Chapter 1

Approximate Geometric Query Structures

Specialized data structures are essential for answering geometric problems. In general one wishes to preprocess a data set so as to efficiently answer certain geometric problems quickly. Of considerable importance is identifying the nearest neighbor to a query point and counting or reporting the number of objects from the data set within a given region. The literature is full of theoretical results on many of these problems. Unfortunately, there are several drawbacks in the theoretical nature of many of these structures. In many cases, the optimal running time or storage requirements are extremely inefficient. For example, Chazelle and Welzl [7] showed that triangle range queries can be solved in $O(\sqrt{n}\log n)$ time using linear space but this holds only in the plane. In higher dimensions, the dependency goes up dramatically. In general, the time needed to perform an exact simple range query and still use small linear space is roughly $\Omega(n^{1-1/d})$, ignoring logarithmic factors [6]. For specialized queries like orthogonal queries, some improvement is given if space is sacrificed. Range trees solve orthogonal range queries in $O(\log^{d-1} n)$ time but use $O(n\log^{d-1} n)$ space [17]. Besides the space requirement, this structure only works well for orthogonal range queries.

As a result, many applied researchers have turned to more practically efficient, if not theoretically efficient, structures. A compromise between theoretical and practical algorithms is often obtained by trying to solve the problem approximately. In recent years several data structures have emerged that efficiently solve several geometric queries approximately. Arya *et al.* [1] introduced the idea of efficiently approximating nearest-neighbor queries in low-dimensional space. Their work developed a new structure known as the balanced box decomposition (BBD) tree. The BBD tree is a variant of the quadtree and octree [14] but is most closely related to the fair-split tree, see [5]. In [3], Arya and Mount extend the structure to the analysis of approximate range queries. Another structure capable of efficiently answering approximate geometric queries is the balanced aspect ratio tree [11, 12, 13], which is similar in nature to the *k*-d tree [16]. Extremely popular in practice *k*-d trees have long been known to exhibit excellent performance bounds in practice but little theoretical results had been known.

In [9, 11], Dickerson *et al.* show that one of the more common variants, the maximum-spread k-d tree, exhibits properties similar to BBD trees and BAR trees and present efficient bounds on approximate geometric queries for this variant. Unfortunately, the bounds are not as efficient as the BBD tree or BAR tree but are comparable. In the following chapter, we shall cover several standard approximate geometric queries and discuss BBD trees, BAR trees, and maximum-spread k-d trees.

1.1 General Terminology

In order to discuss approximate geometric queries and the efficient structures on them without confusion, we must cover a few fundamental terms. First, we distinguish between general points in $\mathbb{R}^d$ and points given as input to the structures.

For a given metric space $\mathbb{R}^d$, the coordinates of any point $p \in \mathbb{R}^d$ are $(p_1, p_2, \ldots, p_d)$. When necessary to avoid confusion, we refer to points given as input in a set S as *data points* and general points in $\mathbb{R}^d$ as *real points*. For two points $p, q \in \mathbb{R}^d$, the L_m *metric distance* between p and q is

$$\delta(p,q) = (\sum_{i=1}^{d} |p_i - q_i|^m)^{\frac{1}{m}}.$$

Although our analysis will concentrate on the Euclidean L_2 metric space, the data structures mentioned in this chapter work in all of the L_m metric spaces.

In addition, we use the standard notions of (convex) regions R, rectangular boxes, hyperplanes H, and hyperspheres B. For each of these objects we define two distance values. Let P and Q be any two regions in $\mathbb{R}^d$, the *minimum* and *maximum* metric distances between P and Q are

$$\delta(P,Q) = \min_{p \in P, q \in Q} \delta(p,q) \text{ and}$$

$$\Delta(P,Q) = \max_{p \in P, q \in Q} \delta(p,q) \text{ respectively.}$$

Notice that this definition holds even if one or both regions are simply points.

Let S be a finite data set $S \subset \mathbb{R}^d$. For a subset $S_1 \subseteq S$, the *size* of S_1, written $|S_1|$, is the number of distinct data points in S_1. More importantly, for any region $R \subset \mathbb{R}^d$, the size is $|R| = |R \cap S|$. That is, the *size* of a region identifies the number of data points in it. To refer to the physical size of a region, we define the outer radius as $O_R = \min_{R \subseteq B_r} r$, where B_r is defined as the hypersphere with radius r. The inner radius of a region is $I_R = \max_{B_r \subseteq R} r$. The outer radius, therefore, identifies the smallest ball that contains the region R whereas the inner radius identifies the largest ball contained in R.

In order to discuss balanced aspect ratio, we need to define the term.

Definition 1.1 *A convex region R in $\mathbb{R}^d$ has aspect ratio $\mathtt{asp}(R) = O_R/I_R$ with respect to some underlying metric. For a given balancing factor α, if $\mathtt{asp}(R) \leq \alpha$, R has* balanced aspect ratio *and is called an* α-balanced *region. Similarly, a collection of regions $\mathcal{R}$ has balanced aspect ratio for a given factor α if each region $R \in \mathcal{R}$ is an α-balanced region.*

For simplicity, when referring to rectangular boxes, we consider the aspect ratio as simply the ratio of the longest side to the shortest side. It is fairly easy to verify that the two definitions are equal within a constant factor. As is commonly used, we refer to regions as being either *fat* or *skinny* depending on whether their aspect ratios are balanced or not.

The class of structures that we discuss in this chapter are all derivatives of binary space partition (BSP) trees, see for example [18]. Each node u in a BSP tree T represents both a **region** R_u in space and the *data subset* $S_u \subseteq S$ of objects, points, lying inside R_u. For simplicity, regions are considered closed and points falling on the boundary of two regions can be in either of the two regions but not both. Each leaf node in T represents a region with a constant number of data objects, points, from S. Each internal node in T has an associated cut partitioning the region into two subregions, each a child node. The root of T is associated with some bounding (rectangular box) region containing S. In general, BSP trees can store any type of object, points, lines, solids, but in our case we focus on points. Typically, the partitioning cuts used are hyperplanes resulting in convex regions. However, the BBD tree presented in Section 1.4 is slightly different and can introduce regions with a single interior hole. Therefore, we have generalized slightly to accommodate this in our definition.

1.2 Approximate Queries

Before elaborating on the structures and search algorithms used to answer certain geometric queries, let us first introduce the basic definitions of approximate nearest-neighbor, farthest-neighbor, and range queries, see [1, 2, 3, 11, 13].

Definition 1.2 *Given a set S of points in $\mathbb{R}^d$, a query point $q \in \mathbb{R}^d$, a (connected) query region $Q \subset \mathbb{R}^d$, and $\varepsilon > 0$, we define the following queries (see Figure 1.1):*

- *A point $p^* \in S$ is a nearest neighbor of q if $\delta(p^*,q) \leq \delta(p,q)$ for all $p \in S$.*
- *A point $p^* \in S$ is a farthest neighbor of q if $\delta(p^*,q) \geq \delta(p,q)$ for all $p \in S$.*
- *A point $p \in S$ is a* $(1+\varepsilon)$-nearest neighbor of q *if $\delta(p,q) \leq (1+\varepsilon)\delta(p^*,q)$, where p^* is the true nearest neighbor of q.*
- *A point $p \in S$ is a* $(1-\varepsilon)$-farthest neighbor of q *if $\delta(p,q) \geq \delta(p^*,q) - \varepsilon O_S$, where p^* is the true farthest neighbor of q.*
- *An* ε-approximate range query *returns or counts a subset $S' \subseteq S$ such that $S \cap Q \subseteq S'$ and for every point $p \in S', \delta(p,Q) \leq \varepsilon O_Q$.*

To clarify further, a point p is a $(1+\varepsilon)$-approximate nearest neighbor if its distance is within a constant error factor of the true nearest distance. Although we do not discuss it here, we can extend the definitions to report a sequence of k $(1+\varepsilon)$-nearest (or $(1-\varepsilon)$-farthest) neighbors. One may also wonder why the approximate farthest neighbor is defined in absolute terms instead of relative terms as with the nearest version. By observing that the distance from any query point to its farthest neighbor is always at

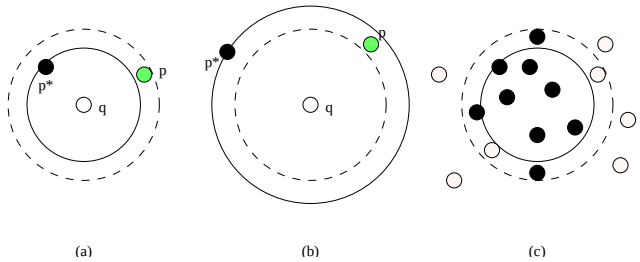


Figure 1.1: Examples of (a) an approximate nearest-neighbor query, (b) an approximate farthest-neighbor query, and (c) an approximate range query, here dark points are report/counted and lighter points are not.

least the radius of the point set O_S, one can see that the approximation is as good as the relative approximation. Moreover, if the query point is extremely far from the point set, then a relative approximation could return any point in S whereas an absolute approximation would require a much more accurate distance.

Although we do not modify our definition here, one can extend this notion and our later theorems to compensate for the problem of query points distant from the point set in nearest-neighbor queries as well. In other words, when the query point is relatively close to the entire data set, we can use the relative error bound and when it is relatively far away from the entire data set, we can use the absolute error bound.

The approximate range searching problem described above has one-sided false-positive errors. We do not miss any valid points, but we may introduce erroneous points near the range boundary. It is a simple modification of the query regions to instead get false-negative errors. That is, we could instead require that we do not include any points outside Q but allow missing some points inside Q if they are near the border. In fact, for any region Q one could define two epsilon ranges ε_i and ε_o for both interior and exterior error bounds and then treat the approximation factor as $\varepsilon = \varepsilon_i + \varepsilon_o$.

There are numerous approaches one may take to solving these problems. Arya *et al.* [1] introduced a priority search algorithm for visiting nodes in a partition tree to solve nearest-neighbor queries. Using their BBD tree structure, they were able to prove efficient query times. Duncan *et al.* [13] extended the priority search and analysis to solving farthest-neighbor queries. The nearest and farthest neighbor priority searching algorithms shown in Figures 1.2 and 1.3 come from [11]. In the approximate nearest-neighbor, respectively farthest-neighbor, search, nodes are visited in order of closest node, respectively farthest node. Nodes are extracted via an efficient priority queue, such as the Fibonacci heap [10, 15].

Introduced by [3], the search technique used for the approximate range query is a modification to the standard range searching algorithm for regular partition trees. We present the algorithm from [11] in Figure 1.4. In this algorithm, we have two different query regions, the inner region Q and the outer region $Q' \supseteq Q$. The goal is to return all points in S that lie inside Q, allowing some points to lie inside Q' but none outside of Q'. That is $Q' - Q$ defines a buffer zone that is the only place allowed for erroneous points. Whenever a node u is visited, if u is a leaf node, we simply check all of u's

```
ApproximateNearestNeighbor(T, q, ε)
    Arguments: BSP tree, T, query point q, and error factor ε
    Returns: A (1+ε)-nearest neighbor p
    Q ← root(T)
    p ← ∞
    do u ← Q.extractMin()
       if δ(u,q) > δ(p,q)/(1+ε)
          return p
       while u is not a leaf
          u1 ← leftChild(u)
          u2 ← rightChild(u)
          if δ(u1,q) ≤ δ(u2,q)
             Q.insert(δ(u2,q),u2)
             u ← u1
          else
             Q.insert(δ(u1,q),u1)
             u ← u2
       end while
       // u is now a leaf
       for all p' in dataSet(u)
          if δ(p',q) < δ(p,q)
             p ← p'
    repeat
```

Figure 1.2: The basic algorithm to perform nearest-neighbor priority searching.

associated data points. Otherwise, if R_u does not intersect Q, we know that none of its points can lie in Q and we therefore ignore u and its subtree. If R_u lies completely inside Q' then all of the data points in its subtree must lie inside Q', and we report all points. Otherwise, we repeat the process on u's two child nodes. For an ε-approximate range search, we define $Q' = \{p \in \mathbb{R}^d | \delta(p,Q) \leq \varepsilon O_Q\}$. We note that this search algorithm can also be modified to return the count or sum of the weights of the points inside the approximate range rather than explicitly reporting the points.

In all of these search algorithms, the essential criteria behind the running time is the observation that a non-terminating node in the search, one that requires expansion of its child nodes, is a node that must cross certain size boundaries. For example, in the approximate range searching algorithm, the only nodes expanded are those whose region lies partially inside Q, else it would be discarded, and partially outside Q', else it would be completely counted in the output size. A slightly more complex but similar argument applies for nearest and farthest neighbor algorithms. In the next section, we discuss a general theorem providing provable running time bounds for partition trees satisfying a fundamental packing argument.

1.3 Quasi-BAR bounds

We are now ready to examine closely a sufficient condition for a data structure to guarantee efficient performance on the aforementioned searches. Before we can proceed, we must first discuss a few more basic definitions presented in Dickerson *et al.* [9].

Definition 1.3 *For any region R, the* region annulus with radius *r, denoted $A_{R,r}$ is the*

```
ApproximateFarthestNeighbor(T, q, ε)
     Arguments: BSP tree, T, query point q, and error factor ε
     Returns: A (1−ε)-farthest neighbor p
     Q ← root(T)
     p ← q
     do u ← Q.extractMax()
        if Δ(u,q) ≤ δ(p,q)+εD
           return p
        while u is not a leaf
           u1 ← leftChild(u)
           u2 ← rightChild(u)
           if Δ(u1,q) ≥ Δ(u2,q)
              Q.insert(Δ(u2,q),u2)
              u ← u1
           else
              Q.insert(Δ(u1,q),u1)
              u ← u2
        end while
        // u is now a leaf
        for all p' in dataSet(u)
           if δ(p',q) > δ(p,q)
              p ← p'
     repeat
```

Figure 1.3: The basic algorithm to perform farthest-neighbor priority searching.

```
ApproximateRangeSearch(u, Q, Q')
     Arguments: Node u in a BSP tree, inner region Q, outer region Q'
           Initially, u ← root(T).
     Reports: All points in the approximate range defined by Q and Q'
     if u is a leaf node
        for all p in dataSet(u)
           if p ∈ Q
              output p
     else if Ru ⊆ Q'
           // The region lies completely inside Q'
           output all points p in the subtree of u
     else if Ru ∩ Q ≠ ∅
        // The region lies partially inside Q
        call ApproximateRangeSearch(leftChild(u), Q, Q')
        call ApproximateRangeSearch(rightChild(u), Q, Q')
```

Figure 1.4: The basic range search algorithm.

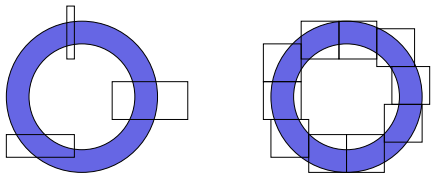


Figure 1.5: An example of a simple annulus region (a) with three other regions which pierce this annulus and (b) with several "fat" square regions. Observe that only a limited number of such "fat" squares can pierce the annulus.

set of all points $p \in \mathbb{R}^d$ *such that* $p \notin R$ *and* $\delta(p,R) < r$. *A region* R' pierces *an annulus* $A_{R,r}$ *if and only if there exist two points* $p,q \in R'$ *such that* $p \in R$ *and* $q \notin R \cup A_{R,r}$.

In other words, an annulus $A_{R,r}$ contains all points outside but near the region R. If R were a sphere of radius r', this would be the standard definition of an annulus with inner radius r' and outer radius $r'+r$. For convenience, when the region and radius of an annulus are understood, we use A. Figure 1.5 illustrates the basic idea of a spherical annulus with multiple piercing regions.

The core of the performance analysis for the searches lies in a critical packing argument. The packing lemmas work by bounding the number of disjoint regions that can pierce an annulus and hence simultaneously fit inside the annulus, see Figure 1.5b. When this packing size is small, the searches are efficient. Rather than cover each structure's search analysis separately, we use the following more generalized notion from Dickerson *et al*, [9].

Definition 1.4 *Given a BSP tree* T *and a region annulus* A, *let* $\mathcal{P}(A)$ *denote the largest set of disjoint nodes in* T *whose associated regions pierce* A. *A class of BSP trees is a* $\rho(n)$-quasi-BAR tree *if, for any tree* T *in the class constructed on a set* S *of* n *points in* $\mathbb{R}^d$ *and any region annulus* $A_{R,r}$, $|\mathcal{P}(A_{R,r})| \leq \rho(n)V_A/r^d$, *where* V_A *is the volume of* $A_{R,r}$. *The function* $\rho(n)$ *is called the* packing *function.*

Basically, the packing function $\rho(n)$ represents the maximum number of regions that can pierce any query annulus. By proving that a class of BSP trees is a $\rho(n)$-quasi-BAR tree, we can automatically inherit the following theorems proven in [1, 3, 13] and generalized in [9]:

Theorem 1.5 *Suppose we are given a* $\rho(n)$-*quasi-BAR tree* T *with depth* $D_T = \Omega(\log n)$ *constructed on a set* S *of* n *points in* $\mathbb{R}^d$. *For any query point* q, *the priority search algorithms in Figures 1.2 and 1.3 find respectively a* $(1+\varepsilon)$-*nearest and a* $(1-\varepsilon)$-*farthest neighbor to* q *in* $O(\varepsilon^{1-d}\rho(n)D_T)$ *time.*

Theorem 1.6 *Suppose we are given a* $\rho(n)$-*quasi-BAR tree* T *with depth* D_T *constructed on a set* S *of* n *points in* $\mathbb{R}^d$. *For any convex query region* Q, *the search algorithm in Figure 1.4 solves an* ε-*approximate range searching query in* T *in* $O(\varepsilon^{1-d}\rho(n)D_T)$

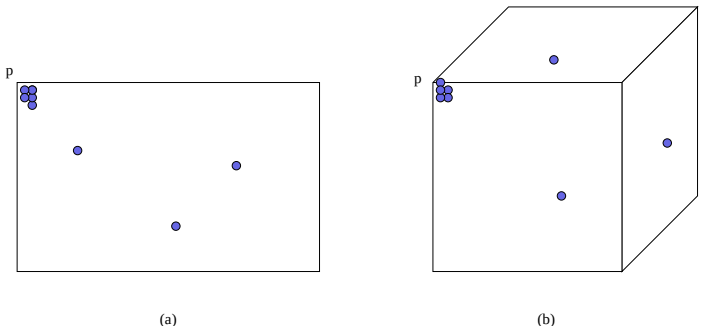


Figure 1.6: (a) A bad corner of a simple rectangular region with nearly all of the points clustered near a corner. Notice a cut in either the x or y direction dividing the points located inside p would cause a "skinny" region. (b) The same situation in $\mathbb{R}^3$.

time (plus output size in the reporting case). For any general non-convex query region Q, the time required is $O(\varepsilon^{-d}\rho(n)D_T)$ *(plus output size).*[1]

Trivially, $\rho(n)$ is always less than n but this accomplishes little in the means of getting good bounds. Having a class of trees with a good packing function helps guarantee good asymptotic performance in answering geometric queries. One approach to finding such classes is to require that all regions produced by the tree be fat. The idea behind this is that there is a limit to the number of disjoint fat regions that pierce an annulus dependent upon the aspect ratio of the regions and the thickness of the annulus. Unfortunately, guaranteeing fat regions is not readily possible using the standard BSP trees like *k*-d trees and octrees. Imagine building a *k*-d tree using only axis-parallel partitioning cuts. Figure 1.6 illustrates such an example. Here the majority of the points are concentrated at a particular corner of a rectangular region. Now, any axis-parallel cut is either too close to the opposing face or does not partition the points in the region well, resulting in large tree depth.

Fortunately, there are structures that can provably be shown to be $\rho(n)$-quasi-BAR trees for good values of $\rho(n)$. The next few sections discuss some of these structures.

1.4 BBD trees

Arya *et al.* [1] introduced the first BSP tree structure to guarantee both balanced aspect ratio and $O(\log n)$ depth. In addition, the aspect ratio achieved allowed them to prove an essential packing constraint. From this, once can verify that the BBD tree has a packing function of $\rho(n) = O(1)$ where the constant factor depends on the dimension d. In the following section, we describe the basic construction of the BBD tree using terminology from [3].

[1] Actually, BBD trees and BAR trees have a slightly better running time for these searches and we mention this in the respective sections.

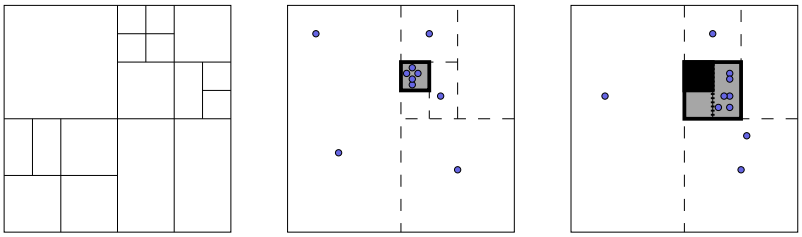
Every region R_u associated with a node u in a BBD tree is either an *outer* rectangular box or the set theoretic difference between an *outer* rectangular box and an *inner* rectangular box. The *size* of a box is the length of its longest side and the *size* of R_u is the size of the outer box. In order to guarantee balanced aspect ratio for these cells, Arya *et al.* [1] introduced a *stickiness* restriction on the inner box. Briefly described, an inner box is *sticky* if the distance between the inner box and every face on the outer box is either 0 or not less than the size of the inner box. Although not essential to the structure, we shall assume that the aspect ratio of the outer box is no more than two.

The construction of the BBD tree is done by a sequence of alternating splitting and shrinking operations. In the *(midpoint) split* operation, a region is bisected by a hyperplane cut orthogonal to one of the longest sides. This is essentially the standard type of cut used in a quadtree or octree. Its simplicity, speed of computation, and effectiveness are major reasons for preferring these operations.

The *shrink* operation partitions a region by a box lying inside the region, essentially creating an inner region. The shrink operation is actually part of a sequence of up to three operations called a *centroid shrink*. The centroid shrink attempts to partition the region into a small number of subregions R_i such that $|R_i| \leq 2|R_u|/3$.

When R_u is simply an outer box, with no inner box, a centroid operation is performed with one shrink operation. The inner partitioning box is found by conceptually applying midpoint split operations recursively on the subregion with the larger number of points. The process stops when the subregion contains no more than $2|R_u|/3$ points. The outer box of this subregion is the inner partitioning box for the shrink operation. The other merely conceptual midpoint splits are simply ignored. Choosing this inner box guarantees that both subregions produced by the split have no more than $2|R_u|/3$ points. This can be seen by observing that the inner box has no more than $2|R_u|/3$ points and also must contain at least $|R_u|/3$ points. The technique as stated is not theoretically ideal because the number of midpoint split operations computed cannot be bounded. Arya *et al.* [1, 3] describe a simple solution by repeatedly computing the smallest bounding midpoint box using a technique due to Clarkson [8].

When R_u has an inner box associated with it, we cannot simply find another inner box as this would violate the restriction on having only one inner box. Let b_i represent the original inner box. The solution is to proceed as in the previous centroid shrink operation, repeatedly applying midpoint split operations on the subregion with the larger number of points. However, we now stop in one of two situations; either the size of the larger subregion has no more than $2|R_u|/3$ points or the subregion no longer contains b_i. In the former case, let b be the outer box of this subregion. In the latter case, or in the event both cases happen, let b represent the outer box of the subregion prior to this final split. We perform a shrink operation using b as the inner box. Since b clearly contains b_i, the subregion associated with the original outer box continues to have one inner box, albeit a slightly larger one than its parent. The subregion R_1, whose outer box is b, also has one inner box, b_i. If $|R_1| > 2|R_u|/3$, we perform a midpoint split on this subregion. Let R_2 be the subregion formed by this last split that does not contain b_i. Since R_2 does not contain an inner box, if R_2 contains more than $2|R_u|/3$ points, we simply perform the previous shrink operation thus dividing R_2 into two smaller subregions as well. Clearly, all the subregions produced by this centroid shrink have less than $2|R_u|/3$ points. Figure 1.7 shows the three main operations, splitting, shrinking,



(a) (b) (c)

Figure 1.7: Examples of (a) multiple iterations of the midpoint split rule, (b) centroid shrinking, with dashed lines representing the conceptual midpoint splits and the highlighted inner box being the actual partition cut, (c) centroid shrinking with an inner box. In the final example, the original inner box is solid, the final midpoint split is shown with dotted lines and the new inner box partition cuts is shown shaded in gray.

and the three-step shrinking process.

In addition to this simple version of the BBD tree, Arya *et al.* [1, 3] present several more practical variations on this approach. The reader should refer to [1, 3] for details on an efficient $O(dn\log n)$ construction algorithm and for discussions on more practical variations on the above approach. To highlight a few options, at any stage in the construction, rather than alternate between shrinking and splitting operations, it is ideal to perform split operations whenever possible, so long as the point set is divided evenly after every few levels, and use the more costly shrinking operations only when necessary. Another approach is to use a more flexible split operation, a *fair split*, which attempts to partition the region more evenly. In this case, more care has to be taken to avoid producing skinny regions and to avoid violating the essential stickiness property; however, as was shown experimentally, the flexibility provides for better experimental performance.

The following theorem summarizes the result:

Theorem 1.7 *Given a set S of n data points in $\mathbb{R}^d$, in $O(dn\log n)$ time it is possible to construct a BBD tree such that*

1. *the tree has $O(n)$ nodes and depth $O(\log n)$,*
2. *the regions have outer boxes with balanced aspect ratio and inner boxes that are sticky to the outer box,*
3. *the sizes of the regions are halved after every $2d$ levels in the tree.*

The above conditions imply that the BBD tree is an $O(1)$-quasi-BAR tree.

The size reduction constraint above is essential in showing a slightly better performance for geometric queries than given for general quasi-BAR trees. In particular, Arya and Mount [3] show that the size reduction allows range queries on BBD trees to be solved in $O(2^d\log n + d(3\sqrt{d}/\varepsilon)^d)$ time, or $O(2^d\log n + d^3(3\sqrt{d}/\varepsilon)^{d-1})$ for

convex queries. Duncan [11] later extended the separation of the n and ε dependencies to nearest and farthest neighbor queries showing that the running time for both is $O(\log n + \varepsilon^{1-d}\log(1/\varepsilon))$ for fixed dimension d.

1.5 BAR trees

The balanced aspect ratio tree introduced in [12] for the basic two-dimensional case and subsequently revised to higher dimensions in [11, 13] can be shown to have a packing function of $\rho(n) = O(1)$ where the constant factor depends on the dimension d and a user-specified aspect ratio parameter α. In the following section, we borrow terminology from [11, 13].

Unlike BBD trees, k-d trees, and octrees, BAR trees do not exclusively use axis-orthogonal hyperplane cuts. Instead, to achieve simultaneously the goals of good aspect ratio, balanced depth, and convex regions, cuts in several different directions are used. These directions are called canonical cuts and the particular choice and size of canonical cuts is essential in creating good BAR trees.

Definition 1.8 *The following terms relate to specific cutting directions:*

- *A* canonical cut set, $\mathcal{C} = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_\gamma\}$, *is a collection of* γ *not necessarily independent vectors that span* $\mathbb{R}^d$ *(thus,* $\gamma \geq d$*).*
- *A* canonical cut direction *is any vector* $\vec{v}_i \in \mathcal{C}$.
- *A* canonical cut *is any hyperplane,* H*, in* $\mathbb{R}^d$ *with a normal in* $\mathcal{C}$.
- *A* canonical region *is any region formed by the intersection of a set of hyperspaces defined by canonical cuts, i.e., a convex polyhedron in* $\mathbb{R}^d$ *with every facet having a normal in* $\mathcal{C}$.

Figure 1.8a shows a region composed of three cut directions $(1,0)$, $(0,1)$, and $(1,-1)$, or simply cuts along the x, y, and $x-y$ directions. After cutting the region at the dashed line c, we have two regions R_1 and R_2. In R_2 notice the left side is replaced by the new cut c, and more importantly the diagonal cut is no longer tangential to R_2. The following definition describes this property more specifically.

Definition 1.9 *A canonical cut* c defines *a canonical region* R*, written* $c \in R$*, if and only if* c *is tangential to* R*. In other words,* c *intersects the border of* R*. For a canonical region* R*, any two parallel canonical cuts* $b, c \in R$ *are* opposing canonical cuts*. For any canonical region* R*, we define the* canonical bounding cuts *with respect to a direction* $\vec{v}_i \in \mathcal{C}$ *to be the two unique opposing canonical cuts normal to* $\vec{v}_i$ *and tangent to* R*. We often refer to these cuts as* b_i *and* c_i *or simply* b *and* c *when* i *is understood from the context. Intuitively,* R *is "sandwiched" between* b_i *and* c_i*. To avoid confusion, when referring to a canonical cut of a region* R*, we* always *mean a canonical bounding cut.*

For any canonical bounding cut, c*, the* facet *of* $c \in R$, $\texttt{facet}_c(R)$*, is defined as the region formed by the intersection of* R *with* c*.*

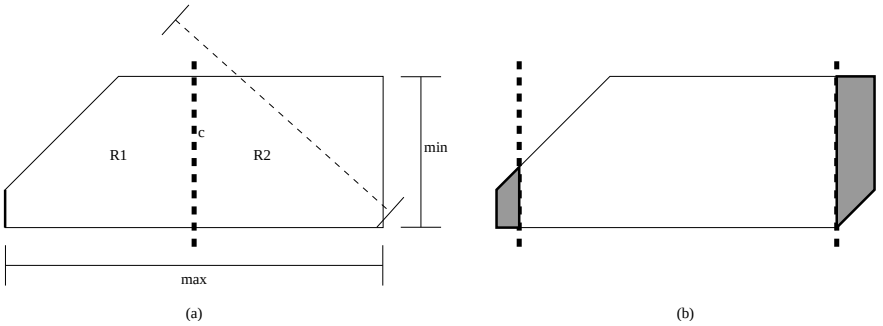


Figure 1.8: (a) An example of a canonical region of three cut directions, x, y, and $x-y$. Observe the three widths highlighted with lines and the min and max widths of the region. The left facet associated with the x direction is drawn in bold. The bold dashed line in the center represents a cut c and the respective subregions R_1 and R_2. (b) An example of a similar region highlighting the two shield regions associated with the x-direction for $\alpha \approx 4$. Notice the size difference between the two shield regions corresponding to the associated facet sizes.

The canonical set used to define a partition tree can vary from method to method. For example, the standard k-d tree algorithm [4] uses a canonical set composed of all axis-orthogonal directions.

Definition 1.10 *For a canonical set $\mathcal{C}$ and a canonical region R, we define the following terms (see also Figure 1.8):*

- *For a canonical direction $\vec{v}_i \in \mathcal{C}$, the* width of R in the direction $\vec{v}_i$, *written* $\texttt{width}_i(R)$, *is the distance between b_i and c_i, i.e.,* $\texttt{width}_i(R) = \delta(b_i, c_i)$.
- *The* maximum side of R *is* $\max(R) = \max_{i \in \mathcal{C}}(\texttt{width}_i(R))$.
- *The* minimum side of R *is* $\min(R) = \min_{i \in \mathcal{C}}(\texttt{width}_i(R))$.
- R *has* canonical aspect ratio, $\texttt{casp}(R) = \max(R)/\min(R)$.

For simplicity, we also refer to the facets of a region in the same manner. We define the following terms for a facet of a region R, $f = \texttt{facet}_c(R)$:

- *The* width of f in the direction $\vec{v}_i$ *is* $\texttt{width}_i(f) = \delta(b_i, c_i)$ *where b_i and c_i are the opposing bounding cuts of f in the direction $\vec{v}_i$.*
- *The* maximum side of f *is* $\max(f) = \max_{i \in \mathcal{C}}(\texttt{width}_i(R))$.
- *In addition, for any canonical cut $c \in R$, the* length *of c,* $\texttt{len}_c(R)$, *is defined as* $\max(\texttt{facet}_c(R))$.

When using a canonical cut c_i to partition a region R into two pieces R_1 and R_2 as the cut gets closer to a side of R, one of the two respective regions gets increasingly skinnier. At some point, the region is no longer α-balanced. This threshold region is referred to as shield region and is defined in [11] as the following:

Definition 1.11 *Given an α-balanced canonical region R and a canonical cut direction $\vec{v}_i$, sweep a cut c' from the opposing cut b_i toward c_i. Let P be the region of R between c' and c_i. Sweep c' until either region P is empty or just before $\texttt{casp}(P) > \alpha$. If P is not empty, then P has maximum aspect ratio. Call the region P the* shield region of c_i in R, $\texttt{shield}_{c_i}(R)$. *Let the* maximal outer shield, $\texttt{mos}_i(R)$, *be the shield region* $\texttt{shield}_{b_i}(R)$ *or* $\texttt{shield}_{c_i}(R)$ *such that* $|\texttt{mos}_i(R)| = \max(|\texttt{shield}_{b_i}(R)|, |\texttt{shield}_{c_i}(R)|)$, *i.e., the maximal outer shield is the shield region with the greater number of points. See Figure 1.8b.*

Definition 1.12 *An α-balanced canonical region, R, is* one-cuttable *with reduction factor β, where $1/2 \leq \beta < 1$, if there exists a cut $c^1 \in \mathcal{C}$, called a* one-cut, *dividing R into two subregions R_1 and R_2 such that the following conditions hold:*

1. *R_1 and R_2 are α-balanced canonical regions,*
2. *$|R_1| \leq \beta|R|$ and $|R_2| \leq \beta|R|$.*

Definition 1.13 *An α-balanced canonical region, R, is k-*cuttable *with reduction factor β, for $k > 1$, if there exists a cut $c^k \in \mathcal{C}$, called a k-*cut, *dividing R into two subregions R_1 and R_2 such that the following conditions hold:*

1. *R_1 and R_2 are α-balanced canonical regions,*
2. *$|R_2| \leq \beta|R|$,*
3. *Either $|R_1| \leq \beta|R|$ or R_1 is $(k-1)$-cuttable with reduction factor β.*

In other words, the sequence of cuts, $c^k, c^{k-1}, \ldots, c^1$, results in $k+1$ α-balanced canonical regions each containing no more than $\beta|R|$ points. If the reduction factor β is understood, we simply say R is k-cuttable.

Definition 1.14 *For a canonical cut set, $\mathcal{C}$, a binary space partition tree T constructed on a set S is a* BAR tree with maximum aspect ratio α *if every region $R \in T$ is α-balanced.*

Figure 1.9 illustrates an algorithm to construct a BAR tree from a sequence of k-cuttable regions.

Theorem 1.15 *For a canonical cut set, $\mathcal{C}$, if every possible α-balanced canonical region is k-cuttable with reduction factor β, then a BAR tree with maximum aspect ratio α can be constructed with depth $O(k \log_{1/\beta} n)$, for any set S of n points in $\mathbb{R}^d$.*

The main challenge in creating a specific instance of a BAR tree is in defining a canonical set $\mathcal{C}$ such that every possible α-balanced canonical region is k-cuttable with reduction factor β for reasonable choices of α, β, and k. The α-balanced regions produced help BAR trees have the following packing function.

Theorem 1.16 *For a canonical cut set, $\mathcal{C}$, if every possible α-balanced canonical region is k-cuttable with reduction factor β, then the class of BAR trees with maximum aspect ratio α has a packing function $\rho(n) = O(\alpha^d)$ where the hidden constant factor depends on the angles between the various cut directions. For fixed α, this is constant.*

```
CreateBarTree(u, S_u, R_u, α, β)
     Arguments: Current node u to build (initially the root),
           S_u is the current point set (initially S)
           R_u is the α-balanced region containing S_u
              (initially a bounding hypercube of S)
           (Optional) node u can contain any of the following:
              region R_u, sample point p ∈ S_u, size |S_u|
     if |S_u| ≤ leafSize then
        (leaf) node u stores the set S_u
        return
     find c^i, an i-cut for R_u, for smallest value of i
     (internal) node u stores c^i
     create two child nodes of u, v and w
     partition S_u into S_v and S_w by the cut s^i
     partition R_u into R_v and R_w by the cut s^i
     call CreateBarTree(v, S_v, R_v, α, β)
     call CreateBarTree(w, S_w, R_w, α, β)
```

Figure 1.9: General BAR tree construction algorithm.

Theorems 1.15 and 1.16 immediately show us that approximate geometric nearest-neighbor and farthest-neighbor queries can be solved in $O(\varepsilon^{1-d}\log n)$ time and approximate geometric range searches for convex and non-convex regions take $O(\varepsilon^{1-d}\log n + k)$ and $O(\varepsilon^{-d}\log n + k)$ respectively. As with the BBD tree, in fact, these structures can also be shown to have running times of $O(\log n + \varepsilon^{1-d}\log\frac{1}{\varepsilon})$ for nearest-neighbor and farthest-neighbor queries [11] and $O(\log n + \varepsilon^{1-d} + k)$ and $O(\log n + \varepsilon^{-d} + k)$ for convex and non-convex range queries [3].

Another examination of Figure 1.6 shows why simple axis-orthogonal cuts cannot guarantee *k*-cuttability. By concentrating a large number of points at an actual corner of the rectangular region, no sequence of axis-orthogonal cuts will divide the points and maintain balanced aspect ratio regions. We can further extend this notion of a bad corner to a general κ-corner associated with a canonical region R.

Definition 1.17 *For a canonical cut set $\mathcal{C}$ and a canonical region R, a* κ-corner $B \in R$ *is a ball with center* ρ *and radius* κ *such that, for every cut direction $\vec{v}_i \in \mathcal{C}$ with bounding cuts b_i and c_i, either b_i or c_i intersects B, i.e.* $\min(\delta(\rho, b_i), \delta(\rho, c_i)) \leq \kappa$.

When $\kappa = 0$, we are not only defining a vertex of a region but a vertex which is tangential to one of every cut direction's bounding planes. As described in [11], these corners represent the worst-case placement of points in the region. These corners can always exist in regions. However, if one of the facets associated with this corner has size proportional to the κ ball, then we can still get close enough to this facet and properly divide the point set without introducing unbalanced regions. The following property formalizes this notion more:

Property 1.18 *A canonical cut set $\mathcal{C}$ satisfies the* κ-Corner Property *if for any $\kappa \geq 0$ and any canonical region R containing a κ-corner $B \in R$, there exists a canonical cut $c \in R$ intersecting B such that* $\mathtt{len}_c(R) \leq \mathcal{F}_\kappa \kappa$ *for some constant $\mathcal{F}_\kappa$.*

```
ComputeTwoCut(u)
     Arguments: An α-balanced node u in a BAR tree
     Returns: A one or two-cut for u
     for all c_i ∈ C
        if c_i is a one-cut, return c_i
     let P be the smallest maximal outer shield of R
     let c = c_i be the bounding cut associated with P
     let c' be the cut parallel to c intersecting R such that
        δ(c,c') = width_i(P) + len_c(R)/σ
     return c'
     // c' partitions R into two α-balanced regions R_1 and R_2
     // |R_2| ≤ β|R|
     // R_1 incident to c_i is one-cuttable
```

Figure 1.10: An algorithm to find either a one or two cut in a region.

In particular, notice that if $\kappa = 0$, one of the bounding cutting planes must intersect at a single point. The advantage to this can be seen in the two-dimensional case. Construct any canonical region using any three cutting directions, for simplicity use the two axis-orthogonal cuts and one cut with slope $+1$. It is impossible to find a κ-corner without having at least one of the three bounding sides be small with respect to the corner. This small side has a matching very small shield region. Unfortunately, having a small shield region does not mean that the initial region is one-cuttable. The points may all still be concentrated within this small shield region. However, it is possible that this small shield region is one-cuttable. In fact, in [11], it is shown that there exist canonical cut sets that guarantee *two-cuttability* for sufficient values of α, β, and σ, where the σ parameter is used in the construction. The sufficiency requirements depend only on certain constant properties associated with the angles of the canonical cut set labeled here as $\mathcal{F}_{\min}, \mathcal{F}_{\max}, \mathcal{F}_{\text{box}}$, and $\mathcal{F}_{\kappa}$. For specific values of these constants, see [11]. Figure 1.10 describes an algorithm to find an appropriate cut.

Theorem 1.19 (Two-Cuttable Theorem) *Suppose we are given a canonical cut set, $\mathcal{C}$, which satisfies the κ-Corner Property 1.18. Any α-balanced canonical region R is two-cuttable if the following three conditions are met:*

$$\beta \geq (d+1)/(d+2), \tag{1.1}$$

$$\alpha\mathcal{F}_{\min}/4(\mathcal{F}_{\text{box}}+1) > \sigma > (2\mathcal{F}_{\max}+\mathcal{F}_{\kappa}), \text{ and} \tag{1.2}$$

$$\alpha > 4(\mathcal{F}_{\text{box}}+1)(2\mathcal{F}_{\max}+\mathcal{F}_{\kappa})/\mathcal{F}_{\min} + \mathcal{F}_{\max}/\mathcal{F}_{\min}. \tag{1.3}$$

Theorems 1.19 and 1.15 can be combined to yield the following theorem:

Theorem 1.20 *Suppose we are given a canonical cut set $\mathcal{C}$ that satisfies the κ-Corner Property and an $\alpha > f(\mathcal{C})$. A BAR tree with depth $O(d\log n)$ and balancing factor α can be constructed in $O(g(\mathcal{C})dn\log n)$ time, where f and g are constant functions depending on properties of the canonical set. In particular, the running time of the algorithm is $O(n\log n)$ for fixed dimensions and fixed canonical sets.*

Let us now present two cut sets that do satisfy the κ-Corner Property. The two cut sets we present below are composed of axis-orthogonal cuts and one other set of cuts. Let us give specific names to a few vector directions.

Definition 1.21 *A vector $v = (x_0, x_1, x_2, \ldots, x_d)$ is*

- *an* axis-orthogonal cut *if $x_i = 0$ for all values except one where $x_j = 1$, e.g. $(0,0,1,0)$,*
- *a* corner cut *if $x_i = \pm 1$ for all values of i, e.g. $(1,1,-1,-1)$,*
- *a* wedge cut *if $x_i = 0$ for all values except two where $x_j, x_i = \pm 1$, e.g. $(0,1,-1,0)$*

The Corner Cut Canonical Set *$\mathcal{C}_c$ is the set of all axis-orthogonal cuts and corner cuts. The* Wedge Cut Canonical Set *$\mathcal{C}_w$ is the set of all axis-orthogonal cuts and wedge cuts.*

Notice that $|\mathcal{C}_c|$ is $\Theta(2^d)$ and $|\mathcal{C}_w|$ is $\Theta(d^2)$. Although the corner cut canonical set does not necessarily have to be as large as this, the complexity of the corner cut itself means sidedness tests take longer than axis-orthogonal and wedge cuts, namely d computations instead of 1 or 2. The above two canonical sets satisfy the κ-Corner Property 1.18 and from Theorem 1.20, we get the following two corollaries [11]:

Corollary 1.22 *For the Corner Cut Canonical set $\mathcal{C}_c$, a BAR tree with depth $O(d \log n)$ and balancing factor $\alpha = \Omega(d^2)$ can be constructed in $O(n \log n)$ time.*

Corollary 1.23 *For the Wedge Cut Canonical set $\mathcal{C}_w$, a BAR tree with depth $O(d \log n)$ and balancing factor $\alpha = \Omega(\sqrt{d})$ can be constructed in $O(n \log n)$ time.*

To get the exact values needed, see [11]. However, it is important to note that the α bound above is a vast overestimate of the minimum value needed. In practice, one should try an initially small value of α, say 6, and when that fails to provide two-cuttability double the value for the lower subtree levels. In this manner, one can arrive at the true minimum value in $O(\log \alpha)$ such iterations, if necessary, without having to calculate it. Since the minimum α needed in both cut sets is $O(d^2)$, this adds only an $O(\log(d))$ factor to the depth.

1.6 Maximum-Spread k-d trees

One very popular class of BSP tree is the k-d tree, see Chapter **??**.[2] Although there are very few theoretical bounds known on these structures, there is a lot of empirical evidence that shows them to be extremely efficient for numerous geometric applications. In particular, one variant the maximum-spread k-d tree has long been considered an ideal k-d tree. Given a set of points S and a particular axis dimension x_d, define the *spread* of S in x_d to be the difference between the minimum and maximum coordinates of the points in that dimension. The maximum-spread k-d tree is formed by choosing at each internal node a cutting plane orthogonal to the axis of maximum

[2] **Is there a specific chapter reference for this in the Handbook?**

spread placed at the median point in this direction, see for example [16]. Arya *et al.* [1] applied the maximum-spread *k*-d tree to their approximate nearest-neighbor searching algorithm and experimentally showed that they were comparable to the theoretically efficient BBD tree. Later Dickerson *et al.* [9, 11] proved the following theorem regarding maximum-spread *k*-d trees, referred to there as longest-side *k*-d trees:

Theorem 1.24 *Suppose we are given a maximum-spread k-d tree T constructed on a set S of n points in $\mathbb{R}^d$. Then the packing function $\rho(n)$ of T for a region annulus A is $O(\log^{d-1} n)$. That is, the class of maximum-spread k-d trees is an $O(\log^{d-1} n)$-quasi-BAR tree.*

Although the bound is not as good as for BBD trees and BAR trees, the simplicity of the structure yields low constant factors and explains why in practice these trees perform so well. Experimental comparisons to BBD trees and BAR trees verified this result and showed that only for very highly clustered data did the dependency on $\log^{d-1} n$ become prominent [1, 11]. In practice, unless data is highly clustered and the dimension is moderately large, the maximal-spread *k*-d tree is an ideal structure to use. However, for such data sets both the BBD tree and the BAR tree revert to the same behavior as the maximal-spread tree, and they perform well even with highly clustered data. Because of its simpler structure, the BBD tree is most likely more practical than the BAR tree.

Bibliography

[1] Arya, Mount, Netanyahu, Silverman, and Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[2] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

[3] Sunil Arya and David M. Mount. Approximate range searching. *Comput. Geom.*, 17(3-4):135–152, 2000.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to *k*-nearest-neighbors and *n*-body potential fields. *J. ACM*, 42:67–90, 1995.

[6] Bernard Chazelle. Lower bounds on the complexity of polytope range searching. *J. Amer. Math. Soc.*, 2:637–666, 1989.

[7] Bernard Chazelle and Emo Welzl. Quasi-optimal range searching in spaces of finite VC-dimension. *Discrete Comput. Geom.*, 4:467–489, 1989.

[8] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 226–232, 1983.

[9] M. Dickerson, C. A. Duncan, and M. T. Goodrich. K-D trees are better when cut on the longest side. In *ESA: Annual European Symposium on Algorithms*, volume 1879 of *Lecture Notes Comput. Sci.*, pages 179–190, 2000.

[10] J. R. Driscoll, H. N. Gabow, R. Shrairaman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31:1343–1354, 1988.

[11] C. A. Duncan. *Balanced Aspect Ratio Trees*. Ph.D. thesis, Dept. of Computer Science, Johns Hopkins Univ., 1999.

[12] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees and their use for drawing very large graphs. *JGAA: Journal of Graph Algorithms and Applications*, 4:19–46, 2000.

[13] C. A. Duncan, M. T. Goodrich, and S. G. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. *ALGORITHMS: Journal of Algorithms*, 38:303–333, 2001.

[14] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Inform.*, 4:1–9, 1974.

[15] M. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization problems. *J. ACM*, 34:596–615, 1987.

[16] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[18] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.