AIList: an effective algorithm for searching genomic interval data

Jianglin Feng[1,2], Aakrosh Ratan[1], and Nathan Sheffield[1,*]

[1] Center for Public Health Genomics, University of Virginia, Charlottesville, Virginia, VA 22908, USA; [2] Structure Solutions & Scientific Computing, P.O. Box 69, Crozet, VA 22932, USA.

ABSTRACT

Motivation: Genomic data are signals recorded as intervals on the genome coordinates. Interval-based comparing operation is fundamental to nearly all genomic analysis. As the available genomic data rapidly grows, developing effective and dedicated algorithms for searching this data becomes necessary.

Results: An augmented interval-list (AIList) algorithm is presented. The interval list is first sorted by interval *start,* then partitioned into two approximately normalized sub-lists, and then an augmenting list of the maximum interval *end* is added to each sub-list. The query time is $O(n+logN)$, where n is the number of overlaps and N is the size of the dataset. Tested with genomic interval datasets, code based on this algorithm runs significantly faster than code based on existing algorithms such as augmented interval-tree (AITree), nested containment list (NCList) and R-tree.

Availability: http://github.com/databio/AIList

Contact: nsheff@virginia.edu

Supplementary information: an alternative algorithm of AIList, source code, genomic data used in the study are provided.

1. Introduction

A genomic interval is defined by two coordinates that represent the *start* and *end* locations of a continuous signal on a chromosome. In general, an interval set can be collected or integrated from many experimental datasets and an interval may have attributes like signal level, tissue/cell type, chromatin state etc. The interval search becomes a problem when the interval set possesses the coverage or containment relationship, i.e., one interval contains another interval; which breaks the total order of the interval set: the order based on the *start* is different from the order based on the *end*. The general interval search problem can be defined as follows:

Given a set of N intervals $R = \{r_1, r_2, \ldots, r_N\}$ and a query interval q, find the subset S of R that intersect q:

$$S(q) = \{r \in R \mid (r.start < q.end \wedge r.end > q.start)\}$$

Following the literature standard, the interval here is defined as half open region $r = [start, end)$ and the number of query intervals is assumed large.

This is a fundamental problem in genomic data analysis (Giardine, B. et al., 2005; Birney et al., 2006; Li, 2009) and several approaches have been developed or practiced (Cormen et al., 2001; Kent et al., 2002; Richardson, 2006; Alekseyenko etal., 2007; Quinlan, A. and Hall, I., 2011; Neph, S., 2012). Currently, the nested containment list (NCList) by Alekseyenko et al., the augmented interval-tree (AITree) by Cormen et al. and the R-tree by Kent et al. have gained popularity and practicality. Comparing to the naive solution that sequentially compares each of the N intervals with the query, these algorithms successfully skip large portion of the comparison. Their time complexities are O(log$N$+n+m), where n is the number of overlaps (the query result set) and m is the average number of *extra* comparisons that are made in order to get the n overlaps; and for genomic interval datasets, N is usually several orders of magnitudes larger than n or m. Clearly, m is the main difference among these algorithms and it is resulted from different sources. For NCList, if the query covers two or more sublists, then *extra* binary searches are needed; for AITree, one needs to compare all interval nodes marked by the augmenting value and not all of them intersect the query; and for R-tree, intervals in the found bin are scanned, in which many intervals may not overlap the query.

In this paper, we present a new algorithm, the augmented interval-list (AIList), that can generally reduce the extra number of comparisons and thus achieve higher search efficiency.

2. AIList data structure and query algorithm

2.1 A simple augmented interval list

Augmenting is a common way to extend the functionality of the basic data structures (Cormen et al., 2001). Here, the genomic interval data is directly treated as an interval list and then it is sorted by the *start*. Then an augmenting value is added to each data element (interval). Two different augmenting values may be used: the running maximum *end* value *MaxE*, or the sorted *end* value *SortedE*. In the following discussion, we will use *MaxE*, which is similar to the interval tree and the *SortedE* method can be found in the *Supplementary Information*.

Figure 1(a) shows a simple augmented interval list (AIList). The augmenting value *MaxE* is the maximum *end* value counting from the first interval. It reflects the containment relationship among the intervals: the 2nd interval [3,8) contains the 3rd interval [5,7) and they have the same *MaxE* value of 8; similarly the 4th interval [7,20) contains the next three intervals and they all have the the same *MaxE* value of 20. In another word, *MaxE* marks a list of containment sublists.

To find the overlaps for a query, such as [*9*,*12*), we first do a binary search over the list to get the index of the last interval I_E that has $start < 12$, which is the 5th element [9,10,20); then we only need to check intervals from the 5th to the 4th because their $MaxE > 9$ and stop at the 3rd [5,7,8) since its $MaxE < 9$ and we can skip all comparisons on the left. This is very efficient since we only checked one extra element at the stop. But if the query is [17,21), then we need to check 6 interval elements from the 8th [19,30,30) back to the 3rd [5,7,8), which contains 4 extras, so it is not as efficient. The first query represents the best case and the second is the worst case for this list. Since there is another long coverage with *MaxE* value of 30 in the list, the overall or average query efficiency would be more close to the worst case.

2.2 Enhanced augmented interval list

To improve the overall query efficiency, we need to reduce the length of the long coverage sublists. This can be achieved by taking out the intervals with longest coverage and put them in a separated list or attach them to the end of original list. Figure 1(b) shows an example. By taking out intervals that covers the next *3* (coverage length criterion) or more elements in the above list, we get two sublists *L1* and *L2.* Now in *L1* there are only two containment sublists and both have the minimum length of *2*, and there is no containment in *L2* ; so query in both lists would be or close to be the best. To get the index I_E in *L1* and *L2* now we need two separated binary searches. Since the index of an item in the original list is the sum of the index in *L1* and the index in *L2*, an optional structure list *Opt* can be used to facilitate the search, see figure 1(c). *Opt* keeps the sorted *start* from 1(a) and the link (*index*) to *L1.* Using the above example query [*9*,*12*), the binary search on *start* of *Opt* is *5*, so the index I_E in *L1* is *Opt* [*5*].*index* = 4 and that in *L2* is *5-4* =*1.* In practice, the time taken by the binary search is only a very small portion of the total time, so it is not very necessary to use this optional structure.

It is apparent that a list can be split into three or more sublists with two or more coverage length criteria in a similar way as above. But this is not necessary for real genomic sequence datasets that we have tested. Also the optimal length criterion for a list to be split is found to be about *20*; the size of the split *L2* is usually much smaller than *L1*, and many datasets do not need to be split. Similar to other algorithms mentioned above, the query time complexity for AIList is $O(\log N + n + m)$, but the average number of extra comparisons m should be, in general, smaller than others'.

Algorithm 1: AIListSearch(*L1, L2, Opt, start, end*)

```
$H \leftarrow 0$
$i \leftarrow$ BinarySearchStart($Opt, end$)
$i1 \leftarrow Opt[i].index$
$i2 \leftarrow i - i1$
while $i1 > 0$ and $L1[i1].MaxE > start$ do
    if $L1[i1].end > start$:
        $H \leftarrow H \cup L1[i1]$
    $i1 \leftarrow i1 - 1$
end while
… repeat the above while section for L2 with i2 …
Return $H$
```

3. Results

AIList is implemented in *C*. To evaluate the efficiency of *AIList*, we compared the performance of *AIList* with interval tree (*AITree*), nested containment list (*NCList*) and *R-tree*. For *AITree* we used the rbtree-based interval tree from Linux kernel, which was implemented by Landley (2007); for NCList we used the C-code intervaldb.c implemented by the original authors; and for R-tree we used the popular C++ implementation *BEDTools* by Quinlan and Hall (2011). Seven genomic datasets that have number of intervals from ~200 thousand to 128 million are used in the evaluation, where the two small datasets are from BedTools site and the other five are from UCSC site. One of the median sized dataset *chainRn4* is used as the query set, the other six are used as database set, see Table 1. Information about all these data and code can be found in Supplementary Information.

Figure 2 shows the total time with respect to the data size for AIList, AITree, NCList and BEDTools. The time includes the dataset loading, data structure construction and searching and result output, so it reflects the overall efficiency of the algorithms. For small size datasets, AIList is *20-50%* times faster than AITree and NCList and *2* times faster than BEDTools (v2.25.0); for large size datasets, AIList can be *4-5* times faster than AITree and NCList and up to *18* times faster than BEDTools.

4. Conclusion

An augmented interval-list (AIList) algorithm is presented. The interval list is first sorted by interval *start,* then partitioned into two approximately normalized sub-lists, and then an augmenting list of the maximum interval *end* is added to each sub-list. The query time is $O(n+logN)$, where n is the number of overlaps and N is the size of the dataset. Tested with genomic interval datasets, code based on this algorithm runs significantly faster than code based on existing algorithms such as augmented interval-tree (AITree), nested containment list (NCList) and R-tree.

REFERENCES

Birney, E. *et al.* (2006) Ensembl 2006. *Nucleic Acids Res.*, **34** (Database issue).

Cormen, T.H., Leiserson, C.E., and Rivest, R.L. (2001) Chapter 14.3. *Introduction to Algorithms (2nd ed.),* MIT Press and McGraw-Hill, Cambridge, MA, pp.311-317.

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

Neph, S. et al. (2012) BEDOPS: high performance genomic feature operations. *Bioinformatics*, **28**, 1919-1920.

Richardson, J.E. (2006) fjoin: simple and efficient computation of feature overalps. J. Computat. Biol., 13, 1457-1464.

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

Li, H., et al. (2009) The sequence alignment/map (SAM) format and SAMtools. Bioinformatics, 25, 2048-2049.

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

Quinlan, A.R., and Hall, I.M. (2011) BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics* **26**, 841-842.

Supplementary Information

1. Interval list augmented with a list of sorted *interval end*

The augmenting list *SortedE* is the sorted list of all interval ends. First find the last interval I_E in R that satisfies $r.start < q.end$ with a binary search, which excludes all intervals on the right. Since *SortedE* is sorted by the *end*, we can find the leftmost element I_S that is equal to or larger than *q.start.* This excludes all elements on the left of I_S. The number of intersections is I_E-I_S $+1$. This algorithm does not directly find I_S, instead it simultaneously enumerates *SortedE* and R from I_E to the left.

Algorithm 1: AIListSearch(*L1, L2, Opt, start, end*)

```
$H \leftarrow 0$
$i \leftarrow$ BinarySearchStart($Opt, end$)
$i1 \leftarrow Opt[i].index$
$i2 \leftarrow i - i1$
$t \leftarrow 0$
while $i1 > 0$ and $L1[i1].SortedE > start$ do
    $t \leftarrow t+1$
    if $L1[i1].end > start$:
        $H \leftarrow H \cup L1[i1]$
        $t \leftarrow t - 1$
    $i1 \leftarrow i1 - 1$
end while
while $t > 0$ do
    if $L1[i1].end > start$:
        $H \leftarrow H \cup L1[i1]$
        $t \leftarrow t - 1$
    $i1 \leftarrow i1 - 1$
end while
… repeat the above for L2 with i2 ...
Return $H$
```

2. Testing Data

3. Source code

REFERENCES

Layer, R., Skadron, K., Robins, G., Hall, I. And Quinlan, A. (2013) Binary Interval Search: a scalable algorithm for counting interval interactions. *Bioinformatics* **29**, 1-7.