Geo-footprint indexing and similarity search

Achilleas Pappas

Diploma Thesis

Supervisor: Nikos Mamoulis

Ioannina, June, 2024

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Abstract

User trajectories in indoor spaces are an effective way of distinguishing common interests and similarities between users. These data can be used in many applications such as recommender systems and data mining tasks. In this thesis we aim to find and implement effective indexing schemes and search methods for these spatial data. We first extract each user's important regions using the concept of geo-footprints, where each footprint contains many Regions of Interest represented by minimum bounding rectangles (MBRs). These geo-footprints are then loaded into an R-tree using two different approaches: either by loading each Region separately or by assigning each leaf node entry to a specific user, using their geo-footprint as a MBR. We then implement three search methods according to each indexing scheme, namely Iterative (naïve approach), Batch and User-Centric, with the goal of answering top-K queries. Finally, we develop a trajectory generator with a focus on generating indoor user trajectories and we test each method using generated data to determine different weaknesses and strengths.

Keywords: geo-footprint, R-tree, MBR, spatial data, Regions of Interest, trajectory

Περίληψη

Οι διαδρομές ατόμων σε εσωτερικούς χώρους είναι ένας αποτελεσματικός τρόπος αναγνώρισης κοινών ενδιαφερόντων και ομοιοτήτων μεταξύ τους. Αυτά τα δεδομένα μπορούν να χρησιμοποιηθούν σε πολλές εφαρμογές, όπως σε συστήματα συστάσεων ή για σκοπούς εξόρυξης δεδομένων. Σε αυτή τη διπλωματική εργασία, στοχεύουμε να βρούμε και να υλοποιήσουμε αποτελεσματικά ευρετήρια και μεθόδους αναζήτησης για αυτά τα χωρικά δεδομένα. Αρχικά, εξάγουμε τις σημαντικές περιοχές κάθε χρήστη χρησιμοποιώντας την έννοια των γεωγραφικών αποτυπωμάτων, όπου κάθε αποτύπωμα περιέχει πολλές Περιοχές Ενδιαφέροντος που αντιπροσωπεύονται από ελάχιστα ορθογώνια περιγράμματα (MBRs). Αυτά τα γεωγραφικά αποτυπώματα στη συνέχεια φορτώνονται σε μια δομή R-tree χρησιμοποιώντας δύο διαφορετικές προσεγγίσεις: είτε φορτώνοντας κάθε περιοχή ξεχωριστά είτε αναθέτοντας κάθε εγγραφή του δέντρου σε έναν συγκεκριμένο χρήστη χρησιμοποιώντας το σχήμα του γεωγραφικού του αποτυπώματος. Στη συνέχεια, υλοποιούμε τρεις μεθόδους αναζήτησης σύμφωνα με κάθε ευρετήριο, δηλαδή την Iterative (επαναληπτική, απλή προσέγγιση), την Batch (ομαδική αναζήτηση) και την User Centric (αναζήτηση μεταξύ μεμονωμένων χρηστών), με στόχο να απαντήσουμε σε ερωτήματα εύρεσης κορυφαίων K. Τέλος, αναπτύσσουμε μια γεννήτρια διαδρομών με έμφαση στη δημιουργία διαδρομών χρηστών σε εσωτερικούς χώρους και δοκιμάζουμε κάθε μέθοδο χρησιμοποιώντας παραγόμενα δεδομένα για να προσδιορίσουμε διάφορα ελαττώματα και πλεονεκτήματα.

Λέξεις Κλειδιά: γεωγραφικό αποτύπωμα, R-tree, MBR, χωρικά δεδομένα, Περιοχές ενδιαφέροντος, διαδρομή

Table of contents

Chapter 1: Introduction

1.1 Motivation

Position tracking methods such as GPS have become an essential tool in everyday life by helping users navigate through maps, locate destinations, calculate distances and more [4]. This has led to a surge of technologies for indoor and outdoor position tracking [5], leading to more research on ways to discern patterns in crowd movement [6] and find similarities between individuals [7].

User trajectories are usually stored using points in 2D or 3D space and can prove to be challenging when it comes to indexing and similarity searching. That is because spatial data cannot be stored to preserve spatial proximity and indexing them in physical media is never optimal. The same goes for similarity searching, where comparing trajectories is a complex task in and of itself and proves to be especially time consuming when done between many trajectories of many users.

When examining indoor data, user trajectories tend to be more chaotic and random when focusing on small areas but provide important information when categorized in bigger areas of interest. By extracting larger areas from user paths, we can store information and calculate similarities seamlessly, without the need for complex calculations. The indexing is focused on capturing the locations of interest of each user (e.g. in a department store) and can later be used in conjunction with a traditional recommendation system to provide more accurate results.

The focus of this thesis is to implement and test different ways of indexing and answering of top-K queries between user trajectories tracked in indoor spaces. Each different indexing scheme implemented will prove to have its advantages and disadvantages depending on the data it is tested on. We aim to find and explain the optimal data for each method, and where possible provide optimizations in the indexing and searching algorithms.

1.2 Indoor Data

1.2.1 Position tracking

Indoor position tracking is a valuable tool, used in many different domains. Apart from recommendation systems, indoor tracking can be used for emergency services, healthcare applications, crowd behavior predictions, crisis management and more [8,9,10]. These applications have led to a rise in demand for indoor data [11] which has led to many new methods of position tracking.

Moreover, the use of GPS in internal spaces has proven to be unreliable, especially in locations with bad satellite coverage. [12] There have been attempts to mitigate the effects of GPS drifting (inaccurate tracking caused by bad signal) [13], but many researchers have opted to alternative ways of tracking indoor locations. [5]

A very common way of indoor tracking is using WiFi beacons [15]. This method works by utilizing the signals emitted by Wi-Fi access points or dedicated Wi-Fi beacons to determine the location of a device within a building. The main problem with this method is the installation of access points and the accuracy limitations posed by signal reflections and interference, therefore making it imperfect for some applications.

Another way of position tracking is using computer vision. [16] This method requires camera setup for image capturing and is followed by feature detection and object and path recognition using complex computer vision algorithms. Although very accurate, this method does not provide much scalability and the use of cameras in indoor spaces can cause privacy concerns in some scenarios.

Finally, some alternative methods of indoor tracking include phone inertial sensors [17], 3d range sensors [18], rf [19], Bluetooth [20] and more. Although there is research being done in all those methods, indoor position tracking remains an unsolved problem with no ideal solutions. This has led many researchers in indoor data management to develop Generators for indoor data. [21, 22] Generated data can be an effective tool when developing and testing indoor data management systems.

1.2.2 Data management and analytics

Following a recent survey in data management, analytics, and learning [23] we can observe the many different trends in spatial storage and analysis, especially when it comes to trajectories.

Concerning indexing, most systems use indexed grids, spatial filling curves or tree structures such as R-trees. The most widely used index is the R-Tree, which is a tree data structure that indexes multidimensional information by their Minimum Bounding Boxes (MBRs). It is scalable and very efficient for range and nearest neighbor queries, but can suffer from overlapping regions, leading to inefficiencies.

Another important task of a spatial data management system is analysis of the spatial relations. When it comes to trajectories, comparisons are performed using pointwise measures, which can be complex to perform and can be affected by factors such as tracking errors, sampling rates and point shifts (points shifting on the path of the trajectory). For that reason, more focus is put on placed on other robust operations that can be performed to extract information from trajectories, such as clustering [24, 25], trajectory routing [26], segmentation [27] and classification [2].

Due to the hard task of comparing trajectories, many researchers have opted for comparing dwell or stay regions [28, 29]. These are regions where points remain close to each other for a time, deeming them important areas of attention. This identification can either performed by clustering or other geometric algorithms. Comparing those regions instead of the trajectories themselves can be a new similarity measure that is much more efficient to calculate and can be very useful in areas where the exact trajectory paths do not hold much significance.

1.3 Related work

Our work involves extracting Regions of Interest from user trajectories, indexing them using a spatial index and performing top-K searches by applying certain similarity measures inspired from information retrieval. In this chapter we present similar approaches for indexing and similarity measures.

There has been much work on storage and indexing, which resolves in the form of trajectory search engines that manage and analyze indoor data [23, 30, 31, 32]. These engines, although able to perform similarity searches and clustering, lack the user centered approach that [1] follows by using geo-footprints to organize regions.

Similar to stay regions [28], Regions of Interest [1] defined later are rectangular and enclose points important to them. Using this approach, we can avoid the complexity of trajectory-to-trajectory comparison and focus on comparing simpler rectangular regions. This, paired with efficient extraction of those regions, can be a new and simple way of finding similarities between users.

Chapter 2: Geo-Footprints

In this chapter we discuss concepts and algorithms presented by [1], which are focused on extracting important spatial information from an indoor user trajectory, with a goal of determining rectangular regions in which they seem interested. These methods are key in devising efficient ways of storing and indexing our data, which is the focus of this thesis.

The first step is defining and extracting user Geo-Footprints and their Regions of Interest. The goal of these is to describe user trajectories in a more abstract way, aiming at effective storage for our users without losing important information about spatial interests and habits. The concept of these regions and their extraction is inspired by stay regions [28, 29] of moving objects. We will be extracting those regions using an algorithm focused on extracting the maximal regions for each sub-trajectory.

After that, we present different ways of similarity computation. We first present an efficient algorithm for computing the Euclidean norms for each footprint. Then, [1] suggests several ways of computing similarity between footprints, with the spatial join approach being the most efficient for our implementation (considering indexing and storage).

Finally, we aim to expand on the presented algorithms by adding time into the equation. Specifically, by using the length of each trajectory contained in a Region of Interest, we can add a weight to each Region relative to the time a user has spent on it. This way, we can improve on the accuracy of our similarities. This change will require subtle edits to the algorithms and equations presented previously, which we discuss and implement.

1.4 Basic concepts

In this section we present the basic definitions used in [1]. These structures are inspired by stay regions, which are regions extracted from trajectories of moving objects. In our case, extracted regions are rectangular and are the building blocks that compose a Geo-footprint. Following the mentioned paper, we explore three definitions:

1.4.1 User Trajectory

A User Trajectory is a sequence of locations that a user has visited. Each location is characterized by a spatial position, in our case a pair of x, y coordinates, and a timestamp t. In our implementation we use trajectories with regularly tracked positions, so the difference between timestamps is constant. This means that we can just use the x, y position for each step.

User trajectories are the input data we receive from an indoor movement of a user. They are usually large in size because they contain many unnecessary positions. This is due to the peculiarity of indoor movements since many users tend to stay stationary or make little to no movement for long periods of time. For this reason, we aim to extract the stay regions of these movements, in order to refine our data and keep only the important regions that the user is interested in. This leads us to our next definition, the Regions of Interest.

1.4.2 Regions of Interest

As mentioned, we are not interested in the entire trajectory of a user, but just the sub-trajectories where our user remains relatively stationary (i.e. when a customer wanders around a specific exhibition of items). The paper [1] defines those stay regions as Regions of Interest, specifically *a minimum bounding box that encloses a set of consecutive locations in a sub-trajectory*, defined by a spatial and a temporal constraint:

i) *epsilon*: for each pair of points l_i and l_j in the trajectory, their Euclidean distance must be less or equal to epsilon

ii) *tau*: each sub-trajectory must contain at least tau number of points

These two constraints are what qualify a sub-trajectory as a region of interest (RoI). The temporal constraint tau ensures that a region has enough steps to be considered as important, and the spatial constraint is used to restrict interesting regions from having too great of a size. Both constraints are important in order to avoid unnecessary information, e.g. when a user wanders around the majority of the space without a specific interest in some area.

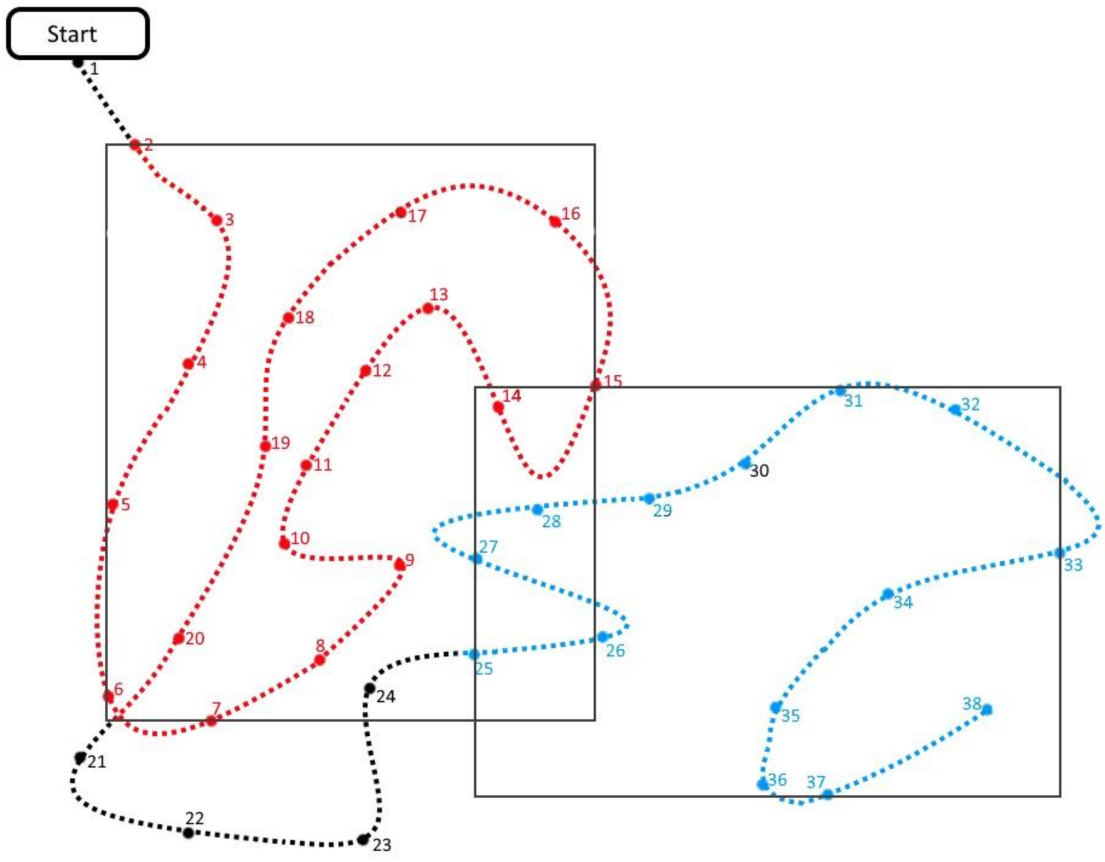


Figure 1: Extraction of two regions.

In the example of Figure 1 we observe the extraction of two regions. We observe that the first region stops at the 20th point, since the distance between point 16 and 21 is greater than *epsilon*. The same happens in the second region, where the 24th point is too far away from 33 to be concluded into the second region. Additionally, both regions must have at least *tau* points (in this case 10) and therefore points 21 to 24 do not form a region. Finally, it is important to note that a region is formed by sequential points, and that is why point 24 is not concluded in a region, although it is inside the first RoI.

Since RoIs are composed of sub-trajectories, they reference consecutive tracked points. This can cause many different possible regions to be formed for a specific sub-trajectory, if there are enough points close together. For this reason, when composing the RoIs for a user, we aim to extract temporally maximal regions. This leads us to our final definition of Geo-Footprints.

1.4.3 Geo-footprints

A user's Geo-Footprint is a set of Regions of Interest, extracted with the goal of keeping the least number of regions without compromising the interests of our user. Since many different RoIs can be extracted from a single sub-trajectory, the final Regions need to be temporally maximal and temporally disjoint, meaning that each region needs to be the largest it can be without breaking the spatiotemporal constraints of an RoI.

For this reason, [1] suggests an algorithm for RoI extraction which computes all Regions that follow the aforementioned conditions. Specifically, the algorithm is a greedy heuristic that, starting from the first *tau* regions of a trajectory, checks if they form a Region according to *epsilon*. If they do, it continuously adds points until the *epsilon* condition is not true, in which case a maximal region has been formed. It then continues from the next *tau* points and repeats this process. If the first [0, *tau*] points do not form a region, it continues to check with [*1, tau+1*] and so on until the epsilon condition is satisfied.

Algorithm 1 Regions of interest extraction

```
Require: trajectory $T$; bounds $\epsilon, \tau$
 1: $R \leftarrow$ null                                        ▷ Current region
 2: for each $l_i \in T$ do
 3:     if $R \cup l_i.p$ does not violate $\epsilon$ then
 4:         add $l_i$ to $R$
 5:     else
 6:         if $|R| \geq \tau$ then                            ▷ Current region has enough points
 7:             add $R$ to user profile
 8:             $R \leftarrow \{l_i\}$                         ▷ Initialize current region
 9:         else
10:             $newR \leftarrow \{l_i\}$                      ▷ Initialize new region
11:             while $newR \cup R.last$ does not violate $\epsilon$ do
12:                 $newR \cup R.last$; delete $R.last$;
13:             end while
14:             $R \leftarrow newR$                            ▷ Initialize current region
15:         end if
16:     end if
17: end for
18: if $|R| \geq \tau$ then                                    ▷ Last region has enough points
19:     add $R$ to user profile
20: end if
```

Algorithm 1: Regions of interest extraction algorithm, reference from [1]

Following the extraction algorithm (Algorithm 1), we start from the 1st point and add points until a region is formed. When arriving at point 6, the epsilon constraint is broken. Since there aren't enough points to form a region (assuming a tau of 10), we repeat the process starting from the 2nd point. This time, points are added without breaking the epsilon constraint until we reach point 21. Here, epsilon is broken, but since there are enough points to form a region, the first region is extracted. Continuing from the 21st point, we add points until a region is formed. Same as before, epsilon is broken before enough points are added, but this time, performing an optimization, the algorithm adds points in reverse order from 30, in an attempt to conclude as many points as possible to the next region without starting over from the 22nd point. The region is again not formed, and the process is continued until a region with at least tau points is formed, leading to the extraction of the second region.

This algorithm is run on all user trajectories, extracting their Regions of Interest. According to the dataset - its users and their movements - *tau* and *epsilon* need to be tuned before extraction in order to extract a balanced number of regions. We aim to avoid footprints with many small regions and footprints with few large regions. Finally, it is important to note that the algorithm is fast and scalable, as the paper notes.

1.5 Geo-Footprint Similarity

In this part, following the measures suggested by [1], we define the similarity between Geo-Footprints, using disjoint segmentations of RoIs. After that we present all suggested similarity algorithms, focusing on the approach which we end up using for all similarity computation in our later storage and indexing implementations.

1.5.1 Definition

Inspired by textual data, [1] represents a user's geo-footprint as a set of disjoint spatial regions and their frequencies. A footprint can be modeled as a set of rectangular regions that result from the overlaps between regions of interest in a footprint. If two or more regions overlap over a specific area, that area's frequency is equal to the number of regions that are over it.

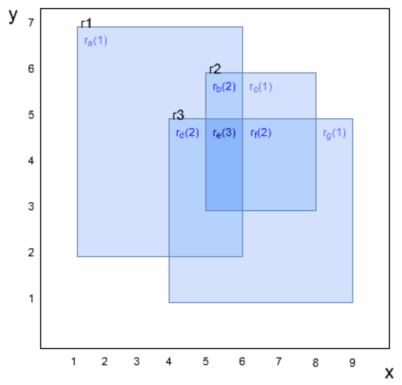


Figure 2: Formation of disjoint regions.

Figure 3 shows an example of 7 disjoint regions formed from 3 different RoIs. Each region has its own frequency depending on how many RoIs it associates with.

This representation is useful when comparing geo-footprints together because it allows the use of sweep line algorithms, which are algorithms that traverse space through a specific axis and make calculations during each stop.

Moreover, since each region is represented by a pair of (Region, frequency) we can define the similarity between two Geo-footprints using a measure like cosine similarity, inspired from Information Retrieval. The paper defines that the similarity between two footprints F(r) and F(s) as:

$$sim(F(r), F(s)) = \frac{\sum_{\substack{(X,fx)\in F(r), \\ (Y,fy)\in F(s)}} (|X \cap Y| \cdot fx \cdot fy)}{||F(r)|| \cdot ||F(s)||}$$

The numerator is the overlapping area between each of the disjoint regions of F(r) and F(s), multiplied by their frequencies. The denominator is the product of the Euclidean norms of our footprints, which is the maximum possible value of the numerator, if all regions of both footprints were totally overlapping. This means that, when two footprints are equal, both denominator and numerator are equal, resulting in similarity of 1. On the other hand, when two footprints are disjoint, the numerator becomes zero and the similarity is also zero.

Specifically, the Euclidean norm for a footprint F(r) is:

$$||F(r)|| = \sqrt{\sum_{(X,fx)\in F(r)} |X| \cdot fx^2}$$

Since the Euclidean norm is solely dependent on the footprint, it can be calculated before comparing each user, as a pre-computation step. For this reason, [1] suggests an algorithm for computing user norms, with the goal of storing all norms and using them in later steps.

1.5.2 Norm computation

Our goal is to compute the Euclidean norm of a geo-footprint. For this, the paper suggests a plane sweep algorithm, which firstly generates a set of disjoint regions and their frequencies, as described earlier. At the same time, it sums the contribution of each region, incrementally constructing the norm. The same process is repeated for each footprint of each user.

More specifically, the norm computation algorithm (Algorithm 2) uses an imaginary plane sweep line, which sweeps the plane along one axis, stopping at the starts and ends of each rectangle. At the same time, it keeps a data structure D which manages the disjoint regions along the other axis (non-sweep axis). D divides the line into intervals, according to the disjoint regions and their frequencies formed by the RoIs, so each entry of D is a pair of y-start and frequency. When the sweep line moves from one position to the next, the algorithm uses D to compute the areas of the disjoint regions, which it then multiplies with their frequencies and sums to calculate their contribution to the norm. At the same

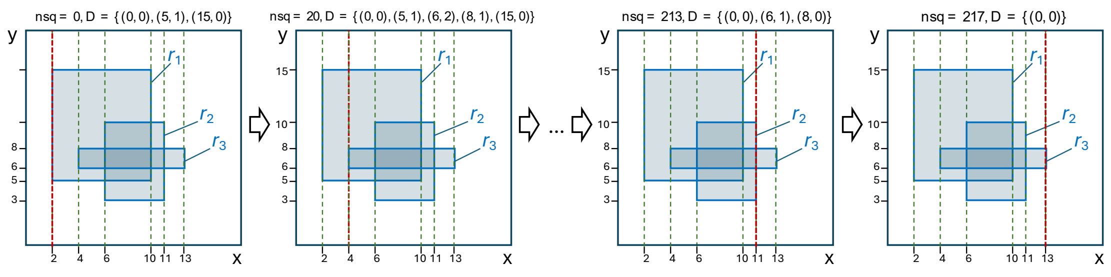

nsq = 0, D = {(0,0),(5,1),(15,0)}
nsq = 20, D = {(0,0),(5,1),(6,2),(8,1),(15,0)}
nsq = 213, D = {(0,0),(6,1),(8,0)}
nsq = 217, D = {(0,0)}
...
y
x
r_1
r_2
r_3
15
10
8
6
5
3
2
4
6
10
11
13

Figure 3: Norm computation snapshot example

time, at each stop D’s entries updated accordingly, either by adding entries if the stop is a start of a Region, or by removing entries if the stop is the end of a Region.

Figure 3 shows snapshots of the Norm computation algorithm. Initially D contains just one entry (0,0). At the first stop, D is updated with two additional entries: (5,1) representing the y interval [5,15] and (15,0) representing the [15, +∞] interval. Note that the only non-zero frequency is the one of the [5, 15] interval, representing the disjoint region found. At the second stop, a new rectangle is added with a y from 6 to 8, so the D structure is updated accordingly. Two new entries are added: (6,2) for the [6,8] y interval, and (8,1) for the [8,15] interval. This time, the (5,1) entry represents the [5,6] interval since a new disjoint region has been formed. At the same stop and before the new D entries are added, the nsq is updated to conclude the area found so far, specifically adding:

$\underbrace{(0*[0,5] + 1*[5,15] + 0*[15, +\infty])} * \underbrace{[2,4]} = 10*2=20.$

*y-intervals times their frequencies * x-interval*

Algorithm 2 Norm computation algorithm

```
Require: set of RoIs (footprint) $F(r)$;
Sort endpoints $\langle v, r_i.id, type \rangle$ of RoIs projections on the x-axis
$D \leftarrow [(0,0)]$; $ssq \leftarrow 0$; $prev \leftarrow$ min x-value;
for each $\langle v, r_i.id, type \rangle$ in $v$-order do
    for each entry $e$ in $D$ in $start$-order do                ▷ update norm square
        $ssq \leftarrow ssq + (e.next.start - e.start) \cdot (v - prev) \cdot e.count^2$
    end for
    if $type = Start$ then                                       ▷ add entries to $D$
        $e \leftarrow e \in D$ with largest $start$, such that $e.start \le r_i.y_{low}$
        $e \leftarrow (r_i.y_{low}, e.count + 1)$; $D.add(e)$
        while $e.next.start < r_i.y_{up}$ do
            $e.next.count \leftarrow e.next.count + 1$
            $e \leftarrow e.next$
        end while
        $D.add((r_i.y_{up}, e.count - 1))$;
    else                                  ▷ $type = End$: remove entries from $D$
        $e \leftarrow e \in D$ with $start = r_i.y_{low}$
        $D.remove(e)$
        while $e.next.start < r_i.y_{up}$ do
            $e.next.count \leftarrow e.next.count - 1$
            $e \leftarrow e.next$
        end while
        $D.remove(e.next)$                                       ▷ entry for $r_i.y_{up}$
    end if
    $prev = v$
end for
return $\sqrt{ssq}$
```

Algorithm 2: Norm computation algorithm, reference from [1]

The same process is followed, with the difference of removing entries when meeting the end of a region. After the final region is closed, the D structure contains only (0,0) and the result returned is the square root of nsq.

When implementing this algorithm, we chose to use a data structure that keeps D sorted and that can be sequentially accessed. For this reason, we use a binary search tree. The tree is constantly updated, with entries being added or removed at each stop of the sweep line. Additionally, at each stop, D needs to be scanned to update the sum of the norm. This leads to a time complexity of $O(n^2)$ and a space requirement of $O(n)$ for the computation of a footprint's norm with n RoIs.

This algorithm can be used before indexing our users by calculating and storing their norms alongside their footprints. This can be very efficient when working with a standard set of Geo-footprints, since norms need to be calculated only once. After the initial pre-computation step, only the query norms need to be calculated for each search, therefore greatly reducing the similarity search time (compared to having to calculate each user norm separately each time). On the other hand, this algorithm does not provide an efficient way of updating a footprint's norm in the case of addition or removal of new regions. Therefore, each user norm needs to be re-calculated in the case of an update, which can be a detriment when working with a live set of users.

1.5.3 Similarity algorithms

Following the calculation of the denominators for the similarity equation, we need to calculate the numerator for each pair of users in order to determine their similarity. More specifically, we need to calculate the sum of the overlapping disjoint regions multiplied by their frequencies. For this the paper suggests two algorithms, one being a version of the Norm computation algorithm that also calculates the overlaps between regions, and the other being a simple join-based similarity algorithm. We will explore why the second method is more suiting for our use case.

The first similarity algorithm presented is based on the norm computation algorithm. Given the user norms, the algorithm uses a sweep line to sweep between the regions of two different users. It also keeps two data structures (as opposed to just one used previously) with the disjoint regions and their frequencies along the non-sweep axis. The main difference is a routine that runs on each stop of the sweep line, which merges the contents of the two data structures, computing the contribution of the current stripe. After completing all stops, the result is divided by the product of the two given norms for our current users and returns their similarity.

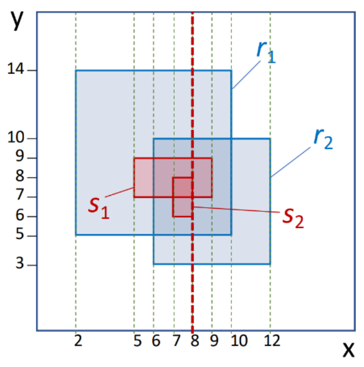


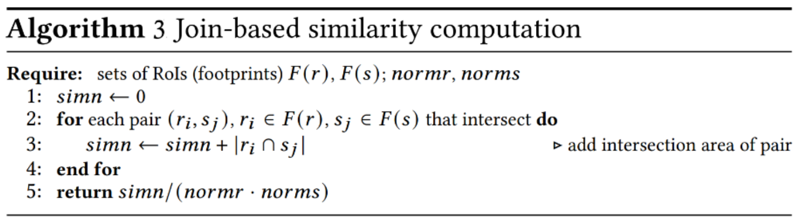
Figure 4: Snapshot of the Similarity computation algorithm.

In the snapshot of Figure 4 from an example explained in the original paper [1], we can see the computation of the similarities between two different users, R and S. At that point, D_r = {(0,0), (3,1), (5,2), (10,1), (14,0)} and D_s = {(0,0), (6,1), (7,2), (8,1), (9,0)}. The subroutine joins the two structures by scanning for the pairs that produce non-zero overlaps. Before updating the structures by removing the entries from that stop, it calculates the contributions of the overlaps and sums them to the total similarity overlap.

Additionally, it is worth mentioning that this algorithm can be modified to calculate the norms simultaneously, as the paper mentions. This means that this algorithm can be used regardless of whether the norms have been precomputed or not. This feature can be very useful when working with a live set of users which are constantly having their regions updated. That way we do not have to re-calculate the user norms each time since they are being calculated on demand.

That said, this algorithm has a complexity of $O((n+m)^2)$ for two users, with n and m Regions respectively. In a standard set of users, this algorithm performs many unnecessary comparisons to calculate their overlap. For this reason, there is a more intuitive approach, based on the concept of a spatial intersection join.

Inspired by [3, 33], the paper suggests an easier method for calculating the numerator of the equation using spatial joins, namely Algorithm 3. Specifically, when calculating the overlap between two footprints F(r) and F(s), each pair of RoIs r_i and s_j that intersect contribute to the numerator as much as the area of their intersection. This can save many additions compared to the previous algorithm, since each overlap's contribution is added to the numerator sum just one time. On the previous algorithm, each region was broken down to multiple disjoint regions, resulting in many comparisons and additions for a single pair of regions. This approach will prove to be much more effective, especially considering all the optimizations for spatial joins that can be applied [3].


Algorithm 3 Join-based similarity computation

```
Require: sets of RoIs (footprints) $F(r), F(s)$; $normr, norms$
1: $simn \leftarrow 0$
2: for each pair $(r_i, s_j), r_i \in F(r), s_j \in F(s)$ that intersect do
3:     $simn \leftarrow simn + |r_i \cap s_j|$          ▷ add intersection area of pair
4: end for
5: return $simn/(normr \cdot norms)$
```

Algorithm 3: Join-based similarity algorithm, reference from [1]

Following the example of Figure 4, we can calculate the overlapping areas using either approach and the results would be the same. Specifically, following the disjoint region approach, the area is calculated to be 18 (a more detailed calculation can be found in chapter 4 of [1]). Following the join-based similarity computation, we get the same area:

$$Overlap = |r1 \cap s1| + |r1 \cap s2| + |r2 \cap s1| + |r2 \cap s2| = 8 + 2 + 6 + 2 = 18$$

Considering that our main goal is to index all users into a database structure, the spatial join approach is more suited for our implementation. If we calculate and store all user norms as a pre-computation step, we can simply load all user rectangles into a spatial database and find similarities between users by searching for a query user and calculating the overlap between them and all stored users. This spatial join algorithm can be seamlessly applied to a spatial index, and in the next chapter, using an R-Tree, we explore how we can achieve this efficiently.

Complexity wise, this algorithm is expected to be more efficient than the previous sweep algorithm. A spatial join can be done at a cost of O(nlogn) + O(mlogm) + O(n+m+K) when searching for the overlapping pairs of regions between two users with n and m RoIs respectively (and K being the output size of the join). For each resulting pair of rectangles, the overlapping area can be calculated at a constant time, which results in a much lower complexity compared to the previous $O((n+m)^2)$.

1.6 Weighted approach

As we presented earlier, a region is determined as important if it contains enough points from the trajectory without surpassing a spatial limit. Although there is a temporal requirement for a region to be selected, there is no upper bound for the time that is spent in that region. This can lead to loss of information, since a user can spend either 5 minutes or 1 hour in a region, and still just get the same single RoI extracted in both cases.

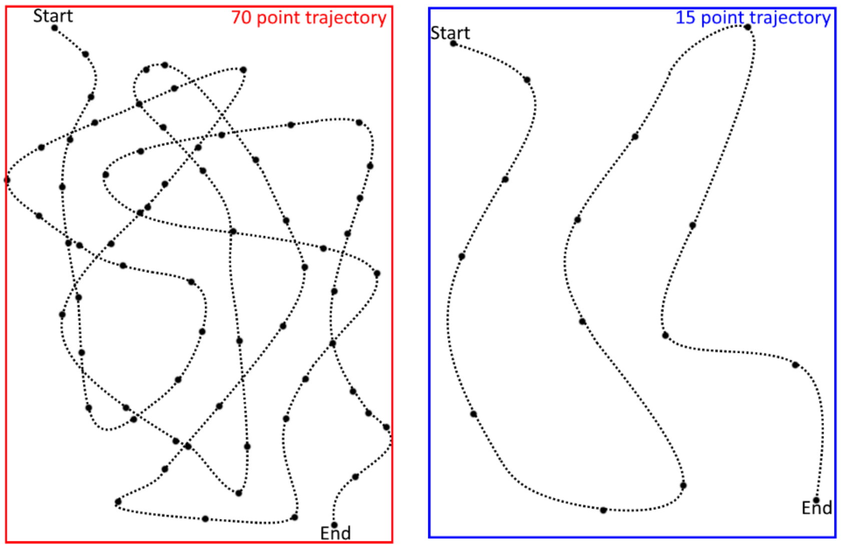


Figure 5: Two RoIs with similar size but different number of points

Observing Figure 5, we see two Regions of Interest that could get extracted with a tau of 10. After reaching the 10-point minimum, the extraction algorithm continues to add points to the same region since there is no upper limit to the points being added. This can result in a situation such as this, where the same exact Region extracted regardless of the amount of time spent. Those two users would be considered very similar, although the amount of interest expressed by the number of points is vastly different.

For this reason, it is our goal to extend and improve the similarity measures between two footprints by including this temporal information. Specifically, we aim to include a weight for each Region that corresponds to the points that are contained in it. During the extraction algorithm, after a region abides by the lower tau limit, we keep a tally of all points that belong in it, until the epsilon condition is met. We then return the extracted rectangle normally but include the number of points found divided by tau. The division is done to avoid larger numbers and to aid later calculations. In the end we are left with the same Regions of Interest extracted earlier, but with the additional information of the time spent in them, in the form of a weight.

Following the example of Figure 5 and using a tau of 10, the first user would have a weight of 7 and the second a weight of 1.5.

The next step is to see how the similarity equation and algorithm would be enhanced using this weight. Concerning the similarity equation, we observe that the frequencies can be replaced by the weights of each rectangle, and we would achieve the same result as previously if all weights were 1. By following this idea, we can replace the frequencies with the sum of the weights of all RoIs that are represented in that disjoint rectangle. These sums have a minimum value of 1 since each region has at least tau points and their final weight is divided by tau. In the following example we can see the difference between a normal and a weighted approach.

Following the example of Figure 6, we can calculate the similarities between users R and S with and without the weights to show the difference in their similarities. This example shows two footprints with substantial overlaps, but in all cases the overlaps are between areas with a big difference in weights: user S has most of their weight in regions B and C, which overlap with user R on a region with a weight of 1. Same goes for user R, which has most of their weight on regions A and B, which overlap with S on region A with a weight of 1 as well. This difference in weights indicates that these two users are not that similar, since their greatly weighted regions of interest do not overlap.

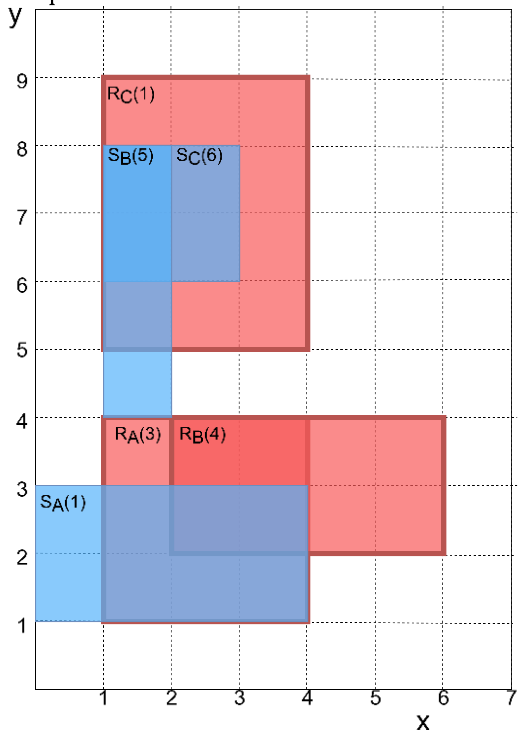


Figure 6: Weighted overlap example

Without weights, the similarity would be:

$$sim(F(r), F(s)) = \frac{15}{\sqrt{35}\sqrt{20}} = 0.567$$

But using weights, we find a similarity of

$$sim_w(F(r), F(s)) = \frac{65}{\sqrt{285}\sqrt{372}} = 0.199$$

Which points to the initial hypothesis that those two users were not that similar.

The updated similarity equation looks like this:

$$sim(F(r), F(s)) = \frac{\sum_{\substack{(X,fx)\in F(r),\\ (Y,fy)\in F(s)}} \left(|X \cap Y| \cdot \sum w_x \cdot \sum w_y\right)}{||F(r)|| \cdot ||F(s)||}$$

The same can be applied to the norm equation, and we observe that the resulting similarity is still a number between 0 and 1. The frequency is once again updated by the sum of the weights.

$$||F(r)|| = \sqrt{\sum_{(X,fx)\in F(r)} |X| \cdot \sum w_x^{\;2}}$$

The next step is to find how we can change the algorithms for norm calculation and similarity between users to calculate the weighted similarity accordingly. Looking at the norm computation algorithm (Algorithm 2), in the lines 9 and 11 we can see the frequency being added because of the addition of a new region. The same way in lines 14 and 19 the frequency is subtracted from the removal of a region. Instead of adding and subtracting just 1s, we can add and subtract the weights of the region being added or removed, resulting in the weights being used instead of the frequencies. For example, following a weighted version of the example in Figure 3 and assuming weights 1, 2 and 3 for regions r1, r2 and r3 accordingly, we get the following example.

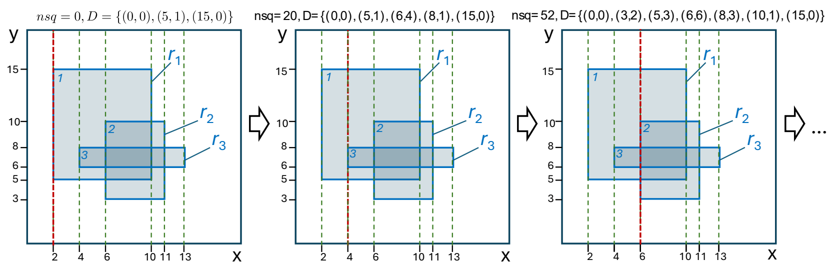

$nsq = 0, D = \{(0,0),(5,1),(15,0)\}$
nsq= 20, D= {(0,0), (5,1), (6,4), (8,1), (15,0)}
nsq= 52, D= {(0,0), (3,2), (5,3), (6,6), (8,3), (10,1), (15,0)}
y
x
r_1
r_2
r_3
1
2
3
15
10
8
6
5
3
2
4
6
10
11
13
...

Figure 7: Norm calculation snapshot with weights

Here we see the difference in the D structure, where instead of increasing by 1 each time a new region is added, we increase by that region's weight. This results in a different D structure and an increased nsq being calculated.

Following the same path as before, the intersection of two regions calculated in line 3 of the similarity algorithm (Algorithm 3) can be multiplied by the product of the weights of each region used. This results in the intersection as presented in the similarity equation. We will not be testing the first similarity algorithm presented in the paper since it is not as efficient, but the changes would be similar to the norm computation algorithm.

Finally, since implementing all those algorithms using an R-Tree, we will be storing each weight alongside its corresponding RoI. As we explain later, each Region is stored in the leaf nodes of our tree. We can simply change the items stored in each node to include a float which represents the weight of the Region. Since overlaps are calculated within the R-Tree structure, we can simply return the same overlap as before, multiplied by the weights accordingly.

After augmenting all aforementioned algorithms, we expect to see different top-K results compared to the previous approach. This, of course, depends on the variance in the amount of time each user spends in each region. If all users spend a similar amount of time in each region, the resulting weights will be similar on all of them, resulting in little to no change in the top-K queries when using the weighted approach. On a more realistic dataset, though, the time each user spends in a region can vary a lot, which can cause great variance in the weights, leading to different results in top-K queries. We will be testing this hypothesis in our final chapter.

Concerning complexity, the only difference is the addition of multiplications in each overlap calculation. This means that the time complexities remain the same, however we do expect a slight increase in total time due to the additional multiplications performed. It is also worth mentioning that the previous norm computation approach also performs multiplications internally, even without the usage of weights. This is done to accommodate for the addition and removal of nodes in the binary search tree, when dealing with multiple regions starting or ending in the same x-axis sweep stop. This results in little to no change in time when using the weights, regarding the norm computation.

Chapter 3: Indexing and Similarity Search

So far, we examined efficient ways of extracting the geo-footprints for each user. The next step is to find and implement efficient ways of indexing the Regions of Interest in order to answer similarity search queries. Specifically, we will focus on top-K queries: given a query user q the search objective is to find the K most similar users to q according to their geo-footprint similarity as defined previously.

We are going to firstly focus on indexing schemes, that is ways of storing the Regions of each user. Starting from the traditional R-tree, a data structure used for spatial data, we are following two different approaches: either storing each Region of Interest separately according to their spatial relation or organizing them by each user footprint so that each element of a leaf node represents a user.

After that, we will examine ways of answering the top-k queries, depending on each indexing scheme. We are going to augment the traditional R-tree search in order to return the overlapping areas for each query-user pair. That way we can more easily calculate the resulting similarities using the aforementioned pre-computed norms.

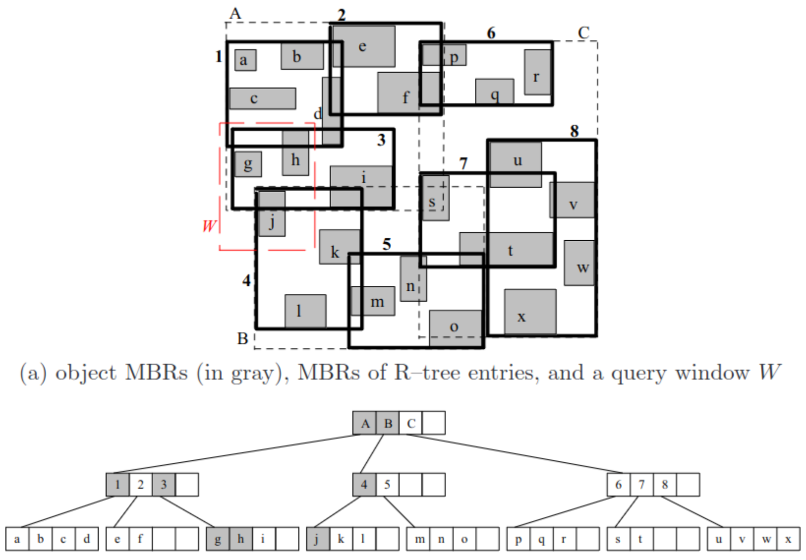
While calculating overlaps, we can use spatial join optimizations such as a plane sweep to improve on the speed of our search. In the end, we will have a baseline method called Iterative search, an improved method called Batch Search for the first tree and a different user-centric approach for the second tree. Of course, each method will have its strengths and weaknesses, which will be outlined along the way.

1.7 Indexing schemes

As seen in the previous section, each user is represented by a geo-footprint, a set of Minimum Bounding Rectangles representing the Regions of Interest of each user. The most common way of indexing MBRs is by using an R-tree.

1.7.1 Using an R-Tree

An R-tree is a popular access method for spatial data, implemented by many commercial database systems. It is a tree structure, where spatial data are stored according to their MBRs, with each node of the tree correlating to a disk block. Each leaf node contains many items that are close together in space, represented by a MBR and a pointer to them. Each non-leaf node is represented by a MBR of all the sub-nodes it points to, and each node apart from the root must be at least 40% full so that the disk space is properly utilized.



(a) object MBRs (in gray), MBRs of R–tree entries, and a query window W

(b) the R–tree; nodes touched by query W are in Gray color

Figure 8: Example of an R-Tree from [34]

When it comes to using an R-Tree for our data, some modifications need to be made in order to achieve efficient storage and indexing. First, since the data stored are the rectangles themselves, there is no need for the leaf node pointers in the R-Tree. Since the tree checks for the overlapping rectangles while searching for each query, we can tally the discovered overlapping areas and return a list of them at the end of each search, instead of returning the entries themselves like the R-tree normally does.

Moreover, each of the Regions that need to be stored will have the user's ID for identification. Since each user may have many Regions of Interest in their footprint, different stored MBRs may have the same ID, which is contrary to a classic R-tree that has unique IDs for each entry. That particularity will not be a problem since the search methods we use later will scan for the entire leaf node each time regardless of the node's ID.

We will be using nushoin's implementation of the R-tree in C++ [35] which is an expanded version of Antonin Gutman's R-tree from [36]. Most modifications will be done later to accommodate the different search methods. For the base R-tree, loading the MBRs and searching them using Rectangle queries can be done without any changes. The final tree will contain all Regions from all users, each identified by the user's ID.

The main disadvantage of this initial scheme is evident when searching for specific users and their Regions. Specifically, since the R-Tree organizes all Regions relative to their location in space regardless of the user they are assigned to, a user may have regions located in many different locations in space. When trying to find a user's footprint, a search of the entire tree is necessary in order to guarantee all of that user's regions have been accounted for. That search can be very costly depending on the tree size. This disadvantage, however, will not be a detriment in our case since we will be focusing on top-K queries, which do not require searching for a specific user in the tree.

Moreover, in the case of top-K queries, this indexing scheme can be very efficient. These types of queries require searching for a set of RoIs according to a given query geo-footprint and the calculation of overlapping areas between each given RoI and each stored RoI of the structure. Either iteratively or in batches, these Regions can be easily searched through the R-tree due to its tree structure, and each overlap found can be traced back to a specific user using the ID of that node. Users with sparsely located Regions in their footprints do not have to be fully searched, since the Region's location in the structure is irrelevant to the user it belongs to.

As far as updating the structure, while insertions and updates can be seamless, removal of users is a more complex process. Specifically, insertions of new Regions and new users can be done by inserting their corresponding MBRs in the structure, considering that the only factor connecting regions to their users is their ID, which does not have to be unique for each Region. Removals, on the other hand, require the search of a specific Region and if trying to remove a user, the whole R-Tree needs to be searched in order to ensure that no regions of that user are left in the structure.

1.7.2 Implementing a User-centric R-Tree

Another way of indexing user geo-footprints is by using the total MBR of a footprint, implementing what we denote as a User-centric R-tree. More specifically, each geo-footprint, being a collection of MBRs, has its own MBR that covers all of its Regions. Using this, we can store the MBR of a geo-footprint as a single entry in the R-tree instead of storing each region independently. Each user can have a pointer to all of their regions and the search method can calculate overlaps with the query rectangles using an efficient spatial join algorithm like a plane-sweep.

A user centric tree centers around faster searches through the structure because of its decreased size. This is because while using this scheme, each user will amount to a single footprint-Region in the R-Tree instead of multiple regions per user. This in turn leads to much quicker searches because of the tree's reduced height, which is the main factor that affects R-tree search time.

A disadvantage of this method is visible when working with users with sparsely placed Regions (e.g. users that traverse a large shopping center) like the users shown in Figure 9. Here, we observe two different cases of footprint overlaps, indicating that there is no direct correlation between a footprint's MBR and its regions. Specifically, in both cases the user centric approach will need to iterate over all pairs of regions from both users while checking many regions that would otherwise be skipped by iterative or batch. This was not a problem with the previous method because a user's spatial extent was irrelevant to the tree structure, and each Region was stored regardless of the user it belonged.

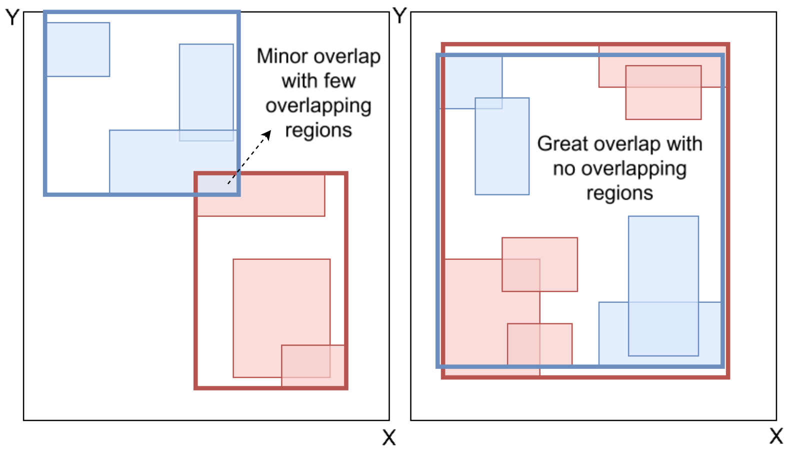

Y
Minor overlap with few overlapping regions
X
Y
Great overlap with no overlapping regions
X

Figure 9: Examples of edge cases of overlaps between footprints.

Concerning updates and removals, the User Centric R-Tree can be much more efficient than the previous structure. Updating a user requires removal of their MBR from the tree and insertion of a new updated one, if the new Region affects the total shape of the geo-footprint. Since regions are stored externally, additions and removals can be performed much faster since there is no need to search the entire structure. When it comes to user insertions and removals, for each user, we can simply add or remove their corresponding MBR from the tree and be assured that no Regions are left over.

1.8 Similarity Search

As we have discussed earlier, we will be using R-trees to store and index our geo-footprints. The next step is to implement search methods to find similar users according to a query user, which will have their own geo-footprint and Regions of Interest. Each search method is responsible for comparing the query user's regions with the structure's regions in order to determine overlapping areas. These areas will be used alongside our pre-computed norms in the similarity computation algorithm (Algorithm 3) to calculate their geo-footprint similarities, so that we can find the top-K most similar users according to their calculated similarities.

As seen previously, we have two different schemes for indexing our users and each approach leads to different search methods. The first approach, namely storing each Region separately, can be used either to search for each query region individually or by searching for the entire query footprint MBR and calculating area overlaps in batches. The second User Centric R-tree can be searched in a similar way but gives us the opportunity to use spatial join optimizations during the overlap calculation of the Regions.

1.8.1 Iterative Search

A first approach and the baseline algorithm for searching is going to be the Iterative Search. Given a query user q with their geo-footprint and Regions, we iteratively search through the structure using each region of q as our query. When searching for a region, the structure returns all users that overlap with that specific region and the corresponding area of that overlap. We aggregate those overlaps while searching for all regions of q with the goal of calculating the final similarities using our pre-computed norms from earlier. After all regions of q have been examined, for each pair (q, u), where each u is a user that was found to overlap with regions of q, we calculate the similarity (q, u) using the similarity algorithm (Algorithm 3). The final similarities are then sorted, and the top-K users are the result of our search.

The R-Tree search needs to be modified in order to calculate and return the overlapping areas for each Region searched. Specifically, for each query user we keep an array of all overlapping users found and their corresponding areas. It is important for the array size to be equal to the number of our unique users, since there is no guarantee of which users each search finds, due to the structure's unorganized placement of users and their regions. This array is passed through our search function and is updated every time a Region that overlaps with our query user is found. This update is done when the R-tree reaches a leaf node, where all of the node's Regions are examined whether they overlap with our query region q or not. If a region, say one that belongs to a user u, overlaps with q, then the area of that overlap is added to the total area of u's overlaps stored in our array. After all of q's regions have been examined, the resulting area sums are stored in our array, which is reset before each new query user is searched.

The Iterative search is mainly used as a baseline search because of its simplicity. It is not expected to be the fastest implementation since it iterates over all query regions and traverses the tree many times for each query. Especially when dealing with large leaf nodes (i.e. nodes that contain many regions), this method is expected to perform many unnecessary iterations since it checks all Regions of that leaf iteratively for possible overlaps. It is also worth mentioning that this method depends heavily on the dataset size because of the search time complexity increasing with the size of the structure. That said, Iterative may perform better in small datasets, where the structure size is small enough that performing MBR calculations required in other methods do not provide an improvement in time over simply searching through the structure iteratively. Moreover, iterative also may excel over other methods when dealing with sparse query footprints, where it is more efficient to search each region individually instead of searching for the whole query MBR.

1.8.2 Batch Search

The main problem with the Iterative search is that it performs many traversals and iterations where the region given may not be overlapping. This is because the query footprint is broken down to its regions and searched separately. Given that our goal is to compare users with similar habits and whereabouts, a user footprint MBR may be a useful guide when searching. Specifically, we can use the query's MBR to traverse the tree and perform spatial joins when reaching leaf nodes.

This method is called a Batch Search because it calculates overlaps in batches every time a leaf node is reached. In more detail, we guide our search using the query's MBR which is found by determining the minimum and maximum bounds of the footprint's regions. When a leaf node is reached, the query's regions are compared with that leaf node and the output is the same as the previous method: an array of all overlapping areas relevant to each user. In the next chapters we will discuss different ways of performing spatial joins when reaching a leaf node, with the goal of determining efficient ways for calculating overlaps.

There are many benefits to this approach over the iterative. For one, users with similar interests are expected to have many regions in the same leaf nodes, due to the spatial relatability the R-Tree uses to store its items. This can result in a query traversing the tree and reaching the same leaf nodes many times when using the Iterative method. When using Batch, we skip those unnecessary traversals and reach each leaf node once when overlapping with our query's footprint. This reduction in traversals can result in a substantial time difference between batch and iterative, especially when working with large datasets that involve a greater R-tree height.

Batch is also expected to outperform Iterative when dealing with larger leaf nodes. Iterative is bound to perform all possible comparisons between each query region and each user region found in a leaf node. As we examine later, batch can use spatial joins to compare the two groups of regions efficiently while also calculating the overlapping areas. That said, as we noted earlier the iterative method may perform similarly or better when dealing with smaller leaf nodes or smaller datasets.

It is worth noting that the batch method efficiency is dependent on the sparsity of our data. When dealing with queries with sparsely placed regions, batch search may have to traverse many unnecessary nodes because of false positive overlaps. In the example of sparsely placed regions in Figure 9, in the second example of two greatly overlapping footprints with no actual overlapping regions, we would get many false positives that would need to be pointlessly compared. These sparse queries are where the iterative method works best, since it separates regions that are far apart and examines them individually, avoiding redundant spatial joins.

Batch may also fall short when working with smaller datasets. Batch needs to perform additional calculations in order to find each footprint's total MBR, therefore needing additional time per query. This additional time is unsubstantial compared to the time it would take for Iterative to perform all of its traversals in a large structure, but when working with a smaller structure, those calculations can be a detriment and result in Iterative performing better.

As mentioned, we can follow many different approaches when comparing each overlapping leaf node with our query. Starting with a simple approach similar to iterative we will be applying different spatial join optimizations and discussing the expected behaviors.

1.8.2.1 Naïve approach

Our objective is to find all intersecting areas between a set of query RoIs and user regions stored in our structure. This can be naively implemented with a double loop, where while traversing the tree and landing on a leaf node, we compare each query region with each leaf node region and calculate their overlaps. This approach is similar to iterative due to the many iterations being done when reaching a leaf node, yet it still traverses the tree in batches, so we expect to see a different behavior. Even without any improvements, this naïve approach can still outperform iterative in various scenarios, especially when dealing with a larger structure. We can of course improve this further by using spatial join optimizations, but it is still important to implement this naïve approach in order to compare batch and iterative methods in terms of tree traversals.

1.8.2.2 Spatial join optimizations

There are many ways of optimizing a spatial join. In our case, we follow the approaches of [3] where we try to consider only regions that overlap with the comparing footprint's MBR. In detail, before performing checks between each region of the query and each region of the leaf, we check to see if our current region overlaps with our query. If not, we can skip those unnecessary comparisons and continue to the next query region. By doing this, we can easily skip many comparisons of false positive overlaps that would have been otherwise iterated over.

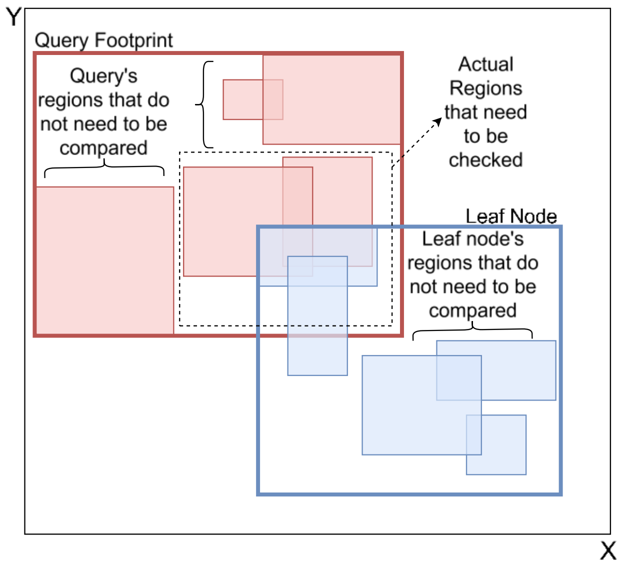

Y
Query Footprint
Query's regions that do not need to be compared
Actual Regions that need to be checked
Leaf Node
Leaf node's regions that do not need to be compared
X

Figure 10: Example of region pruning optimization

This optimization can also be performed from the user's side. Specifically, we can access all regions from the leaf node and remove from consideration all regions that do not overlap with the query's footprint MBR. This can be done before the main comparisons begin by creating a new pruned list of regions and using that instead of the whole leaf node.

These two optimizations save time by performing less comparisons between the rectangles found in the leaf nodes. Because of the small complexity of those comparisons, we do not expect an improvement over time when using these methods, especially considering the additional scan that needs to be performed when pruning rectangles from the user's leaf node. For this reason, and after testing both methods in various structures, we conclude that just using the first optimization is more optimal. We will be comparing this optimization to the rest of the methods.

1.8.2.3 Using a plane sweep join

The last optimization, and possibly the most efficient, is using a plane sweep algorithm to calculate overlaps between our footprints. A plane sweep is a technique from Computational Geometry which uses an imaginary line that scans the space through one axis and performs calculations on each stop. In our case of spatial joins in R-Trees, the sweep line is used to determine overlapping regions between the two footprints.

All rectangles from each set need to be sorted according to their start position along one axis for the sweep to be performed. We chose the x axis, so we sort all rectangles both from the user and the query lists. The line is then swept across our space along the chosen axis and stops whenever a rectangle is met on either list. At each stop, a loop is run to determine whether the current rectangle overlaps with a rectangle from the other list, first along the x and then the y axis. If the intersection is confirmed, their area of intersection is calculated, and the area sums are updated accordingly. We keep each area for each user in a list, the same way as in previous methods. A pseudo-code for the plane sweep in reference from [34] can be found in Figure 11.

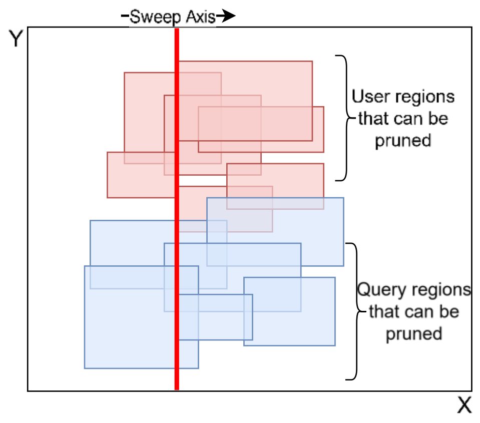
```
**function** $Loop$(rectangle $anchor$, int $fpos$, $x$-axis sorted list of rectangles $L$)
    $k := fpos$;
    **while** $(k \le |L| \wedge L[k] \le anchor.xu)$ /* $x$-intersection */
        **if** $(y\text{-}intersects(anchor, L[k]))$ **then** /* $y$-intersection */
            **output** $(anchor, L[k])$;
        $k := k + 1$;

**function** $forward\_sweep$($x$-axis sorted lists of rectangles $L_R$, $L_S$)
    $i := 1$; $j := 1$; /* point to the first elements of $L_R$, $L_S$ */
    **while** $(i \le |L_R| \wedge j \le |L_S|)$
        **if** $(r_i.xl < s_j.xl)$ **then** /* $r_i$ has smaller sort-key value than $s_j$ */
            $Loop(r_i, j, L_S)$;
            $i := i + 1$;
        **else** /* $s_j$ has smaller sort-key value than $r_i$ */
            $Loop(s_j, i, L_R)$;
            $j := j + 1$;
```

Figure 11: Forward sweep heuristic, pseudocode in reference from [34]

This plane sweep approach can be combined with our previous optimizations. Specifically, we can exclude all user regions from the leaf node that do not overlap with our query footprint and exclude all query regions that do not overlap with the current leaf node MBR. This optimization, as described earlier, would require two additional scans, one for our leaf node and one for our query footprint, with regards to acquiring the non-eliminated Regions of Interest.

Although the plane sweep may be performed in a smaller section of the space after these optimizations, the benefit in performance will not be significant. Since our sets are sorted along the x axis, the plane sweep by nature will skip any Regions that begin before the start of the other set and will also stop the comparisons after the last region of any set has been reached. This yields the same results as the aforementioned optimizations, but only along the x axis, as the overlap checks are done according to the sweep direction. Implementation wise, these optimizations add more complexity than just using plane sweep, since we try to create pruned sets rather than just skip through the regions while performing the sweep.


Sweep Axis
Y
X
User regions that can be pruned
Query regions that can be pruned
Figure 12: Example of prunable regions for plane sweep

One instance where the pruning optimizations may be useful is when dealing with footprints that overlap heavily on the sweep axis but very slightly on the other. In this case, pruning the two sets to keep just the overlapping regions can lead to less stops during the plane sweep, because of the decreased regions along the non-sweep axis. Even in extreme cases of uneven overlaps along the two axes, plane sweep would only benefit by the pruned nodes if their size were substantial enough to beat the time needed to perform those pruning operations. Since our index is focused on indoor user paths, we expect to see more discrete regions being formed, with either heavy or little overlap between them. For this reason, we do not expect to see cases where the pruning optimizations would aid the standard plane sweep approach.

A final part of our implementation involves around the sorting part of plane sweep. Specifically, as discussed, both query footprints and leaf nodes require to be sorted by their starting point along one axis. This of course could be achieved by sorting the nodes each time before comparison, but it would not be very efficient. Instead, we only sort the query once before searching and sort each leaf node once the first time it is traversed. We achieve this by augmenting the leaf nodes of the tree with a boolean variable that indicates whether the node is sorted or not so that each sort is only performed once. This, of course, leads to updates to the structure needing to re-sort the leaf nodes, but we manage this by resetting our boolean variable each time an update is done, with the goal of leaving any sorts to be done during the next search.

We expect the plane sweep method to perform exceptionally well, showing better results than all previous methods. That said, we also expect batch to perform similar to Iterative when dealing with large queries that take up most of the space, because of the increased false positives it would have to iterate through. For this reason, it is important to test our results with unconventional queries and structures to discover any setbacks. Finally, we do not see any instances where the naïve implementation could outperform this method, since plane sweep always performs equal or less comparisons than naïve.

1.8.3 User Centric Search

So far we discussed implementations concerning a straightforward R-tree indexing scheme, specifically one that stores each region individually. As mentioned, this structure may end up sizable since each user has multiple regions that need to be indexed. This led us to the User Centric approach, where only the footprint MBR is stored for each user, resulting in much smaller trees that potentially produce search results much quicker.

When it comes to similarity search, all mentioned Batch search methods can be implemented in the User Centric Tree since the task is similar; compare two nodes and the regions they enclose. The size of nodes may differ in this case compared to the previous structure. In batch, the number of regions per comparison depended on the maximum size of leaf nodes, where in this case, sizes only depend on the activity of a user, making more active users have larger footprints with more regions. This, of course, does not affect our decision in choosing a similarity method, because, as discussed earlier, a plane sweep is expected to perform similarly or better compared to any other implementation. Especially when it comes to larger nodes that tend to be more prevalent when using the User-Centric approach, a sweep is expected to outperform any other approach.

As far as implementation goes, for each user we keep an array containing all their regions. Footprint MBRs are stored in the tree, and each MBR points to its corresponding array. When given a query user, the footprint's MBR is used to guide the search through the tree, the same way done in batch. Any overlapping users are then checked individually by performing a plane sweep between their regions and the query's regions. As explained previously, for the plane sweep to be done efficiently both query and user regions need to be sorted along the sweep axis. This is achieved by sorting all query and user regions before starting our search. That results in the same overlapping areas array as above, which are in turn used to calculate the final similarities. The general workflow of a User Centric search can be seen on Figure 13.

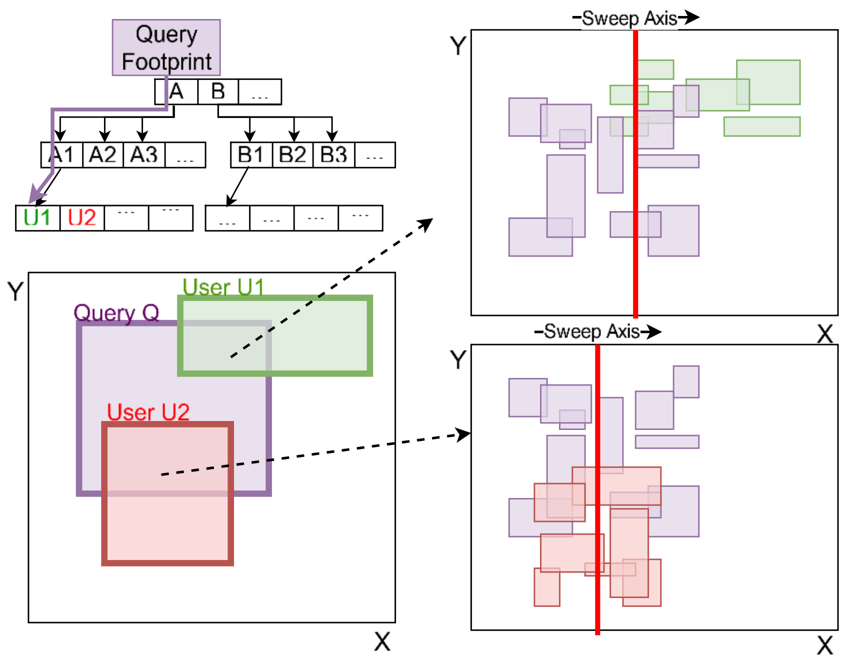


Figure 13: User Centric search workflow

User centric tree search is essentially a version of batch search, if each leaf node represented a specific user. The main difference is the size of the structure; all regions of each user are stored externally, resulting in reduced height in the R-Tree and in turn faster traversals. This difference between the User Centric Tree and the standard approach is amplified when dealing with large datasets.

A plane sweep is the most efficient way of comparing two sets of regions, so we expect to see faster searches when dealing with users with heavily overlapping footprint MBRs. That said, in cases of users with large MBRs that take up a substantial percentage of the plane, user centric can produce many false positives which will result in many unnecessary comparisons.

In scenarios where user footprints take up a small amount of space (i.e. in a large shopping center), we expect user centric to outperform batch search. But in more realistic scenarios, where a user's footprint can take up most of the space, we expect to see worse performance than the batch sweep approach. In most cases where the dataset is large enough, we expect this scheme to beat the standard R-tree scheme in terms of traversal time.

Chapter 4: Experimental Evaluation

So far, we have examined ways of extracting important regions from user trajectories and developed two indexing schemes and efficient search methods for each scheme. We also made observations about the scenarios in which each method would be more effective. In this chapter we will be testing those hypotheses with custom data according to each case.

In order to generate data tailored to each test case, we develop and present a trajectory generator. It generates user paths based on either predetermined or random regions of interest, while ensuring smooth transitions from each region to the next, and a customizable amount of randomness each time using normally distributed random variables.

Using our generator, we design test cases according to each method's expected strengths and weaknesses. For each dataset generated, we follow the suggested procedure from [1] to tune the extraction algorithm and determine the appropriate epsilon and tau variables.

Finally, we perform tests for each method, namely Iterative, Naive Batch, Optimized Batch, Plane Sweep Batch and User Centric. We try all datasets generated for each method and discuss the results. We then name the discovered strengths and weaknesses of each method and conclude that the Batch Sweep approach is the most suitable for most cases of Indoor Trajectories.

4.1 Indoor trajectory generator

In the introductory chapter we discussed about the state of indoor data tracking, concluding that, although promising, indoor tracking is yet to be fully developed and used. For this reason, there is a general shortage of indoor datasets, and most of what exists is not enough to test specific strengths and weaknesses for each of our approaches.

For this reason, we have developed a Customizable Indoor Trajectory Generator [37] in the context of this thesis. This generator can produce user trajectories with customizable length, step size, areas of interest and more. It is also capable of generating paths inside predefined spaces such as supermarkets or shopping centers. We will briefly present how it works and its capabilities.

4.1.1 General Structure

The Custom Trajectory Generator is a trajectory generator developed in java, which produces trajectories for multiple users, in order to simulate the tracking data from an indoor space. The generation of paths is done in layers, with each layer containing many instances of the layers beneath it. The resulting concatenated set of points from the bottom layer is the output Trajectory for that specific user.

The first layer is responsible for creating each user separately. For each user, a day represents a series of walks done in a given space. According to the input, we can generate many different traversals, each separate from the other, each representing a day's trajectory from that user.

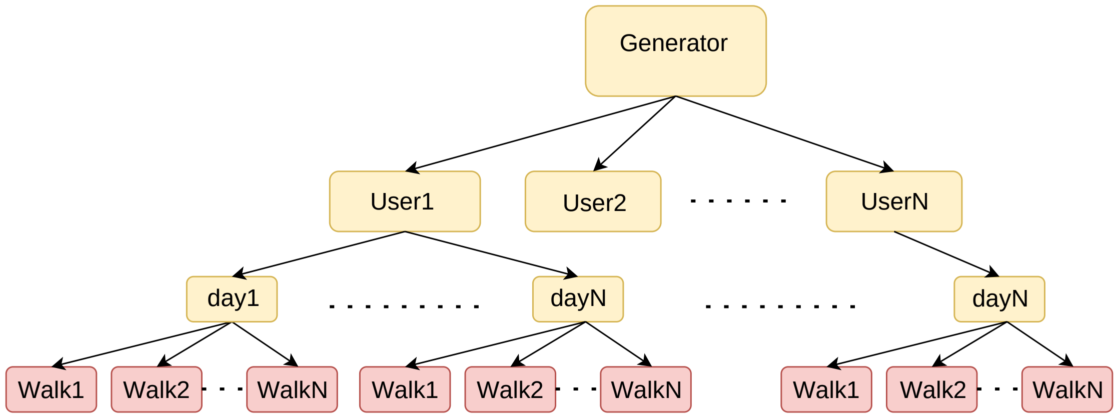


Figure 14: General Structure of the Trajectory Generator

The second layer is responsible for the Walks of our user. A walk is a series of traversals through a specific generated Rectangular Region, which we define as a Region of Interest. Those Regions are generated in the Region of Interest sub-generator, which is a module responsible for creating Rectangular Regions according to specific requirements. We will go over that module and its capabilities later.

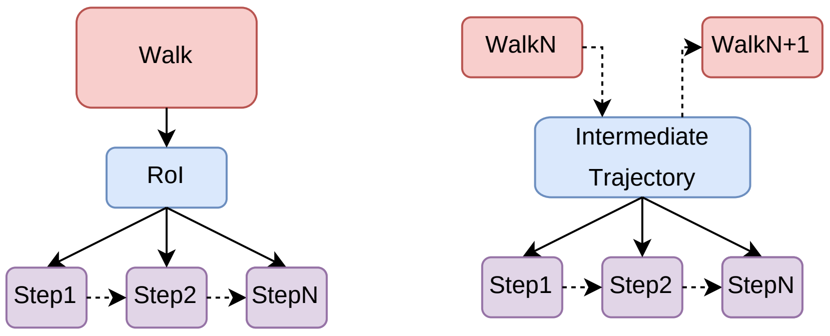


Figure 15: a) Walk generation b) Intermediate Trajectory generation

Between each walk, in order to ensure smooth transitions, an Intermediate Trajectory is generated inside an Intermediate Region that guides the user from the last point visited to the start of the next walk.

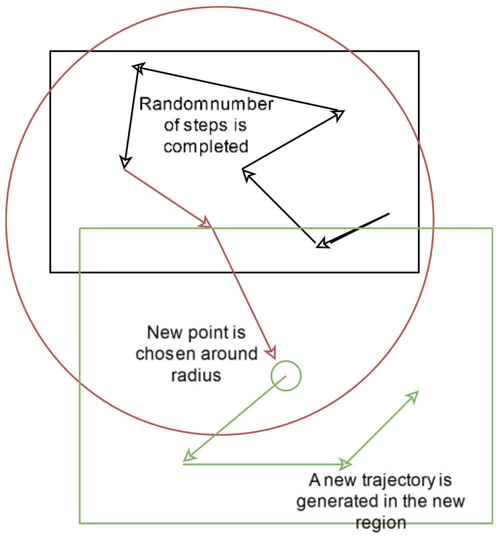
The third and final layer is the Path generator. In this layer, given a specific Region, a series of consecutive points are generated depending on random distances and directions given. This is where a sub-trajectory is generated in relation to its previous trajectory and the region given from the RoI module. The path generation is performed in its own module called Path Generator. We will also go over how it works and its capabilities.

In summary, for each day, each user makes a random number of Walks. During each walk, a user visits a Rectangular Region, denoted as a Region of Interest. During each of those visits, they perform a random amount of steps which are translated as consecutive points in a trajectory. The final points are concatenated into a larger trajectory which is the final output for that specific user. We will now go over the special capabilities of each module, including how we can contain a user's walks into specific sets of areas (to simulate interests).

4.1.2 Region of Interest Generator

The Region of Interest generator is responsible for creating a series of rectangular regions relative to each other, in order to simulate smooth walks around a discrete space. There are two main ways of generating those regions, namely randomly in a radius or through a predefined grid. This module is also responsible for generating the intermediate Regions between each walk.

The Randomly Generated regions are created in a radius around the last point a user has visited. We assume that the space has no obstacles, so that the user can continue his trajectory towards any direction. A random point is selected in a radius, either anywhere or strictly outside the last walk's RoI. Using that point as a center, a new rectangular region is generated using a randomly distributed normal variable to determine its x and y sizes. After finishing the sub-trajectory generation, this process is repeated until all walks have been completed.


Randomnumber of steps is completed
New point is chosen around radius
A new trajectory is generated in the new region
Figure 16: Path generation using random rectangles

The Grid generation works by loading a set of regions at the start of the generation. These regions are given in the form of an adjacency list, where each rectangle is followed by the regions it is connected to. These adjacent rectangles can represent the isles in a supermarket, or the different stores in a shopping center. The list is loaded in the form of a linked list (denoted as a Linked Grid) and is used to indicate the next region that a user will visit. Each time, the next region selected is a region adjacent to the current region, preferably different than the previously visited region.

Finally, the Intermediate Regions are rectangles that cover the distance between the last point of the last completed walk, and the first point of the next walk, indicated by the next region that will be visited. Inside those rectangles an intermediate trajectory is generated, in which the user walks inside the region until they reach a point close to the first step of the next region. This is mainly performed in the path generation module, although the region required is generated here.

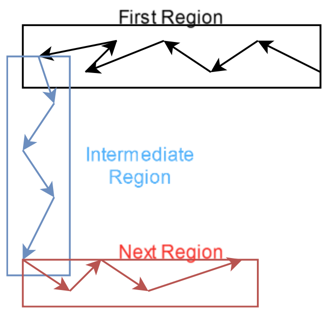


Figure 17: Intermediate Trajectory Generation

It is worth mentioning that all variables are determined using a normally distributed random number generator, with a specific mean and variance for each case. More specifically, the number of Walks performed, the size of the random rectangle and the radius are all selected through normal variables defined at the start of the generation.

The RoI generation can be customized to achieve different results. As mentioned, we can control whether we strictly select the next random rectangle to be inside or outside of our current one during generation through radius. We can also choose to restrict the areas that a user can visit when using a Grid. We can select a custom subdivision of the total areas that a user can visit, in order to simulate a user that is interested in a specific subdivision of the store. We can also choose to keep that subdivision in the subsequent days, to generate a user that visits the same general areas each day.

Given a series of regions, the next step is to generate steps inside of those regions. This is the responsibility of the Path Generator.

4.1.3 Path Generator

The Path Generator is a simple module that generates a set of consecutive points according to given random variables. Given a Region of Interest and a starting point, our user is tasked with making a random number of steps. For each step, our user traverses a random distance towards a specific direction, and then randomly changes towards a new direction. The change in direction is always done with a mean of zero and a customizable variance. Finally, during each step, there is a chance (usually 30%) that our user does not make a step, which is used to simulate a curious user stopping to observe something.

All these random variables are, again, normally distributed, with predefined means and

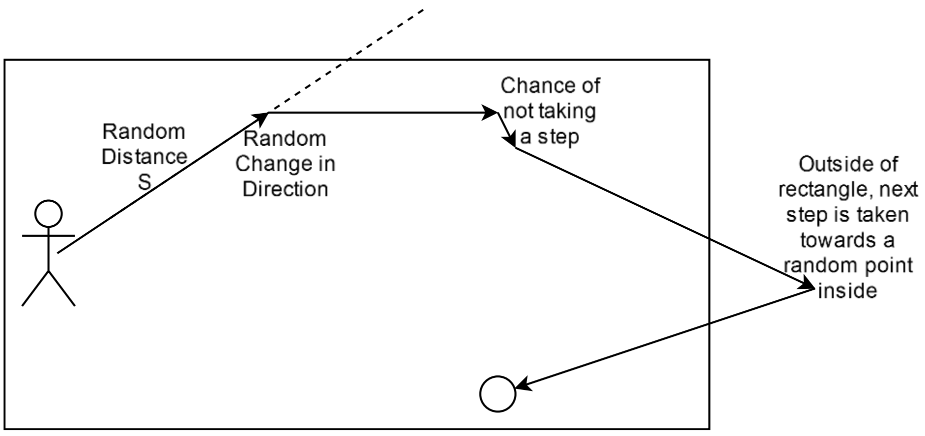

Random Distance S
Random Change in Direction
Chance of not taking a step
Outside of rectangle, next step is taken towards a random point inside

Figure 18: Trajectory Steps Generation

variances. For each variable we can also define a maximum and a minimum value which overwrites the random generated number if it gets out of bounds.

During the Intermediate regions, the direction variables are adapted to guide the user more quickly towards the next point. Specifically, during each step the direction median is always towards the destination point, and the variance is reduced.

4.2 Dataset generation and extraction

In this chapter we go over the factors that affect performance and how they translate to each method. We then explore the datasets produced to test those factors, and how we applied the extraction algorithm to each of them.

4.2.1 Performance Factors

Using the Trajectory Generator we developed, we can focus on creating datasets with tailored aspects to test weaknesses and strengths according to our hypotheses. To achieve this, we first need to go over all possible factors that can affect the performance of our indexing schemes. These factors stem from the structure of each method:

Leaf Node Size: Each structure, being an R-Tree, has a predefined leaf node size which dictates the maximum and minimum number of elements that the leaf node can contain. In the case of Iterative and Batch, a larger leaf node can cause a smaller tree height leading to faster traversals but will also result in more comparisons being done when reaching a leaf node to calculate overlaps. It can also cause a larger leaf MBR which will cause more false positives when using Batch. A smaller leaf node can lead to a taller structure, requiring more time to search for a specific result and causing Batch (and Iterative in a lesser extend) to perform more traversals through the tree. For our tests, we concluded that the standard leaf node size of 16 would be suitable, achieving a balanced performance across all methods.

RoI density: A user with many densely placed, overlapping regions can result in smaller leaf node MBRs in the R-Tree, which in turn aids Batch by causing less false positive results while searching. On the other hand, Iterative would perform better than Batch in a context of sparsely placed regions. Because of its decreased search scope (due to searching for each region separately), it would avoid many leaf nodes with non-overlapping regions, which would otherwise be accessed and compared by Batch. User centric would also benefit from this method since the usage of a plane sweep will yield much faster results in the context of densely placed Regions compared to multiple Iterative comparisons.

User extent: In realistic scenarios, most users traverse only a minority of the space. This can cause smaller footprint MBRs to form, leading to more efficient operations in Batch and User Centric compared to Iterative. User centric would benefit most this context since it inevitably compares the entirety of user and query footprints, unlike batch which selectively compares only parts of each footprint. However, in cases where users traverse a large percentage of the space, or form sparsely placed regions, large MBRs are formed which pose a detriment to the User Centric approach. Although Batch can compensate for this because of its storage of regions irrelative to their users, it can still perform worse than Iterative in cases where the given query user is taking up most of the space.

Region Extraction: As mentioned, the extraction algorithm requires two variables to be tuned to produce the right number of regions. A smaller than optimal tau and epsilon can produce a large amount of small, densely placed regions, which will affect the performance of each method as mentioned earlier. On the other hand, usage of larger than optimal values can produce large, sparse regions with little overlap, which results in loss of information, especially in the case of an unweighted approach. We will go over the tuning process described in [1] and how it led us to picking each extraction variable.

4.2.2 Region Extraction

Paper [1] describes an experimental approach for determining the appropriate number of regions per user. We will be following the recommended target of 15-20 regions per user per day. We aim to find the largest possible values for epsilon and tau while staying within the target number of regions.

To achieve this, for each dataset we keep a small sample size of 1000 users and their trajectories for a single day. We start by keeping tau fixed in a small value of 20 and perform tests with different values for epsilon. For each test, we count the average amount of regions per user. We increase epsilon by a small amount relative to the dataset until the average number of regions is between 15 and 20. After that, we start increasing tau by 10, until the number of regions starts decreasing. At this point, the tau value has been determined. For a final tuning, we try slightly increasing or decreasing epsilon until we reach a specific target average. According to the dataset, this target can be slightly above or below the recommended. We finally check the average x and y extend of each region and if the result is acceptable according to the dataset, we perform the extraction on the entire set of users.

4.2.3 Datasets

Finally, we present the four datasets developed for testing our indexes and methods. Each dataset has a goal, either to simulate realistic conditions or to test a specific hypothesis. In each case, we keep a subset of 1000 users to be used as a query. This ensures that the methods function correctly by producing 100% similarities since each query user is also stored in the structure. It also simulates similarity searching between users of the same kind, since query users stem from the same dataset as our indexed users.

Below is an index of all the variables used for achieving each dataset using our trajectory generator. We will explain later the goal of each dataset, along with design decisions.

	Grid Size	#Days	#Users	#Walks*	#Steps**	Avg Step	Avg Region***
Shopping Center	1500x2200	3	350k	15+-10	50+-40	7+-5	-
Grocery Shop	1850x1150	7	300k	10+-8	30+-25	10+-8	-
SUSR	5000x5000	2	400k	30+-25	150+-100	5+-5	200+-150
SUDR	5000x5000	1	500k	1	10k+-2k	5+-5	600+-100

Table 1: Index of variables used for extraction

* Average number of regions visited per user per day.
** Average number of steps taken during a visit in a specific region.
*** Average size of each axis in a randomly generated region. Each axis is selected independently.

4.2.3.1 Shopping Center Dataset

For this dataset, we followed a layout of a large Shopping Center [38] with multiple stores and large isles. We modeled each region so that there is always a path to go from one region to an adjacent one. This dataset was created to simulate a realistic scenario where users are tracked in a shopping center, while visiting specific stores.

The goal of this dataset is to simulate a realistic scenario of users in a shopping center. Since users are not restricted in any way, they can roam the various regions freely without specific interests in stores. This will cause many footprint MBRs to take up a large percentage of the space, which, as we discussed, will be detrimental to Batch and User Centric

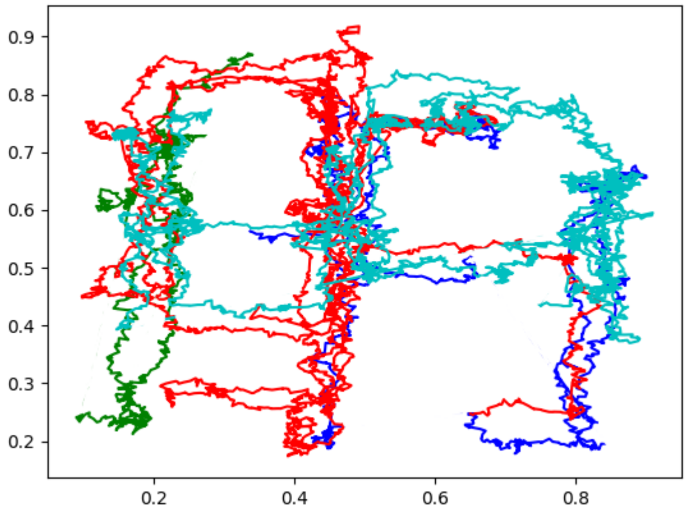


Figure 19: Shopping Center Trajectory Example (4 users)

which use the MBR to guide the search. That said, since the dataset is large with many regions per user, we expect traversal through the large structure to be extremely costly, causing Iterative to be much slower than the rest of the methods.

Following the procedure explained earlier, we extract the regions for each day. Our target is 15 regions, which is on the lower end of the recommended range. This is because of the spatial extent of each extracted region, which was ideal when using this number as a target, relative to the layout of the shopping center. The results are an epsilon of 0.13 and a tau of 50.

4.2.3.2 Grocery Shop Dataset

Another realistic scenario of indoor tracking would be a grocery shop. For this reason, we modeled a grocery store using the layout from [14]. Since this store's extent is much smaller than the shopping center, we opted for a larger step size and decreased number of walks per day. We also increased the number of days, attempting to simulate a week's set of visits.

Since our grocery store is small, an unrestricted set of visits by each user would also produce sizable footprints as before. For this reason, we implemented a restriction between each day. Specifically, during their subsequent visits, each user may only visit the regions that they visited the first day. This ensures that, even in 7 days, a user's footprint will not take up most of the space. This also simulates interests, since it creates users that visit the same locations each time.

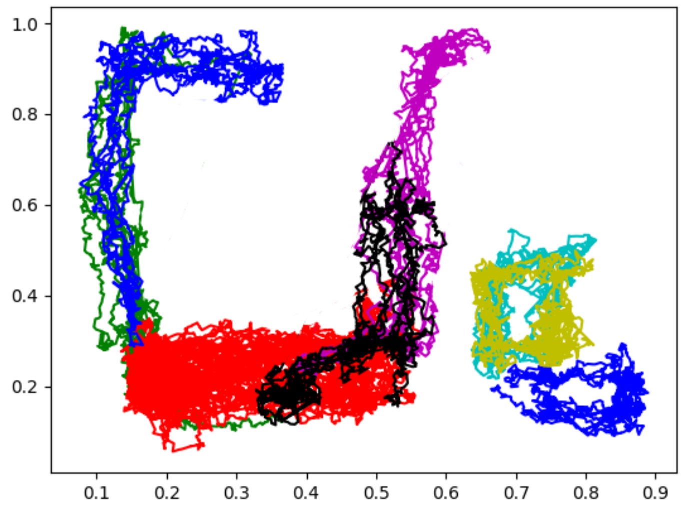


Figure 20: Grocery shop trajectory example (7 users)

We expect this dataset to aid the user centric approach by resulting in smaller extents in footprints. The same improvement will also be visible in batch search, but to a lesser extent. Iterative will still be expected to perform poorly, especially in this case where 7 days of footprints are bound to create a large structure that is slow in traversal.

Keeping the same target of 15 regions as before, we perform the same procedure. The values found suitable were 0.12 for epsilon and 30 for tau.

4.2.3.3 Sparse Users – Sparse Regions Dataset

The sparse users datasets were developed as stress tests, to test cases of small footprints with either many densely placed regions or less, sparsely placed regions. For that reason, aiming for more contained footprints, we opted to generate only for 1-2 days for each user. These datasets are not necessarily realistic but prove useful when trying to uncover strengths and weaknesses of the various MBR related search methods.

This first dataset produces smaller regions with a smaller number of walks and steps, which, with proper extraction, should result in sparser regions. The goal of this dataset is

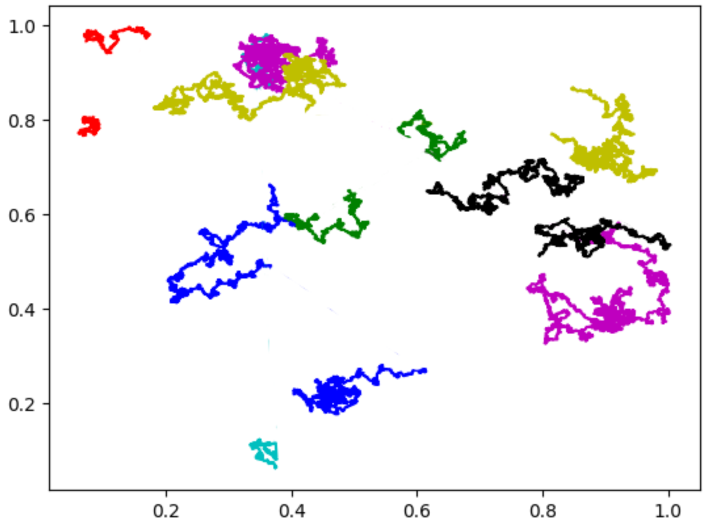


Figure 21: Sparse users trajectory example (7 users)

to observe the behavior of each method when dealing with these unusual types of footprints. We expect this method to mostly benefit Iterative search, since it produces a smaller number of regions that is quick to search individually. We also expect a greater performance from User Centric, since we are dealing with contained user footprints. Batch is expected to perform as usual.

For extraction, we tested and found that a target of 30 regions would produce sparse enough rectangles without compromising on information. The ideal epsilon and tau would be 0.025 and 30 respectively.

4.2.3.4 Sparse Users – Dense Regions Dataset

The goal of this dataset was to produce similarly sparse users as before; therefore, the grid size remained the same. The main difference in this case is the way that the trajectories were generated each day. Specifically, we opted for a single day dataset with a single walk for each user. To compensate for the reduced number of walks, we chose a large number of 10k steps per walk, therefore creating a dataset where users form a lengthy and dense trajectory inside a confined space. This will result in many overlapping regions being formed during the extraction. Although unrealistic, we want to observe the edge cases where batch method won't outperform the other methods in terms of query time. We also expect to see Iterative answer queries faster for the same reasons as the previous dataset.

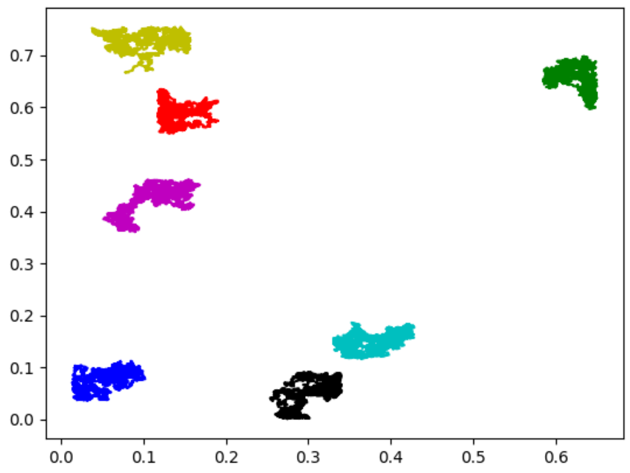


Figure 22: Sparse users trajectory example (7 users)

Because of the unusual nature of our trajectories, we aim for a much greater number of regions. The ideal number found was 90, which produced regions similar in spatial extent as before. Our discovered ideal epsilon and tau where 0.015 and 30.

4.3 Evaluation

In this chapter we perform tests for our methods and discuss the results found in each method. More specifically, we first randomly choose 1000 User footprints to be used as Query users. Then for each of the 5 methods we run a search for those 1000 users, after indexing the rest of the dataset. Each run is performed 12 times, and our resulting time is the average of those runs, excluding the best and worst time. For each run we time the Total Run time but are mainly focused on the Total Indexing Time and Average Query Time (which is the average time it takes to complete a single query).

We compiled all codes in g++ 9.4.0 with the –O3 optimization flag and ran experiments on a 32GB Ubuntu 20.04.3 LTS machine with Intel Core i9-10900K CPU @3.70GHz.

4.3.1 Total Time

Total Run time is a general indicator of the performance of one method. This time can result from many different factors, since there are many operations that are performed during each run. We will later be examining the sub-procedures in which there is a variance in time between each method.

	shopping center	grocery shop	SUSR	SUDR
Iterative	1177.187	2588.018	80.458	79.393
Batch	917.634	1293.311	456.887	110.897
Batch-Opt	895.678	1253.543	431.658	107.769
Batch-Sweep	755.559	1114.609	293.586	92.796
User Centric	985.086	1374.893	273.384	56.118

Table 2: Total runtimes in seconds

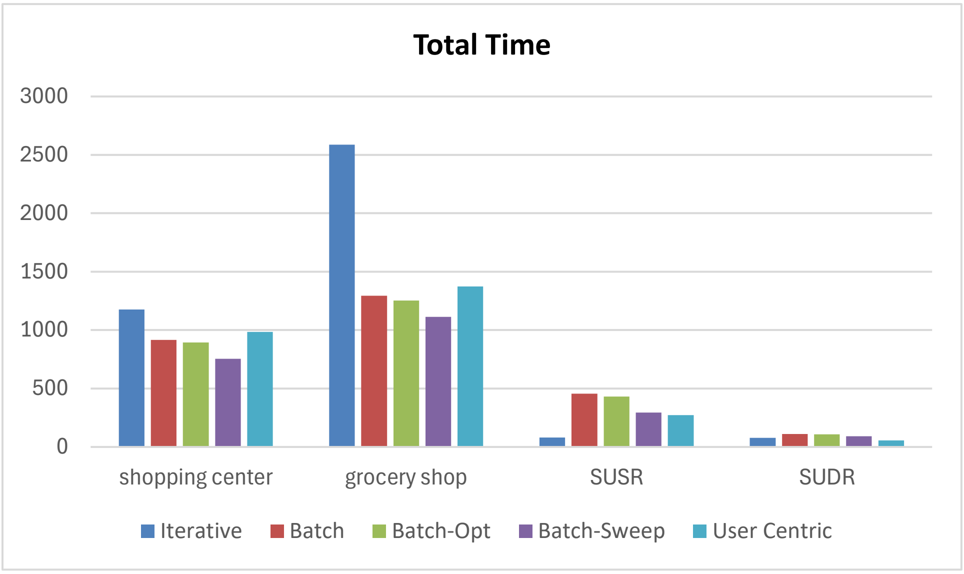


Figure 23: Graph of total runtimes in seconds

In the first two datasets, which attempt to produce realistic results, we observe Batch to perform the best with the sweep version being the most efficient. As the following charts show, this is due to a difference in query times, which are performed more efficiently in small batches using special relativity as a guide. User centric performs worse in both methods, which is a result of the large number of false positives produced when using the footprint MBR as a guide.

In the unrealistic sparse user methods, iterative seems to perform better. This is because of the edge case scenarios, where searching in a sparse space produces less false positives by using individual regions as a guide. This, combined with the smaller sizes of the structures produced leads to an increased performance in Iterative. As expected, User Centric also performs better due to the ideal conditions for performing sweeps between entire users (multiple densely placed regions). The only reason for beating batch, as we will explain next, is the vast difference in indexing time, where user centric performs exceptionally well.

4.3.2 Indexing Time

Indexing refers to the procedure of reading the input data and adding them to the data structure. This time is solely dependent on the structure used; therefore we do not see difference between the first 4 methods which use a standard R-Tree. The fifth method, namely user centric, produces the index much faster since it does not add each region individually, rather it adds just the MBRs of each region to the structure and keeps the regions per user externally.

	shopping center	grocery shop	SUSR	SUDR
Iterative	16.681	27.005	24.098	39.672
Batch	16.807	27.164	24.242	39.852
Batch-Opt	16.75	27.043	24.097	39.618
Batch-Sweep	16.385	26.464	23.867	39.362
User Centric	0.635	0.948	0.881	1.399

Table 3: Total Indexing times in seconds

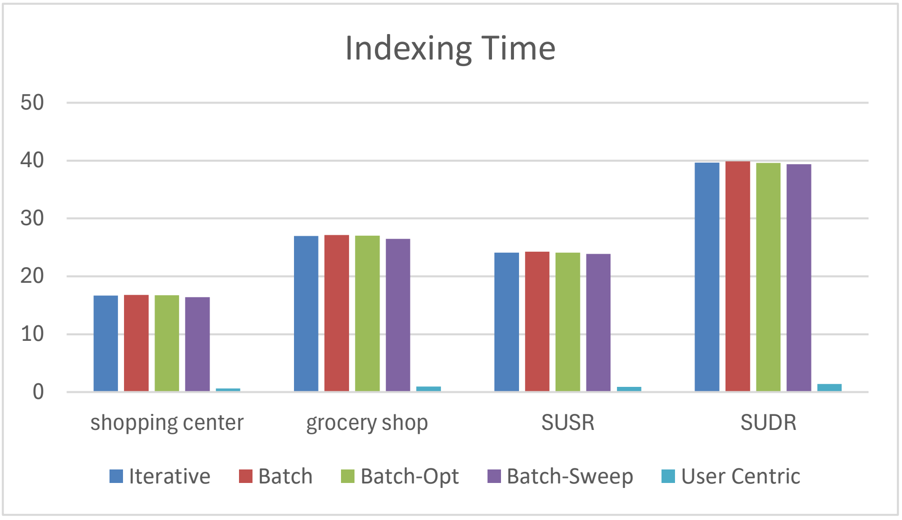


Figure 24: Graph of total Indexing times in seconds

Here lies the main benefit of User Centric; a much faster indexing time compared to the standard R-Tree. This fast-indexing time can, in some cases, compensate for the loss in query time, even producing similar results to Batch, which is the most efficient method along the simulated realistic datasets.

As discussed earlier, in cases of live datasets where users' regions are constantly being appended or removed, user centric is capable of faster and more reliable performance due to its structure. This trade-off between total search time and fast indexing updates can be crucial when picking the ideal approach for an indexing scheme.

4.3.1 Average Query Time

Finally, the most important factor that affects the total search time is the query time, or in other words, the time it takes for a single query to be answered. This time is tied to the search method, producing a variety of results for each dataset.

	shopping center	grocery shop	SUSR	SUDR
Iterative	1.142	2.542	0.0417	0.028
Batch	0.882	1.247	0.418	0.059
Batch-Opt	0.861	1.207	0.393	0.056
Batch-Sweep	0.721	1.069	0.255	0.0419
User Centric	0.966	1.361	0.258	0.054

Table 4: Average Query times in seconds

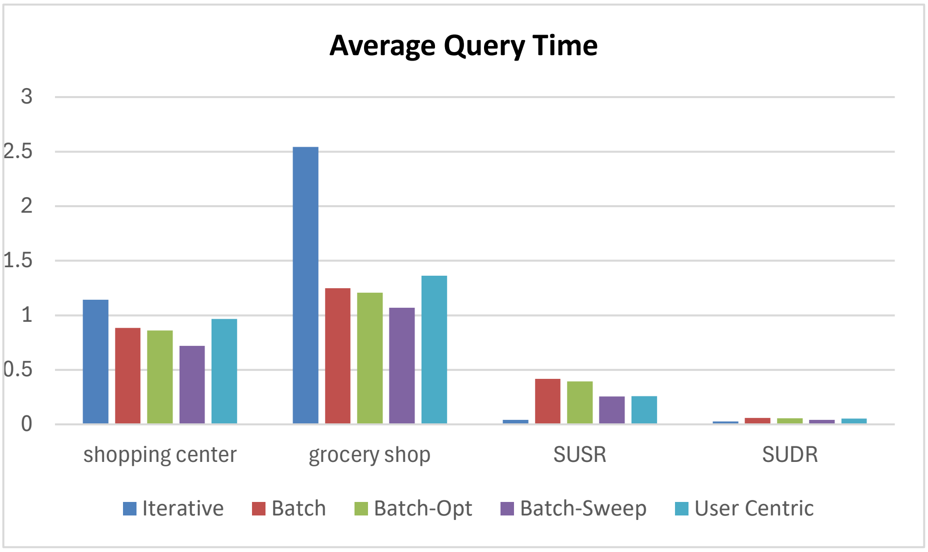


Figure 25: Graph of average query times in seconds

The first two datasets show similar behavior, with batch being the most efficient. User centric is worse compared to all approaches of batch, winning only against iterative which is forced to perform many more traversals through the structure. These cases of realistic users have footprint MBRs that take up a considerable amount of space, therefore causing User Centric to perform poorly against the more guided traversals of batch search.

The sparse datasets are more ideal for User Centric because of their contained users. By producing more confined footprint MBRs, user centric deals with less false positives, therefore reaching the performance of batch. That said, we observe a much greater performance from iterative because of the small structure size and the sparsely placed users. That confirms the hypothesis that the performance of iterative is heavily dependent on the structure size, which is much smaller in the sparse datasets due to the lesser number of days.

4.3.2 Weighted vs Unweighted Comparison

Using the Grocery and Shopping Datasets, which we concluded caused the most time-consuming tests, we performed multiple runs to examine two theories related to the weighted similarity search. First, we hypothesized that a weighted set of regions would result in a difference in the top-K results, because of the new similarity measure that is heavily affected by the weights of each Region. To test this, we performed a top-10 search in the Grocery and Shopping datasets, with the goal of observing what percentage of the top-10 results were affected by using the weighted approach. Below is an example of two specific query searches compared between the weighted and unweighted versions:

In this specific example, we can observe that only two users struck similar positions in the top 5 most similar users. On average between all users from the queries tested on both datasets, 7.96 out of 10 results are different between the two measures, indicating the impact of applying temporal weights to our similarity measures.

Weighted	Unweighted
269755	271565
186087	**269755**
287022	279725
288679	356906
246774	**288679**

Table 5: Example of results in a top-5 search between the two approaches

The second hypothesis was concerning time. Due to the multiplications already being done in the norm computation algorithm, we expected the additional weight multiplication not to affect the norm time. The only way that norm time would be affected is by some optimizations performed by the g++ compiler. As far as query times go, we expect to see a slight difference in time since we avoid those multiplications when dealing with unweighted regions.

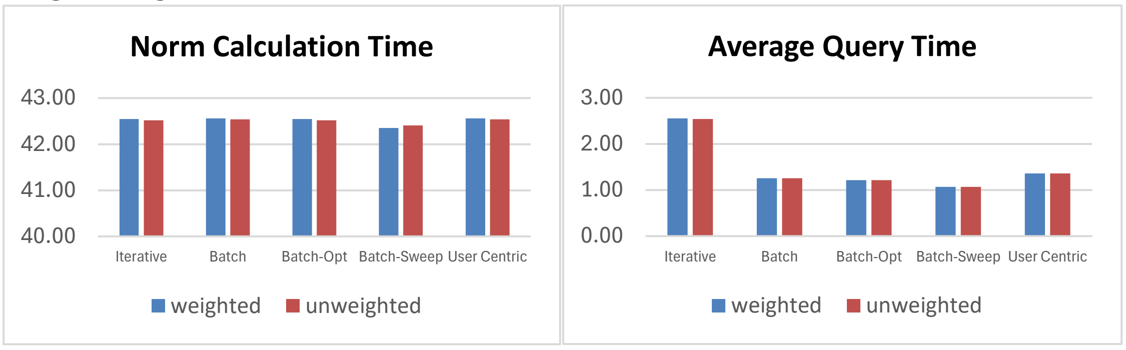


Figure 26: Comparison between the two approaches, a) Norm calculation times b) Average query times

As expected, total times do not differ between weighted and unweighted versions. The total time depends on the norm calculation times and query times, which we can compare further. Norm calculation time performance is similar between the two methods, due to the same number of multiplications being done in both cases. The average query time is where we expected to see slightly worse performance from the weighted approach. However, we observe little to no differences between the two approaches, indicating that the time needed for the additional multiplications is unimportant.

4.4 Conclusion

In conclusion, as we hypothesized and observed from the test results, there is no perfect method for all possible use cases. Each method proved to have its own strengths and weaknesses depending on the factors discussed in 4.2.1.

Iterative proves to be efficient when it comes to small structures, where the R-Tree traversal time is reduced. It also performs well in sparse datasets where large footprint MBRs cause many false positive results that need to be double checked. When it comes to large datasets, iterative performs poorly, and combined with its inefficient additions and removals, it is not considered the best of the three methods presented.

Batch is a lot more efficient than Iterative and showed the best performance of the three. The optimizations used were crucial in reducing the time that region comparisons take, by pruning the dataset and reducing false positives. The main disadvantage of this method is the updatability of the structure, since it is very inefficient to add and remove users from the R-Tree.

Finally, User Centric structure and search underperformed in the more realistic scenarios generated. However, in some unusual test cases, the time saved in indexing time was enough to beat the other two methods. In general, the advantage of this method comes with indexing and updatability. Since each user is stored separately, removals, updates and additions of users could be performed much faster compared to the other two methods, making this method suitable for a live dataset where users are constantly updated. Further tests need to be performed concerning updates in the structure to prove the advantages of user centric.

In conclusion, the R-Tree approach followed by [1] proved to be very effective when it comes to geo-footprints. Instead of the usual R-Tree, other spatial indexes could be tested, such as a grid, in order to test possible advantages or disadvantages. The weighted approach proved to affect the top-10 results greatly and its usage in real world use cases would prove to be much more accurate than the initial approach.

References

[1] Achilleas Michalopoulos, Konstantinos Lampropoulos, George Kelantonakis, Chrysostomos Zeginis, Kostas Magoutis and Nikos Mamoulis, “Similarity Search based on Geo-footprints”, EDBT (March 18, 2024), pp. 610–616, https://dx.doi.org/10.48786/edbt.2024.52

[2] Jiang Bian, Dayong Tian, Yuanyan Tang, and Dacheng Tao. 2019. Trajectory Data Classification: A Review. ACM Trans. Intell. Syst. Technol. 10, 4, Article 33 (July 2019), 34 pages. https://doi.org/10.1145/3330138

[3] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient processing of spatial joins using R-trees. SIGMOD Rec. 22, 2 (June 1, 1993), 237–246. https://doi.org/10.1145/170036.170075

[4] Shen, Li & Stopher, Peter. (2014). Review of GPS travel survey and GPS data-processing methods. Transport Reviews. 34. 316-334. 10.1080/01441647.2014.903530.

[5] F. Zafari, A. Gkelias and K. K. Leung, "A Survey of Indoor Localization Systems and Technologies," in IEEE Communications Surveys & Tutorials, vol. 21, no. 3, pp. 2568-2599, thirdquarter 2019, doi: 10.1109/COMST.2019.2911558.

[6] Chen, J. & Hu, T. & Zhang, P. & Shi, W. & Shan, J.. (2014). Trajectory Clustering for People's Movement Pattern Based on Crowd Souring Data. ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. XL-2. 55-62. 10.5194/isprsarchives-XL-2-55-2014.

[7] Y. Zhang, Y. Li and W. Ji, "A Trajectory-Based User Movement Pattern Similarity Measure for User Identification," in IEEE Transactions on Network Science and Engineering, vol. 10, no. 6, pp. 3834-3845, Nov.-Dec. 2023, doi: 10.1109/TNSE.2023.3274516.

[8] Wesolowski A, Eagle N, Tatem AJ, Smith DL, Noor AM, Snow RW, Buckee CO. Quantifying the impact of human mobility on malaria. Science. 2012 Oct 12;338(6104):267-70. doi: 10.1126/science.1223467. PMID: 23066082; PMCID: PMC3675794.

[9] Z. Wang, M. Lu, X. Yuan, J. Zhang and H. Van De Wetering, "Visual Traffic Jam Analysis Based on Trajectory Data," in IEEE Transactions on Visualization and Computer Graphics, vol. 19, no. 12, pp. 2159-2168, Dec. 2013, doi: 10.1109/TVCG.2013.228.

[10] B. Zhou, X. Wang and X. Tang, "Understanding collective crowd behaviors: Learning a Mixture model of Dynamic pedestrian-Agents," 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, 2012, pp. 2871-2878, doi: 10.1109/CVPR.2012.6248013.

[11] https://www.marketsandmarkets.com/Market-Reports/indoor-location-market-989.html

[12] D. Dardari, P. Closas and P. M. Djurić, "Indoor Tracking: Theory, Methods, and Technologies," in *IEEE Transactions on Vehicular Technology*, vol. 64, no. 4, pp. 1263-1278, April 2015, doi: 10.1109/TVT.2015.2403868.

[13] W. Yao *et al.*, "Impact of GPS Signal Loss and Its Mitigation in Power System Synchronized Measurement Devices," in *IEEE Transactions on Smart*

Grid, vol. 9, no. 2, pp. 1141-1149, March 2018, doi: 10.1109/TSG.2016.2580002.

[14] Dong, Jia & Li, Li & Huang, Jiaxuan & Han, Dongqing. (2017). ISOVIST BASED ANALYSIS OF SUPERMARKET LAYOUT——Verification of Visibility Graph Analysis and Multi-Agent Simulation.

[15] Krishna Chintalapudi, Anand Padmanabha Iyer, and Venkata N. Padmanabhan. 2010. Indoor localization without the pain. In Proceedings of the sixteenth annual international conference on Mobile computing and networking (MobiCom '10). Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/1859995.1860016

[16] Alper Yilmaz, Omar Javed, and Mubarak Shah. 2006. Object tracking: A survey. ACM Comput. Surv. 38, 4 (2006), 13–es. https://doi.org/10.1145/1177352.1177355

[17] Li, Fan & Zhao, Chunshui & Ding, Guanzhong & Gong, Jian & Liu, Chenxing & Zhao, Feng. (2012). A Reliable and accurate indoor localization method using phone inertial sensors. UbiComp'12 - Proceedings of the 2012 ACM Conference on Ubiquitous Computing. 421-430. 10.1145/2370216.2370280.

[18] Bršćić, Dražen & Kanda, Takayuki & Ikeda, Tetsushi & Miyashita, Takahiro. (2013). Person Tracking in Large Public Spaces Using 3-D Range Sensors. Human-Machine Systems, IEEE Transactions on. 43. 522-534. 10.1109/THMS.2013.2283945.

[19] P. Bahl and V. N. Padmanabhan, "RADAR: an in-building RF-based user location and tracking system," *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, Tel Aviv, Israel, 2000, pp. 775-784 vol.2, doi: 10.1109/INFCOM.2000.832252.

[20] L. Bai, F. Ciravegna, R. Bond and M. Mulvenna, "A Low Cost Indoor Positioning System Using Bluetooth Low Energy," in *IEEE Access*, vol. 8, pp. 136858-136871, 2020, doi: 10.1109/ACCESS.2020.3012342.

[21] Kong, Xiangjie & Chen, Qiao & Hou, Mingliang & Wang, Hui & Xia, Feng. (2023). Mobility trajectory generation: a survey. Artificial Intelligence Review. 56. 10.1007/s10462-023-10598-x.

[22] Huan Li, Hua Lu, Xin Chen, Gang Chen, Ke Chen, and Lidan Shou. 2016. Vita: a versatile toolkit for generating indoor mobility data for real-world buildings. Proc. VLDB Endow. 9, 13 (September 2016), 1453–1456. https://doi.org/10.14778/3007263.3007282

[23] Wang, Sheng & Bao, Zhifeng & Culpepper, J. & Cong, Gao. (2021). A Survey on Trajectory Data Management, Analytics, and Learning. ACM Computing Surveys. 54. 1-36. 10.1145/3440207.

[24] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Timos Sellis, and Xiaolin Qin. 2019. Fast large-scale trajectory clustering. Proc. VLDB Endow. 13, 1 (September 2019), 29–42. https://doi.org/10.14778/3357377.3357380

[25] Siyuan Liu, Yunhuai Liu, Lionel M. Ni, Jianping Fan, and Minglu Li. 2010. Towards mobility-based clustering. In Proceedings of the 16th ACM

SIGKDD international conference on Knowledge discovery and data mining (KDD '10). Association for Computing Machinery, New York, NY, USA, 919–928. https://doi.org/10.1145/1835804.1835920

[26] C. Guo, B. Yang, J. Hu and C. Jensen, "Learning to Route with Sparse Trajectory Sets," 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 2018, pp. 1073-1084, doi: 10.1109/ICDE.2018.00100.

[27] Buchin, Maike & Driemel, Anne & Kreveld, Marc & Sacristan, Vera. (2011). Segmenting trajectories: A framework and algorithms using spatiotemporal criteria. J. Spatial Information Science. 3. 33-63. 10.5311/JOSIS.2011.3.66.

[28] Issa, Hamza & Cagnacci, Francesca. (2014). Extracting stay regions with uncertain boundaries from GPS trajectories: a case study in animal ecology. 10.1145/2666310.2666417.

[29] Reaz Uddin, Mehnaz Tabassum Mahin, Payas Rajan, Chinya V. Ravishankar, and Vassilis J. Tsotras. 2023. Dwell Regions: Generalized Stay Regions for Streaming and Archival Trajectory Data. ACM Trans. Spatial Algorithms Syst. 9, 2, Article 9 (June 2023), 35 pages. https://doi.org/10.1145/3543850

[30] Xin Ding, Lu Chen, Yunjun Gao, Christian S. Jensen, and Hujun Bao. 2018. UlTraMan: a unified platform for big trajectory data management and analytics. Proc. VLDB Endow. 11, 7 (March 2018), 787–799. https://doi.org/10.14778/3192965.3192970

[31] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Zizhe Xie, Qizhi Liu, and Xiaolin Qin. 2018. Torch: A Search Engine for Trajectory Data. In The 41st International ACM SIGIR Conference on Research &amp; Development in Information Retrieval (SIGIR '18). Association for Computing Machinery, New York, NY, USA, 535–544. https://doi.org/10.1145/3209978.3209989

[32] Zhou Shao, Muhammad Aamir Cheema, David Taniar, and Hua Lu. 2016. VIP-Tree: an effective index for indoor spatial queries. Proc. VLDB Endow. 10, 4 (November 2016), 325–336. https://doi.org/10.14778/3025111.3025115

[33] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. 1998. Scalable Sweeping-Based Spatial Join. In Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 570–581.

[34] Mamoulis N. (2012). Spatial data management (1st ed.). Morgan & Claypool Publishers. https://doi.org/10.1007/978-3-031-01884-8

[35] https://github.com/nushoin/RTree/

[36] https://superliminal.com/sources/

[37] https://github.com/acmwl/CustomisableTrajectoryGenerator

[38] https://www.edrawsoft.com/template-mall-floor-plan.html