AIList: effective algorithms 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: Two augmented interval-list (AIList) algorithms and their alternatives are presented. The interval list is first sorted by interval *start*, then an augmenting list of either a sorted interval *end* or maximum interval *end* is added. These algorithms are significantly faster than existing algorithms.

1. Introduction

1.1 The genomic interval search problem

A genomic interval or region is defined by two coordinates that represent the *start* and *end* locations of a continuous signal on a chromosome; for single nucleotide polymorphisms (SNPs), the *start* and *end* locations are the same. In general, a region set can be collected from many experimental data sets and different regions may intersect; also a region may have attributes like signal level, tissue/cell type, chromatin state etc. The region search problem can be defined as follows.

Given a set of N regions $rList = \{r_1, r_2, \ldots, r_N\}$ and a query region q from a set $qList = \{q_1, q_2, \ldots, q_M\}$, find the subset S of $rList$ that intersect q:

$$S(q) = \{r \in rList \mid (r.start \leq q.end \wedge r.end \geq q.start)\}$$

Figure 1(a) shows an example of the problem with a single query. A simple but inefficient solution to the problem is to sequentially compare each q with each region in $rList$, which requires O($N\,M$) comparing operations. Assume that the region set $rList$ can be sorted with complexity O($N\,log(N)$), then using M binary searches with complexity O($M\,log(N)$) would be an efficient solution. Since a region has two numbers, the region set cannot be sorted totally and binary search cannot be applied directly. The maximum number of intersections is N in the worst case when q covers the whole genome, so the complexity of the search is always O($N\,M$). However, for genomic region data q is normally several orders of magnitude smaller than the genome size and the number of intersections is several orders of magnitude smaller than N; so, the best solution in practice is to enumerate the least number (K) of non-intersecting regions in the set. If we define the search efficiency per query as the ratio of (N-K) over N, then for an efficient algorithm we expect to get an efficiency of ~1 (K<<N) instead of ~0 (K~N).

There has not been a dedicated and efficient solution to this problem. The recursion-based search algorithm of the interval-tree structure [Cormen et al., 2002] is not efficient because it involves too many extra checks of the node's left and right links and also it has significant functional calling overhead; the binary interval search algorithm [Layer et al, 2013] only counts the number of overlaps; and BEDTools [Quinlan et al., 2011] sequentially enumerates all intervals in a subset.

2. Methods

Augmenting is a common way to extend the functionality of the basic data structures. The genomic interval data can be represented by an **interval list** and in the following, two different augmenting lists are proposed. In both algorithms, the interval list is sorted first. The sorting can be based on either the *start* or the *end*. In the following discussion, we assume it is sorted by the *start*.

2.1 Augmenting with a list of maximum *region end*

Start from the first element, the augmenting list *MaxE* is the running maximum of the region *end*. The last interval I_E that satisfies $r.start <= q.end$ can be obtained with a binary search, then all the intersections can be obtained by enumerating only the intervals marked by *MaxE.*

Algorithm 1: Interval-list augmented with maximum end

```
Input: interval list $rList$, query interval list $qList$
Output: list of intersections $Hits$ for each query $q$ in $qList$
Function: AISearchMaxEnd($rList$, $qList$) begin
    SortIntervalListByStart($rList$)
    $MaxE \leftarrow$ ConstructAugmentingListFromEnd($rList$)
    for each $q$ in $qList$:
        $I_E \leftarrow$ BinarySearchListStart($rList$, $q.end$)
        while $I_E >= 0$ & $MaxE[I_E] >= q.start$:
            if $rList[I_E].end >= q.start$:
                add $rList[I_E]$ to $Hits[q]$
            $I_E$--
    return $Hits$
```

Figure 1(b) shows an example with a single query interval *[7, 9]*. Binary search finds the last interval that satisfies $r.start<=9$ as *region 4 [8, 12]*, so regions on the right don't need to check. From *region 4* to the left, there are *3* regions that satisfy $MaxE >= 7$. These are the only regions that need to be checked for intersections. Among these *3* regions, *region 2* and *region 4* are the intersections since they also satisfy $r.end >= 7$.

2.2 Augmenting with a list of sorted *region end*

The augmenting list *SortedE* is the sorted list of all region ends. First find the last interval I_E in *rList* that satisfies $r.start <= q.end$ with a binary search, which excludes all regions 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*, which excludes all elements on the left. The number of intersections is $I_E - I_S + 1$. This algorithm does not directly find I_S, instead it simultaneously enumerates *SortedE* and *rList* from I_E to the left.

Algorithm 2: Interval-list augmented with sorted end

```
Input: interval list $rList$, query interval list $qList$
Output: list of intersections $Hits$ for each query $q$ in $qList$
Comment: $t$ is the number of intersections to be found
Function: AISearchSortedEnd($rList$, $qList$) begin
    SortIntervalListByStart($rList$ )
    $EList \leftarrow$ ConstructListFromEnd($rList$)
    $SortedE \leftarrow$ SortList($EList$)
    for each $q$ in $qList$ :
        $I_E \leftarrow$ BinarySearchListStart($rList$,  $q.end$)
        $t \leftarrow 0$
        while $I_E >= 0$ & $SortedE[I_E] >= q.start$ :
            $t{+}{+}$
            if $rList[I_E].end >= q.start$ :
                add $rList[I_E]$ to $Hits[q]$
                $t{-}{-}$
            $I_E{-}{-}$
        while $t > 0$ :
            if $rList[I_E].end >= q.start$ :
                add $rList[I_E]$ to $Hits[q]$
                $t{-}{-}$
            $I_E{-}{-}$
    return $Hits$
```

Figure 1(c) shows an example. Binary search finds the last interval that satisfies *r.start<=9* is *region 4.* From *region 4* to the left, check *2* regions that satisfy *SortedE >=7* , which indicates that there are a total of *2* intersections. These two intersections are *region 2* and *region 4* , which satisfy *r.end>=7.*

2.3 Alternative algorithms

Since both *MaxE* and *SortedE* are sorted, another binary search may be used to identify the leftmost interval I_S that satisfies $MaxE[i]>=q.start$ and $SortedE[i]>=q.start$, respectively.

Algorithm 1a: Interval-list augmented with maximum end

```
Input: interval list $rList$ , query interval list $qList$
Output: list of intersections $Hits$ for each query $q$ in $qList$
Function: AISearchMaxEndDualBinary($rList$, $qList$) begin
    SortIntervalListByStart($rList$ )
    $MaxE \leftarrow$ ConstructAugmentingListFromEnd($rList$)
    for each $q$ in $qList$ :
        $I_E \leftarrow$ BinarySearchListStart($rList$,  $q.end$ )
        $I_S \leftarrow$ BinarySearch($MaxE$,  $q.start$ )
        while $I_E >= I_S$:
            if $rList[I_E].end >= q.start$ :
                add $rList[I_E]$ to $Hits[q]$
            $I_E$--
    return $Hits$
```

Algorithm 2a: Interval-list augmented with sorted end

```
Input: interval list $rList$ , query interval list $qList$
Output: list of intersections $Hits$ for each query $q$ in $qList$
Function: AISearchSortedEndDualBinary($rList$, $qList$) begin
    SortIntervalListByStart($rList$ )
    $EList \leftarrow$ ConstructListFromEnd($rList$)
    $SortedE \leftarrow$ SortList($EList$)
    for each $q$ in $qList$ :
        $I_E \leftarrow$ BinarySearchListStart($rList$,  $q.end$)
        $I_S \leftarrow$ BinarySearch($SortedE$,  $q.start$ )
        $t \leftarrow I_E - I_S + 1$
        while $t > 0$ :
            if $rList[I_E].end >= q.start$ :
                add $rList[I_E]$ to $Hits[q]$
                $t$--
            $I_E$ --
    return $Hits$
```

2.4 Time complexity analysis and efficiency comparison

The overall complexity for all 4 algorithms above is $O(N \log(N) + MK)$, where N, M are the number of intervals in *rList* and *qList* respectively and K is the average number of intervals per query that need to be checked. Here we compare their efficiencies.

In the following we mainly consider the operation of comparison or its equivalents and ignore simpler operations like loop indexing. The construction part of **algorithm 1** involves $N \log(N) + N$ operations, and the search part does $M \log(N) + 2M K$. Factor 2 reflects that one needs to check both *MaxE* and *rList.end*.

Because *SortedE* requires sorting, the construction part of **algorithm 2** involves $2N\log(N) + N$ operations, and the search part is $M \log(N) + MK + MK_1$, where K_1 is the average number of intersections per query and $K_1 <= K$. The overall performance of **algorithm 2** is generally worse than **algorithm 1** because of the extra sorting. In a database system where *rList*, *MaxE* and *SortedE* are all presorted or pre-constructed, **algorithm 2** can be slightly faster than *algorithm 1* because it involves less checkings.

For **algorithm 1a** and **algorithm 2a** the search part involves $2M \log(N) + MK$ operations. The average intersections per query (size *100 bp*) is about*1* for single genomic interval datasets, and keeps small for small-scale genome databases. So **algorithm 1** and **algorithm 2** are more efficient for routine genomic dataset analysis. For a heavily integrated database where $K >> \log(N)$, **algorithm 1a** and **algorithm 2a** should be more efficient.

3. Results

REFERENCES

Cormen, T.H., Leiserson, C.E., and Rivest, R.L. (2002) Introduction to Algorithms.

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

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.

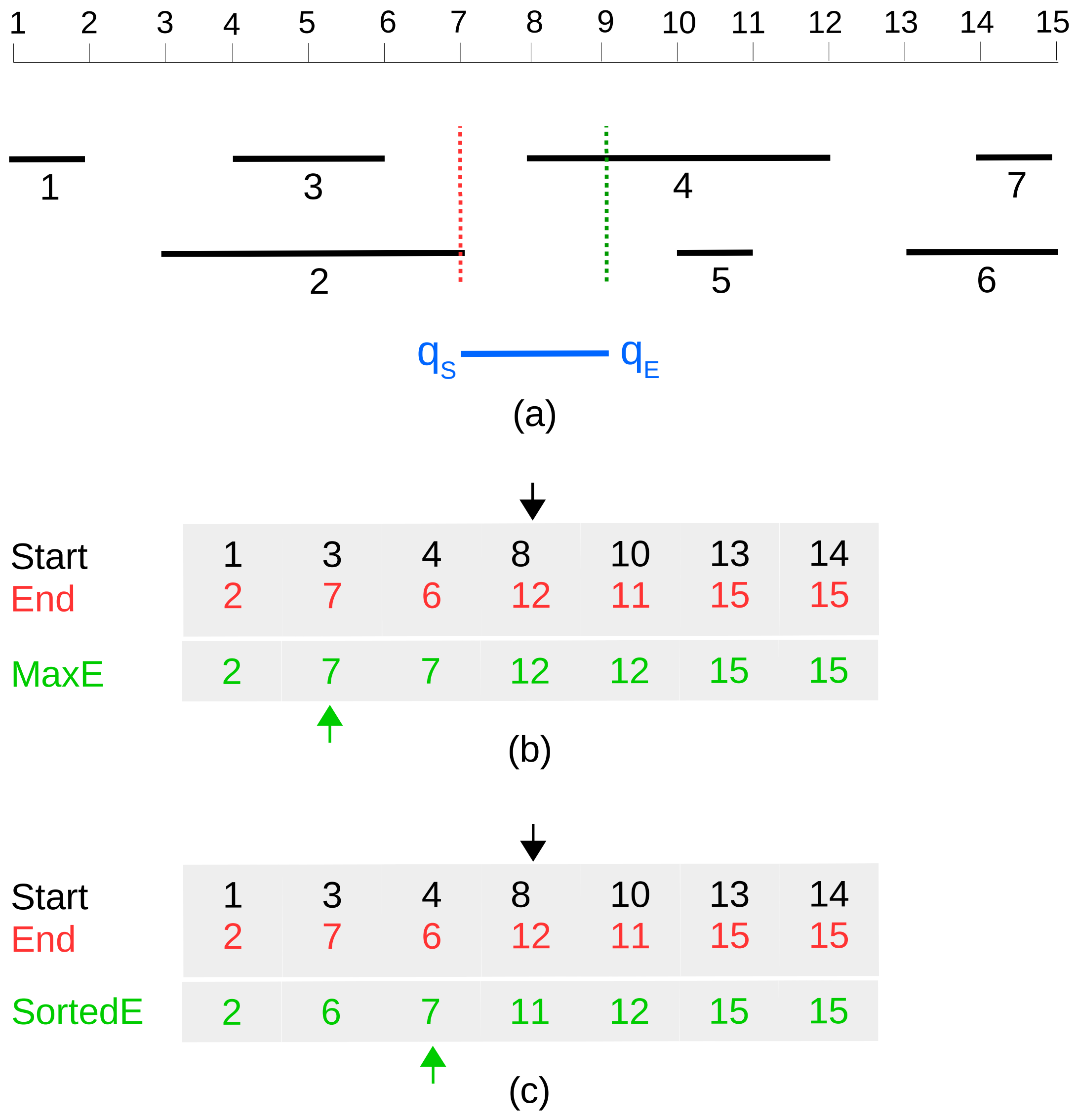

Figure 1. An interval set with a single query [7, 9] (**a**) and two ways of augmenting for effective search: maximum-end list (**b**) and sorted-end list (**c**). The interval list is sorted by the *start* in both (**b**) and (**c**).