SCIENTIFIC REPORTS

OPEN

Efficient Genomic Interval Queries Using Augmented Range Trees

Chengsheng Mao[1], Alal Eran[2,3] & Yuan Luo[1]

Received: 7 June 2018
Accepted: 5 March 2019
Published online: 25 March 2019

Efficient large-scale annotation of genomic intervals is essential for personal genome interpretation in the realm of precision medicine. There are 13 possible relations between two intervals according to Allen's interval algebra. Conventional interval trees are routinely used to identify the genomic intervals satisfying a coarse relation with a query interval, but cannot support efficient query for more refined relations such as all Allen's relations. We design and implement a novel approach to address this unmet need. Through rewriting Allen's interval relations, we transform an interval query to a range query, then adapt and utilize the range trees for querying. We implement two types of range trees: a basic 2-dimensional range tree (2D-RT) and an augmented range tree with fractional cascading (RTFC) and compare them with the conventional interval tree (IT). Theoretical analysis shows that RTFC can achieve the best time complexity for interval queries regarding all Allen's relations among the three trees. We also perform comparative experiments on the efficiency of RTFC, 2D-RT and IT in querying noncoding element annotations in a large collection of personal genomes. Our experimental results show that 2D-RT is more efficient than IT for interval queries regarding most of Allen's relations, RTFC is even more efficient than 2D-RT. The results demonstrate that RTFC is an efficient data structure for querying large-scale datasets regarding Allen's relations between genomic intervals, such as those required by interpreting genome-wide variation in large populations.

Annotating functional elements in genomic datasets is fundamental for understanding genome biology, interpreting genomic variation, and advancing precision medicine. Genomic features, such as genes, exons, or regulatory regions, can be represented as genomic intervals, comprised of a chromosome ID with a start and an end position. Genomic intervals serve to anchor numerous diverse genomic datasets and their experimental results on a common basis, thereby facilitating their comparison and integration. With the advance of next-generation sequencing, multiple online resources including the UCSC genome browser[1] and the Encyclopedia of DNA Elements (ENCODE) project[2] provide billions of interval-based genomic annotations. Sifting through a large number of genomic intervals often involves identifying the set of intervals satisfying certain interval relations with query genomic intervals. This is a challenging task due to the extremely large number of genomic intervals present and due to the multiple different relations that can hold between genomic intervals. Most existing studies focus on the overlapping relations between genomic intervals[1,3–9]. However, more refined relations between genomic intervals can also be suggestive of relations between corresponding genomic annotations. For example, a putative promoter that regulates transcription of a particular gene is located in the vicinity of the transcription start sites of that gene, and more specifically on the same strand, and upstream of the gene. Developing efficient methods for identifying the set of genomic intervals satisfying the often complex relations with a genomic interval of interest is a major unmet need that is crucial for biomedical discoveries.

In many cases, we need to know all the intervals in a large set that satisfy a certain relation with a given interval. This is known as an interval query problem. Given a query interval q, a set of data intervals S, and one relation r, an interval query is to retrieve all the intervals $x \in S$, such that the relation $x\ r\ q$ holds. In interval query, the most common and widely studied problem is the intersection query problem where the relation r refers to the intersection relation. Usually, two intervals a and b intersect if and only if $a.\ start \leq b.\ end$ and $a.\ end \geq b.\ start$ where for example, $a.\ start$ denotes the start position of interval a. BEDTools[7] and BEDOPS[10] are two widely used tools used for interval queries with this type of intersection relation. However, in many cases, the intersection relation can be rather coarse, and the relative position of the overlapping genomic intervals may also be of interest. Figure 1

[1]Department of Preventive Medicine, Feinberg School of Medicine, Northwestern University, Chicago, IL, USA. [2]Department of Biomedical Informatics, Harvard Medical School, Boston, MA, USA. [3]Department of Life Sciences, Ben Gurion University of the Negev, Beersheba, Israel. Correspondence and requests for materials should be addressed to Y.L. (email: yuan.luo@northwestern.edu)

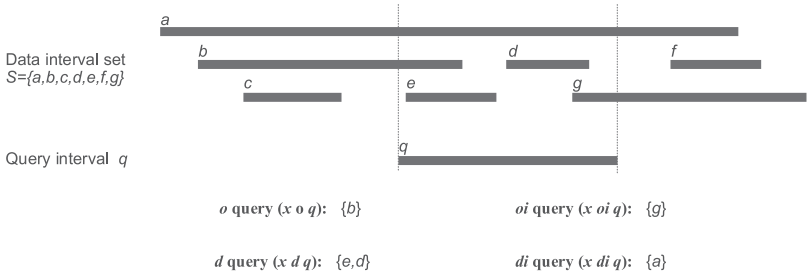


Figure 1. An example of interval queries regarding four different intersection relations, i.e., overlapping from the front (*o*), overlapping from behind (*oi*), contains (*di*) and contained in (*d*). The four types of interval queries are based on the query interval q and the data interval set S.

illustrates such an example of interval queries regarding four different types of intersections including: overlapping from the front (*o*), overlapping from behind (*oi*), contains (*di*), and contained in (*d*).

Figure 1 is a simple interval query example with a small data interval set where the result set can be obtained by comparing each interval in the set with the query interval one by one. However, in practical applications, the brute-force enumeration scales poorly for large datasets in terms of efficiency. Developing efficient methods for interval queries regarding more refined relations is also crucial, e.g., the putative promoter example.

Previous studies on genomic interval query made use of interval trees[11], binning approaches (based on R-trees)[1,6,7], nested containment lists[4,8] or linear sweeps[3,9,10], but mostly focused on the coarse intersection queries. BEDTools[7] and BEDOPS[10] are two popular tools for the coarse interval intersection query. However, even the latest versions of both tools do not support the more refined interval relations in Allen's interval algebra. Seok *et al.*[5] reviewed a number of interval query algorithms from the above categories and analyzed their time complexities on intersection queries: most of them cannot achieve $O(logn + k)$ time, where n is the size of interval set and k is the number of result intervals. Though interval tree algorithms can achieve $O(logn + k)$ time on intersection queries, however, theoretical analysis showed that conventional interval tree based algorithms have sub-optimal speed (time complexity larger than $O(logn + k)$ in general) for more refined interval relations in Allen's interval algebra[12]. In this paper, we propose query rewriting and adapt the range trees to design and implement an efficient interval query method that can achieve the optimal $O(logn + k)$ time complexity for interval queries regarding all Allen's interval relations.

Objectives

Efficient queries for genomic intervals that have a certain relation with a given interval are essential for various bioinformatic applications, especially for large genomic datasets. According to Allen's interval algebra, there are 13 possible relations between two intervals, 11 out of which are associated with the intersection, the other two are associated with non-intersection. An interval query regarding one of the two non-intersection relations can be recast as a certain interval query regarding one of Allen's intersection relations, so we only consider the interval queries regarding Allen's intersection relations. Though the coarse intersection query is widely studied, and the studies have made some achievements, interval queries regarding more refined relations are also of interest. Unfortunately, when applying existing interval query methods designed for the coarse intersection to a more refined relation in Allen's algebra, they need to find all the intervals satisfying the coarse intersection relations, and then search in the results for intervals satisfying the refined relation. Thus, the query would become less efficient even the result intervals become much less. Suppose in a total of n data intervals there are m intervals overlapping with the query interval, and in the m intervals there are k intervals satisfying the o relationship ($m > k$). Usually, an efficient interval query algorithm like interval tree needs take time $O(logn + 2 \times m)$ to query the k result intervals for the o query, which is more than the time of interval query with the coarse relationship i.e., $O(logn + m)$, even though query results size become much less ($k < m$). Our objective is to improve the interval query efficiency regarding refined relations in Allen's algebra using query rewriting and the range tree data structure. Our method can achieve the optimal $O(logn + k)$ time complexity for interval queries regarding all Allen's interval relations.

Materials and Methods

We applied Allen's interval algebra to refine the relations between two genomic intervals into 13 categories. To efficiently retrieve all the genomic intervals satisfying a certain Allen's relation with a given genomic interval from a large dataset, we regarded an interval as a 2-dimensional point and transformed the interval query problem to the range query problem by rewriting the definition of Allen's interval relations. We then applied the range tree data structure and the corresponding query algorithm to perform efficient range queries. Besides the basic 2-Dimensional Range Tree (2D-RT), we also implemented an augmented Range Tree structure with the technique of Fractional Cascading (RTFC). The current state-of-the-art interval tree (IT) algorithm was also implemented as a baseline. We tested interval query efficiency of the above algorithms with ENCODE[2] genomic annotation intervals as the data interval dataset, and Genome Aggregation Database (gnomAD)[13] variant intervals as the query set. We have made our code publicly available at https://github.com/mocherson/range-tree.

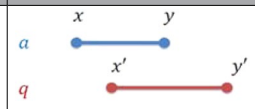
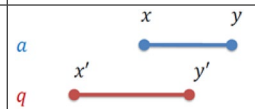
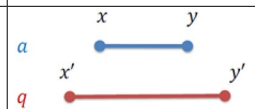
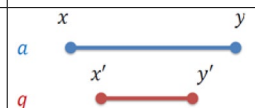
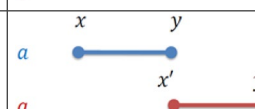
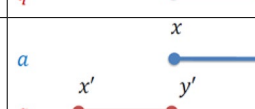
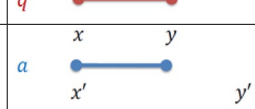
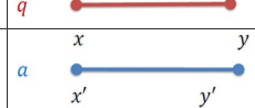
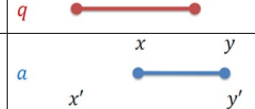
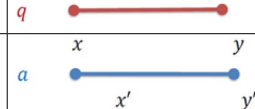
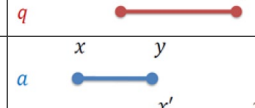
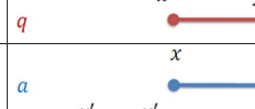
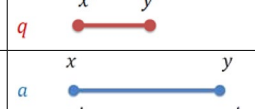
Symbols	Relation	Illustration	Definition	Rewriting as range query
o	$a\ o\ q$		$x<x'<y<y'$	$0<x<x'$ $x'<y<y'$
oi	$a\ oi\ q$		$x'<x<y'<y$	$x'<x<y'$ $y'<y<\infty$
d	$a\ d\ q$		$x'<x<y<y'$	$x'<x<y'$ $x'<y<y'$
di	$a\ di\ q$		$x<x'<y'<y$	$0<x<x'$ $y'<y<\infty$
m	$a\ m\ q$		$x<y=x'<y'$	$0<x<x'$ $y=x'$
mi	$a\ mi\ q$		$x'<y'=x<y$	$x=y'$ $y'<y<\infty$
s	$a\ s\ q$		$x'=x<y<y'$	$x=x'$ $x'<y<y'$
si	$a\ si\ q$		$x=x'<y'<y$	$x=x'$ $y'<y<\infty$
f	$a\ f\ q$		$x'<x<y=y'$	$x'<x<y'$ $y=y'$
fi	$a\ fi\ q$		$x<x'<y'=y$	$0<x<x'$ $y=y'$
$<$	$a<q$		$x<y<x'<y'$	$0<x<x'$ $0<y<x'$
$>$	$a>q$		$x'<y'<x<y$	$y'<x<\infty$ $y'<y<\infty$
$=$	$a=q$		$x=x'<y'=y$	$x=x'$ $y=y'$

Table 1. Allen's interval relations and their transformation to the bound range for range query. Note: $\boldsymbol{a}=[\boldsymbol{x}, \boldsymbol{y}]$ is an interval from the data interval set, $\boldsymbol{q}=[\boldsymbol{x}', \boldsymbol{y}']$ is the query interval.

Allen's Interval Algebra. In 1-dimensional cases, an interval is usually defined by two numbers corresponding to the start and the end, where the end is supposed to be greater than the start. In this paper, we use $[x, y]$ to denote an interval with start x and end y. Based on the three relations between two numbers, i.e., greater, equal and less, Allen[14] proposed 13 relations between two temporal intervals that are distinctive, exhaustive and qualitative. Distinctive and exhaustive because each pair of definite intervals must be described by one and only one of the relations; qualitative because no numeric spans are considered. The relations between intervals and the

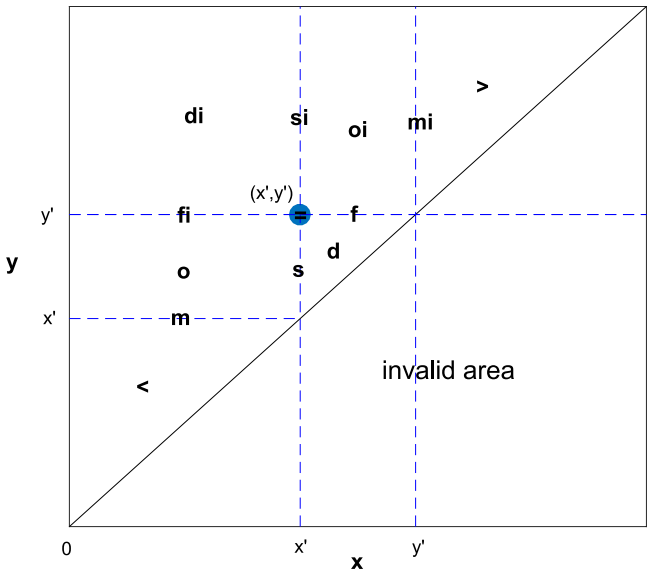


Figure 2. If an interval is associated with a 2-dimensional point, an interval query can be transformed to a range query. The query interval $[x', y']$ is associated with the point (x', y'), and the interval query regarding each of Allen's interval relations corresponds to the range query regarding the indicated area (marked in the figure by the corresponding relation symbol, the "=" query exactly corresponds to the point (x', y')). The start is less than or equal to the end for an interval, thus, points in the lower right area under the line $y = x$ is invalid to associate an interval with.

operations based on them form Allen's interval algebra. Though Allen's interval algebra was originally proposed for temporal intervals, it applies to spatial intervals such as genomic intervals. If we consider two intervals, $a = [x, y]$ and $q = [x', y']$, the 13 relations between them can be defined and illustrated in Table 1. From Table 1, Allen's interval algebra provides a more refined interval relation category method, based on which the intersection relation consists of 11 Allen's interval relations (i.e., *o, oi, d, di, s, si, f, fi, m, mi* and =).

For an interval query regarding relation '<' or '>', there usually are a very large number of result intervals. In practice, only a subset of these result intervals are of interest, and the query can be regarded as a *d* query. For example, to find a putative promoter in the upstream of the gene with interval $[x', y']$ requires a '<' query. Since a promoter that regulates transcription of the gene is located in the vicinity (e.g., *l* bases) of the transcription start sites of that gene (i.e., x'), we only need to perform the *d* query regarding $[x' - l, x']$. Analogously, a '>' query also usually takes the form of a *d* query regarding $[y', y' + l]$ in practice. Thus we only consider the 11 intersection related queries. For another example on practical usage of Allen's interval algebra, it is an active area of research to build genome wide interval-based score databases for function prediction score and evolutional constraint score for different cell lines. When one does a whole-genome sequencing, he may get multiple millions of SNPs and indels, as well as thousands of structural variants. If he wants to scan all of them against multi-cell-line score databases, then there are significant computational challenges. Even the ability to speed up indel and structural variant queries will be quite useful (i.e. the *di* query).

Rewriting Interval query to range query. An interval can be mapped to a 2-dimensional point with the 2 endpoints of the interval as the 2 coordinates of the point, i.e., interval $[x, y]$ corresponds to the point (x, y). Thus, a 2-dimensional range tree can be adapted for interval query by transforming interval relations to relations between 2-dimensional points. Through rewriting the definition of each Allen's interval relation as shown in the last column of Table 1, a satisfying interval can be mapped to a point satisfying a certain range constraint, and thus an interval query problem is transformed to a range query problem, as shown in Figure 2. Figure 2 illustrates the satisfying regions of range queries that are transformed from interval queries with respect to a certain query interval. From Figure 2, it is natural to apply a range tree for interval queries after query rewriting. Note that an interval should have its start less than its end, thus a point associated with an interval must be above the line $y = x$. This is the reason why the lower right area in Figure 2 is invalid for interval query.

Basic range tree. Range trees are originally designed for efficient range queries[15,16]. In range query problems, range trees are usually applied to query the set of points that lie in a given range, especially in a rectangular area. A *d*-dimensional range tree RT^d on a set of *d*-dimensional points is actually an augmented balanced Binary Search Tree (BST) with the following recursive structure. Each node *v* uses the first of the *d* coordinates as its key and contains an associated *(d-1)*-dimensional range tree RT_v^{d-1} on the rest *d*-1 coordinates of the points stored in the subtree rooted at *v*. Each node *u* of RT_v^{d-1} uses the second of the *d* coordinates as its key and contains an associated *(d-2)*-dimensional range tree RT_u^{d-2} on the rest *d-2* coordinates of the points stored in the subtree rooted at *u*. The recursive structure continues analogously as we go through each of the *d* coordinates. Eventually, a

1-dimensional range is exactly a traditional balanced BST on the last coordinate. Generally, the points stored in a range tree are stored in the leaves, and each internal node stores the largest value contained in its left child.

In 1-dimensional cases, a range query is to list all the points that lie in a certain interval, denoted as $[x1, x2]$, from a set of given points. One can use a 1-dimensional range tree to efficiently perform the query by searching for the two endpoints $x1$ and $x2$ respectively and reporting all the points between them. Since the range tree is balanced, the search for $x1$ and $x2$ takes $O(logn)$ time, where n is the number of data points. Reporting all the points between $x1$ and $x2$ needs to traverse all the subtrees between their search paths, and it can be done in linear time. Thus, a range query on the basic 1-dimensional range tree has the time complexity $O(logn + k)$, where k is the number of result points.

Range queries in d-dimensional cases are similar. The main difference is that for d-dimensional trees we need to traverse the recursive tree structure as defined above. For example, using the two endpoints of the first coordinate, we identify a set of subtrees S between their search paths (excluding the nodes in the search paths). For the root v of each subtree in S, we perform a *(d-1)*-dimensional range query on RT_v^{d-1}. To perform a *(d-1)*-dimensional range query on RT_v^{d-1}, we identify a set of subtrees S_v using the two endpoints of the second coordinate analogously, then perform a *(d-2)*-dimensional range query on RT_u^{d-2} for the root u of each subtree in S_v. We continue with further lower dimensional queries in a recursive manner. Eventually, a series of 1-dimensional range queries will be performed, and the correct points will be reported. Since a d-dimensional range query consists of $O(logn)$ *(d-1)*-dimensional range queries[17,18], by recursive time complexity analysis, the time required to perform a d-dimensional range query is $O(\log^d n + k)$.

Fractional cascading. Fractional cascading (FC) is a technique to speed up the searching for the same value from multiple sequences[19,20]. With FC, searching for the same value from a number of sets would need only one search for that value from the union of these sets. For example, if we have two arrays of numbers, $A1$and $A2$ (let $A2 \subseteq A1$), both sorted ascendingly, then FC takes the following steps to search for the minimum values no less than v in $A1$ and $A2$ respectively. This process is also shown in Figure 3.

1. Create an indexing array I from $A1$ to $A2$. The ith element in $A1$ (i.e., $A1[i]$) corresponds to the ith element in I (i.e., $I[i]$) which is the index of the smallest element in $A2$ no less than $A1[i]$ (-1 if no such elements). We refer to such an index as FC-index.
2. Perform a binary search on $A1$ for v and return the smallest element no less than v in $A1$, say $A1[j]$, and its position j.
3. Directly obtain the smallest element no less than v in $A2$, i.e., $A2[I[j]]$. $A2[-1]$ indicates all elements in $A2$ are less than v.

Through FC, searching for the same value from multiple sequences can be achieved by only one binary search from the union set, and the search results of each subset can be directly obtained; this is more efficient than multiple searches from all the sets.

Range tree with fractional cascading. By the fractional cascading technique, we can improve the query time from $O(log^d n + k)$ to $O(log^{d-1} n + k)$ for range tree[17,21], and thus a 2-dimensional range query can be done in $O(\log n + k)$ time. Let x and y indicate the coordinates of the 2 dimensions, we construct a 2-dimensional range tree with fractional cascading (RTFC) using the following steps. The x-tree is first constructed on the x coordinates of all the points as the basic range tree. Instead of building a y-tree RT_v^1 in each node v, RTFC stores the corresponding points as an array $A(v)$, sorted by y-coordinate. In addition, each node v stores two FC-index arrays, containing the FC-indices from $A(v)$ to $A(l(v))$ and $A(r(v))$, where $l(v)$ and $r(v)$ are the left and right children of node v respectively. Since $A(v) = A(l(v)) \cup A(r(v))$, by the fractional cascading technique described in Section 3.4, once we obtain the range query results in v, the subsequent query results in its children $l(v)$ and $r(v)$ can directly be obtained through their corresponding FC-index arrays. The query results in its grandchildren are readily obtained, recursively for all descendants until we reach the leaf nodes. Since range queries on RTFC avoid the 1-dimensional range query on y-trees, it has the time complexity $O(logn + k)$ in 2-dimensional cases and $O(log^{d-1} n + k)$ in d-dimensional cases.

Data structure implementation. Besides the RTFC, we also implement the traditional 2-dimensional range tree (2D-RT) and the interval tree (IT) as baselines. All the three tree structures are implemented in C++. Unlike IT in which each interval is referenced only once, in RTFC and 2D-RT, each interval is referenced multiple times. To reduce memory consumption, we use a static vector to store the points (i.e., rewritten intervals) and refer to the indices of the point in RTFC and 2D-RT. We design and implement the tree structures for RTFC, 2D-RT and IT as shown in Figure 4.

The tree building algorithm for RTFC can be found in Table 2. The range tree is constructed by splitting the original interval set P into P_{left} and P_{right} corresponding to the two child by the x-values (Line 5), and each child is built recursively (Lines 9–10). A simple example to illustrate the construction of RTFC is shown in Figure 5. Figure 5(a) shows the original interval set and the corresponding indices sorted by x-value; Figure 5(b) represents the resulting range tree structure with fractional cascading. In Figure 5(b), in each node of RTFC, the field "the index of the key point" represents the index of the median x-value in the original interval set, and the field "indices sorted by y-value" links to the original interval set, indicating the indices of all the data interval in this node.

The interval query algorithm by RTFC is shown in Table 3. We first transform the interval query to range query (Line 1). Figure 5(c) shows an example to transform an interval query with d relation and interval [2, 10] to

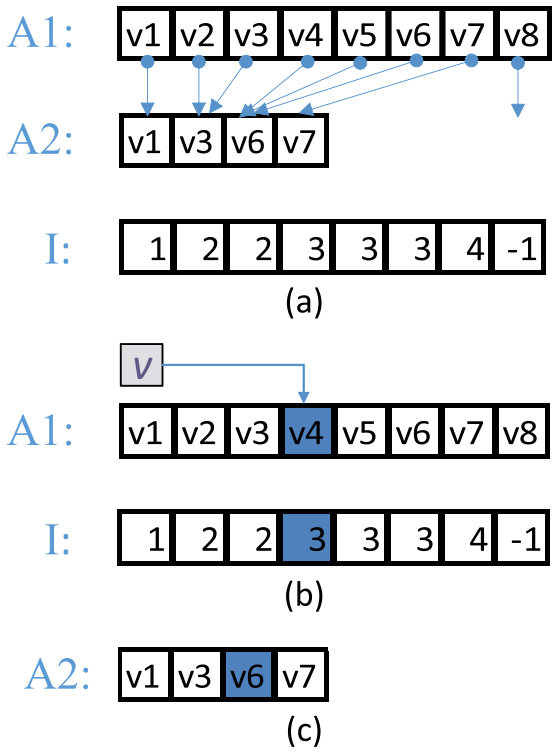


Figure 3. A simple example of search by fractional cascading. Given a value v, the query searches for the minimum values no less than v in A1 and A2 respectively. (**a**) Create the indexing array I from A1 to A2. Refer to such an index as FC-index. (**b**) Suppose $v \in (v3, v4]$, the binary search for v from A1 will return $v4$ and its position 4, then the corresponding FC-index in A2 is $I[4] = 3$. (**c**) Directly return $A2[I[4]] = v6$ as the search result from A2.

a range query. To eliminate the intervals that start with 2 or end with 10, we modify the query interval to [2.5, 9.5], since all the gnomic intervals are all based on integers. Figure 5(b) shows the query processes for the corresponding range query $x \in [2.5, 9.5]$, $y \in [2.5, 9.5]$ on the constructed RTFC. The search paths for $x_1 = 2.5$ and $x_2 = 9.5$ are $4 \rightarrow 2 \rightarrow 1 \rightarrow 2$ and $4 \rightarrow 6 \rightarrow 7 \rightarrow 8$, respectively (marked by red arrow line in Figure 5(b)), thus the split node is 4 (Line 3). All the intervals in the nodes between the two search paths can meet $x \in [2.5, 9.5]$. Since the split node is not a leaf, search for $y_1 = 2.5$ in the data of the split node, the returned index $i = 1$ (Line 9), bounded by a red box in the split node in Figure 5(b). For the path from the split node to x_1, excluding the split node and the leaf $(2 \rightarrow 1)$, cascade the index of $y_1 = 2.5$ to each node in the path (node 2 and 1) and the nodes (node 3) that are not in the path but are the right child of any nodes in the path (Lines 10, 18 and 20), the result in each of these nodes is bounded by a red box in Figure 5(b). For each of the nodes that are not in the path but are the right child of any nodes in the path (node 3), report the intervals in this node from the cascaded index to the end of the data array until the y-value of the interval is greater than y_2 (Lines 14–17), i.e., report intervals 3 and 4 in this example. For the leaf node in the path, report the interval if it is in the ranges (Lines 23–25), i.e., report interval 2 in this example. Similarly, for the path from the split node to x_2, report the intervals in the ranges in the nodes left of the path (Lines 26–41), this will report interval 6 in this example. Thus, the query results consist of 4 intervals (i.e., intervals 3, 4, 2, and 6, shaded by green in Figure 5(b)).

Dataset. As we move closer to practicing precision medicine, one of the main challenges remains the interpretation of noncoding genomic variants[22,23]. Since the vast majority of the human genome is noncoding, the vast majority of human variation is noncoding. To assess the functionality of noncoding regions and enable personal noncoding variant interpretation, the ENCODE project has systematically identified functional genomic intervals in the human genome at scale[2]. These include transcription factor binding sites, chromatin structures, and histone modification sites. Here we demonstrate the ability of RTFC to rapidly annotate noncoding variants in these ENCODE regions, thereby facilitating timely personal genome interpretation.

For each of the 10,791 bed files representing high-quality ENCODE ChIP-seq data (Supplementary Dataset), we extract the "chromStart" and "chromEnd" fields (start and end positions of a region in a chromosome) to construct intervals. We then categorize all the intervals from all the bed files by the chromosome (i.e., the "chrom" field). For each chromosome, all its intervals are regarded as a data interval set to be queried (Supplementary S2). We focus our analysis on annotating two types of genomic variants, of varying lengths: single nucleotide variants (SNVs) and insertions/deletions (indels), both detected in the 123,136 exome sequences and the 15,496 whole-genome sequences of the gnomAD[13]. The sizes of these population-level variant datasets are summarized in Table 4, and described in details per chromosome in Supplementary S2.

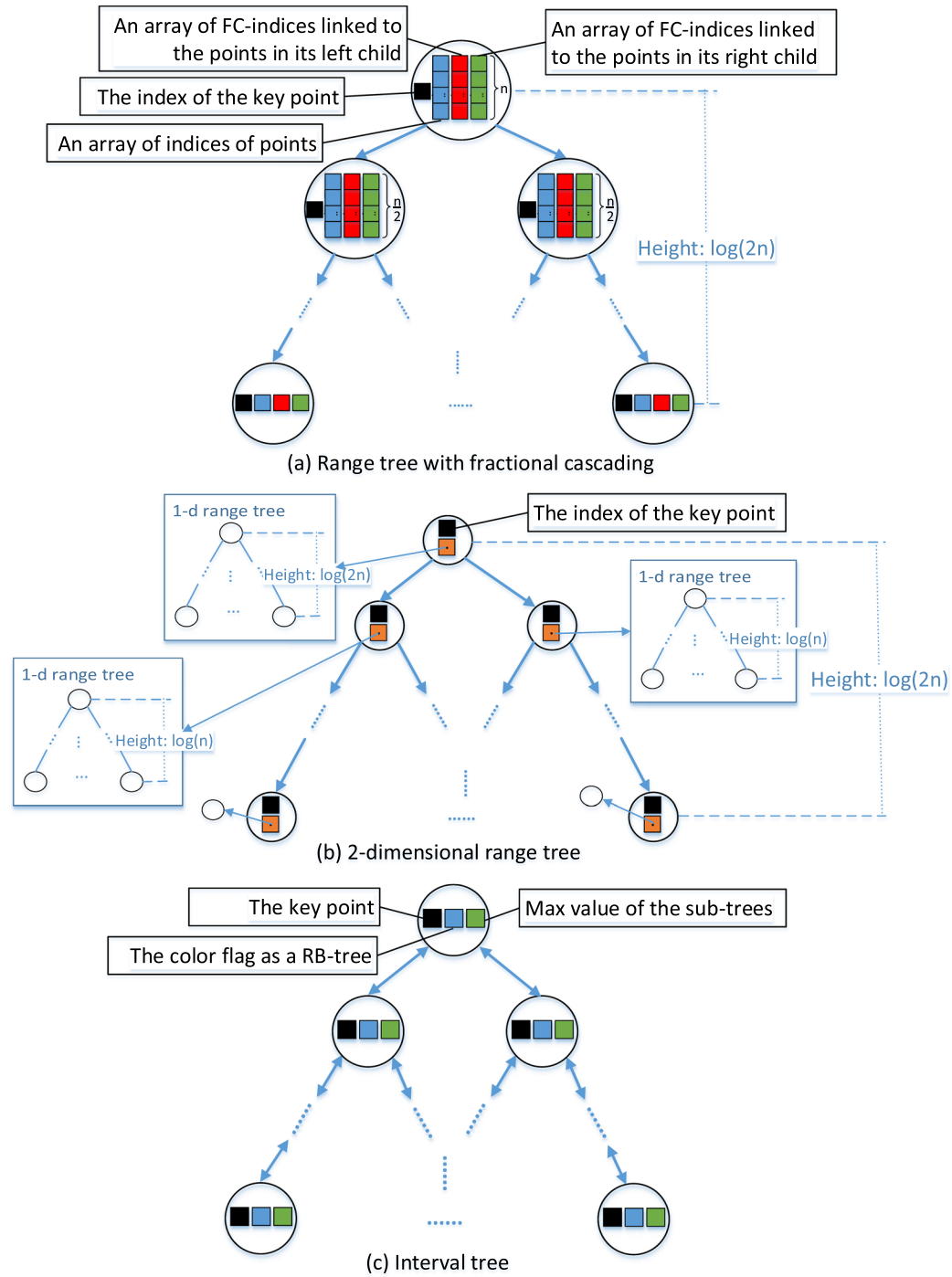


Figure 4. The three tree structures for a data interval set with n intervals. For brevity of illustration, the pointer fields of each node pointing to the left, right or parent node are omitted inside the node, and represented by the corresponding arrows between nodes. (**a**) The data structure of range tree with fractional cascading (RTFC). (**b**) The data structure of the basic 2-dimensional range tree (2D-RT). (**c**) The data structure of interval tree (IT) implemented as an augmented RB-tree (red-black tree) with n nodes.

Experiments and Results

For each of the 23 chromosomes (22 autosomes and X chromosome), we first constructed the three tree structures based on the ENCODE interval sets. Then we performed the interval query with respect to each of Allen's relations using each interval from the gnomAD datasets as query interval. The running time was recorded for each step in order to evaluate the efficiency of the three tree structures.

We followed the range tree building algorithm BUILD2DRANGETREE to build 2D-RT[17], and RTFC was built following the algorithm *BuildRTFC* in Table 2. IT was built by iteratively inserting a node corresponding to a point into an initially empty tree. Since IT is an augmented RB-tree (Red-Black tree), IT insertion algorithm is

Algorithm *BuildRTFC*(P)	
1.	Sort P by y-value, return an array of intervals P_y.
2.	**if** P contains only one interval i **then**
3.	Creating a leaf node v_{leaf} storing this interval. i.e., $v_{leaf}.interval = i$
4.	**else**
5.	Split P into P_{left} and P_{right}, the subsets $\leq$ and $>$ the median x-value x_{mid} of P.
6.	Sort P_{left} and P_{right} by y-value.
7.	Create an FC-index I_{left} from P_y to P_{left}.
8.	Create an FC-index I_{right} from P_y to P_{right}
9.	$v_{left} \leftarrow BuildRTFC(P_{left})$
10.	$v_{right} \leftarrow BuildRTFC(P_{right})$
11.	Create a node v storing x_{mid}, I_{left} and I_{right}. $v.x = x_{mid}$; $v.data = P_y$; $v.lfc = I_{left}$; $v.rfc = I_{right}$; $v.lchild = v_{left}$; $v.rchild = v_{right}$
12.	**end if**
13.	**return** v

Table 2. The algorithm to build range tree with fractional cascading. Input $\boldsymbol{P}$ is the original interval set. Return the root node $\boldsymbol{v}$ of the resulting range tree.

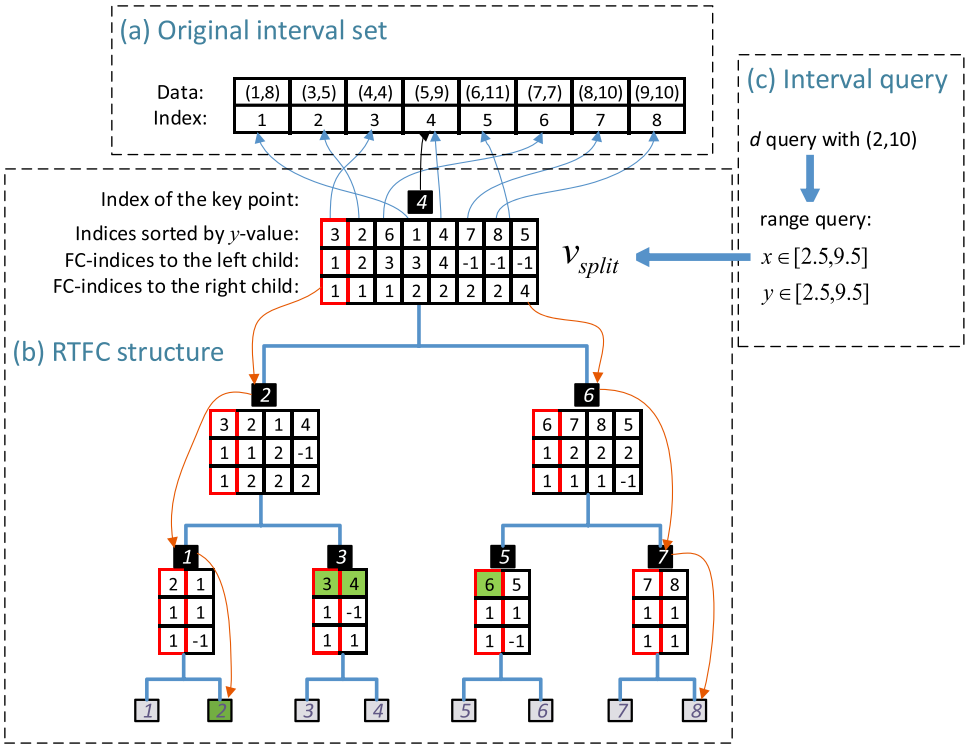


Figure 5. An example to illustrate the construction of RTFC and the query using RTFC. (**a**) The original interval set and the corresponding indices. (**b**) The resulting tree structure and the query processes. The red arrow lines represent the search path of $x_1 = 2.5$ and $x_2 = 9.5$. The item bounded by a red box in a node represents the searched index in the data array for $y_1 = 2.5$. The green shaded intervals represent the result intervals. (**c**) Range query transformed from interval query.

a well-defined modification of the RB-INSERT algorithm outlined by Cormen *et al.*[24]. In our experiments, the building times for RTFC, 2D-RT and IT are summarized in Table 5, and shown in detail in Supplementary S3. From Table 5, IT took the shortest time and 2D-RT took the longest time to build the corresponding tree structures. According to Figure 4, IT has the least complex structure among the three trees. For RTFC, each node needs to additionally maintain one index array and two FC-index arrays. For a 2D-RT, each node additionally maintains a 1-dimensional range tree. Since a range tree has a more complex structure than an array, constructing a 2D-RT takes more time than constructing an RTFC. Consistently, in our experiments, we saw that IT < RTFC < 2D-RT in building time. On the other hand, we note that once the tree structure was built, it could be used for interval query regarding any relations for any given query intervals. Thus, for all queries on one chromosome, building

Algorithm *QueryRTFC*(T, I, R)	
1.	Transform interval query with respect to interval I and relationship R to range query with x range $[x_1, x_2]$ and y range $[y_1, y_2]$ according to Table 1.
2.	Initialize the result set $S = \{\}$
3.	Find the split node v_{split} in range tree T where the paths to x_1 and x_2 split, or the leaf where both paths end.
4.	**if** v_{split} is a leaf node **then**
5.	**if** $v_{split}.interval.x \in [x_1, x_2]$ and $v_{split}.interval.y \in [y_1, y_2]$ **then**
6.	Report the interval in v_{split}, $S = S \cup \{v_{split}.interval\}$
7.	**end if**
8.	**return** S
9.	Perform binary search on $v_{split}.data$ for y_1 by y-value, find the index i of the smallest element no less than y_1 in $v_{split}.data$.
10.	$v_{split} = v_{split}.lchild$, $i = v_{split}.lfc[i]$
11.	**while** v is not a leaf **and** $i \neq 1$ **do**
12.	**if** $x_1 \leq v.x$ **then**
13.	$j = v.rfc[i]$
14.	**while** $j \neq -1$ **and** $v\ rchild.data[j].y \leq y_2$ **and** $j \leq v.rchild.data.size$ **do**
15.	Report interval, $S = S \cup \{v.rchild.data[j]\}$
16.	$j = j + 1$
17.	**end while**
18.	$i = v.lfc[i]$, $v = v.lchild$
19.	**else**
20.	$i = v.rfc[i]$, $v = v.rchild$
21.	**end if**
22.	**end while**
23.	**if** v is a leaf **and** $v.interval.x \in [x_1, x_2]$ **and** $v.interval.y \in [y_1, y_2]$ **then**
24.	Report the interval in v, $S = S \cup \{v.interval\}$
25.	**end if**
26..	$v = v_{split}.rchild$, $i = v_{split}.rcf[i]$
27.	**while** v is not a leaf **and** $i \neq -1$ **do**
28.	**if** $x_2 \geq v.x$ **then**
29.	$j = v.lfc[i]$
30.	**while** $j \neq -1$ **and** $v.lchild.data[j].y \leq y_2$ **and** $j \leq v.lchild.data.size$ **do**
31.	Report interval, $S = S \cup \{v.lchild.data[j]\}$
32.	$j = j + 1$
33.	**end while**
34.	$i = v.rfc[i]$, $v = v.rchild$
35.	**else**
36.	$i = v.lfc[i]$, $v = v.lchild$
37.	**end if**
38.	**end while**
39.	**if** v is a leaf **and** $v.interval.x \in [x_1, x_2]$ **and** $v.interval.y \in [y_1, y_2]$ **then**
40.	Report the interval in v, $S = S \cup \{v.interval\}$
41.	**end if**
42.	**return** S

Table 3. The algorithm to execute interval query using range tree with fractional cascading. Input ***T*** is the range tree; ***I*** is the query interval; ***R*** is the relationship. Return the result interval set ***S*** corresponding to all the intervals in range tree ***T*** satisfying the relationship ***R*** with interval ***I***.

the tree data structure was just one-time up-front effort and we focused our time complexity analysis on repeated query processes.

Since '<' and '>' queries can be transformed to d queries as explained in Section 3.1, we only considered the queries regarding the 11 intersection relations in Allen's algebra in our experiments. The time complexity evaluations of these query types on all the 23 chromosomes are summarized in Table 6. From Table 6, RTFC and 2D-RT are more efficient than IT for most of the interval query types, and RTFC consistently consumes less time than 2D-RT for the 11 intersection queries. For *s, si, mi* and = queries, since the result intervals must have a fixed start, these queries mainly perform binary searches to find the tree nodes corresponding to the start position (subsequent searches are within these nodes only). This procedure is similar to query on IT with interval starts as keys, which is consistent with the observation that IT and range trees have comparable query efficiency.

	ENCODE intervals	gnomAD intervals
Total number	1,340,125,581	241,056,551

Table 4. Total number of intervals in our experiments.

	RTFC	2D-RT	IT
Total building time (s)	6971.58	11569.15	**3904.27**

Table 5. The total building time (in seconds) of the three tree structures on ENCODE genomic intervals for the 23 chromosomes. Abbreviations: RTFC = range tree with fractional cascading; 2D-RT = basic 2-dimensional range tree; IT = interval tree.

We also tested the coarse interval query performance of "*findOverlaps*" function in the IRanges R package in Bioconductor version 3.4, where the interval query is implemented based on Nested Containment Lists (NCList)[8]. Since the "*findOverlaps*" function can only perform some coarse interval queries that contain a number of interval relationships in Allen's Algebra, e.g., the coarse type "within" consists of four refined relationships (i.e., *d, s, f, e*) in Allen's Algebra. The interval relation types in "*findOverlaps*" and the corresponding Allen's interval relationships are shown in Table 7. When testing "*findOverlaps*", we pre-constructed the NCList structure for the data intervals by "*NCList()*", and then counted the time cost of "*findOverlaps*" for different types of interval queries, excluding the data structure construction time from query time. The R code that calculated the "*findOverlaps*" time cost is in the Supplementary S1. Table 7 also shows the performance of NCList for the corresponding interval queries using "*findOverlaps*" and the performance using RTFC for the corresponding refined interval queries, with gnomAD intervals as query intervals and ENCODE intervals as data intervals. From Table 7, we can see that, even for the coarse interval queries, RTFC can outperform NClist by performing all the corresponding refined interval query. It should be noted that "*findOverlaps*" function provides five separate query types. The "*within*", "*start*", "*end*" and "*equal*" queries are not implemented through filtering of the "*any*" query. NCList can handle these query types separately. But NCList cannot handle the more refined interval queries defined by Allen's interval algebra directly. If applying NCList to a more refined interval query defined by Allen's interval algebra, it needs to perform a coarse interval query, and then search in the results for intervals satisfying the refined relation, this would take even more time than the coarse interval query.

We also tested BEDTools and BEDOPS for the coarse interval intersection on the same datasets and compared the performances with RTFC, as shown in Table 8. For each query interval, RTFC can return all the overlapping intervals from a set of intervals. BEDTools integrates the data structure construction process and interval query process in one command. To ensure fair comparison, we also report total time for RTFC in Table 8 including building and query time. BEDOPS takes in two bed files each consisting of a set of intervals, but returns only the subset of overlapping intervals in the first bed file and ignores the overlapping intervals in the second bed file. In addition, BEDOPS requires sorted BED files as input, thus, the BED files need to be sorted before interval query. The time consumptions of the three tools for coarse interval query are shown in Table 8, where we can see that RTFC is more efficient than BEDTools and BEDOPS in interval query time. Though the single sort operation in BEDOPS is faster than RTFC construction, when finding how the intervals in two bed files overlap, we need to repeat BEDOPS sort and query operations for each interval in the second bed file, whose time quickly dwarfs that of RTFC.

Discussion

Allen's interval algebra can be applied to genomic intervals to refine the relations between two genomic intervals. There are 13 possible relations between two intervals according to Allen's interval algebra. Through rewriting the definition of Allen's interval relations, we transform the interval query problem to the range query problem and efficiently solve the problem using the range tree data structure with fractional cascading. Though there are many studies on the interval queries for the coarse intersection relation in the literature, this is the first study with implementation, to our knowledge, that tries to improve the query efficiency for more refined interval relations defined in Allen's algebra. Our results show that the range tree data structure can be more efficient than an interval tree for the genomic interval queries in most cases, and the technique of fractional cascading can further improve the query efficiency for range trees.

From the results in Table 6, we can also observe that different queries on the same tree structure can consume quite different times even though their result sizes are approximately equal, e.g., *o vs. oi*, *si vs. fi*, and *m vs. mi*. Though the theoretical time complexities for range trees ($O(logn+k)$ for RTFC, $O(log^2n+k)$ for 2D-RT) are the same across different query relations, the search orders for the two coordinates in our implementation and the widths of the query intervals make the actual query times different (i.e., they affect the constant factors hidden in the $O(\cdot)$ notations). In our implementation of the range tree, we built the range tree using the start of an interval as the key of a node in the first level tree (i.e., x-tree). During the query, we first searched the x-tree then searched the y-trees (for 2D-RT) or a point array (for RTFC). If constraint on the start is rewritten to correspond to a narrow range, we can eliminate many intervals whose starts are not in this range by the first search on the x-tree, and the following search on y-trees that have less intervals become more efficient. For example, for a query interval $[x', y']$, an *oi* query ($x' < x < y'$) usually has a narrower range for the start than an *o* query ($0 < x < x'$). This is consistent with our observation that *oi* queries consume less times than *o* queries in our implementation. Similarly, *si* queries ($x = x'$) are more efficient than *fi* queries ($0 < x < x'$) and *m* queries ($0 < x < x'$) are less efficient than *mi* queries ($x = y'$).

Query	RTFC	2D-RT	IT	Result size
o	**97.19**	258.18	1774.81	36,731,416
oi	**26.83**	30.57	128.20	36,573,171
d	**26.63**	29.00	132.00	96,633
di	**551.69**	986.35	1935.54	47,218,140,890
s	**35.60**	137.67	128.68	3,005
si	144.63	156.85	**136.54**	113,873,940
f	**40.08**	43.68	126.52	3,220
fi	**319.12**	554.89	1733.57	113,684,059
m	**303.59**	548.73	1729.51	113,785,360
mi	**132.77**	155.19	134.88	114,013,875
=	147.69	157.90	**124.21**	417

Table 6. The query time of the 11 intersection queries (in seconds) and the corresponding result set sizes with gnomAD intervals as query intervals and ENCODE intervals as data intervals. Abbreviations: RTFC = range tree with fractional cascading; 2D-RT = basic 2-dimensional range tree; IT = interval tree.

Coarse types	Refined relations	NCList query time	RTFC query time
"any"	*o, oi, d, di, m, mi, s, si, f, fi, =*	5654.42	**1825.82**
"within"	*d, s, f, e*	942.71	**250.00**
"start"	*s, si, e*	1007.73	**327.92**
"end"	*f, fi, e*	1033.35	**506.89**
"equal"	*e*	968.37	**147.69**

Table 7. The coarse interval query types in "*findOverlaps*" function and the corresponding refined relations in Allen's interval algebra. Column "NCList query time" indicates the query time (in seconds) using "findOverlaps" function with gnomAD intervals as query intervals and ENCODE intervals as data intervals. Column "RTFC query time" indicates the query time to execute all the corresponding interval queries with respect to the refined relations using RTFC.

		BEDOPS*			RTFC		
Tools	BEDTools	sort	query	total	build	query	total
Time consumption	126911.72	2111.65	2388.56	4500.21	6971.58	**1825.82**	8797.40

Table 8. The time consumptions (in seconds) of BEDTools and BEDOPS for coarse interval intersection query compared with RTFC. The data intervals are ENCODE intervals and query intervals are gnomAD intervals. *BEDOPS cannot return the detailed intersection information as RTFC and BEDTools, it returns only the subset of overlapping intervals in the first bed file and ignores the overlapping intervals in the second bed file.

For simple queries where the start is fixed, such as *s*, *si*, *mi*, and =, as explained in Section 4, their major tasks are to search all the intervals that have a certain start essentially. In these cases, one can use a simple binary search tree such as an interval tree to perform the query as efficiently as a range tree, or even more efficiently due to interval tree's simpler data structure. For example, a range tree stores all the points in leaf nodes, while an interval tree can store points in internal nodes. Thus a range tree is one level higher than an interval tree when storing the same number of intervals.

The range tree data structure is more complex than the interval tree structure as shown in Figure 4; constructing a more complex data structure usually takes more time. However, the tree building process is upfront, once the tree structure is constructed, it can be used for interval queries with respect to any relations and any query intervals repeatedly. Thus, as a general case in genomics, when performing a large number of interval queries, improving the query efficiency is much more important than improving the building efficiency. Nonetheless, we plan to improve the building efficiency for range trees by optimizing the insertion algorithm in our future study.

Conclusion

Allen's interval algebra can provide more refined relations between intervals. These more refined relations can provide more detailed information for genomic annotations and facilitate future discovery on genomics. Improving the efficiency of interval query regarding relations in Allen's algebra is essential for multiple bioinformatics applications. In this study, we developed a novel approach for efficient interval queries, which is optimal in theory and fast in practice. Our approach transforms an interval query problem to a range query problem by rewriting the definition of Allen's interval relations. We then developed and implemented a basic 2-dimensional range tree (2D-RT) and an augmented range tree with fractional cascading (RTFC) to efficiently solve the range

query problem. In particular, RTFC can have an optimal theoretical time complexity of $O(\log n + k)$. Our experimental results show that 2D-RT is more efficient than an interval tree for most of the queries and RTFC can further improve the query efficiency. Thus, the RTFC provides a data structure that can perform the interval queries efficiently on genomic datasets and can facilitate faster genomic data analysis and knowledge discovery.

Data Availability

The ENCODE metadata, including the data file download URLs, is provided in the Supplementary Dataset. gnomAD data is publicly available at http://gnomad.broadinstitute.org/downloads. Our implementation of range tree with fractional cascading, traditional 2-dimensional range tree and interval tree are available at https://github.com/mocherson/range-tree.

References

1. Kent, W. J. *et al*. The human genome browser at UCSC. *Genome Res* **12**, 996–1006 (2002).
2. Dunham, I. *et al*. An integrated encyclopedia of DNA elements in the human genome. *Nature* **489**, 57–74 (2012).
3. Layer, R. M., Skadron, K., Robins, G., Hall, I. M. & Quinlan, A. R. Binary interval search: a scalable algorithm for counting interval intersections. *Bioinformatics* **29**, 1–7 (2013).
4. Wiley, L. K., Sivley, R. M. & Bush, W. S. Rapid storage and retrieval of genomic intervals from a relational database system using nested containment lists. *Database* **2013**, bat056 (2013).
5. Seok, H. S., Song, T., Kong, S. W. & Hwang, K. B. An efficient search algorithm for finding genomic-range overlaps based on the maximum range length. *Ieee Acm T Comput Bi* **12**, 778–784 (2015).
6. Li, H. *et al*. The sequence alignment/map format and SAMtools. *Bioinformatics* **25**, 2078–2079 (2009).
7. Quinlan, A. R. & Hall, I. M. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics* **26**, 841–842 (2010).
8. Alekseyenko, A. V. & Lee, C. J. Nested containment list (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. *Bioinformatics* **23**, 1386–1393 (2007).
9. Richardson, J. E. fjoin: Simple and efficient computation of feature overlaps. *J Comput Biol* **13**, 1457–1464 (2006).
10. Neph, S. *et al*. BEDOPS: high-performance genomic feature operations. *Bioinformatics* **28**, 1919–1920 (2012).
11. Lawrence, M. *et al*. Software for computing and annotating genomic ranges. *Plos Comput Biol* **9**, e1003118 (2013).
12. Luo, Y. & Szolovits, P. Efficient queries of stand-off annotations for natural language processing on electronic medical records. *Biomed Inform Insigh* **8**, BII–S38916 (2016).
13. Lek, M. *et al*. Analysis of protein-coding genetic variation in 60,706 humans. *Nature* **536**, 285–291 (2016).
14. Allen, J. F. Maintaining knowledge about temporal intervals. *Commun Acm* **26**, 832–843 (1983).
15. Bentley, J. L. Decomposable searching problems. *Inform Process Lett* **8**, 244–251 (1979).
16. Lueker, G. S. A data structure for orthogonal range queries. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)* 28–34 (IEEE, 1978).
17. De Berg, M., Van Kreveld, M., Overmars, M. & Schwarzkopf, O. C. Orthogonal Range Searching. In *Computational geometry* 105–109 (Springer, 2000).
18. Edelsbrunner, H. A new approach to rectangle intersections .1. *Int J Comput Math* **13**, 209–219 (1983).
19. Chazelle, B. & Guibas, L. J. Fractional cascading: I. A data structuring technique. *Algorithmica* **1**, 133–162 (1986).
20. Chazelle, B. & Guibas, L. J. Fractional cascading: II. Applications. *Algorithmica* **1**, 163–191 (1986).
21. Willard, D. E. The super-B-tree algorithm. (Cambridge, MA: Aiken Computer Lab, Harvard University, 1979).
22. Khurana, E. *et al*. Role of non-coding sequence variants in cancer. *Nat Rev Genet* **17**, 93–108 (2016).
23. Vorstman, J. A. S. *et al*. Autism genetics: opportunities and challenges for clinical translation. *Nat Rev Genet* **18**, 362–376 (2017).
24. Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. *Introduction to algorithms*, (MIT press Cambridge, 2001).

Acknowledgements

This study is supported in part by the NIH grant R21 LM012618-01.

Author Contributions

Y.L. and A.E. originated the study. Y.L. and A.E. procured and curated the datasets. Y.L. designed the algorithms. C.M. implemented the algorithms and conducted the experiments. C.M. and Y.L. analyzed the results and prepared the manuscript. All authors contributed to the revision of the manuscript.

Additional Information

Supplementary information accompanies this paper at https://doi.org/10.1038/s41598-019-41451-3.

Competing Interests: The authors declare no competing interests.

Publisher's note: Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.