BIOINFORMATICS ORIGINAL PAPER

Vol. 29 no. 7 2013, pages 933–939
doi:10.1093/bioinformatics/btt070

Data and text mining

Advance Access publication February 14, 2013

Efficient comparison of sets of intervals with NC-lists

Matthias Zytnicki*, YuFei Luo and Hadi Quesneville

URGI, INRA Versailles, Plant Biology and Breeding Division, 78026 Versailles Cedex, France

Associate Editor: Michael Brudno

ABSTRACT

Motivation: High-throughput sequencing produces in a small amount of time a large amount of data, which are usually difficult to analyze. Mapping the reads to the transcripts they originate from, to quantify the expression of the genes, is a simple, yet time demanding, example of analysis. Fast genomic comparison algorithms are thus crucial for the analysis of the ever-expanding number of reads sequenced.

Results: We used NC-lists to implement an algorithm that compares a set of query intervals with a set of reference intervals in two steps. The first step, a pre-processing done once for all, requires time $\mathcal{O}[\#R\log(\#R)+\#Q\log(\#Q)]$, where Q and R are the sets of query and reference intervals. The search phase requires constant space, and time $\mathcal{O}(\#R+\#Q+\#M)$, where M is the set of overlaps. We showed that our algorithm compares favorably with five other algorithms, especially when several comparisons are performed.

Availability: The algorithm has been included to S–MART, a versatile tool box for RNA-Seq analysis, freely available at http://urgi.versailles.inra.fr/Tools/S-Mart. The algorithm can be used for many kinds of data (sequencing reads, annotations, etc.) in many formats (GFF3, BED, SAM, etc.), on any operating system. It is thus readily useable for the analysis of next-generation sequencing data.

Contact: matthias.zytnicki@versailles.inra.fr

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on March 27, 2012; revised on December 20, 2012; accepted on February 7, 2013

1 INTRODUCTION

With the advent of high-throughput sequencing, bioinformatics must analyze a large amount of data every day. Modern sequencers can generate several hundred millions of sequences in a week for a price that is affordable to more and more labs. When a reference genome is available, the first task is to map the reads on the genome. Many mapping tools are now available and research is active on this topic [see Langmead *et al*. (2009) for instance]. For RNA-Seq, the second step may be the assignment of the mapped read to the transcripts they originate from, to estimate the expression of the genes (Anders, 2011). In general, the genomic comparison of the mapped reads with a reference annotation is the basis of many analyses: comparison of putative transcription factor binding sites with up-regulated genes (Blankenberg *et al*., 2010; Giardine *et al*., 2005; Goecks *et al*., 2010); detection of the single-nucleotide polymorphisms that are located in coding regions (Renaud *et al*., 2011); processing *de novo* transcript sequences to determine if they represent known or novel genes (Roberts *et al*., 2011). These three examples involve a comparison of two annotations, and the problem has been addressed often. However, high-throughput sequencing, for the amount a data it produces, requires optimized algorithms for its analysis.

Most tools model the reads or annotation as intervals, or lists of intervals when different elements are modeled (exons, UTRs, etc.). These intervals are considered along a reference, which usually is a chromosome or a scaffold. Thus, comparing RNA-Seq reads with known transcripts reduces to comparing a set of query intervals (the reads) with a set of reference intervals (the exons of the transcripts).

Every efficient algorithm requires a dedicated data structure, such as an indexed database, an indexed flat file [such as a BAM file (Li *et al*., 2009)], an R-tree or NC-lists (nested containment lists) (Alekseyenko and Lee, 2007). These structures are usually built once during the pre-processing step, and can be reused for other analyses. Although these structures may take considerable amount of time to build, the balance is usually favorable to pre-processed structures when several comparisons are performed, as the time spent for the comparison itself is considerably reduced. This observation leads to the conception of the BAM format, now widely used in the bioinformatics community.

With the notable exception of the fjoin algorithm (Richardson, 2006), almost all the algorithms previously described only get all the reference intervals that overlap with one given query interval: most algorithms have been designed to retrieve all the intervals a user can see when he selects a given window in a genome browser (Kent *et al*., 2002). Whereas these algorithms can be used to compare two sets by comparing each query interval, one after the other, with the reference intervals, we will show here how comparing the whole query set with the reference set can be more efficient.

Among the possible data structures presented to compare intervals, NC-lists (Alekseyenko and Lee, 2007) are one of the most promising. NC-lists have been first described to retrieve all the reference intervals that overlap with a single interval. Their structure is compact (a simple set of two arrays, L and H), the algorithm is fast in practice and the search phase requires only constant space, which is compulsory when handling several hundreds of millions of reads. The key idea of NC-lists is to perform binary dichotomic search on the list of reference intervals. But dichotomic search cannot be performed when some intervals are contained (or nested) inside other intervals, so NC-lists arrange intervals into lists—the L array—where no two intervals are nested. If some intervals are nested inside an ancestor interval,

*To whom correspondence should be addressed.

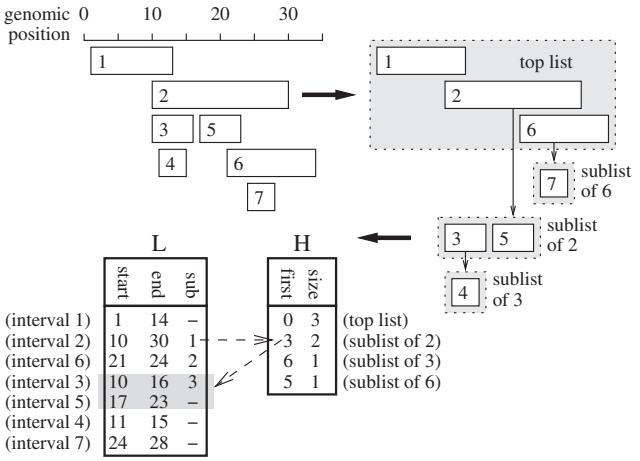

Fig. 1. Transforming a set of ordered intervals an NC-list. All the intervals have been previously sorted according to their increasing start position and, in case of tie, decreasing end position. Because intervals 3, 4 and 5 are nested inside interval 2, they are removed from the top list (which consists in intervals 1, 2 and 6) and inserted into another sublist. Intervals 3 and 5 are moved to the sublist of 2. Similarly, interval 4 is nested inside interval 3, and thus moved to another sublist. When an interval is nested into two intervals (as it is the case for the interval 7, which is nested in 2 and 6), the right-most interval that contains it is chosen. Here, it is interval 6. An NC-list is a set of two arrays, L and H. Each line of L stores the start and end positions of an interval, as well as an index to the H array. The L data are stored so that the intervals that are in the same list appear contiguously. For each sublist, a corresponding line of the H array stores the index of its least interval and the size of the list. As highlighted by the arrows, the sublist of interval 2 (line 1 of the L array, which is a zero-based structure) is the line 1 of the H array. The sublist starts at index 3 of the L array and contains 2 intervals (the intervals 3 and 5)

they are stored in a separate sublist using the H array (see Fig. 1). NC-lists can be built in linearithmic time [i.e. of the form $\mathcal{O}(n \log n)$], using linear space (actually, only five integers are stored per interval). In their article, the authors presented a recursive dichotomic algorithm, equivalent to Alg. 1, which uses NC-lists. It is claimed that getting all the reference intervals that overlap with a query interval could be done in time $\mathcal{O}[\log(\#R) + \#M]$, where R is the reference set and M the pairs query/reference that overlap, but this is not accurate for some cases (see section 3.1).

In this article, we will present an algorithm, which relies on NC-lists, and provides all the pairs query intervals/reference intervals that overlap. In a pre-processing step, the algorithm sorts the query and the reference intervals. It then builds a NC-list for the reference intervals. In the search phase, the algorithm compares every query interval with the reference intervals in time $\mathcal{O}(\#R + \#Q + \#M)$. All together, the algorithm takes $\mathcal{O}[\#R \log(\#R) + \#Q \log(\#Q) + \#M]$. Although the complexity of the whole algorithm is not better than already known algorithms, the runtime complexity is significantly lower than other constant-space algorithms. As such, our algorithm is especially useful when performing multiple comparisons on large sets of data, such as in an RNA-Seq data analysis.

Algorithm 1 Original algorithm

```
/* The algorithm is initiated with search(top_list, q) */
search(list, q):
    M ← ∅
    r ← findFirstOverlap(list, q)
    /* findFirstOverlap implements dichotomic search to
       find the first element of list that overlaps with q */
    while r ∈ list ∧ (r overlaps with q) do
        M.add({r} ∪ search(r.children, q))
        r ← r.next
    return M
```

Algorithms 2 Simplified algorithm

```
nfo ← R[0], M = ()
for each q in Q.sorted() do
    r ← nfo, nfo ← None
    loop
        if r.end < q.start then r ← r.next
        else if q.end < r.start then
            if nfo = None then nfo ← r
            break
        else    /* q and r overlap */
            if nfo = None then nfo ← r
            M.add((q, r))
            r ← r.firstChild or r.next
return M
```

2 METHODS

To compare two sets of intervals, we also used a NC-list for the reference set, and query intervals are simply sorted by their start position. Our aim is to find all the query intervals that overlap with at least one reference interval. The main idea of the algorithm is that knowledge from the comparison between a query interval and a reference interval will be used for the comparison of the next query interval. A sketch of the algorithm, which provides all the pairs of query/reference intervals that overlap, is presented in Alg. 2. The actual algorithm is slightly more complex, and is described in section 3.2. It uses a special variable, *nfo* (for *next first overlap*), which stores the first reference query that may overlap with the next query interval.

3 ALGORITHMS

3.1 Original algorithm

Definitions. We will describe and analyze here the problem of the comparison of genomic intervals. We will first formally define our data and the NC-list structure.

DEFINITION 1. *An interval $i = (a, b)$ is an element of $\mathbb{N}^2$ such that $a \leq b$. By convention, we set $i.start = a$ and $i.end = b$.*

For two intervals i and j, we define:

$i < j \quad \Leftrightarrow (i.end < j.start)$ (*i is before j*)

$i <> j \quad \Leftrightarrow ((i.start \leq j.end) \wedge (j.start \leq i.end))$ (*i and j overlap*)

$i \subset j \quad \Leftrightarrow ((j.start \leq i.start) \wedge (i.end \leq j.end))$ (*i is contained in j*)

The NC-list construction algorithm supposes that the intervals have been previously sorted following a total order:

Definition 2. *[O_L, (Alekseyenko and Lee, 2007)]. A total order $\leq$ is defined, such that*

$$\forall (r, r') \in R^2, r \preceq r' \Leftrightarrow \begin{cases} r.start \leq r'.start \\ \vee \\ (r.start = r'.start) \wedge (r.end \geq r'.end) \end{cases}$$

The associated asymmetric relation is defined by

$$\forall (r, r') \in R^2, (r \prec r') \Leftrightarrow \neg (r' \preceq r)$$

If two different intervals, r and r', have the same coordinates $[(r.start = r'.start) \wedge (r.end = r'.end)]$, we define $r \prec r'$ or $r' \prec r$ arbitrarily.

To avoid ambiguity, the $\prec$ relation is subscripted by the set it relates to (namely $\prec_Q$ for the query set and $\prec_R$ for the reference set).

The successor of an element $r \in R$ with respect to the order $\prec_R$ will be noted succ(r), when it exists.

The construction phase of the NC-list groups the sorted intervals into lists, such that an interval that is contained in another interval is moved into the sublist of the container interval.

Definition 3. *We define the subelement of an interval by*

$$\forall r \in R, r.sub = \{s \in R : r \prec_R s \ \wedge s \prec_R \min_{\prec_R} \{r' \in R : (r \prec_R r') \wedge (r' \not\subset r)\}\}$$

The children of an interval are as follows:

$$\forall r \in R, r.children = \{c \in r.sub : \forall c' \in r.sub \backslash \{c\}, c \not\subset c'\}$$

r.children is also called the sublist of r.
The parent of an interval is defined as follows:

$$\forall (c, p) \in R^2, (c.parent = p) \Leftrightarrow (p \in r.children)$$

*Finally, r.ancestors is the list of ancestors of $r \in R$, i.e. the list $(r_1, r_2, r_3, \ldots, r_n)$ such that r_1 has no parent, $r_k = r_{k+1}$.*parent *and $r_n = r.parent$.*

The previous definition provides a way to build the nested containment structure from a sorted list of intervals: given an interval r, all its successors that are nested into r should be found in a list under r. They are the subelements of r. The children of r are the subelements that are right under r (i.e. there is no other interval nested in r that contains a child of r).

Note that an interval r may have no parent. In this case, we set $r.parent = \emptyset$ and all the intervals that have no parent form the top list.

Definition 4. *The NC-list of a set of intervals is a tree-like data structure such that*

- *each node contains sorted intervals,*
- *the root node is the list of intervals that has no parent,*
- *there is an edge between every interval and the list of its children.*

Notice that a NC-list is not a tree because an edge connects a node (the parent interval) to a list of nodes (the children intervals).

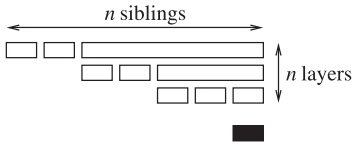

Fig. 2. Pathological case concerning the algorithm between one query interval (in black) and several reference reads (white)

Revised complexity. The original algorithm, equivalent to Alg. 1, considers a query interval q and a set of reference intervals R. It gives the elements of R that overlap with q. We will show here that the algorithm presented by Alekseyenko and Lee (2007) does not have the complexity claimed in the article. In the example in Figure 2, the announced complexity does not hold.

The example has a nested structure, where each reference interval has the same number of siblings. For each list of siblings, none but the last one has children. The query overlaps every last sibling of each list. The number of layers is equal to the number of siblings, n. Here, $\#M = n$ and $\#R = n^2$. Executing the algorithm yields a time complexity of $\mathcal{O}[n \log(n)]$, as n binary searches are performed (one for each layer). However, the expected complexity is $\mathcal{O}[\log(n^2) + n] = \mathcal{O}(n) < \mathcal{O}[n \log(n)]$.

3.2 New algorithm

Definition 5. *The problem of the comparison of sets of intervals considers two sets of intervals, Q and R (hereafter named the query set and the reference set) and finds all the pairs $(q, r) \in Q \times R$ such that q and r overlap.*

Notice that there is no assumption on the two sets: elements from the query or reference sets may be nested or not, have different sizes, etc.

Definition 6. *Consider a query read q. Let*

- *$B[q] = \{r \in R : r < q\}$ be the set of reference intervals that are before q.*
- *$M[q] = \{r \in R : r <> q\}$ be the set of reference intervals that overlap with q.*
- *$A[q] = \{r \in R : r > q\}$ be the set of reference intervals that are after q.*

Because $B[q]$, $M[q]$ and $A[q]$ are disjoint, and cover the entirety of R, $\{B[q], M[q], A[q]\}$ is a partition of R. Moreover, any optimized algorithm would of course try to compute $M[q]$ as fast as possible, while avoiding scanning $B[q]$ and $A[q]$. The two following lemma (their proof are omitted for they are straightforward) will help us skipping reading these sets.

Lemma 1. *If an interval is in $B[q]$, then all its subelements also are.*

A consequence, if a reference element is in $B[q]$, then its children intervals will not be compared with q.

Lemma 2.

$$\forall a \in A[q], \forall r \in R, (a \prec_R r) \Rightarrow (r \in A[q])$$

The previous lemma implies that if we scan the reference interval using the ordering $\prec_R$, the search can stop when the least

element of $A[q]$ is found. In other words, the greatest element of $M[q]$, when this set is not empty, is the predecessor of the least element of $A[q]$. There is no similar rule concerning the least element of $M[q]$ and $B[q]$, and characterizing the 'left frontier' of $M[q]$ is slightly more complex.

To do so, we will define here *nfo*. Informally, this variable is the least (using the ordering $\prec_R$) lowest (meaning that none of its children does) interval that overlaps with q. Because *nfo* overlaps with q, all its ancestors also do. Because it is the least variable that overlaps with q, the successors of *nfo* either overlap with q or are after q. In the algorithm, this variable is set when we compare a query interval q with the set of reference intervals, and it is the first interval that will be compared with the successor of q. We will prove the previous claims here.

Definition 7.

$$nfo[q] = \min_{\prec_R}\{\{r \in M[q] : r.children \subset B[q]\} \cup A[q]\}$$

nfo[q] may be undefined if B[q] and A[q] are empty. In this case, we define nfo[q] = None.

To help the reader, different configurations of the *nfo[q]* are described in Figure 3. We will prove that the predecessors of *nfo*, except for its ancestors, are all in $B[q]$.

Lemma 3. *Let* $m[q] = \min_{\prec_R}\{M[q] \cup A[q]\}$.
If $m[q]$ is undefined, we set $m[q] = None$.
If $m[q]$ is *None*, the $nfo[q]$ also is. *Otherwise*,

$$m[q] \in \{nfo[q]\} \cup nfo[q].ancestors$$

Proof. Let us suppose that $m[q]$ is not None (the proof is clear otherwise). If $m[q] \in A[q]$, then $m[q] = nfo[q]$ and the lemma is proved. Otherwise, let r be a reference element such that $(m[q] \in r.ancestors) \wedge (r.children = \emptyset)$. Such an element exists, otherwise the number of sublists would be infinite. Clearly, $r \in \{(r' \in M[q] : r'.children \subset B[q])\}$, so $nfo[q] \preceq_R r$. We have thus $m[q] \preceq_R nfo[q] \preceq_R r$ and $r \in m[q].sub$, which implies, by definition of the subelements, that $nfo[q] \in m[q].sub$, or $nfo[q] = m[q]$. This proves the lemma.

Corollary 4.

$$\forall r \in R, (r \prec_R nfo[q]) \Rightarrow (r \in B[q] \cup nfo[q].ancestors)$$

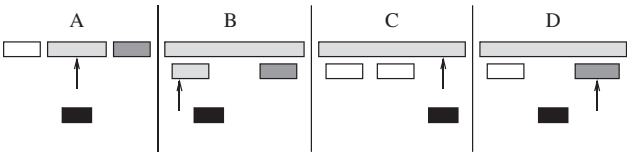

Fig. 3. Different configurations of the interval comparison problem. In every case, the query interval (q) is in black, and the other colors refer to the reference intervals. $nfo[q]$ is indicated by the arrow. To help the reader, reference intervals in $B[q]$ are white; the intervals in $M[q]$ are light gray; dark gray intervals are in $A[q]$. Case (**A**) is the simple case, the other cases are less intuitive. In case (**B**), we can observe that the first overlapping interval is not $nfo[q]$: it is the bottom-most overlapping element. In case (**C**), all the children of $nfo[q]$ are in $B[q]$. In case (**D**), $nfo[q]$ is in $A[q]$

As a result, suppose that we have found $nfo[q]$ and that we are looking for $nfo[q']$, with $q' = succ(q)$. Because $B[q] \subset B[q']$, the previous corollary implies that $nfo[q']$ is either a parent of $nfo[q]$ or one of its successors.

Lemma 5.

$$\forall m \in M[q], \forall r \in R,$$
$$((m \prec_R r) \wedge (m \notin r.ancestors)) \Rightarrow (r \in M[q] \cup A[q])$$

Proof. Let r be a reference interval such that $m \prec_R r \wedge m \notin r.ancestors$. The following assertions hold:

(1) $m.start \leq q.end \wedge q.start \leq m.end$ (q and m overlap),
(2) $m.start \leq r.start$ ($m \prec_R r$),
(3) $m.start > r.start \wedge m.end < r.end$ ($m \notin r.ancestors$).

From 2 and 3, we deduce that $m.end < r.end$. Comparing with 1, we deduce that $q.start < r.end$, and so $r \in M[q] \cup A[q]$. This proves the lemma.

Proposition 6.

$$\forall r \in R, r \succ_R nfo[q] \Rightarrow \begin{cases} r \in B[q] \text{ if } (nfo[q] \in M[q]) \vee \\ \quad (nfo[q] \in r.ancestors) \\ r \in (M[q] \cup A[q]) \text{ otherwise} \end{cases}$$

Proof. If $nfo[q] \in A[q]$, then Lemma 2 proves that r will be in $A[q]$. Otherwise, $nfo[q] \in M[q]$. In this case, by definition of $nfo[q]$, all its children are in $B[q]$, and by application of Lemma 1, all the intervals that are under $nfo[q]$ are in $B[q]$. Finally, by application of Lemma 5, all the reference intervals that are after $nfo[q]$, but not under it, are in $M[q] \cup A[q]$. This proves the proposition.

This last proposition implies that in general, all the elements greater than $nfo[q]$ could overlap with the successor of q. The only exception is when $nfo[q]$ overlaps with q. In this case, children intervals must be skipped. This is why we use a variable *skip*, which stores this configuration.

Algorithm. From the previous propositions, we can directly infer an algorithm, which is completely presented in supplementary materials. A loop iterating over the query elements is described in `findOverlap`. The algorithm that compares a query interval with the reference intervals is described in `findOverlapIter`. A last algorithm, `getNext`, shows how to get the successor of a reference interval (considering the ordering $\prec_R$).

Informally, the main algorithm directly jumps to the *nfo* reference element that had been computed by the previous query interval. It then checks the ancestors. Then, it scans forward. If the current reference is in $B[q]$, it jumps to the next interval. If the current reference is in $M[q]$, it goes down to the sublists, except if the variable *skip* is true. In such case, it directly jumps to the next interval. If the current reference is in $A[q]$, it stops. The variable *nfo* is updated when necessary.

Proposition 7. *$nfo[q]$ is the nfo computed in the algorithm.*

Proof. Let us consider a query interval q' and its successor q. We have $(M[q] \cap A[q]) \subset (M[q'] \cap A[q'])$. By corollary 4,

$m[q] = \min_{\prec_R}\{M[q] \cup A[q]\}$ is either a parent of $nfo[q']$ or one of its successors.

Besides, we have previously proved that $m[q] \in \{nfo[q]\} \cup nfo[q].ancestors$. Thus, starting from the previous nfo, checking its ancestors, then possibly going right until $m[q]$ is found, and then finally going down is enough to find $nfo[q]$. This is what the algorithm does.

PROPOSITION 8. *The time complexity of the algorithm is* $\mathcal{O}(\#Q + \#R + \#M)$.

PROOF. Let us consider the reference intervals that will be compared with q. Let q' be its predecessor. The reference intervals that are scanned are $B[q]\backslash B[q']$, $M[q]$, and the least element of $A[q]$. Because the sets $\{B[q]\backslash B[q'] : (q, q') \in Q^2, succ(q') = q\}$ are all disjoint, the total number of comparisons is $\mathcal{O}(\#Q + \#R + \#M)$.

Notice that the algorithm `findOverlap` sometimes needs to go from the child to the parent, and thus be able to visit the tree from bottom to top, whereas the original algorithm described in Alg. 1 is a typical top-down algorithm. To be able to go up, we added in the L table a new cell, which contains the address of the parent element in the L table.

*Transcript modeliz*ation. Transcripts usually are not simple interval, but a succession of several intervals, which are the exons. Similarly, the reads can also be splitted in several parts if they overlap the exon/exon junction. In our implementation, we modeled the query and the reference element as a single interval (the smallest interval that contains all the exons), and store these intervals into the NC-list. To avoid reporting the reads that are the introns, we also store, for each interval, a pointer to the memory address where the transcript or read is completely described. To do so, we simply added a new column in the L table, which stores the address. When an overlap is found, the full structure is retrieved and the query and reference intervals are compared in detail to report only true matches.

4 RESULTS

Comparison to other implementations. We show here the results of our algorithm when compared with several other published methods. The first is a simple NC-list algorithm, as presented by Alekseyenko and Lee (2007), which does not use any information between two consecutive query intervals, hereafter called 'nc'. The second method implements binning (Kent *et al*., 2002) using an indexed SQLite table, hereafter called 'bin'. We also implemented another flavor of this algorithm, called 'has', where the database has been replaced by a hash structure, such that the keys are the bins, and the values are lists of intervals. A forth algorithm is a binning table with segment tree, as described in Segtor (Renaud *et al*., 2011), called 'seg'. We also added FJoin (Richardson, 2006) ('fj'), which scans the previously sorted query intervals and reference intervals simultaneously to find overlaps. Our algorithm will simply be called 'new'.

Among the presented algorithms, only 'bin', 'nc' and 'new' have constant space complexities. The other algorithms, 'has', 'seg' (where the trees are stored in memory) and 'fj' (which has a linear space complexity), are thus not likely to work on the large amount of data modern sequencers generate, with a standard computer. For instance, in our implementation, the 'has' algorithm fills our RAM (4 GB) when the reference dataset contains 30 M intervals. Still, as they rely on in-memory data, they usually run faster on the sets they can handle.

For a fair comparison of all the algorithms, and to exclude any bias that would originate from the choice of the programming language used by the different methods, we re-implemented all the algorithms carefully as described by the articles. All the algorithms have exactly the same input, output and functionalities, which reflect a usual mapped reads/annotation comparison study. First, strand is ignored (as many RNA-Seq data have no strand information, and most algorithms, when described in their original articles, do not deal with this case). Second, each feature (hereafter a read or a transcript) is stored as a single interval. If an overlap is detected, the transcript is extracted from the input file (each method keeps track of the memory address of the features) and a second comparison is performed to check if the overlap is not located in the introns of the transcript, in which case the overlap is not reported. Last, the output file is a GFF3 file, which contains the query intervals that overlap with at least one reference element, and the list of the overlapping elements are added in the tags of the ninth field. These implementations, as well as the benchmark itself, are available in the S–MART toolbox. See supplementary materials for more information about these implementations.

Example on a real dataset. We downloaded three different publicly available RNA-Seq datasets: on yeast, fly and cress (available as SRR014335, SRR030228 and SRR346552 datasets in GEO). We mapped the reads with Bowtie (Langmead *et al*., 2009) on the reference genome and we compared the mapped reads with the annotation (the genome sequence and the annotations are both available from the Bowtie website). For each dataset, we reported the number of annotated transcripts (which are the reference intervals) as well as the number of reads (the query intervals). We used the six different algorithms previously mentioned. Run-time results are shown in Table 1. The first columns give the characteristics of the datasets: number of reads, number of transcripts and number of overlaps. The following columns give the run-time spent by the algorithms when the genes are the reference and the reads are the query.

As expected, 'has' and the 'fj' algorithms usually perform well on this dataset because the intervals are stored in memory.

Table 1. Characteristics of three real datasets, and run-time results (in thousands of seconds) for the six algorithms

Dataset	No. of reads	No. of transc.	No. of ov.	bin	has	seg	fj	nc	new
Yeast	10 M	9 k	20 M	5.1	3.2	4.3	[a]	4.8	3.4
Fly	3 M	183 k	10 M	2.5	1.3	1.9	1.1	2.1	1.4
Cress	20 M	245 k	58 M	17	9.2	13	[a]	14	9.1

[a]The program aborted for it needed too much memory (>4 GB).
No. of transc., number of annotated transcripts, used as reference; No. of ov., number of overlaps.

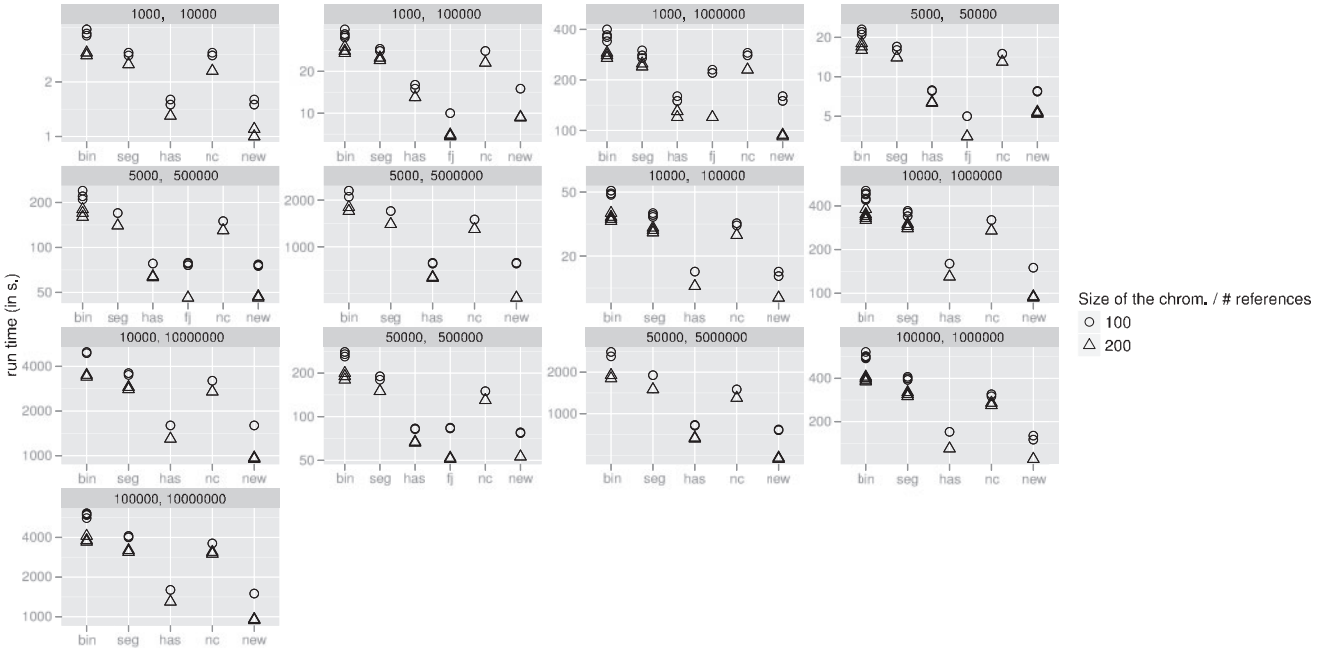

Fig. 4. Runtime of the algorithms. Each cell provides the runtime of each algorithm in seconds. The numbers of reference and query intervals are provided on top of each cell. Each configuration has been repeated five times with a genome size of 100× the number of reference intervals, and 200× the number of reference intervals. The 'fj' required too much RAM (>4 GB) to work on the largest datasets and is therefore not provided in these configurations

Our algorithm is still among the fastest ones. However, the pre-processing of our algorithm is by far the slowest one (see Supplementary Data). This is a typical trade-off between run-time speed and pre-processing-time speed because the 'bin' algorithm, the slowest algorithm in the comparison step, is the fastest algorithm in the pre-processing step among constant space methods.

Example on simulated datasets. We also generated several datasets to compare the algorithms in detail. The intervals ranged from 36 to 100 nt, the genome contained a single chromosome, ranging from 10 k to 2 M bp. The number of reference and query intervals varies from 100 to 100 k and 100 to 10 M elements, respectively. Each configuration was generated five times. The results in Figure 4 give the run-time results of each method. Our algorithm is still the fastest among the constant space complexity algorithms. The 'fj' required too much RAM (more than 4 GB) to work on the largest datasets.

Regarding the pre-processing step, our algorithm is the slowest one (see Supplementary Information) but overall, the balance is always favorable to our algorithm after three comparisons when compared with the 'bin', the 'seg' or the 'nc' algorithm.

Insertion in S–MART. S–MART (Zytnicki and Quesneville, 2011) is a versatile tool box for the analysis of RNA-Seq data. It contains many useful tools for the comparison of RNA-Seq data with respect to a given annotation: number of reads for each transcript, distance distribution between the reads and the closest transcripts, discovery of previously unknown transcribed loci, etc. We added a new tool, called `FindOverlapsOptim`, which implements the algorithm presented in this article. As a consequence, the algorithm can be used for many kinds of data (such as RNA-Seq reads, but also annotation of any feature) in many formats (GFF3, BED, SAM, etc.).

We included a so-called 'nclist' format in S–MART, which contains several NC–lists (one per chromosome), so that pre-processing can be done once for all. This pre-processing step can be performed using a separate tool called `ConvertToNCList`. These files can be used as input file by most tools of the S–MART suite, much like BED or GFF3 files.

We also implemented a second version of our algorithm in the S–MART tool called `CompareOverlapping`. This version is more flexible and accepts many different parameters: it may output the query elements only if they are collinear (or antisense) to the overlapping reference element, the query elements that are nested inside reference elements, the query elements that overlap the first 100 bp of the reference elements, etc. Because `CompareOverlapping` is much more flexible than `FindOverlapsOptim`, it is also substantially slower. Last, we added two versions of the much faster 'has' algorithm in S–MART, to be used when the query or the reference have moderate sizes.

The encapsulation of the algorithms within S–MART ensures that the presented method is not only a theoretical work, but also used in a tool that is readily available to biologists. For the computer scientists, we also implemented an API and executables in C++ so that they can embed them in their algorithms.

5 DISCUSSION

The method presented here uses NC-lists and provides a fast algorithm that compares two large sets of intervals efficiently.

To our knowledge, it is the first time that an algorithm with both linear time complexity and constant space complexity during the search phase is presented. This low run-time complexity comes at the cost of a high pre-processing time complexity, where the intervals should be sorted. However, this step is done only once and is far from untractable (the `samtools sort` algorithm is used routinely to sort BAM files). As a result, the algorithm presented in this article is adapted to multiple comparisons.

When we designed the algorithm, we had the idea in mind that it could help comparing features such as RNA-Seq data, which can amount to several hundreds millions reads. While this algorithm presents a theoretical interest by itself, we also encapsulated it in the S–MART tool box, which includes all the features to handle usual file formats. As a consequence, we hope this work will be useful for both computer scientists and biologists.

Funding: Y.L. was supported by the Plant Breeding and Genetics research division of the INRA and by the Groupement d'intérêt scientifique IBISA.

Conflict of Interest: none declared.

REFERENCES

Alekseyenko,A.V. and Lee,C.J. (2007) Nested containment list (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. *Bioinformatics*, **23**, 1386–1393.

Anders,S. (2011) HTSeq: analysing high-throughput sequencing data with python.

Blankenberg,D. *et al.* (2010) *Galaxy: A Web-Based Genome Analysis Tool for Experimentalists*. John Wiley & Sons Inc. Chapter 19, Unit 19.10.1–21.

Giardine,B. *et al.* (2005) Galaxy: a platform for interactive large-scale genome analysis. *Genome Res.*, **15**, 1451–1455.

Goecks,J. *et al.* (2010) Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biol.*, **11**, R86.

Kent,W.J. *et al.* (2002) The human genome browser at UCSC. *Genome Res.*, **12**, 996–1006.

Langmead,B. *et al.* (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**, R25.

Li,H. *et al.* (2009) The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, **25**, 2078–2079.

Renaud,G. *et al.* (2011) Segtor: rapid annotation of genomic coordinates and single nucleotide variations using segment trees. *PLoS ONE*, **6**, e26715.

Richardson,J. (2006) fjoin: simple and efficient computation of feature overlaps. *J. Comput. Biol.*, **13**, 1457–1464.

Roberts,A. *et al.* (2011) Improving Rna-Seq expression estimates by correcting for fragment bias. *Genome Biol.*, **12**, R22.

Zytnicki,M. and Quesneville,H. (2011) S-MART, a software toolbox to aid RNA-seq data analysis. *PLoS ONE*, **6**, e25988.