Optimized 2D Ball Trees

Luis Carlos dos Santos Coutinho Retondaro
CEFET/RJ - Centro Federal de Educação Tecnológica
do Rio de Janeiro
Petrópolis, RJ
Email: luis.retondaro@cefet-rj.br

Claudio Esperança
Programa de Engenharia de Sistemas e Computação
COPPE - Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ
Email: esperanc@cos.ufrj.br

***Abstract*—Ball trees are hierarchical bounding structures – usually binary trees – where each node consists of a ball (circle, sphere, etc) enclosing its children. Approaches for building an optimal ball tree for a given set of leaves (points or balls enclosing other geometric primitives) typically rely on minimizing some function of the shape of the tree, regardless of the intended application. In this paper we examine the problem of building ball trees for 2D primitives, trying to balance construction time with the efficiency of the produced trees with respect to a set of distance-based queries. In particular, we present three new construction algorithms, propose an optimization whereby each internal node is the smallest ball enclosing all leaves rooted at that node, and describe enhancements to several distance query algorithms. Moreover, an extensive experimental study was conducted in order to evaluate our algorithms with different kinds of data sets, including ball collections that approximate 2D shapes.**

I. Introduction

Querying spatial relationships in two-dimensional geometric models is a common activity performed by many applications. Typically, data structures such as bounding volume hierarchies (BVH) are used to model spatial distributions. These tree structures employ a variety of simple volumes, such as balls[1] [1], [2], axis-aligned bounding boxes [3], [4] or oriented bounding boxes [5], [6].

Many varieties of BVHs have been proposed for efficient processing of spatial queries, such as M-trees [7], VP-trees [8], R-trees [9], [10], and Ball trees [11], [12]. In this work we explore binary ball trees in two dimensions; thus, we use the term *ball* as a synonym for *circle*.

Ball trees are classic data structures used in many applications, such as collision detection [13], [14], image matching [15], data anonymization [16], non-parametric methods [17], and searching spatial data [10], [11].

Our main motivation for studying ball tree construction algorithms is our interest in measuring relative distances between irregular two-dimensional shapes at interactive rates. In particular, we intend to use ball trees to deal with scanned documents that are segmented into pieces that can be grouped independently of their original position on the page.

[1]Here the term is used as a generalization of circles (or disks) in two dimensions, spheres in three dimensions, or hyperspheres in 4 or more dimensions.

A. Contributions

In this work we propose the following contributions that build on the algorithms and experiments on ball trees as described in the seminal paper by Omohundro [12]:

- The three best-performing ball tree construction algorithms described by Omohundro were extended with what we call the *enclosing leaves optimization - (EL)*, i.e., internal nodes built as the smallest bounding circles of all descendant leaf nodes. This is in contrast with the usual practice of bounding just the two immediate descendants.
- Three new ball tree construction algorithms are proposed, also extended with the *EL* optimization.
- In addition to experiments with different random distributions of points, we also use data sets containing collections of circles approximating test shapes, since these arise in many important applications.
- Experiments were conducted to gauge the quality of ball trees as they are used in six different types of queries. This experimentation supplements the usual practice of equating quality to the total ball area.
- The *branch and bound* strategy used in tree distance and tree intersection queries employs a new distance bound metric that is tighter than the usual maximum distance metric (see Section IV).

II. Fundamentals

A ball tree is constructed from an input set of balls. Depending on the intended application, input balls may represent different entities, such as the shape of an object, bounding balls of some collection of discrete objects, point clouds, etc. An *optimal* ball tree for a given input clearly depends on its performance in the application at hand, which, in turn, depends on the distribution of nodes in the tree.

Construction algorithms for BVHs such as ball trees can be divided into three classes: Top-down, Bottom-up and Insertion-based construction [14]. The top-down approach is the most popular, where the algorithm recursively subdivides the collection into smaller and smaller partial collections until only one element is left and placed in a leaf node. Bottom-up algorithms locate pairs of nodes that are close together, creating parent nodes pointing to them; this is done recursively from individual leaf nodes until reaching the root of the tree. The insertion (also called incremental) method starts with an

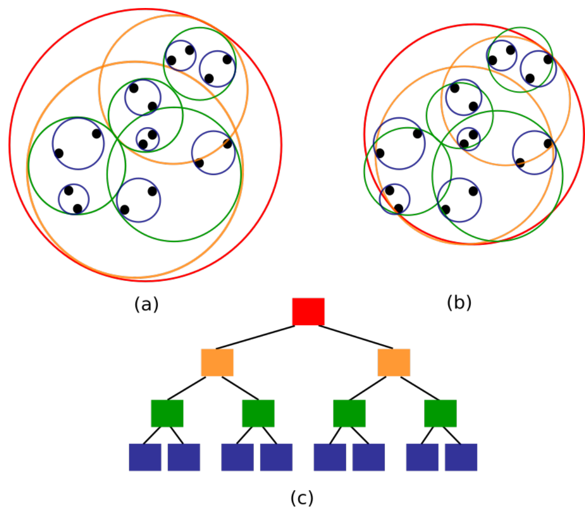


Fig. 1. Typical binary ball tree representation. (a) shows a ball tree where internal nodes enclose the two immediate children, while in (b) they enclose all descendant leaf nodes. (c) depicts the tree topology, which is the same for both trees.

empty tree, inserts one ball at a time at the best location in the tree.

Omohundro [12] states that bottom-up strategies are generally better, despite having a high construction cost, because they tend to be more efficient in finding optimal grouping. Omohundro uses the total volume of all balls in a ball tree as a measure of its quality and concludes that minimizing the total volume may be the most efficient criterion for most applications.

Let us consider the problem of constructing a ball tree for a collection of n balls. To simplify the analysis, we assume a bottom up approach, where all leaf nodes are paired together in order to build the next higher level. Assuming that n is even, the lowest level of the tree will consist of $m = \frac{n}{2}$ pairs of balls. Let's call P the total number of possible pairings, such that $P(n)$ is related to Stirling numbers of the second kind $\left\{ {n \atop k} \right\}$, with $k = 2$. Then,

$$P(n) = \frac{1}{m!} \prod_{j=0}^{m-1} \binom{n-2j}{2} = \frac{n!}{m! 2^m}. \tag{1}$$

By examining Equation 1, it can be shown that $P(n) \in O(c^n)$ for some constant $c > 1$. Table I shows a few values of $P(n)$, which makes this clearer. As the search space for this problem is huge, a deterministic approach to find a tree that can be considered *optimal* in some sense is out of the question and we must rely on heuristics.

TABLE I
NUMBER OF PAIRINGS OF n ELEMENTS FOR SOME VALUES OF n.

n	4	6	8	10	30	50
$P(n)$	6	90	2,520	113,400	8×10^{27}	9×10^{56}

A. Enclosing leaves optimization

The strategy used by Omohundro to define the ball corresponding to a given node follows that of most approaches found in the literature (e.g. [11], [18]), i.e., the root of a subtree is defined as the smallest ball containing the two immediate descendants, which can be computed in constant time.

However, circles (and balls, in general) do not enjoy the enclosing properties of other shapes used as bounding structures. For instance, if two axis-aligned rectangles R_1 and R_2 are the smallest such rectangles which enclose two point sets S_1 and S_2, then it follows that the smallest axis-aligned rectangle that contains $R_1 \cup R_2$ is also the smallest with respect to $S_1 \cup S_2$. The same cannot be said for circles.

In this work, once a ball tree is built, an additional step – which we call the *Enclosing Leaves* optimization – is performed to ensure that each internal node corresponds to the smallest circle containing all leaves rooted at that node. We note that finding the smallest circle which contains a collection of circles can be done in $O(n)$ [19]–[22], and thus this step takes $O(n^2)$ for a tree with n leaf nodes.

B. Related problems

Ball trees can be seen as level-of-detail approximations of the object, in the sense that nodes at lower levels of the hierarchy (closer to the root) provide less accurate but more concise approximations than those at deeper levels. For example, collision detection between two objects represented by ball trees A and B is calculated by a recursive descent process, where pairs of nodes at increasingly deeper levels are tested for intersection.

The concept of tightness can be characterized in several ways for BVHs in general and for ball trees in particular. For instance, clearly, the root node of a ball tree in 2D containing n leaves (circles) must be the smallest circle containing all leaves (see Fig 1). However, depending on how the collection is partitioned to form the two children of the root, we might have considerably different results. We might strive to obtain the smallest circles for each subcollection, i.e., obtaining the smallest total area for their bounding circles. Alternatively, we might try to obtain subcollections with disjoint bounding circles, or at least with the smallest possible overlap, since queries considering points within the intersection of the two circles would necessarily traverse both subtrees. Another appealing heuristic is to split the collection into two subcollections with roughly the same number of elements, since this would lead to more balanced trees. Once we settle on the criterion for minimality, however, finding the optimal ball tree for a given input is a combinatorial problem of exponential complexity. Note that in this paper we focus on 2D problems and adopt the criterion of minimum total area for the circles of all nodes.

Another related problem is that of approximating a given shape S by a minimal set of balls B such that the distance from a point to S is bounded by the distance from the point to B within a given error margin ϵ.This way, shallow levels of the tree provide a rough approximation of the object. For

instance, finding the closest pair of points between two objects represented by balanced ball trees takes time $O(\log^2 n)$ rather than $O(n^2)$ [23].

Typically, the approximation algorithm first computes the distance transform (*DT*) of S, extracting a skeleton based on the discrete medial axis (*DMA* [2], [24]–[26]) and using points along the skeleton as circles of maximum radius [27]–[29].

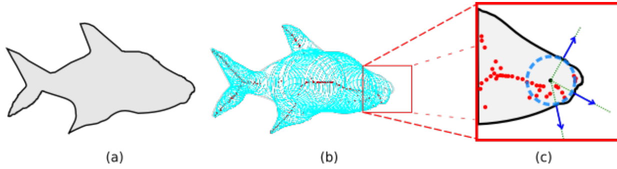


Fig. 2. Example of our own algorithm for covering a shape with circles: (a) Original shape. (b) Object approximated by a union of balls. (c) A maximal circle with center on the *DMA*.

III. Ball tree Construction

Although ball trees have been proposed as a type of *metric trees* [30] and used as spatial decomposition devices, we do not concentrate on construction algorithms proposed for that purpose (e.g., [31]). Rather, we focus on their use in tasks such as collision detection and assorted Euclidean distance-related queries, using on Omohundro's seminal work on ball tree construction [12] as a yardstick.

Six ball tree construction algorithms were implemented[2]. The first three follow the discussion of Omohundro [12] and we refer to them as the *KD construction*, *Online* and *Bottom up* algorithms. We note that, although two other algorithms were tested by Omohundro, these three were considered the best performing. Our implementations differ from those described in [12] mainly in two aspects: the first is the use of *EL optimization*, described earlier. The second is that leaf nodes may consist of balls, rather than points.

Next, we present a quadratic algorithm (*Q*) and two others that use Voronoi neighborhoods [32], [33] which we call *V0* and *V1*.

A. *KD construction algorithm*

This is a top-down offline algorithm, similar to the method proposed by Friedman [23] for building KD-trees. According to [12], if the data set is large and relatively uniform, then the KD approach is fast and simple and more suitable. If the data is clustered or sparse or has extra structure, the KD approach tends not to reflect that structure in its hierarchy. The tree is built by recursively splitting each set into two subsets with roughly the same number of balls. At each stage, the algorithm determines the axis of largest dispersion and the median value with respect to that axis is used to perform the split. The overall complexity of this algorithm is $O(n \log n)$. However, adding *EL* optimization increases its complexity to $O(n^2)$.

[2]The implementation and demo code for all these algorithms can be found at https://observablehq.com/@lretondaro/optimized-2d-ball-trees.

B. *Online algorithm*

This algorithm builds the tree incrementally. It tries to insert each new ball into the tree, searching for the ideal location, i.e., the one that contributes the least to the total volume of the tree. This ball tree structure tends to group the closest nodes as siblings and leave those that are more distant or that are much larger near the top of the tree and in different branches. Once the sibling of the node is determined, the algorithm yields a parent and inserts it together with the new leaf into the tree. The ball tree is built in $O(n \log^2 n)$. Adding *EL* optimization raises the complexity to $O(n^2)$.

C. *Bottom up algorithm*

Initially, the algorithm finds the pair of balls that has the smallest enclosing ball for each node. These pairs are kept in a priority queue and at each iteration the best partners are recalculated with the help of an auxiliary insertion tree built with the online algorithm. Nodes that have already been combined are removed from the insertion tree and when there is only one node left in the insertion tree, construction is complete in $O(n^2 \log^2 n)$.

D. *Q algorithm*

In this algorithm, the input ball collection B is initially sorted by radius. Then, the optimal pair for each element in this sorted order is determined, i.e., for B_i, all elements B_j such that $j > i$ — and such that j has not yet been paired — are examined and the element yielding the smallest bounding circle is chosen. The internal nodes formed by pairing balls of B are again sorted by radius and a new collection B' is created, containing only a fraction $0 < \alpha \leq 1$ of these having the smallest radii is kept, whereas the other pairs are dissolved and their immediate children are also put in B'. The whole process is then repeated, now considering the new collection B', until B' contains a single ball - the root of the tree. The complexity of this algorithm is $O(n^2)$, but the constant depends on α: larger values result in a smaller number of repetitions.

E. *V0 algorithm*

Similar to the *Q* algorithm, the input balls are placed in a list L sorted by radius. This algorithm builds the Voronoi diagram V, whose sites correspond to the centers of the balls $\in L$. For each element $V_i \in V$, there is a set of points that depict its neighborhood sites $N(i)$. Then, for each V_i element, all sites $j \in N(i)$ are examined, i.e., the respective corresponding balls $\in L$. The optimal pair is determined when yielding their smallest enclosing circle. The balls i and j are removed from the L list and the new parent ball is inserted. So, the list L is sorted again and the algorithm rebuilds V. As long as there are balls in L, the algorithm continues until it finishes building the tree from the last single ball. The complexity of the algorithm *V0* is $O(n^2 \log n)$.

F. *V1 algorithm*

Using the same approach as *V0*, this algorithm selects the optimal pair of balls in L list, from the Voronoi neighborhood comparison for each iteration, i.e., examining all elements of $N(i)$ elements for each V_i. However, the elements $\in L$ are not ordered. Instead, they are shuffled only once at start, so that the elements are always taken in random order. The procedure to update list L is the same as used in *V0* and the ball tree is built in $O(n^2 \log n)$

IV. Query Algorithms

The evaluation of a particular ball tree construction algorithm, in addition to an analysis of its time and space complexity, typically includes measurements of certain characteristics of trees built from assorted inputs. As mentioned earlier, the *de facto* standard for this kind of measurement is the total area of all balls in the tree. In the present study, however, we add other empirical measures to this mix, namely, we conduct additional experiments where trees built with each algorithm listed in Section III are used for answering a variety of proximity queries.

A. *Query processing strategy*

Distance-related queries between two trees A and B are routinely computed by a concerted descent of the two trees, analyzing one pair of nodes $(a, b)$for$a \in A, b \in B$ at a time. The process clearly must start with the two root nodes, but from this point on, choosing the next pair to be examined must be informed by some heuristic. The idea is to prioritize pairs of nodes with a greater chance of leading to a better solution to the query than what has been seen so far.

Our query algorithms involving two ball trees A and B follow a *branch-and-bound* strategy, where, at each step, a pair $(a, b) \mid a \in A, b \in B$ of nodes is considered only if combinations of their descendants (if any) can still improve the objective function of the search. The traversal of the search space is performed with the help of a priority queue P ordered with respect to some function $f_P(a, b)$. This function ranks a pair of nodes a, b according to its chance of containing the solution among pairs built with a and b or their descendants. As an additional requirement, $f_P(a, b)$ must support an early termination of the search, that is, it must be possible to determine whether P still contains pairs worthy of examination after examining the pair at the head of the queue.

As an example, consider Algorithm 1, which computes the shortest distance between A and B, i.e., the shortest distance between two leaves $a \in A, b \in B$. Clearly, the early termination clause expressed in line 6 implies that no pair of balls (a', b') still in the queue may yet yield a pair of leaves whose distance is smaller than $minDist$. In other words, if a' is the set of all leaves in the tree rooted at a (and analogously for b'), and $dist(a, b)$ stands for the Euclidean distance between a and b, then we define

$$d_{min}(a, b) = \min\{dist(a'_i, b'_j) \mid a'_i \in a, b'_j \in b\},$$

Algorithm 1 - Tree Distance

Input: A,B {ball trees}
Input: f_P {priority queue ranking function}

```
1:  P ← priority queue ordered by f_P
2:  P.push((root(A), root(B)))
3:  minDist ← ∞
4:  while P ≠ ∅ do
5:    (a, b) ← P.pop()
6:    if dist(a, b) > minDist then
7:      return minDist
8:    end if
9:    if isLeaf(a) and isLeaf(b) then
10:     minDist ← dist(a, b)
11:   else if isLeaf(a) and not isLeaf(b) then
12:     P.push((a, b.left))
13:     P.push((a, b.right))
14:   else if not isLeaf(a) and isLeaf(b) then
15:     P.push((a.left, b))
16:     P.push((a.right, b))
17:   else
18:     P.push((a.left, b.left))
19:     P.push((a.left, b.right))
20:     P.push((a.right, b.left))
21:     P.push((a.right, b.right))
22:   end if
23: end while
24: return minDist
```

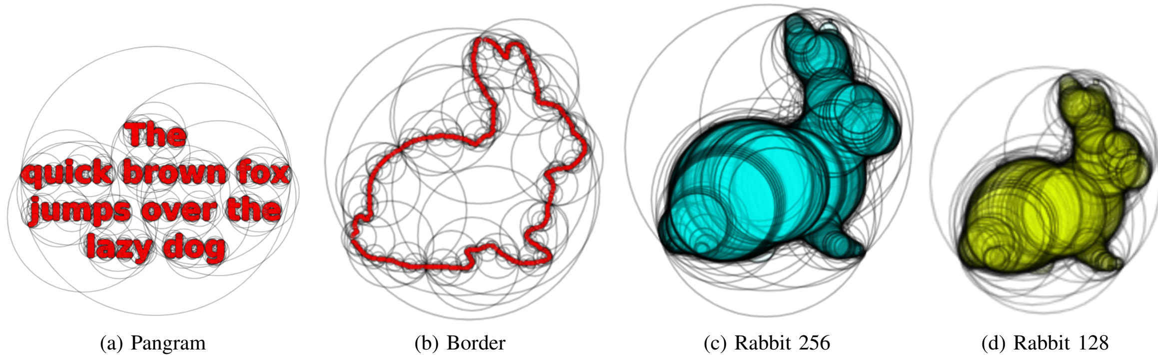
and require that $f_P(a, b) \geq d^+_{min}(a, b) \geq d_{min}(a, b)$, where d^+_{min} is some heuristic that bounds the real value of d_{min}.

On the other hand, if two pairs (a_1, b_1) and (a_2, b_2) are such that $d^+_{min}(a_1, b_1) = d^+_{min}(a_2, b_2)$, it may be advantageous to assign a different priority to them. This can be done by adding a second criterion to break the tie. We propose using a maximum distance estimate $d^+_{max}(a, b)$ for that purpose, i.e., a heuristic that bounds

$$d_{max}(a, b) = \max\{dist(a'_i, b'_j) \mid a'_i \in a, b'_j \in b\}.$$

The rationale is that a pair with lower d^+_{max} should be considered before another with higher d^+_{max}.

Some authors propose using a ranking function based on the area of intersection between the two nodes [34], but this scheme does not establish a distinction between pairs that do not intersect. A more natural choice for $d^+_{min}(a, b)$ is $dist(a, b)$, but we can do better if a and/or b are internal nodes by considering the minimum distance between the up to four possible pairings of their children. Similarly, $d^+_{max}(a, b)$ can be estimated by the maximum distance between a and b but a better (tighter) estimate might be the smallest of the maximum distances for the up to four pairings of their children. In our experiments, a ranking function f_P using the minimum distance between children with ties broken with the maximum distance between children gave the best results.



(a) Pangram (b) Border (c) Rabbit 256 (d) Rabbit 128

Fig. 3. Ball trees of shape approximation - colored leaf balls cover the object shape.

B. Other queries

In addition to the Tree Distance (*TD*) query, we also conducted tests for the following queries:

- ***Tree intersection (TI)*** - Finds a pair of leaf nodes (a, b) such that $a \cap b \neq \emptyset$. This algorithm is analogous to Algorithm 1, except that (1) node pairs popped from P which do not intersect are discarded right away; (2) if a pair of intersecting leaf nodes is found, the algorithm returns `true`; and (3) when P becomes empty, `false` is returned.
- ***Point Intersection (PtI)*** - Given a point p, finds a leaf node a, such that $a \supset p$. This is analogous to ***TI***, regarding p as a tree with a single node.
- ***Distance to Point (DtP)*** - Given a point p, finds a leaf node a, such that $dist(a, p)$ is minimal. This is analogous to ***TD***, regarding p as a tree with a single node.
- ***Line intersection (LnI)*** - Given a line r, finds a leaf node a, such that $a \cap r \neq \emptyset$. This algorithm is similar to ***TI***, except that (1) f_P uses distance estimates between balls and line r; (2) a node popped from P which does not intersect r is discarded right away; (3) if a leaf node intersecting r is found, the algorithm returns `true`; and (4) when P becomes empty, `false` is returned.
- ***Distance to Line (DtL)*** - Given a line r, finds a leaf node a such that $dist(a, r)$ is minimal. This algorithm adapts ***TD*** in much the same way ***LnI*** is an adaptation of ***TI***.

It should be mentioned that Omohundro [12] also proposes an algorithm for DtP queries which does not require a priority queue. Rather, it is a simple recursive depth-first search where at each step the child closest to the query point is traversed first. In our experiments, though, the global node selection criteria provided by a priority queue yields better results in terms of the number of visited nodes. Of course, the cost of maintaining a priority queue for this type of query may offset the gains obtained thus. At any rate, however, the adoption of priority queues in all query algorithms provides a uniform standard for judging the impact of the construction algorithm in the query results.

V. Experiments

In order to evaluate the construction algorithms described in Section III, we conducted a host of different experiments. Besides evaluating directly these algorithms by measuring construction time and total ball area, we also conducted experiments to gauge how well the constructed trees behave with respect to the proximity queries described in Section IV. Since more than $3.5M$ data points were collected in 218 different types of experiments, here we comment only the most relevant findings[3]. All algorithms were coded in JavaScript and run in a Chrome v90.04 browser on a workstation with a Intel i7 1.8GHz processor (8 cores) with 12 GB main memory, running Ubuntu Linux v20.04 64 bits.

A. Test data

For the experiments, we used input ball sets falling in two general classes of distributions: (1) *random uniform and Cantor distributions*, similar to those used in [12] ; and (2) *shape approximations*, i.e., ball sets that approximate 2D shapes as discussed in Section II-B (see Fig. 3).

The balls in the random collections have radius 0.01 distributed inside a square of unit size. Two collections with 500 balls each were used and these are called 500-R and 500-C according to the distribution (uniform or Cantor).

The four *shape approximation* ball collections are representative of pictures, symbols and text that we can extract from documents: *Pangram*, *Border*, *Rabbit 128* and *Rabbit 256*. The balls in each collection were produced from source images using our own ball approximation algorithm with $\epsilon = 0.01$ (see Section II-B) and are shown in Fig. 3, whereas relevant statistics are shown in Table II.

TABLE II
THE *shape approximation* BALL COLLECTIONS

Shapes	Rabbit 128	Rabbit 256	Pangram	Border
Image Resolution	124×122	255×250	144×81	247×243
# Balls	190	391	997	$1,254$

[3]All data, as well as interactive charts that allow its exploration can be found at https://observablehq.com/@lretondaro/optimized-2d-ball-trees-results.

B. Query testing methodology

In order to provide a fair sampling of the queries, these were performed considering a fixed test area equivalent to a square of side 8. This space is subdivided into a 40×40 regular grid, yielding 41×41 points. All ball trees are scaled so that their root node has unit radius. For each query type, one object is placed at the center of the grid, while the comparison object varies in position, generating a total of 1681 individual queries. For queries involving a tree and a point, the point remains fixed at the center while the tree is translated so that the root ball's center coincides with each grid point. Likewise, for queries involving a line, this is placed centered on the vertical axis of the grid.

We also distinguish query samples taken when the objects are close together from those taken when they are distant. We use the term *close* to refer to queries taken when there is at least some chance of intersection between the two objects, otherwise we classify the query as *distant*. This distinction is useful since the overall distance between the objects clearly impacts the time complexity of the query. For instance, a ***PtI*** *distant* query can be answered with just one comparison.

Since all query algorithms rely on a priority queue, the time complexity of each individual query is measured in terms of the total number of operations (*totalOps*) executed on the queue, i.e., the number of *push* and *pop* operations. This is reasonable, since these are the most expensive operations executed for each iteration of the loop (see Algorithm 1), having complexity $O(\log n)$ each, where n is the size of the queue.

C. Tree construction

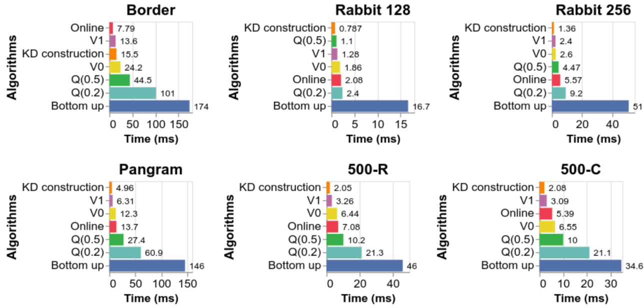


Fig. 4. Ball tree construction algorithms by run time (without EL optimization).

Initially, we present a study of the running time of ball tree construction algorithms. Two versions of the Q algorithm were used, with $\alpha = 0.2$ and 0.5, respectively. All algorithms, except for the Bottom up, exhibit a similar speed, with KD construction running the fastest in most tests (see Figure 4). The Bottom up algorithm is significantly slower than all other algorithms. For instance, for the Rabbit 256 data set, it runs more than 37 times slower than the fastest algorithm (KD construction) and more that 5 times slower than the second slowest algorithm – Q(0.2). Also notice that decreasing Q's α value from 0.5 to 0.2 impacts severely the its running time, more than doubling this measure for all data sets.

The running times shown in Fig. 4 exclude the EL optimization step, which, although quadratic, is quite fast for the tested data sets (see Table III). Indeed, even when paired with the fastest construction algorithm (KD construction), it does not take more than 12% of the total time even for the largest data set (Border). Clearly, when run after a slow algorithm, its impact in the total running time becomes negligible.

TABLE III
EL OPTIMIZATION TIME (MS)

	Bottom up	KD	Online	Q(0.2)	Q(0.5)	V0	V1
500-C	0.69	0.62	0.71	0.67	0.68	0.69	0.68
500-R	0.72	0.66	0.69	0.65	0.63	0.67	0.69
Border	1.74	1.92	2.22	1.69	1.59	1.62	1.61
Pangram	1.47	1.36	1.60	1.35	1.32	1.48	1.37
Rabbit 128	0.24	0.22	0.26	0.22	0.19	0.21	0.19
Rabbit 256	0.47	0.39	0.53	0.46	0.44	0.45	0.46

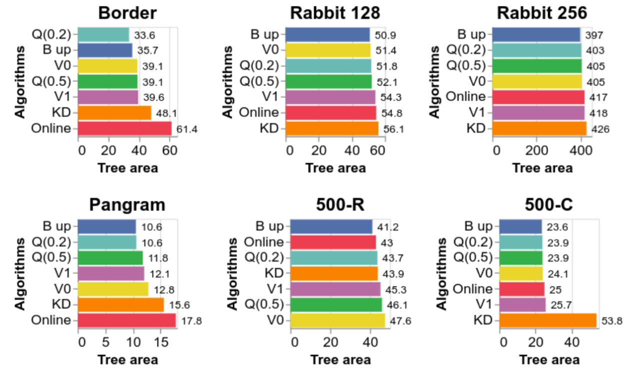


Fig. 5. Ball tree construction algorithms by total tree area (with EL optimization)

With respect to the total area, we confirm that the Bottom-up algorithm produces trees with the smallest overall area for most data sets (see Fig. 5). This advantage is largest in the case of the 500-C data set, where KD construction produces a tree with total area more than 2.2 times larger than that produced by the Bottom-up algorithm. We also note that the construction algorithms introduced in this paper fare rather well for all data sets. For instance, the worst result obtained with the V0 algorithm was for the Pangram data set, with an area only 20% larger than that of the Bottom up, whereas for all other data sets, its result was at most from 2% to 6% larger. Still for Pangram, the best result was obtained with the Q(0.2) with an area 6% smaller than that of the Bottom up. We note further that reducing α from 0.5 to 0.2 produces trees with smaller areas, but the gain is rather modest in general.

The effect of EL optimization on the total area of the tree also depends on the data set type. Data sets with large balls such as the Rabbits are the least benefited. For instance, whereas the Rabbit 128 tree built with the Bottom-up algorithm had its area improved by 4% (from 53 to 50.9), the tree for the Border data set built with Q(0.2) algorithm improved more than 61% (from 54.4 to 33.6).

D. Priority queue ranking functions

We have conducted all query tests with four variants of f_P, as discussed in Section IV-A. These variants correspond to using only the minimum distance, or using minimum distances as the main criterion, but breaking ties with the maximum distance. Minimum and maximum distances, in turn, can be estimated from the distance between the nodes themselves, or by the minimum among the distances between their children. Figure 6 shows a chart for Tree distance queries between a pair of Pangram trees built with the Bottom up algorithm, where the number of queue operations is shown as a function of the distance computed between the two trees for each of four f_P variants. Notice that using the children to estimate distances results in less queue operations being necessary overall. Moreover, using minimum distances as a primary key with maximum distances as a secondary key makes queries at zero distance require dramatically less operations. Since these findings are repeated for all trees and all queries, our discussion henceforth will only consider results computed with this last f_P variant.

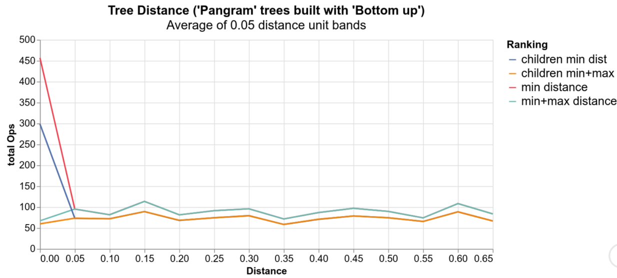


Fig. 6. Tree distance (*TD*) – totalOps by few *close* distances . Pangram ball trees built with the Bottom up algorithm.

E. Query performance

We focus next on the performance of the Tree Distance query for the various data sets and construction algorithms. Table IV show the average total number of priority queue operations required to answer that query for a pair of each data set. Again we observe that the trees built with the Bottom up algorithm behave best, followed closely by some of the other algorithms, notably Q. The worst of the lot seems to be the Online algorithm. We notice that decreasing the value of α for the Q algorithm does not improve the results in all cases, with Q(0.2) and Q(0.5) performing similarly.

TABLE IV
TD QUERIES AVERAGE NUMBER OF QUEUE OPERATIONS

	B up	KD	Online	Q(0.2)	Q(0.5)	V0	V1
500-C	**66.79**	218.36	70.48	69.34	68.10	72.47	72.69
500-R	**83.38**	102.71	84.45	88.67	84.76	87.04	103.33
Border	**149.17**	195.20	822.74	160.32	167.02	225.43	214.95
Pangram	**68.02**	93.26	170.46	68.80	72.97	91.00	81.08
Rabbit 128	**65.55**	78.22	138.18	77.90	68.35	66.10	82.68
Rabbit 256	**87.68**	118.35	208.16	98.94	102.95	101.25	134.95

As for the DtP queries (distance to point), the Online algorithm surprisingly fares rather well, besting the Bottom up algorithm for the 500-C and 500-R data sets, as indicated in Table V, which shows the average number of queue operations for all construction algorithm/data set combinations. Another unexpected result is the worsening observed for the trees obtained with the Q algorithm as α is reduced from 0.5 to 0.2. Since the difference is small, $\alpha = 0.5$ is clearly indicated in most cases, since it allows for construction costs significantly lower than $\alpha = 0.2$.

TABLE V
DTP QUERIES AVERAGE NUMBER OF QUEUE OPERATIONS

	B up	KD	Online	Q(0.2)	Q(0.5)	V0	V1
500-C	29.31	37.28	**29.07**	30.28	29.91	30.31	31.75
500-R	30.11	33.27	**29.44**	32.12	30.55	30.85	33.63
Border	**48.21**	52.13	67.13	48.93	48.56	51.37	50.45
Pangram	**33.07**	37.23	44.57	33.47	34.57	35.69	34.47
Rab. 128	32.98	**31.35**	36.30	35.42	32.28	31.65	32.12
Rab. 256	39.58	38.64	45.15	42.64	38.95	37.01	**35.22**

Experiments with Tree intersection (*TI*) queries for *close* distances reveal that the trees built with the Bottom up approach lead to the best results in almost all cases, except for the Border data set, where Q(0.2) yields a slightly better result. Either the KD construction or the Online approach yields the worst results in all data sets, requiring in some cases more than twice the number of operations. All construction algorithms proposed here fare relatively well. For instance, the trees built with Q(0.5) requires between 4% and 40% more operations than those built with the Bottom up approach.

Focusing now on experiments with point intersection queries (*PtI*). In Figure 7, we observe that for *close* distances the algorithms behave similarly, especially if look at the results obtained for the 5 best performing algorithms. For instance, for the Rabbit 256 tree, V1 requires only 31% more operations than the best result, which is obtained by the Bottom up tree. In the case of the 500-C data set, we observe that the result for KD construction is significantly worse than what is obtained by the other algorithms. This can be attributed to the fact that (1) our query sampling missed all balls in the data set, i.e., no point intersections were detected, and (2) KD is the only top down construction algorithm, which leads to many large balls in the top levels of the tree.

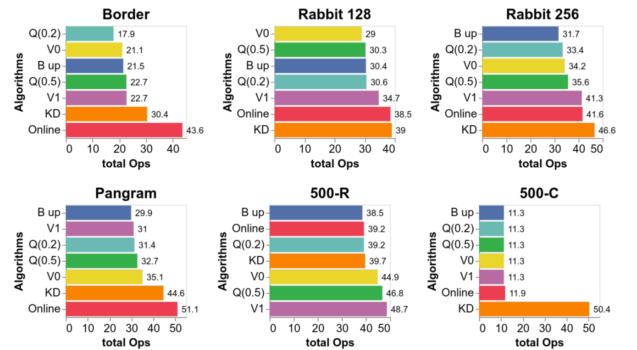


Fig. 7. Point intersection (*PtI*) - Ball tree construction algorithms by totalOps (filtered by *Close* distance).

VI. Conclusion

In this work, we examined the problem of building ball trees for 2D primitives, with the purpose of balancing construction time and efficiency of the trees with respect to a set of distance-based queries. Three new ball tree construction algorithms were presented (*Q*, *V0* and *V1*), in addition to the original implementation of three classic algorithms proposed by Omohundro [12] (*KD construction*, *Online* e *Bottom up*). Besides evaluating directly these algorithms by measuring construction time and total ball area, we also conducted a host of different experiments to gauge how well the constructed trees behave with respect to the proximity queries. All algorithms were extended with *EL optimization*, ensuring an improvement in query processing times. We use a *branch and bound* strategy in tree distance and tree intersection queries that employs a priority queue ranked by a function that combines minimum and maximum distances, leading to better overall results.

For most of the experiments performed, the ball trees built with *Bottom up* algorithm obtained the best performance, despite its slow build time. The results show that algorithms V0, V1 and Q yield query performances comparable with those obtained with the Bottom Up approach, but with a much lower construction time. The Q algorithm, in particular, yields uniformly good results, even beating the Bottom up approach in some cases when tuned with $\alpha = 0.2$.

As a future work, we intend to repeat this study for ball trees in three dimensions, also called sphere trees, which are abundantly employed for collision detection. Also, since algorithms *V0* and *V1* are quick and got reasonable results for some specific queries, we consider studying a way to improve their heuristics.

Acknowledgement

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] L. Källberg and T. Larsson, "Ray tracing using hierarchies of slab cut balls." in *Eurographics (Short Papers)*, 2010, pp. 69–72.

[2] A. Broutta, D. Coeurjolly, and I. Sivignon, "Hierarchical discrete medial axis for sphere-tree construction," in *International Workshop on Combinatorial Image Analysis*. Springer, 2009, pp. 56–67.

[3] D. Takeshita, "Aabb pruning: Pruning of neighborhood search for uniform grid using axis-aligned bounding box," *The Journal of the Society for Art and Science*, vol. 19, no. 1, pp. 1–8, 2020.

[4] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, pp. 6–es, 2007.

[5] S. Gottschalk, D. Manocha, and M. C. Lin, "Collision queries using oriented bounding boxes," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2000.

[6] X. Dong, P. Wang, P. Zhang, and L. Liu, "Probabilistic orientated object detection in automotive radar," *arXiv preprint arXiv:2004.05310*, 2020.

[7] M. Abu-Ata and F. F. Dragan, "Metric tree-like structures in real-world networks: an empirical study," *Networks*, vol. 67, no. 1, pp. 49–68, 2016.

[8] S.-g. Liu and Y.-w. Wei, "Fast nearest neighbor searching based on improved vp-tree," *Pattern Recognition Letters*, vol. 60, pp. 8–15, 2015.

[9] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-trees: Theory and Applications*. Springer Science & Business Media, 2010.

[10] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 47–57. [Online]. Available: https://doi.org/10.1145/602259.602266

[11] M. Dolatshah, A. Hadian, and B. Minaei-Bidgoli, "Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces," *arXiv preprint arXiv:1511.00628*, 2015.

[12] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[13] M. Kong and Y. Bai, "An efficient collision detection algorithm for the dual-robot coordination system," in *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. IEEE, 2018, pp. 1533–1537.

[14] H. A. Sulaiman and A. Bade, "Bounding volume hierarchies for collision detection," *Computer Graphics*, pp. 39–54, 2012.

[15] W. Wan and H. J. Lee, "Deep feature representation and ball-tree for face sketch recognition," *International Journal of System Assurance Engineering and Management*, pp. 1–6, 2019.

[16] C. Cheng, L. Xiaoli, W. Linfeng, L. Longxin, and W. Xiaofeng, "Algorithm for k-anonymity based on ball-tree and projection area density partition," in *2019 14th International Conference on Computer Science & Education (ICCSE)*. IEEE, 2019, pp. 972–975.

[17] T. Liu, A. W. Moore, A. Gray, and C. Cardie, "New algorithms for efficient high-dimensional nonparametric classification." *Journal of Machine Learning Research*, vol. 7, no. 6, 2006.

[18] F. Nielsen, P. Piro, and M. Barlaud, "Tailored bregman ball trees for effective nearest neighbors," in *Proceedings of the 25th European Workshop on Computational Geometry (EuroCG)*, 2009, pp. 29–32.

[19] J. Matoušek, M. Sharir, and E. Welzl, "A subexponential bound for linear programming," *Algorithmica*, vol. 16, no. 4-5, pp. 498–516, 1996.

[20] E. Welzl, "Smallest enclosing disks (balls and ellipsoids)," in *New results and new trends in computer science*. Springer, 1991, pp. 359–370.

[21] N. Megiddo, "Linear-time algorithms for linear programming in r^3 and related problems," *SIAM journal on computing*, vol. 12, no. 4, pp. 759–776, 1983.

[22] K. Fischer, "Smallest enclosing balls of balls," Ph.D. dissertation, Citeseer, 1975.

[23] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[24] D. Attali, J.-D. Boissonnat, and H. Edelsbrunner, "Stability and computation of medial axes-a state-of-the-art report," in *Mathematical foundations of scientific visualization, computer graphics, and massive data exploration*. Springer, 2009, pp. 109–125.

[25] N. Amenta, S. Choi, and R. K. Kolluri, "The power crust, unions of balls, and the medial axis transform," *Computational Geometry*, vol. 19, no. 2-3, pp. 127–153, 2001.

[26] T. K. Dey and W. Zhao, "Approximate medial axis as a voronoi subcomplex," in *Proceedings of the seventh ACM symposium on Solid modeling and applications*. ACM, 2002, pp. 356–366.

[27] R. Fabbri, L. D. F. Costa, J. C. Torelli, and O. M. Bruno, "2d euclidean distance transform algorithms: A comparative survey," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 2, 2008.

[28] T. Saito and J.-I. Toriwaki, "New algorithms for euclidean distance transformation of an n-dimensional digitized picture with applications," *Pattern recognition*, vol. 27, no. 11, pp. 1551–1565, 1994.

[29] F. Chazal and R. Soufflet, "Stability and finiteness properties of medial axis and skeleton," *Journal of Dynamical and Control Systems*, vol. 10, no. 2, pp. 149–170, 2004.

[30] J. K. Uhlmann, "Metric trees," *Applied Mathematics Letters*, vol. 4, no. 5, pp. 61–62, 1991.

[31] A. Moore, "The anchors hierarchy: Using the triangle inequality to survive high dimensional data," *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, 01 2013.

[32] F. Aurenhammer, "Power diagrams: properties, algorithms and applications," *SIAM Journal on Computing*, vol. 16, no. 1, pp. 78–96, 1987.

[33] M. L. Gavrilova, *Generalized voronoi diagram: a geometry-based approach to computational intelligence*. Springer, 2008, vol. 158.

[34] R. Weller and G. Zachmann, "Inner sphere trees and their application to collision detection," in *Virtual realities*. Springer, 2011, pp. 181–201.