Augmented Interval List: a novel data structure for efficient genomic interval search

Supplementary Information

Authors: Jianglin Feng, Aakrosh Ratan, and Nathan C. Sheffield

1. AIList augmented with `SortedE`

In this variant of the AIList data structure, the AIList is augmented directly with $SortedE$, the sorted list of interval ends, instead of the $MaxE$. So the basic data structure contains 3 elements: interval $start$, interval end and $SortedE$. Let aiL be a constructed AIList, $hSub$ be the head info containing the start of the sublists in aiL. We seek to identify overlaps of AIList with a given *query* $q = [start, end)$. The search algorithm is listed in Algorithm S1.

For each sublist, we first find the index of the last interval I_E that has $start < q.end$ with a binary search, which excludes all intervals on the right. Since $SortedE$ is sorted by the end, we can find the index of the leftmost element I_S that has $end > q.start$. This indicates that there are I_S-1 elements on the left of I_S should be excluded and the number of intersections is, thus, I_E - I_S +1 (Layer *et al.*, 2013). This algorithm does not directly find I_S, instead it simultaneously enumerates $SortedE$ and aiL from I_E to the left, so only one binary search is needed.

This search algorithm can slightly outperform the search algorithm of the `MaxE` version in cases such as wide queries ($q.end >> q.start$), but the sorting of the interval ends for `SortedE` is slower than finding the running maximum ends for `MaxE`. Generally `MaxE` version runs slightly faster than `SortedE` version, so the `MaxE` version is preferred.

Algorithm S1 AIList Search Algorithm

Input: AIList aiL, sublist header $hSub$, query $[start, end)$
Output: Overlaps H

procedure AILISTSEARCH($aiL, hSub, start, end$)
  $H \leftarrow \emptyset$
  for $i \leftarrow 1$ *to* $|hSub| - 1$ **do**
    $k \leftarrow BinarySearch(aiL, hSub[i], hSub[i+1] - 1, end)$
    $t \leftarrow 0$
    while $k > hSub[i]$ *and* $aiL[k].SortedE > start$ **do**
      $t \leftarrow t + 1$
      if $aiL[k].end > start$ **then**
        $H \leftarrow H \cup aiL[k]$
        $t \leftarrow t - 1$
      end if
      $k \leftarrow k - 1$
    end while
    while $t > 0$ **do**
      if $aiL[k].end > start$ **then**
        $H \leftarrow H \cup aiL[k]$
        $t \leftarrow t - 1$
      end if
      $k \leftarrow k - 1$
    end while
  end for
  return H
end procedure

2. Relation of AIList with AITree and NCList

The AIList can be considered as a combination of AITree and NCList. It is helpful to get a detailed runtime break-down for their construction and search algorithms in order to understand their differences.

For the data structur construction algorithm, AITree involves a tree-balancing operation, so it is slower than AIList and NCList on data structure construction, see Table 1. For the flat dataset (Dataset 1) AIList takes slightly longer than NCList because of the coverage length len computing during decomposition, but in all other cases, AIList construction is more efficient than NCList.

FOr the query algorithm, AITree is faster than NCList for highly contained datasets but slower for simple datasets; and in all cases, AIList is more efficient than both AITree and NCList, see Table 2. AIList is more efficient than AITree because AIList has fewer extra comparison m due to the decomposition. Although both AIList and NCList use sublists, the numbers of their sublists are very different, see Table 3. AIList maintains a very small number of sublists ($<$10 in all datasets we have tested), while NCList has 19 million linked sublists for a dataset of 128 million intervals. Since each sublist access involves a binary search, the query is slowed down. Thus, the key advantage of AIList over NCList is that it does not require *complete* decomposition, because it works relatively efficiently even with some containment; in contrast, the NCList data structure requires recursive sublist containment construction because it relies on a non-overlapping *guarantee* in the query algorithm.

Runtime(s)	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Dataset 6
AIList	0.086	0.12	0.681	2.352	14.560	34.418
AITree	0.117	0.25	1.187	5.533	42.672	111.536
NCList	0.075	0.157	0.883	3.391	22.983	57.657

Table 1. Construction time for AIList, AITree and NCList for datasets listed in Table 1 of the main text.

Runtime(s)	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Dataset 6
AIList	0.427	0.465	6.310	17.530	66.256	107.732
AITree	0.666	0.705	22.671	68.706	303.796	482.278
NCList	0.590	0.642	25.381	100.360	401.934	611.847

Table 2. Query time of AIList, AITree and NCList for datasets listed in Table 1 of the main text.

Runtime(s)	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5	Dataset 6	Dataset 0
AIList	1	2	6	8	7	7	6
NCList	1	1,006	364,025	2,500,263	8,898,006	19,294,893	203,576

Table 3. Number of total sublist of AIList and NCList for the seven datasets listed in Table 1 of the main text.

3. Data sources

The test datasets “exons” and “fBrain-DS14718” were downloaded from the BEDTools website `http://quinlanlab.org/tutorials/bedtools/bedtools.html#what-are-these-files`. The other datasets used in comparisons were downloaded from the UCSC website `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/database`. For simplicity, AITree, NCList and AIList ignore unplaced contigs, chrM and alternate haplotypes. We stripped off intervals from such contigs prior to use in comparisons. 1-3% regions of these datasets were stripped off, and these datasets (in BED format) are available at code.databio.org/AIList.

4. AIList source code

Source code for AIList is available at `http://github.com/databio/AIList`. We also provide the source code for AITree, and NCList which was downloaded from websites `https://github.com/biocore-ntnu/kerneltree/tree/master/src` and `https://github.com/hunt-genes/ncls/tree/master/ncls/src` respectively, along with methods to use them in comparative analysis to AIList.

References

Layer,R.M. *et al.* (2013) Binary interval search: A scalable algorithm for counting interval intersections. *Bioinformatics*, **29**, 1–7.