Metrics and Ambits and Sprawls, Oh My
Another Tutorial on Metric Indexing

Magnus Lie Hetland

Norwegian University of Science and Technology, mlh@ntnu.no

Abstract. A follow-up to my previous tutorial on metric indexing, this paper walks through the classic structures, placing them all in the context of the recently proposed *sprawl of ambits* framework. The indexes are presented as configurations of a single, more general structure, all queried using the same search procedure.

Keywords: Metric indexing · Tutorial · Sprawls · Ambits

1 Introduction

About ten years ago, I wrote a tutorial on metric indexing [12], and last year, I finally finished a unifying framework for metric indexing and other comparison-based indexing [13]. That paper, however, is perhaps not the most inviting, containing quite a bit of detail and formalism, so in this paper, I'll revisit my earlier tutorial, in light of this new framework. This approach has two main benefits. First, the result should ideally be a streamlined, unified tutorial, rather than a smorgasbord of disjoint techniques; and, second, it provides an example-based introduction to the framework of sprawls and ambits, which might be useful to researchers who are already familiar with metric indexing. I focus primarily on "the classics"; for an overview of many variants, see, e.g., the recent paper by Chen et al. [6].

In contrast to the full paper introducing sprawls and ambits [13], I will try to keep this tutorial brief and to the point—more so, even, than my previous tutorial. In keeping with that, let's get going!

2 Framework

This section presents a thumbnail sketch of the framework used throughout. It may be easier to understand *after* you've read some of the example applications, so feel free to skim it, then skip ahead to section 3, returning here later.

We have a data set V drawn from some universe U with an associated *metric*, i.e., a symmetric function $\delta : \mathrm{U} \times \mathrm{U} \to \mathbb{R}_{\geqslant 0}$, where $\delta(u, v) = 0$ iff $u = v$, and

$$\delta(u, v) \leqslant \delta(u, w) + \delta(w, v)\,, \tag{2.1}$$

for all $u, v, w \in \mathrm{U}$. The problem we are trying to solve is storing V in some data structure, which we may later traverse to efficiently extract those points relevant to some query. Intuitively, we view this data structure as a bipartite digraph of points and *regions*, i.e., *sets* of points. This is referred to as a *sprawl* of regions.[1]

A region R with parents $p_1, \ldots, p_m$ is then *defined* in terms of these. That is, whether $u \in \mathrm{R}$ depends on the distances $x = [\delta(u, p_1), \ldots, \delta(u, p_m)]$, a vector in the so-called *pivot space* of $p_1, \ldots, p_m$. Specifically, we use a linear function $f(x)$ and a threshold, or *radius* r, so $u \in \mathrm{R}$ iff $f(x) \leqslant r$. Such a region is called a linear *ambit*.

The regions partition the space, representing a coarsening of the data. For a query in the form of a ball $\mathrm{Q} = \{u : \delta(q, u) \leqslant s\}$ of relevant points, we are only interested in the contents of a region R *if it intersects* Q. The idea, then, is to have the children of R point the way to smaller subsets of the data set. Search becomes a traversal of our graph, where each region is checked for overlap with the query before possibly traversing its children.

What is more, because a region is defined by its parents, we require *all* the parents to be traversed before traversing the region, and possibly its children. When we traverse a point u, we compute $\delta(q, u)$, so that when we traverse a region, we have all of $\delta(q, p_1), \ldots, \delta(q, p_m)$ available, giving us a distance vector z representing the query. If we assume, for now, that f is nondecreasing, Q and R intersect only if:[2]

$$f(z) \leqslant r + f(s) \tag{2.2}$$

For more advanced queries (kNN), and when we permit elimination, the order of traversal is significant. In these cases, we'd use a priority queue of nodes to traverse, updating their priorities each time we encounter them. In the basic scenario sketched out here, though, we might as well use a depth-first approach, as in the following mutually recursive procedures:

Simplified sprawl search algorithm

```
VISIT-POINT($u, q, s$)
1  if $\delta(q, u) \leqslant s$
2      print $u$
3  for $\mathrm{R} \in u.children$
4      $\mathrm{R}.count = \mathrm{R}.count + 1$
5      if $\mathrm{R}.count == |\mathrm{R}.parents|$
6          VISIT-REGION($\mathrm{R}, s$)
7  $u.color$ = BLACK

VISIT-REGION($\mathrm{R}, q, s$)
1  get $z$ from $\mathrm{R}.parents$
2  get $f$ and $r$ from R
3  if $f(z) > r + f(s)$
4      return
5  for $v \in \mathrm{R}.children$
6      if $v.color$ == WHITE
7          VISIT-POINT($v, q, s$)
```

In general, the idea is that δ is memoized in some way, so once $\delta(q, u)$ is computed on line 1 of Visit-Point, it is subsequently available when we gather up z in Visit-Region. Normally, one would have one or more designated root nodes, and call Visit-Point on them in turn to initiate the search.

The way this is set up, one would need to run a reinitialization in-between queries, resetting the memo, coloring nodes white and setting counts to zero.

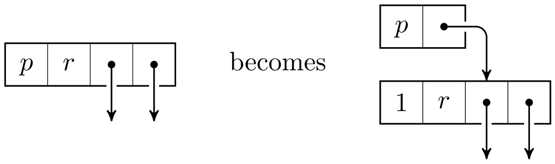
[1] Equivalently, a hyperdigraph on V, with one region per hyperedge [13, Rem. 2.3.12].

[2] Here $f(s)$ is shorthand for $f(s, \ldots, s)$.

There are many ways of handling this, of course. One could have actual attributes in the nodes, and maint a list of those that need resetting, requiring constant amortized time. An even simpler approach might be to simply use hash tables that are reset between searches. With some additional memory, one could even do the reset in actual constant time, using the standard trick for constant-time array initialization. In this case, one could keep a stack of nodes whose attributes are valid, and let each node keep its index in the stack. Then the reset would simply require setting the stack length to zero.

3 Ball Trees

A metric ball tree is a form of search tree where subtrees and their points are enclosed in balls. A subtree is then only explored if its ball intersects the query. For example, the simple BS-tree is a tree where each node is associated with a single point and a radius that covers the points below it in the tree [14]. The idea of a sprawl is for the graph (in this case, a tree) to express dependencies, where we have edges from points to the regions they tell us about, and from regions to the points they tell us to explore (*if* we intersect them). In the case of the BS-tree, then, each BS-tree node would be split into two sprawl nodes: one for the point, and one for the radius (i.e., region). For example:



Handling a BS-node then means first computing $\delta(q, p)$ and considering p for inclusion in the result, and then determining whether $\delta(q, p)$ is greater than $r + s$. If so, no further action is taken, as the query ball Q does not overlap the region (i.e., ball) R. Otherwise, the two child pointers are followed recursively.

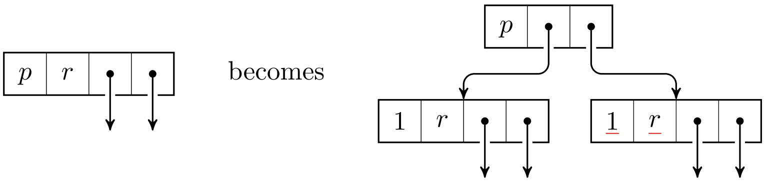
In the sprawl version, we've split out the point p as a parent node of the region. Initially, we visit this node, compute $z = \delta(q, p)$, and increment a counter associated with the child node. In general, we'll need to hang on to the z value as well as the counter; we could keep those in some separate memo, or perhaps store them in the nodes themselves. The counter is only useful if a region has multiple parents, so we know when we've visited them all; in this case, as soon as the counter goes from 0 to 1, we're done. Also, storing z is mostly useful if we're not going to use it immediately, and so it may be a bit wasted in this case.

Be that as it may, once the counter hits the threshold m (the number of parents of the region), we visit the region node. Here we store the radius r, but also one or more coefficients in a vector a. Note that m here is the number of entries in a, stored as part of the vector (or implicit, if the length is fixed). In this case, $m = 1$, $a = [1]$ and $f(x) = ax = x$. That means the overlap check reduces to that of the BS-tree:

$$f(z) \leqslant r + f(s) \iff az \leqslant r + as \iff z \leqslant r + s$$

There is no magic in the use of two children here; we may very well increase this number, as in the M-tree, for example [7]. (The M-tree adds another twist, which we'll return to in section 4.)

There's also the VP-tree [22,24] and its relatives such as LC [5], where there's a single ball that separates the inside from the outside. In that case, we get a different transformation:



The idea here is that the center point p is shared between the ball (left subtree) and its complement, the outside (right subtree). The only difference between the two region nodes is that the outside one has its coefficient and radius negated.[3] At this point, a slight revision is in order. We have previously assumed that f is nondecreasing, i.e., that $a \geqslant 0$. That is no longer the case! The more general version of the overlap check then uses $|a|s$, rather than as. What happens, then, is that the overlap criterion for the left subtree is still $z \leqslant r + s$, but for the right one, we get:

$$az \leqslant -r + |a|s \iff -z \leqslant -r + s \iff z \geqslant r - s$$

This is exactly as in the VP-tree, except that the surface of the ball is included both for the inside *and* the outside; we'd really like $z > r - s$. This is a detail not handled by the framework (though it easily could be amended to); however, it could only (presumably in rare cases) lead to false positives, i.e., exploring subtrees unnecessarily, which won't produce any wrong results. However, except for the goal of emulating the VP-tree, there is no need to use the same radius in both regions. One could use r_1 and $-r_2$, for example, and adapt each to cover only the points in each subtree.

4 Intersections

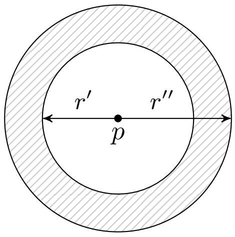
In section 3, our regions were individual balls and their complements.[4] We can combine these two kinds of regions to create *shell* regions, by turning a and r into column vectors:

$$a = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \qquad r = \begin{bmatrix} -r' \\ r'' \end{bmatrix}$$

[3] Here $\underline{x}$ is a space-saving shorthand for $-x$.

[4] Strictly speaking, the closure of their complements, as we don't use strict inequalities.

This gives a shell region around the single parent point p, as follows:



The membership check for a point u with distance $x = \delta(p, u)$ is still $ax \leqslant r$, but in this case, that means:

$$\begin{aligned} -x &\leqslant -r' \\ x &\leqslant r'' \end{aligned}$$

This is, of course, equivalent to $r' \leqslant x \leqslant r''$. For the overlap check, we take the absolute value for each row separately, so we still have $az \leqslant r + s$, which becomes (with some simplification):

$$\begin{aligned} z + s &\geqslant r' \\ z - s &\leqslant r'' \end{aligned}$$

That is, q must be so far away (z) that the s-ball around it reaches the inside radius (r') but not so far away that it ends up beyond the outside radius (r'').

A classic metric index—the Burkhard–Keller tree—branches out using multiple shells around a single center [3]. In this case, we'd simply use multiple shell regions, all with the same parent point.

There's not much point in using more than two rows when we have a single focus, i.e., a center, as we'll only end up with a single ball, inverted ball or shell, anyway. However, if we have more than one focus, we can add multiple *columns* to represent the intersection of multiple shells with *different* centers, yielding a coefficient matrix A. For simplicity, let's say we wish to represent the intersection of two balls, with respective centers p_1 and p_2. We use those points as the region's parents, and region membership becomes $\mathrm{A}x \leqslant r$, with coefficients and radii as follows:

$$\mathrm{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad r = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}$$

The intersection of multiple shells has been used in, e.g., Brin's GNAT [2] and its descendants, as well as the PM-tree family of structures [20] (see also section 5), and was later dubbed a *cut region* by Lokoč et al. [15].

The M-tree combines balls and shells in an interesting way. Before even computing $\delta(q, u)$ to perform the overlap check $\delta(q, u) \leqslant r + s$, it executes a preliminary filtering step, with the check

$$|\delta(q, p) - \delta(p, u)| \leqslant r + s\,,$$

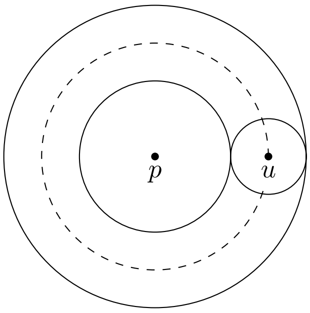
or, with our established notation, $|z - x| \leqslant r + s$. The intuition here is that $|z - x|$ is a lower bound for $d(q, u)$, a fact used in the standard pivot filtering check $|z - x| \leqslant s$ (see section 5). Here, however, it's plugged in as a lower bound in our *ball* overlap check (with u as our ball center), creating a weakened, preliminary version. This might seem like it requires introducing some new concept or indirection, but that is not so. The check is still linear and is equivalent to a standard shell region. This is easily seen by rewriting the check as follows:

$$\begin{aligned} -z + x &\leqslant r + s \\ z - x &\leqslant r + s \end{aligned}$$

We can rewrite this to match our previous shell overlap check:

$$\begin{aligned} z + s &\geqslant x - r \\ z - s &\leqslant x + r \end{aligned}$$

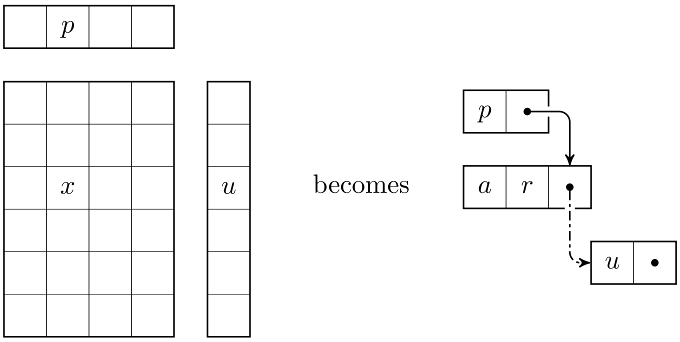
In other words, we here simply have a shell region with inner radius $x - r$ and outer radius $x + r$, corresponding to our knowledge of the r-ball around u before computing $\delta(q, u)$:



It would seem like we now have to store additional distances. Rather than just keeping x and r, we need to store r, $x - r$ and $x + r$. But is that really so? Given our M-tree to sprawl translation, each point node is now the center of multiple (quite possibly overlapping) shell regions, as well as a single ball region enclosing them all. The only reason to keep this ball region is if its radius is lower than the greatest radius of the shells. If we stuck rigidly to our translation, this could happen—but if we simply kept our shells as tight as possible around the subtrees, it could not. We then end up storing just two distances per subtree, once more, and have a structure with a behavior quite similar to, and no worse than, the M-tree.

5 Elimination

The most common purpose of a pointer in an index structure is to lead you toward further data to explore. There is a certain genre of structures, however, that do the exact opposite—where instead of discovering data, you *eliminate* it. Take, for example, the LAESA structure [17]: a table of distances between so-called *pivots* and the other points in the data set. The query is compared to each of these pivots, and the computed query–pivot distances, along with the stored pivot–data-point distances, are used to determine whether any given data point may possibly be relevant. In sprawl terms, each pivot–data-point distance represents a region:



In this case, the region is a *sphere*, a shell of width zero (i.e., with identical inner and outer radii):

$$a = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \qquad r = \begin{bmatrix} x \\ -x \end{bmatrix}$$

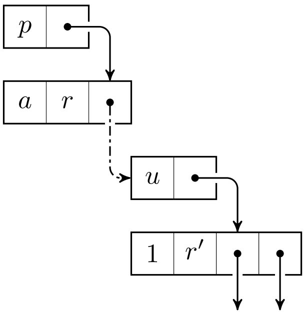
As before, our overlap check is $az \leqslant r + s$, or:

$$\begin{aligned} z &\leqslant x + s \\ -z &\leqslant -x + s \end{aligned}$$

Combined, this is the standard pivoting bound, $s \geqslant |x - z|$. Now, however, we get to the more interesting point. The dotted pointer indicates that we've got a potential *elimination* on our hands. That is, rather than saying "if there's overlap, let's look at u, otherwise, let's ignore it" we turn it around, and say "if there's overlap, let's ignore it, otherwise, let's eliminate it." In the terminology of our earlier pseudocode, that essentially means setting $u.\mathit{color}$ to BLACK. The exact implementation here could be done in several ways. One could have different region node types, for positive and negative regions (leading to discovery and elimination, respectively), or have separate lists of positive and negative child-edges, so the same region could both discover and eliminate points. These are possible optimizations, but they don't substantially change the behavior of the search.

It's possible to combine discovery and elimination, such as in the PM-tree. A simplified version would consist of a ball tree, such as in section 3, along with a set of pivots with eliminating regions around the subtrees. Specifically, the

PM-tree uses shells around each subtree, with global pivots, yielding something like the following:



An important thing to note about elimination is that it may be performed *lazily*. That is, we need not check for overlap with the various shell regions associated with the shared, global points in the PM-tree until we've established that we intersect the ball region in the tree itself. This kind of laziness could be implemented by having pointers in the reverse direction, without a need for counter updating. When considering a point, we would simply look at the region parents and see if they had been examined yet (i.e., if they were colored black).

It's possible to implement such things in different ways, of course; one could, for example, have some *parents* of a region be lazy, explored on demand, or the like. Having such a mechanism, one could simply use the global pivots of a PM-tree as lazy parents of every region in the tree, turning them from balls into cut regions, removing the need for elimination altogether.

The elimination perspective in LAESA could similarly be turned on its head, if instead of multiple regions, we use a single region for each point, with all pivots as its parents. This region would then be the intersection of all the spheres, and a point would simply not be discovered if there were no intersection.

This does not mean that we can do entirely without elimination, however. In any scenario where we at one time are able to traverse a point, and at a later time are not, this is the result of elimination. To my knowledge, the only current structure where it is truly needed, even if one were to introduce various forms of laziness optimization, is the AESA family of indices [23], where all points are available initially, and the set of candidates is gradually whittled down. The order of traversal then becomes crucial, as discussed in the next section.

6 Priority

The AESA family of indexing methods are all based on the same simple data structure: a complete distance matrix between the data points.[5] The points are

[5] Because of symmetry, one need only store half of it, of course.

explored one by one, and at every step we eliminate any of the remaining points we're able to. The elimination works just as in LAESA; the difference is that the pivots aren't kept separate from the objects. Rather than simply examining all the pivots in an arbitrary order, we now need to be quite careful about which object to examine next, to minimize the number of objects explored overall. A ubiquitous simplification here is to only focus on the elimination power of the next object and select the one that will give us the most bang for our buck.

We don't know which one that is, though. Rather, we must perform this choice *heuristically*, based on the information gathered so far, i.e., the distances from each examined point to the query and to the candidates for examination. If we represent these distances by vectors z and x, the original AESA used $\|z-x\|_1$ while a revised version used $\|z-x\|_\infty$, simply choosing the object whose filtering lower bound is the smallest, i.e., the one that's furthest away from being eliminated. Later, there was iAESA [11], which instead used Spearman's footrule between permutations of the previously examined objects, sorted by distance to the query and the candidate. Even more recently, Socorro et al. introduced the two-phase PiAESA method [21], which initially uses a set of preselected pivots (like LAESA), chosen for their general filtering power; once enough objects have been explored, it switches to the classic AESA behavior. Many variations are possible here, of course; for example, one might use regression or machine learning to estimate distances or filtering power or the like [10,18].

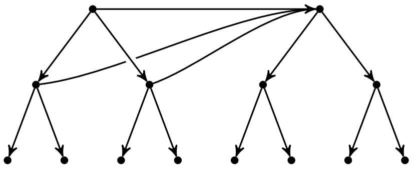
From a sprawl-of-ambits point of view, these methods are essentially the same: A complete directed graph of elimination edges, where each edge has a single sphere region. The priority or heuristic used to select the next available point is left unspecified. What *is* relevant, however, is when and how to compute or update the heuristic. In the simplest, most naive implementation, on might merely iterate over all available objects in each step, computing an arbitrary black-box priority for each, based on the knowledge gathered so far. It's possible, however, to let priority updates piggyback on other traversal operations.

For example, if the heuristic is based on how hard a point is to get rid of, one might update the priority every time the point is rediscovered and every time one fails to eliminate it. In each of these cases, a lower bound on the distance is computed, and one may then simply keep the sum or maximum, as in AESA.

For structures without elimination, such as the majority of search trees, priority is not relevant to the number of distance computations needed to resolve a range query; the behavior will be the same, regardless of the ordering. For kNN queries, however, priority can be crucial, as the covering radius of the result set tends to shrink as good candidates are found, and this will improve the chances of eliminating subtrees.

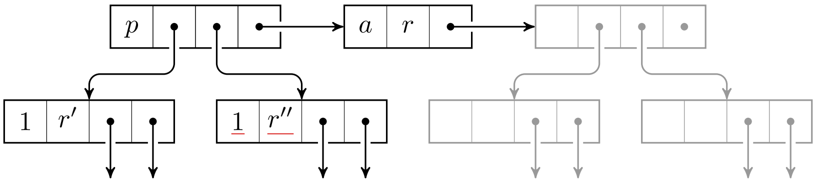
7 Non-trees

Index structures tend to be tree-shaped, more or less, especially if we ignore the eliminating parts. One early exception is the *excluded middle vantage point forest* introduced by Yianilos [25]. This structure is still *mostly* tree-shaped—or, as the name implies, forest-shaped. That is, it primarily consists of a collection of trees. However, these trees are connected to each other, making the whole thing a directed, acyclic graph.



The trees are essentially VP-trees with three regions rather than two: an inner ball, a middle shell, and an outer inverted ball. For queries whose radius is less than half the width of the middle shell, the search will never traverse more than one of the inner and outer subtrees—a major selling-point of the structure. There may still be points located in the separating shell, though, and these must also be indexed!

The idea is to gather up all the points that end up in any separating shell throughout the tree, and build a *new* tree from those, in the same manner (possibly leading to a third tree, and so on). We then simply make the root of this new tree the single child of every shell region in the first tree, as in the following, where $a = [-1\ 1]^t$ and $r = [-r'\ r'']^t$:



An essentially equivalent structure, at least from a bird's-eye view, is the D-index [9]. There, too, we have a multitude of shell regions separating inner and outer subsets, with the shells leading to a secondary structure, and so on. The main difference is that where the excluded middle vantage point forest uses tree traversal to determine which intersection of inner and outer regions a given point falls into, with the centers found along the path from the root, the D-index provides a fixed set of shared centers from the beginning, in a manner similar to the so-called fixed-queries tree [1]. Several levels are, in essence, collapsed, and the correct subtrees or leaves, representing the intersection of multiple shell or ball regions, are found directly, using hashing. This is an optimization that does not affect the high-level behavior (i.e., which points are examined).

8 Hyperplanes

In section 5, we created sphere and shell regions by having two radii, and thus two rows in our coefficient matrix, ending up with a column vector $[1\ -1]^t$. But we could also just use a row vector $a = [1\ -1]$, along with a single radius. This means we need two parents, or *foci*, p_1 and p_2, and we finally get a pivot vector $z = [z_1\ z_2]^t$. The overlap check becomes:

$$az \leqslant r + \|a\|_1 s$$

Here $\|a\|_1$ is the sum of absolute values. If we use $r = 0$, this corresponds to a metric half-space, separated by a midset or hyperplane. The overlap check then simplifies to the standard one [22]:

$$z_1 - z_2 \leqslant 2s$$

We are here defining the region of points closer to p_1 than p_2. If we wish to have *multiple* contrasting objects, modeling general Voronoi cells or Dirichlet domains [19], we can just add parent points, as well as some rows and columns. Let's say, for example, we wish to describe the region of points that are closer to p_1 than both p_2 and p_3. We'd then use all three as parents of our region, and use the following coefficients and radii:

$$\mathrm{A} = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \qquad r = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This corresponds to the following overlap check, where both inequalities must hold for there to be overlap:

$$\begin{aligned} z_1 - z_2 &\leqslant 2s \\ z_1 - z_3 &\leqslant 2s \end{aligned}$$

One may extend this to an arbitrary number of foci in the obvious manner.

9 Other Conics

The hyperplane case is easy enough to extend to (generalized) ellipses [22,8], by using coefficients $a = [1\ 1]$ and the appropriate radius, yielding the following overlap check:

$$az \leqslant r + \|a\|_1 s \iff z_1 + z_2 \leqslant r + 2s$$

Or we can get shifted half-spaces, what amounts to metric hyperbolas [8,16], by adjusting the radius away from 0. That is, we still have $a = [1\ -1]$ but we have $r \neq 0$, yielding the following slightly more general check:

$$z_1 - z_2 \leqslant r + 2s$$

This, then, represents not the points that are closer to z_1 than to z_2, but where the distances differ by a given value (i.e., the radius). That is, membership for a point with distance vector x is $ax \leqslant r$, i.e., $x_1 - x_2 \leqslant r$.

10 Other Queries

Nearest neighbor queries (kNN) have been mentioned briefly already. A general approach is to maintain the (up to) k points closest to q found so far, letting the search radius s be an upper bound on the distance to the kth nearest neighbor. Beyond updating s during the search, the procedure is the same.

A generalization that does not seem to have been explored is using other regions than balls as queries. After all, if a ball query works well in a tree built from hyperplanes, there's nothing stopping us from using a hyperplane query in a tree built from balls. That is, we might have a prototypical example object q, and a prototypical *counter*-example q', and we then search our index for objects closer to q than q'. (Such queries were briefly mentioned by Uhlmann [22].) Or maybe we have two prototypes, and wish to find the k objects with the lowest average distance to q and q', resulting in an ellipsoid query.

More generally, our query might consist of a *weighted combination* of query objects, looking for points with a low weighted sum of query distances. In other words, we may use an arbitrary linear ambit as our query [13, § 3.2.1]. As long as the ambit coefficients of the query, *or* those in the tree, are all non-negative, determining query–region overlap is straightforward, as we shall see.

11 . . . and Beyond

It ought to be quite clear that the sprawls and ambits used so far have been quite limited. The sprawls have mostly been tree-like, and the coefficients of the ambits have been 1 or -1, with at most two nonzero coefficients to a row. Countless variations are possible, both in how the sprawls are put together and in how the ambits are parameterized.

Determining whether an arbitrarily constructed sprawl is correct is a hard problem [13, Thm. 2.3.2]. However, extrapolating from existing index structures, we may quite easily ensure that the sprawls we construct are *responsible*, in which case they are guaranteed to be correct. Roughly, responsibility means that for every point p, there is a set of edges we can traverse that will lead us to it, and that the regions of those edges contain p, as do the regions of any negative edges that might disrupt that traversal. For the case where the positive edges of our structure are acyclic, this can be dealt with locally, where the responsibilities of a node's incoming edges depend only on those of the outgoing ones [13, Obs. 2.3.10]. Thus it ought to be possible to mix and match quite freely, perhaps even using heuristic search to look for efficient structures automatically.

As for regions, any coefficient matrix and radius vector yields a valid linear ambit, usable as a region or a query. For a query ambit Q with coefficient vector c and radius s, and a region ambit R with coefficient vector a and radius r, with a or c non-negative and $\|a\|_1, \|c\|_1 = 1$, if R and Q intersect, then

$$r + s \geqslant a\mathrm{Z}c^t \,, \tag{11.1}$$

where z_{ij} is the distance between focus p_i of R and focus q_j of Q [13, Thm 3.1.2]. With this overlap check, one can use ambit queries with existing index structures, and one could extend existing indexes with additional regions, without adding any distance computations. In an tree structure where several points are explored when deciding which subtrees to visit, arbitrary subsets of these could be used to construct additional filtering predicates for any subtrees, merely by adding radii and possibly coefficients.[6]

Finally, one may go beyond the limits of linearity. For example, using any (multi-parameter) non-decreasing *metric-preserving* function f to calculate remoteness, we may still use the original overlap check (2.2) [13, § 3.5]. This opens the door to a wide range of learning and optimization methods for adapting regions to points in ways that improve search performance.

References

1. Baeza-Yates, R., Cunto, W., Manber, U., Wu, S.: Proximity matching using fixed-queries trees. In: Annual Symposium on Combinatorial Pattern Matching. pp. 198–212. Springer (1994). doi:10.1007/3-540-58094-8_18
2. Brin, S.: Near neighbor search in large metric spaces. In: Proceedings of the 21th International Conference on Very Large Data Bases. pp. 574–584 (1995)
3. Burkhard, W.A., Keller, R.M.: Some approaches to best-match file searching. Communications of the ACM **16**(4), 230–236 (1973). doi:10.1145/362003.362025
4. Carlsen, S.M.Ø., Moe, H.H.: Similarity Search in Metric Spaces with Weighted Multi-Focal Regions: Using the Ambit Region Type to Improve the Performance of the SSS-Tree. Master's thesis, Norwegian University of Science and Technology (2020)
5. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters **26**(9), 1363–1376 (2005). doi:10.1016/j.patrec.2004.11.014
6. Chen, L., Gao, Y., Song, X., Li, Z., Miao, X., Jensen, C.S.: Indexing metric spaces for exact similarity search. arXiv preprint arXiv:2005.03468 (2020)
7. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An effcient access method for similarity search in metric spaces. In: Proceedings of the 23rd VLDB conference, Athens, Greece. pp. 426–435 (1997)
8. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: Separable splits of metric data sets. In: Proceedings of the Nono Convegno Nazionale Sistemi Evoluti per Basi di Dati (2001)
9. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: Distance searching index for metric data sets. Multimedia Tools and Applications **21**(1), 9–33 (2003). doi:10.1023/A:1025026030880
10. Edsberg, O., Hetland, M.L.: Indexing inexact proximity search with distance regression in pivot space. In: Proceedings of the Third International Conference on SImilarity Search and APplications. pp. 51–58 (2010). doi:10.1145/1862344.1862353
11. Figueroa, K., Chávez, E., Navarro, G., Paredes, R.: Speeding up spatial approximation search in metric spaces. Journal of Experimental Algorithmics (JEA) **14**, 3–6 (2010). doi:10.1145/1498698.1564506

[6] This approach has been tentatively explored by my students Carlsen and Moe [4].

12. Hetland, M.L.: The basic principles of metric indexing. In: Swarm intelligence for multi-objective problems in data mining, pp. 199–232. Springer (2009). doi: 10.1007/978-3-642-03625-5_9
13. Hetland, M.L.: Comparison-based indexing from first principles. arXiv preprint arXiv:1908.06318 (2019)
14. Kalantari, I., McDonald, G.: A data structure and an algorithm for the nearest point problem. IEEE Transactions on Software Engineering (5), 631–634 (1983). doi:10.1109/TSE.1983.235263
15. Lokoč, J., Moško, J., Čech, P., Skopal, T.: On indexing metric spaces using cut-regions. Information Systems **43**, 1–19 (2014). doi:10.1016/j.is.2014.01.007
16. Lokoč, J., Skopal, T.: On applications of parameterized hyperplane partitioning. In: Proceedings of the Third International Conference on Similarity Search and Applications. pp. 131–132. ACM (2010). doi:10.1145/1862344.1862370
17. Micó, M.L., Oncina, J.: A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. Pattern Recognition Letters **15**(1), 9–17 (1994). doi:10.1016/0167-8655(94)90095-7
18. Murakami, T., Takahashi, K., Serita, S., Fujii, Y.: Probabilistic enhancement of approximate indexing in metric spaces. Information Systems **38**(7), 1007–1018 (2013). doi:10.1016/j.is.2012.05.012
19. Navarro, G.: Searching in metric spaces by spatial approximation. The VLDB Journal **11**(1), 28–46 (2002). doi:10.1007/s007780200060
20. Skopal, T., Pokornỳ, J., Snasel, V.: PM-tree: Pivoting metric tree for similarity search in multimedia databases. In: ADBIS (Local Proceedings) (2004)
21. Socorro, R., Micó, L., Oncina, J.: A fast pivot-based indexing algorithm for metric spaces. Pattern Recognition Letters **32**(11), 1511–1516 (2011). doi:10.1016/j.patrec.2011.04.016
22. Uhlmann, J.K.: Metric trees. Applied Mathematics Letters **4**(5), 61–62 (1991). doi:10.1016/0893-9659(91)90146-M
23. Vidal Ruiz, E.: An algorithm for finding nearest neighbours in (approximately) constant average time. Pattern Recognition Letters **4**(3), 145–157 (1986). doi:10.1016/0167-8655(86)90013-9
24. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms. pp. 311–321. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1993)
25. Yianilos, P.N.: Excluded middle vantage point forests for nearest neighbor search. In: In DIMACS Implementation Challenge, ALENEX'99 (1999)