers, Cov the coverage, t the threshold used in the counting, and $outp$ the assembly output.

	Compact Flye	Original Flye
Appr. Count	$GSF * 4^{15} * t$	$GSF * 4^{15} * 8$
Exact Count	$O(S)$	$O(S^2)$
Indexing	$O(S * Cov)$	$O(S^2 + S * Cov)$
Assembly	$O(S * Cov + outp)$	$O(S^2 + S * Cov + outp)$

Table 2 shows the space complexity. In the approximate counting, we use t bits (or $\log\lceil(t-1)\rceil + 1$, if $k > 15$) per entry instead of 8 bits of the original Flye. The second and more important difference is that in the exact count and the index, the size of the structure used by the original approach is tied to the number of collisions, doubling the data structure when the number of collisions surpasses a threshold. However, our data structure is only tied to the simple bitmap b, of constant size, and only increases the size of the entries linearly with the number of solid k-mers that passed the first filter (S). On the contrary, in the case of the original Flye, collisions and therefore duplications pose a big price in space. These duplications of the hash table are inherited during the indexing phase, therefore, in the space complexity the original Flye has an additional S^2 term.

As summary, we trade off time for the sake of reducing space consumption. In the time complexity, the only significant difference is the presence of an additional S^2 term caused by the fact of having linked lists instead of a constant-time Cuckoo hashing, which is a very pessimistic scenario that does not occur in practice, as lists are generally short. The counterpart is that the original Flye has in its space complexity that additional S^2 term in the exact counting, indexing, and assembly phases, precisely due to the use of that very efficient Cuckoo hashing strategy. However, in

practice, the price in space that Flye has to pay is much bigger in comparison with the worsening in times of the compact version. Compact data structures are more memory friendly, and this yields even slight improvements in time in some cases, as we will see in the experimental section.

4 Experimental evaluation

4.1 Experimental framework

All experiments were run on an Intel® Core™ Xeon es2470 CPU @ 2.3 GHz (32 cores), 64 GB of RAM, over Debian GNU/Linux 10 (buster). All programs were coded in C++. We use version 2.4 of Flye. Our code is available at https://bitbucket.org/bfreirec1/compactflye. To ensure the availability of all datasets, we have collected them and uploaded them into the following repository: https://bitbucket.org/bfreirec1/datasets-compactflye.

We have run the tests with two different k-mer sizes: the k value recommended by Flye for each dataset ($k = 15$ for the smallest datasets and $k = 17$ for the largest datasets), and a higher k value, more concretely, $k = 31$, for all datasets. All experiments were run five times for the smallest datasets and three times for the largest datasets, and we report the average results.

We have used five datasets from Oxford Nanopore Technology (ONT) and PacBio (PB) sequencers, some of which were used also by the authors of Flye [18]. We describe now the datasets, and include some properties in Table 3.

- `BACTERIA-PB` dataset[12] contains data gathered with a PacBio RS II System and P4-C2 chemistry on a size selected 20kb library of *E. coli* K12 substr. MG1655.
- `BACTERIA-ONT` dataset[13] also contains reads from whole-genome shotgun sequencing of the model organism *E. coli* K-12 substr. MG1655, but generated on a MinION device.
- `WORM` dataset[14] contains Pacific Biosciences reads (coverage 40x) from a Bristol mutant strain of *C. elegans* genome of length 100 Mb (6 chr.).
- `DROSOPHILA-PB` dataset[15] contains Pacific Bioscience reads (coverage 120x) from a subline of the ISO1 strain of Drosophila melanogaster.
- `DROSOPHILA-ONT` dataset[16] contains reads from Oxford Nanopore technology (coverage 30x) from a subline of the ISO1 strain of Drosophila melanogaster.

4.2 Results

4.2.1 Memory consumption

Figure 4 shows the memory consumption of the original Flye and that of our improved version. In all cases we obtain important improvements, ranging from 22% to 47% less space. We require almost half the space for almost all experiments, except for `WORM` with $k = 31$, which uses 78% of the space needed by the original version. The most remarkable result is observed for `DROSHOPILA-ONT` with $k = 31$. In this case, the physical memory of the machine (64 GB) was not enough for the original Flye. On the contrary, our version finished successfully, which shows that our version is more scalable due to better memory usage.

4.2.2 Time performance

Figure 5 shows the processing times of the complete process. One of the most significant improvements is obtained for `DROSHOPILA-ONT` with $k = 17$ (the recommended setup), where our method is 13% faster, whereas, as explained, with $k = 31$, original Flye did not run in our machine. In `WORM` dataset, the results are on a par, whereas in the smallest datasets, our method is between 5% and 11% faster with $k = 15$ and between 8% and 18% with $k = 31$. For `DROSHOPILA-PB`, results are on a par when using $k = 17$ and 25% faster for $k = 31$. These results show that improved memory usage and, therefore better scalability, can even lead to better runtimes with large datasets when physical memory is nearly exhausted, or when the datasets are smaller, but significant parts of the data structures can be kept in higher levels of the memory hierarchy, like in our `BACTERIA` datasets.

We have measured the time spent at each of the four phases of the assembly process. We show the results in Figure 6. In the small datasets, the pre-assembly phases are faster with our bitmap-based data structures compared to the Cuckoo based hash table of Flye, whereas for the assembly phase, both methods are on a par.

The time complexity of the approximate count is the same in both approaches, $O(1)$ per processed k-mer, still compact Flye obtains an improvement of 33% in the `WORM` dataset, but it is on a par in the case of `DROSOPHILA-ONT`. However, in the exact count, here we can see clear differences. Our method has a worst-case cost of $O(S)$ per processed k-mer, while the original Flye has $O(1)$ access time if we exclude the duplications of the Cuckoo hash table needed to keep that $O(1)$ access time. However, as seen, each duplication requires a considerable waste of time. This implies that compact Flye is between 1.20 times and 4.24 times faster than the original version. The counterpart is in the indexing phase, where compact Flye has $O(L_r * N_r + S^2)$ time due to the sorting of the arrays and the mixture of data from several k-mers in the lists of the index, which is worse than the $O(L_r * N_r)$ of the original FLye, and this can be seen in the largest datasets, where our method is between 37% and 48% slower. Nevertheless, our index based on a bitmap does not slow down the assembly. As it can be seen from the figures, both approaches are on a par in all cases.

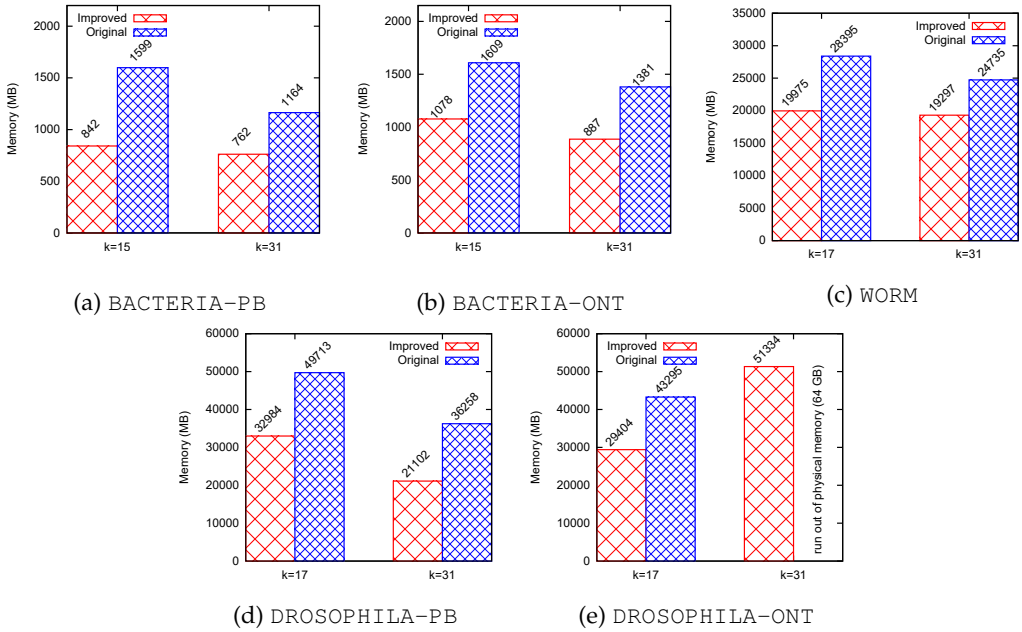
4.2.3 Analysis of underlying data structures

One question that may arise is that, during the approximate counting, if instead of using t bits (or $\log\lceil(t-1)\rceil + 1$ bits) for counters, a 1-byte counter would be faster. Moreover, is that solution scalable when the threshold increases? For answering these questions, we present an experiment varying the value of threshold t, and show the results in Figures 7–8.

12. https://github.com/PacificBiosciences/DevNet/wiki/E.-coli-20kb-Size-Selected-Library-with-P4-C2
13. http://lab.loman.net/2015/09/24/first-sqk-map-006-experiment/
14. https://github.com/PacificBiosciences/DevNet/wiki/C.-elegans-data-set
15. https://github.com/PacificBiosciences/DevNet/wiki/Drosophila-sequence-and-assembly
16. https://www.ebi.ac.uk/ena/data/view/SRR6702603

TABLE 3: Datasets used in the experiments.

Dataset	Species	Est. Gen. Size	Techn.	Cover.	Number of reads	Aver. bp	Max bp
BACTERIA-PB	*E. coli*	5 Mbp	PB	50x	48,048	8,637	41,331
BACTERIA-ONT	*E. coli*	5 Mbp	ONT	50x	50,966	9,753	57,229
WORM	*C. elegans*	100 Mbp	PB	40x	1,481,552	10,958	55,460
DROSOPHILA-PB	*D. melanogaster*	175 Mbp	PB	120x	3,227,724	9,303	44,766
DROSOPHILA-ONT	*D. melanogaster*	175 Mbp	ONT	30x	663,784	6,956	446,050



(a) BACTERIA-PB (b) BACTERIA-ONT (c) WORM

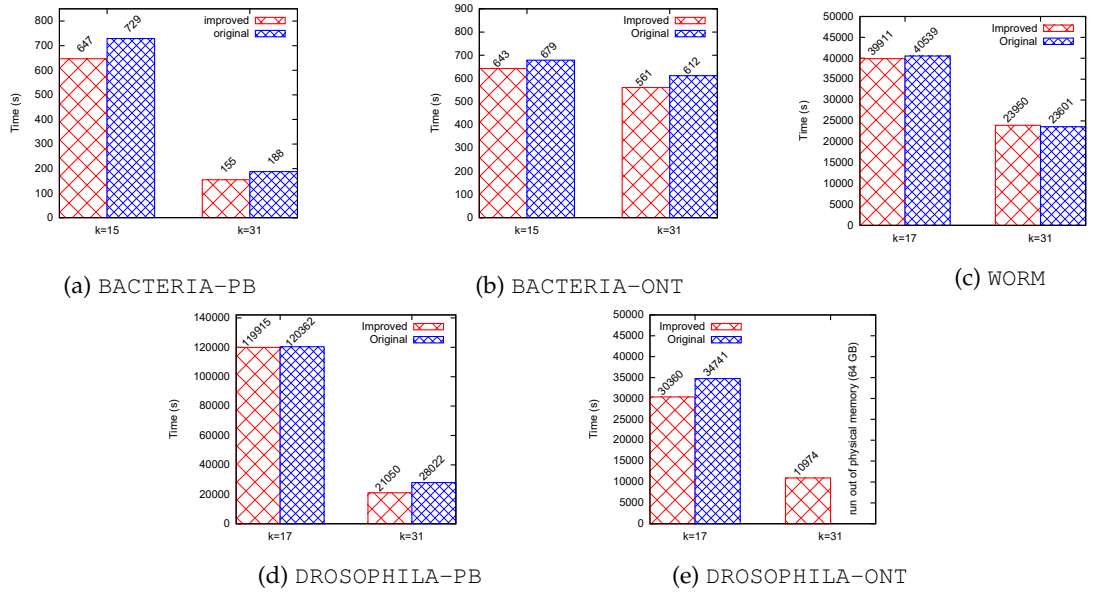
(d) DROSOPHILA-PB (e) DROSOPHILA-ONT

Fig. 4: Main memory peak (in Megabytes).

Recall that Flye chooses automatically the correct value for this threshold, ranging between 2 and 5. More specifically, Flye sets $t = 2$ for WORM and DROSOPHILA-ONT, $t = 4$ for BACTERIA-PB and BACTERIA-ONT, and $t = 5$ for DROSOPHILA-PB. For this experiment, we hard-coded the values between 2 and 5. However, we must note that, by doing this, the assembly obtained can be of a lower quality, or even Flye may not get an assembly at all.

As explained, our method for the approximate counting is based on identifying the first bitmap with a 0-bit in the corresponding entry and changing its value to 1. However, instead of a sequential scan, our implementation requires constant time by using bitwise shifts. We compare our approach with that of the original Flye and also with the use of a 1-byte counter. We can see the time results for BACTERIA-PB with $k = 15$ in Figure 7(a), where a counter using 1 byte is slightly faster for the approximate counting step, but requiring much more memory, as shown in Figure 8(a). We can also observe that times for our approach remain stable, regardless of the value of t, and always below the times of the original Flye. In addition, there are no significant differences between using t-bits counters and 1-byte counters when we take into account the subsequent phases (exact counting and indexing). In terms of space, we can see in Figure 8(a) that, during the approximate counting step, the solution using t bits requires more space when using a larger t, but not as much as the 1-byte counters. In fact, the space used for the 1-byte counters only for the approximate counting is even larger than the space required by the rest of the phases of our improved version. Moreover, the 1-byte counters also require more space than the original solution of Flye for the approximate counting. In any case, the original Flye requires much more memory for the following steps. In general, we can see that the global memory consumption tends to decrease when t grows. This is due to the fact that if the threshold augments, then fewer k-mers pass the first filter, and thus this may lead to a decrease in memory consumption.

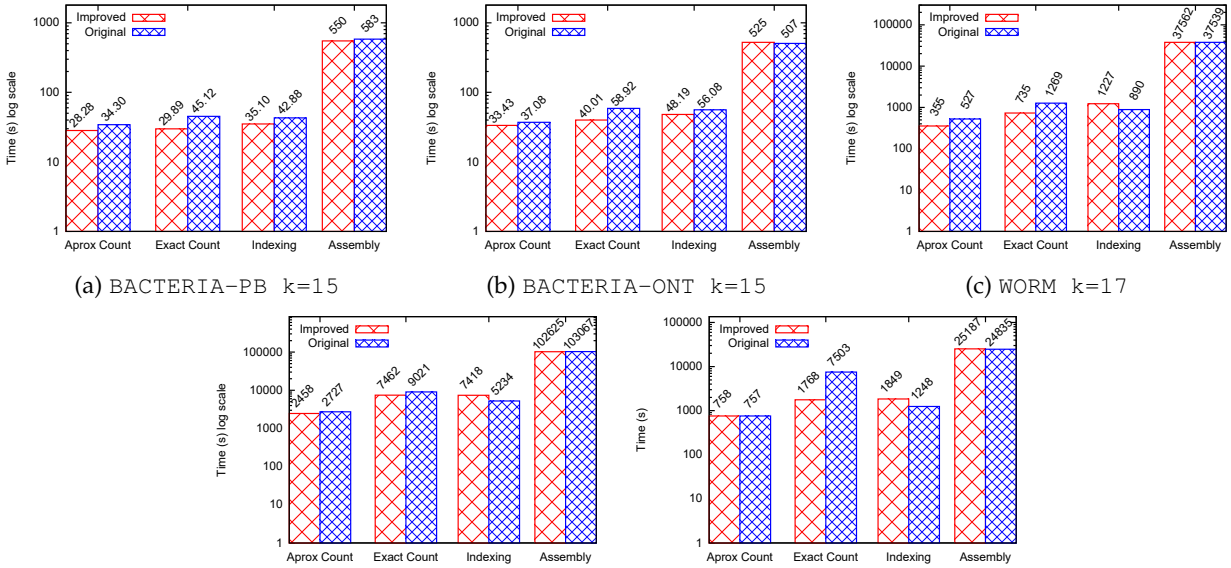
In Figures 7(b) and 8(b), we can see the same experiment for WORM dataset with $k = 17$. In this case, compact Flye uses a counter of $\log\lceil(t-1)\rceil$ bits. Times for the approximate counting remain in the same order as using 1-byte counters when t increases. This is expected, as the only difference is the use of counters using bits not aligned to bytes, and, as seen, this does not significantly affect the times. In any case, our approach is clearly faster than the original method, between 37–70% for this step. In terms of peak memory consumption, we observe the same pattern as the one described for the smallest dataset. However, for this dataset, there are no big differences when t decreases, as the number of k-mers that pass the first filter remains similar for the different values of t: there is a reduction only of 12% comparing those k-mers that pass the filter when $t = 2$ and $t = 5$ for WORM, but 83% reduction for BACTERIA-PB. Thus, we can see that our approach is not only more efficient in terms of space and time compared to the original Flye, but also more stable



(a) BACTERIA-PB (b) BACTERIA-ONT (c) WORM

(d) DROSOPHILA-PB (e) DROSOPHILA-ONT

Fig. 5: Processing time (in seconds).



(a) BACTERIA-PB k=15 (b) BACTERIA-ONT k=15 (c) WORM k=17

(d) DROSOPHILA-PB k=17 (e) DROSOPHILA-ONT k=17

Fig. 6: Processing time (in seconds with log scale) of the different phases.

when the value of t varies, as the space/time results of the original Flye are very dependant on the number of k-mers that pass the first filter.

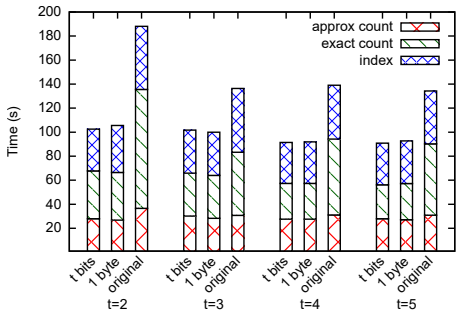
4.2.4 *Energy study*

Even though energy was not considered during the design of our proposal, we also measured the energy consumption of both tools. It was measured using Perf, which uses the Intel RAPL (Running Average Power Limit) energy estimates. As it can be seen in Figure 9, our proposal obtains reductions in energy consumption for all datasets, with improvements of 3–8% when $k = 15$ or $k = 17$, and reaching an improvement of 26% in the case of DROSOPHILA-PB when $k = 31$.

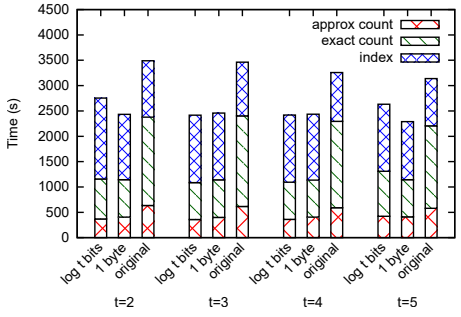
5 Discussion

As seen in the previous section, the improved version has better memory consumption. Flye allocates considerably large amounts of new memory when an insertion in the hash table, which uses a Cuckoo strategy, reaches a given number of collisions. Therefore, as the input size grows, the amount of memory needed by Flye grows much faster. With our approach, when the input datasets are huge, our growth speed is the same as for small cases. Thus, at the early stages of the process, the allocated memory grows fast, although not even close to the original Flye, but in the last stages our rhythm falls heavily.

In principle, better memory usage yields a more scalable system. A prove of this is that our improved version suc-

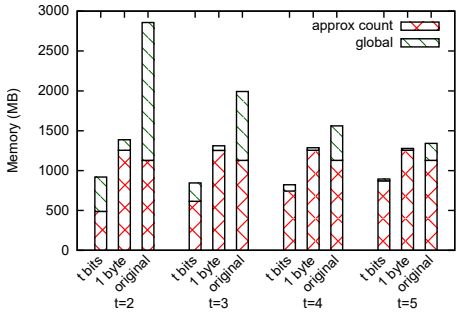


(a) BACTERIA-PB k=15

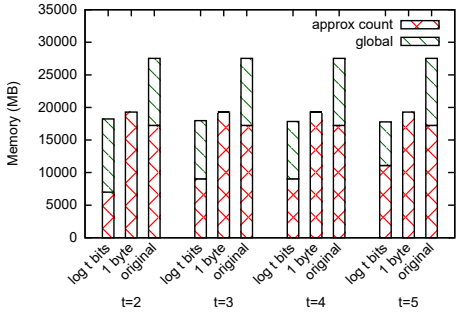


(b) WORM k=17

Fig. 7: Processing time (in seconds) of each of the steps (approximate counting, exact counting, and indexing) when using different data structures and varying the threshold value (t) for the approximate counting step.



(a) BACTERIA-PB k=15



(b) WORM k=17

Fig. 8: Peak memory consumption for the approximate counting and the complete process when using different data structures and varying the threshold value (t) for the approximate counting step.

cessfully assembled DROSOPHILA-ONT with $k = 31$ in our machine of 64 GB, whereas the original Flye was not able.

Moreover, as can be observed in Figure 6, the counting phases are faster in the new version. Observe that our enhancements are designed, in principle, to save space. Indeed, observe that in the approximate counting phase, the process is the same in both implementations, except that in our version, we do not use a hash table, but a bitmap. In the exact count and the indexing, we have an additional cost due to collisions that are kept in the same entry.

Therefore, why our version is faster? We measured the different procedures included in the exact count, and the gain is due to the insertion and update procedure of the hash table. In BACTERIA-PB, this procedure consumes 11.82 seconds in our version, whereas the original version requires 17.82; in WORM, the new version consumes 663.50 seconds versus the 1077.43 seconds of the original.

To better determine the origin of this improvement, we measured the cache references using *cachegrind*.[17] In BACTERIA-PB, our code made 157 billions references to the instructions cache, whereas the original one required 192 billions references; in WORM, the new version issued 1,840 billions references versus the 2,733 billions of the original. Moreover, our version required much fewer accesses to the data. In BACTERIA-PB, it performed 26 billions references versus the 44 billions of the original; and in WORM, 287 billions versus 717 billions. This shows that Cuckoo hash is penalized by the doubling procedures.

6 Conclusions

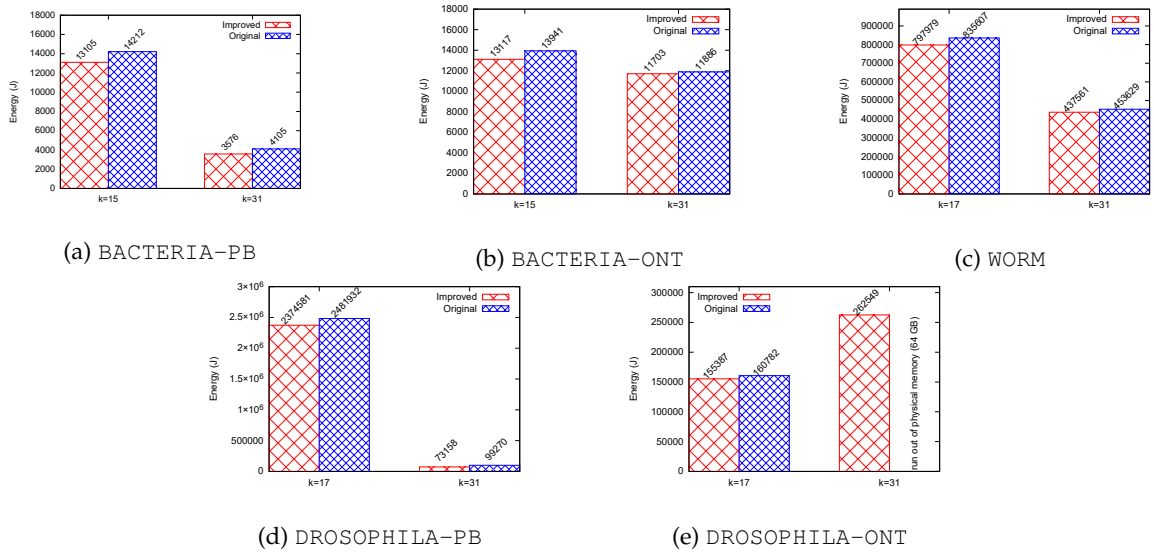
We have successfully modified the original Flye software in order to obtain a more efficient version of the same software, both in terms of space usage and execution time. The enhancements are mainly found in the k-mer counting phase, where we were able to obtain the exact same results with less memory consumption and even faster.

The improvements in memory consumption are considerable, halving the space required in most cases, and in the processing time from being on a par up to obtaining decreases of 25%. More importantly, we are able to assemble datasets that the original Flye is not able to process. In addition, as a side effect, our method saves between 3–8% of energy in general, and up to 26% for one of the experiments.

This implies a more scalable and faster software, which also requires less energy consumption. These memory-, time- and energy-efficient approaches will contribute to the advance of in-field analysis that are now becoming possible thanks to the advances on portable and real-time DNA sequencing and the appearance of affordable and portable handheld devices, such as the Oxford Nanopore's MinION and SmidgION.

As future work, we plan to further reduce the space consumption by counting, selecting and indexing the k-mers in compressed form in main memory. This is not an easy task, and traditional compressors cannot be used, as we must be able to decompress a given k-mer individually. Moreover, this problem becomes even harder, as it requires an on-line compression of the k-mers, that is, compressing

17. https://www.valgrind.org/docs/manual/cg-manual.html



(a) BACTERIA-PB (b) BACTERIA-ONT (c) WORM

(d) DROSOPHILA-PB (e) DROSOPHILA-ONT

Fig. 9: Energy consumption (in Joules).

them during the traversal of the reads, and without storing all the k-mers in main memory.

Acknowledgments

This research has received funding from: the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie [grant agreement No 690941]; CITIC Research Center, funded by "Consellería de Cultura, Educación e Universidade from Xunta de Galicia", supported in an 80% through ERDF Funds, ERDF Operational Programme Galicia 2014-2020, and the remaining 20% by "Secretaría Xeral de Universidades" (Grant ED431G 2019/01); Xunta de Galicia/FEDER-UE under Grants [IG240.2020.1.185; IN852A 2018/14] and Ministerio de Ciencia e Innovación under Grants [TIN2016-78011-C4-1-R; PID2019-105221RB-C41; PID2020-114635RB-I00; FPU17/02742].

References

[1] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, J. Zhang, G. M. Weinstock, F. Isaacs, J. Rozowsky *et al.*, "The real cost of sequencing: scaling computation to keep pace with data generation," *Genome biology*, vol. 17, no. 1, p. 53, 2016.

[2] A. Sboner, X. J. Mu, D. Greenbaum, R. K. Auerbach, and M. B. Gerstein, "The real cost of sequencing: higher than you think!" *Genome biology*, vol. 12, no. 8, p. 125, 2011.

[3] H. Stevens, *Life out of sequence: a data-driven history of bioinformatics*. USA: University of Chicago Press, 2013.

[4] R. Leinonen, H. Sugawara, M. Shumway, and I. N. S. D. Collaboration, "The sequence read archive," *Nucleic acids research*, vol. 39, no. suppl_1, pp. D19–D21, 2010.

[5] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *PLOS Biology*, vol. 13, no. 7, pp. 1–11, 07 2015.

[6] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de bruijn graphs," *BMC Bioinformatics*, vol. 12, no. 1, p. 354, 2011.

[7] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE, 2014, pp. 437–448.

[8] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "Hipmer: An extreme-scale de novo genome assembler," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, 2015, pp. 14:1–14:11.

[9] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "Swap-assembler: scalable and efficient genome assembly towards thousands of cores," *BMC Bioinformatics*, vol. 15, no. 9, p. S2, 2014.

[10] C. Gamboa-Venegas and E. Meneses, "Comparative analysis of de bruijn graph parallel genome assemblers," in *2018 IEEE International Work Conference on Bioinspired Intelligence (IWOBI)*. IEEE, 2018, pp. 1–8.

[11] D. Kleftogiannis, P. Kalnis, and V. B. Bajic, "Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures," *PloS one*, vol. 8, no. 9, p. e75505, 2013.

[12] R. Chikhi, A. Limasset, and P. Medvedev, "Compacting de bruijn graphs from sequencing data quickly and in low memory," *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 2016.

[13] N. R. Brisaboa, R. Cao, J. R. Paramá, and F. Silva-Coira, "Scalable processing and autocovariance computation of big functional data," *Software: Practice and Experience*, pp. 123–140, 2018.

[14] H. Plattner and A. Zeier, *In-memory data management: technology and applications*. Springer, 2012.

[15] G. Jacobson, "Succinct static data structures," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, Jan. 1989, tech Rep CMU-CS-89-112.

[16] G. Navarro, *Compact Data Structures – A practical approach*. New York, NY: Cambridge University Press, 2016.

[17] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," in *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, 2005, pp. 27–38.

[18] Y. Lin, J. Yuan, M. Kolmogorov, M. W. Shen, M. Chaisson, and P. A. Pevzner, "Assembly of long error-prone reads using de bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 113, no. 52, pp. E8396–E8405, 2016.

[19] M. Kolmogorov, J. Yuan, Y. Lin, and P. A. Pevzner, "Assembly of long, error-prone reads using repeat graphs," *Nature biotechnology*, vol. 37, no. 5, pp. 540–546, 2019.

[20] A. D. Tyler, L. Mataseje, C. J. Urfano, L. Schmidt, K. S. Antonation, M. R. Mulvey, and C. R. Corbett, "Evaluation of oxford nanopore's minion sequencing device for microbial whole genome sequencing applications," *Scientific Reports*, vol. 8, no. 1, 07 2018, 11907.

[21] L. E. Kafetzopoulou, S. T. Pullan, P. Lemey, M. A. Suchard, D. U. Ehichioya, M. Pahlmann, A. Thielebein, J. Hinzmann *et al.*, "Metagenomic sequencing at the epicenter of the Nigeria 2018 Lassa fever outbreak," *Science*, vol. 363, no. 6422, pp. 74–77, 2019.

[22] R. M. Idury and M. S. Waterman, "A new algorithm for dna sequence assembly," *Journal of Computational Biology*, vol. 2, no. 2, pp. 291–306, 1995, pMID: 7497130.

[23] J. D. Kececioglu and E. W. Myers, "Combinatorial algorithms for dna sequence assembly," *Algorithmica*, vol. 13, no. 1, p. 7, Feb 1995.

[24] Z. Li, Y. Chen, D. Mu, J. Yuan, Y. Shi, H. Zhang, J. Gan, N. Li, X. Hu, B. Liu, B. Yang, and W. Fan, "Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph," *Briefings in Functional Genomics*, vol. 11, no. 1, pp. 25–37, 2012.

[25] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.

[26] B. G. Jackson, P. S. Schnable, and S. Aluru, "Parallel short sequence assembly of transcriptomes," *BMC Bioinformatics*, vol. 10, no. 1, p. S14, 2009.

[27] A. I. Tomescu and P. Medvedev, "Safe and complete contig assembly through omnitigs," *Journal of Computational Biology*, vol. 24, no. 6, pp. 590–602, 2017.

[28] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.

[29] P. A. Pevzner, H. Tang, and G. Tesler, "De novo repeat classification and fragment assembly," *Genome research*, vol. 14, no. 9, pp. 1786–1796, 2004.

[30] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, p. 552–581, Jul. 2005.

[31] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, no. 3, p. R25, 2009.

[32] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[33] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.

[34] N. Välimäki and E. Rivals, "Scalable and versatile k-mer indexing for high-throughput sequencing data," in *Bioinformatics Research and Applications*. Springer Berlin Heidelberg, 2013, pp. 237–248.

[35] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *Journal of Algorithms*, vol. 48, no. 2, pp. 294–313, 2003.

[36] F. Claude, A. Fariña, M. Martínez Prieto, and G. Navarro, "Compressed q-gram indexing for highly repetitive biological sequences," in *Proceedings of the 10th Int. Conf. on Bioinformatics and Bioengineering (BIBE)*, 2010, pp. 86–91.

[37] T. C. Conway and A. J. Bromage, "Succinct data structures for assembling large genomes," *Bioinformatics*, vol. 27, no. 4, pp. 479–486, 01 2011.

[38] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de bruijn graphs," in *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, 2012, pp. 225–235.

[39] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, "On the representation of de bruijn graphs," in *Research in Computational Molecular Biology*, R. Sharan, Ed. Cham: Springer International Publishing, 2014, pp. 35–55.

[40] E. A. Rødland, "Compact representation of k-mer de bruijn graphs for genome read assembly," *BMC Bioinformatics*, vol. 14, no. 1, p. 313, 2013.

[41] J. He, H. Yan, and T. Suel, "Compact full-text indexing of versioned document collections," in *Proceedings of the 18th ACM conference on Information and knowledge management*, 2009, pp. 415–424.

[42] J. He, J. Zeng, and T. Suel, "Improved index compression techniques for versioned document collections," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, 2010, pp. 1239–1248.

[43] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Relative lempel-ziv compression of genomes for large-scale storage and retrieval," in *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, 2010, pp. 201–206.

[44] G. Navarro and V. Sepúlveda, "Practical indexing of repetitive collections using relative lempel-ziv," in *2019 Data Compression Conference (DCC)*, 2019, pp. 201–210.

[45] S. Deorowicz and S. Grabowski, "Robust relative compression of genomes with random access," *Bioinformatics*, vol. 27, no. 21, pp. 2979–2986, 2011.

[46] S. Kreft and G. Navarro, "On compressing and indexing repetitive sequences," *Theoretical Computer Science*, vol. 483, pp. 115–133, 2013.

[47] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Optimized relative lempel-ziv compression of genomes," in *Proceedings of the 34th Australasian Computer Science Conference*, 2011, pp. 91–98.

[48] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi, "A faster grammar-based self-index," in *International Conference on Language and Automata Theory and Applications*. Springer, 2012, pp. 240–251.

[49] G. Jacobson, "Space-efficient static trees and graphs," in *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989, pp. 549–554.

[50] Raman, Raman, and Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings Symposium on Discrete Algorithms (SODA)*, 2002.

[51] D. Clark, "Compact pat trees," Ph.D. dissertation, University of Waterloo, 1997.

[52] M. Kokot, M. Długosz, and S. Deorowicz, "Kmc 3: counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, 2017.

[53] G. Rizk, D. Lavenier, and R. Chikhi, "Dsk: k-mer counting with very low memory usage," *Bioinformatics*, vol. 29, no. 5, pp. 652–653, 2013.

[54] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in dna sequences using a bloom filter," *BMC Bioinformatics*, vol. 12, no. 1, p. 333, 2011.

[55] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970.

[56] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122 – 144, 2004.

[57] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 05 2018.

Borja Freire received his bachelor degree in Computer Science at the University of A Coruña in 2016 and master degree in Bioinformatics at the same university in 2018. He is now a PhD student of the Doctorate Program in Computer Science at University of A Coruña, and he has been awarded a FPU fellowship to complete his doctorate.

Susana Ladra received the bachelor's degree in mathematics from the National Distance Education University (UNED), in 2014, and the master's in computer science engineering and the Ph.D. degree in computer science from the University of A Coruña, in 2007 and 2011, respectively. She is currently an Associate Professor with the Universidade da Coruña. She is the Principal Investigator of several national and international research projects. She has published more than 40 articles in various international journals and conferences. Her research interests include design and analysis of algorithms and data structures, and data compression and data mining in the fields of information retrieval and bioinformatics.

José R. Paramá has PhD in computer science from the University of A Coruña. Since 1997 he is a professor at the University of A Coruña, and since 2008, Associate Professor. He has participated in more than twenty research projects funded by European, regional and national administrations, and in more than thirty R&D contracts. He is the author of more than thirty scientific publications and more than sixty scientific conferences.