To nest or not to nest, that is the question. An argument that NCList is unnecessary.

Just for discussion; based on *Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases*, Alexander V. Alekseyenko and Christopher J. Lee, Vol. 23 no. 11 **2007**, pages 1386–1393. doi:10.1093/bioinformatics/btl647

These authors present a nested approach to finding overlapping intervals within a potentially very large set of intervals. They argue that in a situation such as given in their Fig. 2 (Figure 1), it is better to nest interval *y* into *x* than to leave it in the initially searched list.

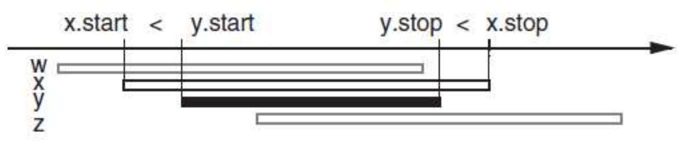

Fig. 2. Any contained interval breaks sortedness. If intervals (boxes) are sorted on their start coordinate, then any interval *y* (filled box) that breaks sortedness of the stop coordinate is properly contained in another interval *x*.

Figure 1. The original issue leading to the development of NCList.

The authors state the problem in this way (bold mine):

> *Interval query can be slow because the overlapping intervals for any given query may not be contiguous in standard indexing.* ***Therefore, the database query cannot stop at the first non-overlapping interval, but must scan the rest of the database.***

And their solution:

> *We can easily solve this problem by realizing that it is caused solely by the intervals that are contained within other intervals, i.e. x. start < y. start y.stop < x.stop. (see Fig. 2). If a sorted list of intervals has both start and stop coordinates in ascending order, then the overlapping intervals for any query are guaranteed to be contiguous in the list.*

The solution is clever. By partitioning the set into sets of fully contained intervals, NCList's findOverlap(start,end) method involves *n* binary searches that are of sets potentially smaller than *N*, each followed by two linear contiguous scans. For example, using NCList, if our search interval *I* is actually *y* itself, then a binary sort for the nested list will find *x*. A second binary search through the subintervals of *x* finds *y*. After each binary search, a scan contiguously both forward and backward finds all overlapping intervals of *I* (my Figure 2):

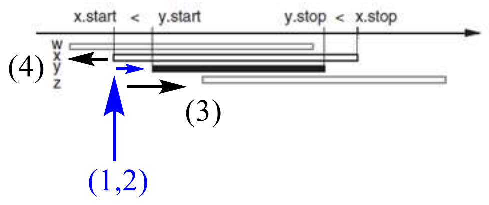

Figure 2: NCList looking for *y* finds *x* first, then *y,* using a series of two binary searches (1,2). After each binary search, it then scans contiguously forward (3) and backward (4) to find all overlapping intervals. (The two scans after the second binary search are not shown here.)

Jalview's NCList implementation – IntervalStore

The implementation of NCList in Jalview, IntervalStore<SequenceFeature> is identical to the original specification, with the added efficiency that during initialization all intervals that have no subintervals are set aside in a separate list. While this adds complications when adding or removing intervals from the set, it is more efficient in terms of storage (just a simple ArrayList<SequenceFeature>). The Jalview implementation is further optimized by separating different *types* of features (plaf, variant, etc.) into separate FeatureStore objects, each with their own IntervalStore. This substantially reduces the number of levels of nesting as well as the number of intervals in any single binary search.

But do we have to nest?

There is another, considerably simpler, solution involving a linked list with similarly scalable performance. The problem addressed by NCList is that once the first binary search is over, we still don't know which intervals that start ahead of our found interval overlap with it. But this is just a problem of initialization. If we allow for one object to point to another, all we have to do is to add a single "starting point contained by" pointer to the intervals such that

z -> y -> x -> w

where "*z -> y*" indicates that the starting point of *z* is contained by interval *y.* Now one binary search for the closest interval *C* such that *C.start >= I.start* will give *y*. From here, we simply scan backward through our pointers and forward through our list until we are out of range. Both scans are *contiguous*.

We have saved (n – 1) binary searches and (n – 1) * 2 contiguous scans. The simple linked list avoids the necessity for all the nesting structure that comes with NCNodes and NCList, and also provides an opportunity for "lazy initialization" of the list.

The advantage of NCList is that it pre-partitions the binary search of N objects into a set of n binary searches of $m_i \le N$ objects, where SUM(i){m_i} = N. Depending upon the extent of nesting, this could be significant. With minimally nested sets, however, it is unlikely that this advantage would be noticeable.

The disadvantage of NCList is that it requires substantial initialization involving at least 3*N* objects (*N* NCNodes, each with a reference to its associated interval as well as a reference to its containing NCList). Thus, it is trivially shown that a single pointer added to each of *N* intervals requires far less initialization and storage than a set of *N* Node/List objects. The storage is a single object reference, and initialization is simply the same initial binary sort required for NCList, followed by a relatively short *contiguous* connection to the *first* overlapping interval in the backward direction only.

In addition, though, the "NoNCList" option allows for lazy initialization. We can do all the loading of the list, including minor addition/removal with the option to not sort the actual list or build the links until it is absolutely necessary (the first findOverlap() call, generally).