Finding Frequently Visited Indoor POIs Using Symbolic Indoor Tracking Data

Hua Lu Chenjuan Guo Bin Yang Christian S. Jensen
Department of Computer Science, Aalborg University, Denmark
{luhua, cguo, byang, csj}@cs.aau.dk

ABSTRACT

Indoor tracking data is being amassed due to the deployment of indoor positioning technologies. Analysing such data discloses useful insights that are otherwise hard to obtain. For example, by studying tracking data from an airport, we can identify the shops and restaurants that are most popular among passengers. In this paper, we study two query types for finding frequently visited Points of Interest (POIs) from symbolic indoor tracking data. The snapshot query finds those POIs that were most frequently visited at a given time point, whereas the interval query finds such POIs for a given time interval. A typical example of symbolic tracking is RFID-based tracking, where an object with an RFID tag is detected by an RFID reader when the object is in the reader's detection range. A symbolic indoor tracking system deploys a limited number of proximity detection devices, like RFID readers, at preselected locations, covering only part of the host indoor space. Consequently, symbolic tracking data is inherently uncertain and only enables the discrete capture of the trajectories of indoor moving objects in terms of coarse regions. We provide uncertainty analyses of the data in relation to the two kinds of queries. The outcomes of the analyses enable us to design processing algorithms for both query types. An experimental evaluation with both real and synthetic data suggests that the framework and algorithms enable efficient and scalable query processing.

1. INTRODUCTION

Indoor spaces such as shopping malls, office buildings, libraries, metro stations, and airports serve as the settings of significant parts of people's daily lives. The indoor movements of people are increasingly datafied due to advances in indoor positioning [1]. As a result, a new type of data—indoor tracking data—is being accumulated in a variety of formats determined by the particular indoor positioning technologies used.

As in the case for outdoor tracking data [3], analyzing indoor tracking data can reveal how different parts of an indoor space are used by its inhabitants, e.g., the number of visits to a particular part of space over time can be determined. The findings are potentially useful in practical scenarios. For example, the lease prices of different shop locations in a large shopping mall may be set according to the numbers of people passing by the location. As another example, information on the behavior of past visitors to a museum with multiple exhibitions may be used for making recommendations to new visitors and for planning. These and other example scenarios may benefit from flow counting using indoor tracking data.

Flow counting is non-trivial in indoor spaces, where new, unique technical challenges exist that are different from those in outdoor contexts. These in turn call for novel data management techniques.

First of all, indoor positioning systems differ fundamentally from GPS that is prevalent outdoors but that generally does not work indoors. Having to use wireless technologies such as Wi-Fi, Bluetooth, and RFID that originally are designed for data communication, indoor positioning systems work according to different principles and offer positioning accuracies that are below that of GPS. For example, in an RFID based system, an object with an RFID tag is detected by an RFID reader only when the object is in the reader's detection range. Due to their costs, a limited number of readers are deployed, covering only part of the host indoor space. Consequently, the tracking data is inherently uncertain and only enables the discrete capture of the trajectories of indoor moving objects in terms of coarse regions. Such uncertainty renders flow counting techniques based on GPS data unsuitable for indoor spaces.

Further, indoor spaces are characterized by entities like doors, rooms, and hallways that enable and constrain the movements of indoor objects. Compared to outdoor Euclidean or spatial network space, indoor spaces have more complex topologies. When counting flows in indoor spaces, their complex topologies must be taken into account.

This paper considers flow counting based on symbolic indoor tracking data where object locations are captured as circular regions centered at pre-selected indoor locations. Such circular regions correspond to the detection ranges of proximity detection devices, e.g., RFID readers. We define appropriate ways of counting flows based on the uncertain tracking data. Our definitions capture probabilistically how frequently indoor POIs are visited by tracked visitors. Based on the definitions of indoor flow counting, we define two query types for finding indoor POIs that are visited frequently at a given time point and during a given time range, respectively. We analyze carefully the data uncertainty with respect to the query types, which enables us to design algorithms for both query types. We use synthetic and real data to evaluate the proposals experimentally. The experimental results show that our proposals are efficient and scalable.

We make the following contributions in this paper.

- We define indoor flow counting methods on symbolic indoor tracking data, and we formulate two types of queries for finding frequently visited indoor POIs.
- We derive query related object uncertainty regions by analysing the relationship between the queries and the tracking data.
- We make use of the uncertainty analysis results to design algorithms for the two query types.
- We perform extensive experiments to evaluate the proposed techniques.

The remainder of the paper is organized as follows. Section 2 presents the format of symbolic indoor tracking data and formulates the research problems. Section 3 derives uncertainty regions for objects. Section 4 details the query processing algorithms. Section 5 reports on the experimental studies. Section 6 reviews the related work, and Section 7 concludes and discusses research directions.

2. PROBLEM FORMULATION

We formulate the research problems in this section. Section 2.1 details the symbolic indoor tracking data, and Section 2.2 gives the problem definitions. Table 1 lists notation used throughout the paper.

Symbol	**Meaning**
p	An indoor POI
P	A set of indoor POIs
o	An indoor moving object
O	A set of indoor moving objects
t	A time point
rd	An indoor tracking record
$\Phi_t(p)$	The flow of p at time t
$\Phi_{t_s,t_e}(p)$	The flow of p during interval $[t_s, t_e]$
V_{max}	The maximum speed of indoor moving objects

Table 1: Notation

2.1 Symbolic Indoor Tracking Data

In symbolic indoor object tracking, raw position readings are reported in the format $\langle objectID, deviceID, t\rangle$. Such a 3-tuple means that the object identified by *objectID* is seen by the device *deviceID* at time t. As the positioning works at a configured sampling frequency, an object is typically seen in multiple, consecutive raw readings by the same device. Such consecutive raw readings are merged [10] to form a tracking record of the form $\langle ID, objectID, deviceID, t_s, t_e\rangle$. Such a tracking record means that the object is continuously seen by the device from time t_s to time t_e, i.e., t_s is the start time for the continuous detection and t_e is the end time. An object tracking table (OTT) is used to store such historical tracking records, as exemplified in Table 2, where attribute *ID* is a record identifier.

ID	*objectID*	*deviceID*	t_s	t_e
rd_1	o_1	dev_4	t_1	t_2
rd_2	o_2	dev_4	t_1	t_2
rd_3	o_1	dev_2	t_5	t_6
rd_4	o_2	dev_1	t_7	t_8
rd_5	o_1	dev_1	t_9	t_{10}
rd_6	o_1	dev_{12}	t_{15}	t_{16}
rd_7	o_2	dev_{13}	t_{20}	t_{21}
rd_8	o_1	dev_{13}	t_{21}	t_{22}
rd_9	o_2	dev_{13}	t_{29}	t_{30}
...	...	...	...	...

Table 2: Object Tracking Table (OTT)

2.2 Problem Definitions

We assume that each indoor POI p has some fixed extent modeled by a polygon, and for simplicity, we equate a POI p with its polygon. As an entire indoor space is typically not fully covered by proximity detection devices like RFID readers due to economic constraints, there are considerable time intervals during which an object is not seen at all. Consequently, flow counting is not straightforward in the setting of uncertain indoor tracking data. Since object locations are uncertain, exact counting of objects does not make sense. Instead, we estimate how many objects that appeared in an indoor POI's range at a particular past time point or during a past time range. For that purpose, we need to determine an indoor moving object's *uncertainty region* that tells where the object can possibly be located. We differentiate between snapshot and interval uncertainty regions.

Given an object o, we use $UR(o, t)$ to denote o's *snapshot uncertainty region* at time point t. In particular, $UR(o, t)$ captures an indoor region in which object o can possibly be at time t. Next, we use $UR(o, [t_s, t_e])$ to denote o's *interval uncertainty region* during time interval $[t_s, t_e]$. Here, $UR(o, [t_s, t_e])$ captures the indoor region in which object o can possibly be during time interval $[t_s, t_e]$. We describe how to derive these uncertainty regions in Section 3. Based on the uncertainty regions, we define the following concepts for flow counting.

First, we consider an object o's presence in the range of an indoor POI p. The presence stipulates how we "count" objects for a POI p.

Definition 1 (Object Presence). *During a given time interval $[t_s, t_e]$, an object o's* interval presence *in a POI p is*

$$\phi_{t_s,t_e,p}(o) = \frac{area(UR(o, [t_s, t_e]) \cap p)}{area(p)}. \quad (1)$$

Similarly, $\phi_{t,p}(o)$ is defined by replacing $UR(o, [t_s, t_e])$ with $UR(o, t)$. The idea of object presence is to capture the intersection between the object's uncertainty region and the POI's range. Intuitively, the larger the intersection, the more likely it is that the object was in the POI. For an arbitrary object o and an arbitrary POI p, it is apparent that $0 \le \phi_{t,p}(o) \le 1$ and $0 \le \phi_{t_s,t_e,p}(o) \le 1$. Therefore, o's object presence can be regarded as the probability that o is in POI p at t or during $[t_s, t_e]$.

Next, we define the concept of flow for indoor POIs.

Definition 2 (Flow). *Suppose O is the set of all objects in an indoor space of interest. Given an indoor POI p and a time interval $[t_s, t_e]$, p's* interval flow *is defined as*

$$\Phi_{t_s,t_e}(p) = \sum_{o \in O} \phi_{t_s,t_e,p}(o). \qquad (2)$$

Similarly, $\Phi_t(p)$ is defined by replacing $\phi_{t_s,t_e,p}(o)$ with $\phi_{t,p}(o)$. The flow definitions perform weighted counting of objects that stay in POI p at time t or during time interval $[t_s, t_e]$, and the weight assigned to each object is its object presence.

With the above definitions, we formulate two types of queries that return the top-k frequently visited indoor POIs.

Problem 1 (Snapshot Top-k Indoor POIs Query). *Given a set P of indoor POIs, a time point t, and an integer k $(0 < k \leq |P|)$, return a k-subset P_T of P such that $\forall p \in P_T$ $(\forall p' \in P \setminus P_T$ $(\Phi_t(p) \geq \Phi_t(p')))$.*

Problem 2 (Interval Top-k Indoor POIs Query). *Given a set P of indoor POIs, a time interval $[t_s, t_e]$, and an integer k $(0 < k \leq |P|)$, return a k-subset P_T of P such that $\forall p \in P_T$ $(\forall p' \in P \setminus P_T$ $(\Phi_{t_s,t_e}(p) \geq \Phi_{t_s,t_e}(p')))$.*

These queries return the indoor POIs that are visited by the largest number of visitors at a time point or during a time interval. This functionality has many applications. For example, it can be used to identify the most popular shops in a shopping mall. Shop rental fees can take such information into account. As another example, it can be used to identify possible bottlenecks that slow down movement in an airport.

3. DERIVING UNCERTAINTY REGIONS

We elaborate on how to derive uncertainty regions for an given object and specified time parameters. Section 3.1 presents the basic terminology, including uncertainty regions for snapshot queries. Section 3.2 focuses on uncertainty regions for interval queries. Section 3.3 addresses how to finalize the uncertainty regions in a given indoor space.

3.1 Basic Terminology

We first cover basic terminology adopted from previous work [10, 17, 25] that is necessary for understanding the paper's contributions.

3.1.1 *Tracking States*

Given a time point t, an object o may be in either an active state or an inactive state [25]. Specifically, if there is a tracking record rd_{cov} for o in OTT (see Table 2 in Section 2.1) covering time t, we say o is in an *active state* at t. We use rd_{pre} to refer to rd_{cov}'s predecessor record in OTT. If no tracking record for o exists in OTT covering time t, object o is in an *inactive state* at t. In this case, we identify two relevant tracking records. Record rd_{pre} is the tracking record for o immediately before o becomes inactive. Record rd_{suc} is the tracking record for o immediately after o leaves the inactive state, i.e., rd_{suc} is the first tracking record when o becomes active again after time t. Note that $rd_{pre}.t_e < t < rd_{suc}.t_s$ if object o is inactive at time t.

Object o_1's tracking records in Table 2 are plotted along the time axis in Figure 1 (originally from [17]). It is in an active state at time t_5, and it is in an inactive state at time t_{19}. Relevant particular tracking records are also marked in

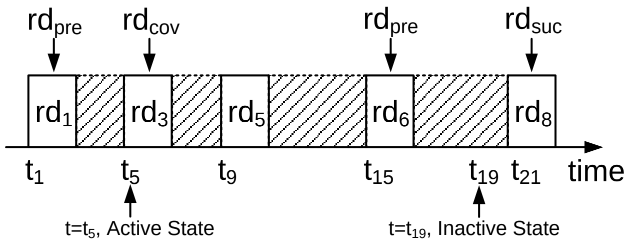


Figure 1: States in Symbolic Object Tracking

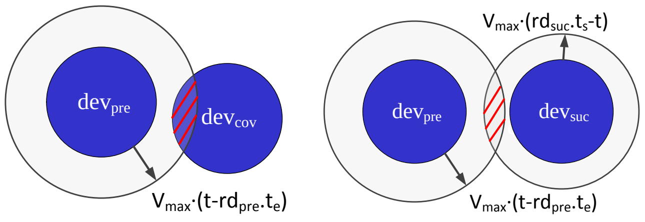
the figure. For simplicity, we use dev_{pre} (dev_{cov} or dev_{suc}) to refer to rd_{pre}'s (rd_{cov}'s or rd_{suc}'s) device.

An object is either in an active or an inactive state at a time point t, whereas it may change state during a time interval $[t_s, t_e]$. In Figure 1, object o_1 changes state five times during $[t_5, t_{19}]$.

3.1.2 *Snapshot Uncertainty Regions*

Two cases exist for the snapshot uncertainty region of an object o at a time point t [17]. We suppose that V_{max} is the maximum speed that object o can move at in the given indoor space.

Case 1: Object o is in an active state at t. In this case, $UR(o, t) = Ring(dev_{pre}, V_{max} \cdot (t - rd_{pre}.t_e))$ [1]$\cap dev_{cov}.Range$. I.e., o's uncertainty region is the intersection of dev_{cov}'s detection range and the ring in which o can be after leaving dev_{pre}'s detection range. Specifically, this ring is determined by dev_{pre}'s range and the maximum distance o can move from $rd_{pre}.t_e$ to t. This case is illustrated in Figure 2(a).



(a) Active State (b) Inactive State

Figure 2: Snapshot Uncertainty Regions

Case 2: Object o is in an inactive state at t. In this case, $UR(o, t) = Ring(dev_{pre}, V_{max} \cdot (t - rd_{pre}.t_e)) \cap Ring(dev_{suc}, V_{max} \cdot (rd_{suc}.t_s - t))$. Here, $UR(o, t)$ is the intersection of two rings: one involves rd_{pre} and is the same as that in Case 1; the other involves rd_{suc} and the maximum distance o can move from t to $rd_{suc}.t_s$. This case is illustrated in Figure 2(b).

3.1.3 *Uncertainty Region Involving Two Consecutive Readings*

Without loss of generality, let rd_i and rd_j be two consecutive tracking records for object o. Records rd_i and rd_j involve devices dev_i and dev_j, respectively. For time intervals $[rd_i.t_s, rd_i.t_e]$ and $[rd_j.t_s, rd_j.t_e]$, object o is in dev_i's and dev_j's detection range, respectively. For time interval $[rd_i.t_e, rd_j.t_s]$, object o's location can be constrained by an

[1] $Ring(dev, \rho)$ denotes the ring whose inner circle is device dev's detection circle and whose outer circle extends the inner circle's radius by ρ.

extended ellipse [10] whose two foci are two points on the boundaries of the two detection ranges (see the two circular regions in dark in Figure 3). Furthermore, the length of the ellipse's major axis is $2a = V_{max} \cdot (rd_j.t_s - rd_i.t_e)$. Such an extended ellipse is illustrated in Figure 3. Object o's uncertainty region for a time interval is represented by the extended ellipse excluding the two circular regions for the two proximity detection devices. The details of deriving such an extended ellipse can be found in the literature [10, 19]. We use $\Theta(dev_i, dev_j, rd_i.t_e, rd_j.t_s)$ to refer to the complete region covered by the ellipse, which will be used in our subsequent discussions.

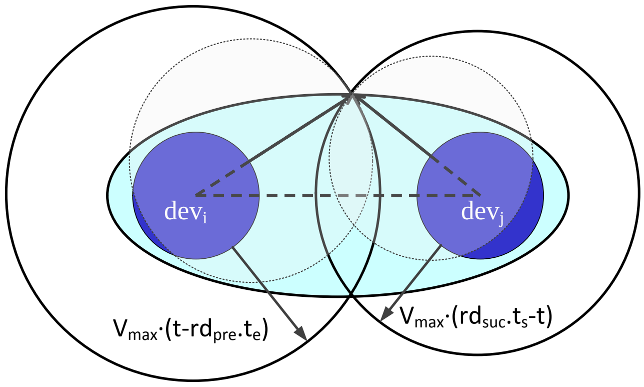


Figure 3: Uncertainty Region Examples

3.2 Interval Uncertainty Regions

In order to derive the uncertainty region of object o in a given time interval $[t_s, t_e]$, we need to find all its relevant tracking records in OTT. Temporally, all those records form a chain of detections of object o. In particular, we need to find the start record, the end record, and all the records in-between for object o. We use rd_s and rd_e to refer to the start (first in the chain) and end (the last) records, respectively. Although there may be multiple in-between records, we use rd_b to refer to a concrete record between rd_s and rd_e when it is of particular interest in our discussion. Table 3 gives the start and end records for all cases regarding object o's state at t_s and t_e.

t_e / t_s	Active		Inactive	
	rd_s	rd_e	rd_s	rd_e
Active	$rd_{cov}(t_s)$	$rd_{cov}(t_e)$	$rd_{cov}(t_s)$	$rd_{suc}(t_e)$
Inactive	$rd_{pre}(t_s)$	$rd_{cov}(t_e)$	$rd_{pre}(t_s)$	$rd_{suc}(t_e)$

Table 3: Start and End Records for $[t_s, t_e]$

Next, we derive $UR(o, [t_s, t_e])$, object o's uncertainty region during $[t_s, t_e]$, for the four cases given in Table 3.

Case 1: Object o is active at both t_s and t_e, i.e., $rd_s = rd_{cov}(t_s)$ and $rd_e = rd_{cov}(t_e)$. Without loss of generality, we assume that there are two records in-between. An illustration is shown in Figure 4, where dev_s is $rd_s.deviceID$ and dev_e is $rd_e.deviceID$. Object o's uncertainty region is then the union of the ellipse regions associated with the consecutive tracking records from rd_s to rd_e. Formally, $UR(o, [t_s, t_e]) = \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i.t_e, rd_{i+1}.t_s)$, where rd_i is a tracking record from sequence $R = \langle rd_{cov}(t_s), rd_{b1}, \ldots, rd_{cov}(t_e)\rangle$ and $dev_i = rd_i.deviceID$.

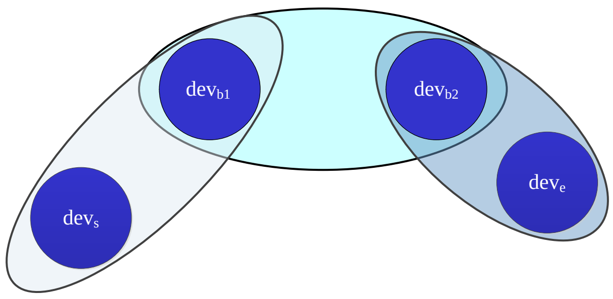


Figure 4: Interval Uncertainty Region for Case 1

Case 2: Object o is inactive at t_s but active at t_e, i.e., $rd_s = rd_{pre}(t_s)$ and $rd_e = rd_{cov}(t_e)$. As for Case 1, an illustration is shown in Figure 5. Here, we need to pay particular attention to the time interval $[t_s, rd_{b1}.t_s]$ before object o becomes detected by device dev_{b1}. For this interval, due to the maximum speed constraint, the object can only be in the intersection of the ellipse region $\Theta_s = \Theta(dev_s, dev_{b1}, rd_s.t_e, rd_{b1}.t_s)$ and the ring captured as $Ring_s = Ring(dev_{b1}, V_{max} \cdot (rd_{b1}.t_s - t_s))$.

As a result, object o's uncertainty region for the entire interval $[t_s, t_e]$ is this intersection unioned with the union of all other ellipse regions associated with all other consecutive tracking records from rd_{b1} to $rd_e = rd_{cov}(t_e)$. Formally, $UR(o, [t_s, t_e]) = (\Theta_s \cap Ring_s) \cup \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i.t_e, rd_{i+1}.t_s)$, where rd_i is a tracking record from sequence $R = \langle rd_{b1}, \ldots, rd_{cov}(t_e)\rangle$ and $dev_i = rd_i.deviceID$.

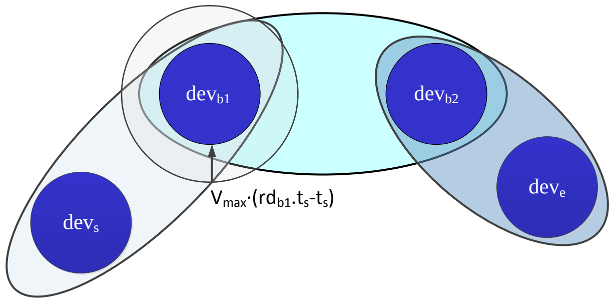


Figure 5: Interval Uncertainty Region for Case 2

Case 3: Object o is active at t_s but inactive at t_e, i.e., $rd_s = rd_{cov}(t_s)$ and $rd_e = rd_{suc}(t_e)$. An illustration is shown in Figure 6. In this case, we need to pay particular attention to the time interval $[rd_{b2}.t_e, t_e]$ after object o is last seen by device $rd_{b2}.deviceID$. For this interval, due to the maximum speed constraint, the object can only be in the intersection of the ellipse region $\Theta_e = \Theta(dev_{b2}, dev_e, rd_{b2}.t_e, rd_e.t_s)$ and the ring $Ring_e = Ring(dev_{b2}, V_{max} \cdot (t_e - rd_{b2}.t_e))$.

As a result, object o's uncertainty region for the entire interval $[t_s, t_e]$ is the above intersection unioned with the union of all other ellipse regions associated with all other consecutive tracking records from $rd_s = rd_{cov}(t_s)$ to rd_{b2}. Formally, $UR(o, [t_s, t_e]) = (\Theta_e \cap Ring_e) \cup \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i.t_e, rd_{i+1}.t_s)$, where rd_i is a tracking record from sequence $R = \langle rd_{cov}(t_s), rd_{b1}, \ldots, rd_{b2}\rangle$ and $dev_i = rd_i.deviceID$.

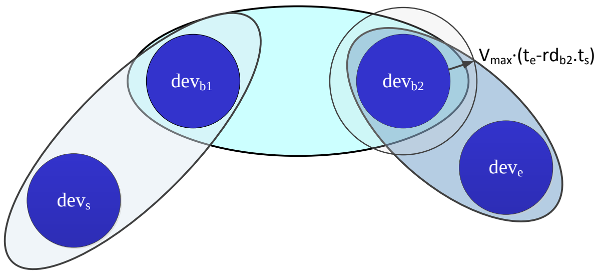


Figure 6: Interval Uncertainty Region for Case 3

Case 4: Object o is inactive at both t_s and t_e, i.e., $rd_s = rd_{pre}(t_s)$ and $rd_e = rd_{suc}(t_e)$. An illustration is shown in Figure 7. This case combines the handling of the beginning and end from Cases 2 and 3, respectively. Therefore, object o's uncertainty region in $[t_s, t_e]$ is $UR(o, [t_s, t_e]) = (\Theta_s \cap Ring_s) \cup (\Theta_e \cap Ring_e) \cup \bigcup_{i=1..|R|-1} \Theta(dev_i, dev_{i+1}, rd_i.t_e, rd_{i+1}.t_s)$, where rd_i is a tracking record from sequence $R = \langle rd_{b1}, \ldots, rd_{b2}\rangle$ and $dev_i = rd_i.deviceID$.

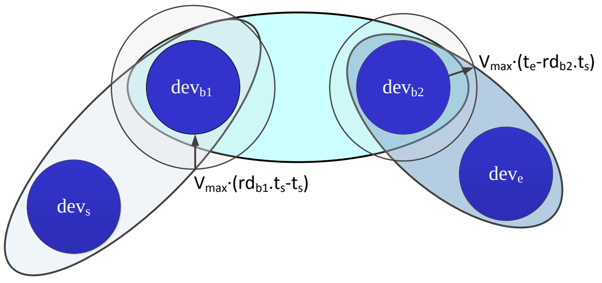

dev$_{b1}$
dev$_{b2}$
$V_{max}\cdot(t_e\text{-}rd_{b2}.t_s)$
dev$_e$
dev$_s$
$V_{max}\cdot(rd_{b1}.t_s\text{-}t_s)$

Figure 7: Interval Uncertainty Region for Case 4

3.3 Indoor Topology Check

Our coverage so far does not consider the topology of the indoor space. Specifically, we need to check $UR(o, t)$ against the given indoor space and exclude all parts of the space that are not accessible to object o at time t. Also, we need to check $UR(o, [t_s, t_e])$ against the given indoor space and exclude all parts of the space that are not accessible to object o from time t_s and t_e without exiting $UR(o, [t_s, t_e])$. For each type of uncertainty region, the part that remains after the indoor topology check is object o's uncertainty region.

Examples are shown in Figure 8, where the shaded parts must be excluded from the object's uncertainty regions. In Figure 8(a), an object o is inactive at time point t. Suppose that it was detected by device 1 at time $t_1 < t$ and then by device 3 after t. According to the discussion illustrated in Figure 2(b), o's snapshot uncertainty region $UR(o, t)$ is the intersection of the two large circular regions constrained by the maximum speed V_{max}. However, the shaded part should be excluded as it is too far away for object o to be able to reach it at time point t. After leaving device 1's detection range, object o must go through door 2 to enter the shaded part, which would yield an indoor walking distance that exceeds the maximum Euclidean distance o can move from t_1 to t, i.e., $V_{max} \cdot (t - t_1)$. Therefore, this part should be excluded from $UR(o, t)$. If the topology check is skipped in this example, the object would be "counted" for room 2 whose flow in turn would be increased incorrectly. Such a miscalculation can result in room 2 entering a top-k query result as a false positive.

It is worth noting that if room 2 had had a door in the region given by the intersection of the two large circular regions (see Figure 8(a)), further checking would be needed to exclude parts of space that are too far away for an object in those parts to be able to move from one device to the other via the door. Based on the maximum speed V_{max}, we can calculate the earliest time t_2 ($t_1 < t_2 < t$) for an object to reach the assumed door. Then, moving from the door, the possible region for the object to reach before t is constrained by the distance $V_{max} \cdot (t - t_2)$. Any part of space beyond that distance from the assumed door should be excluded from the object's uncertainty region.

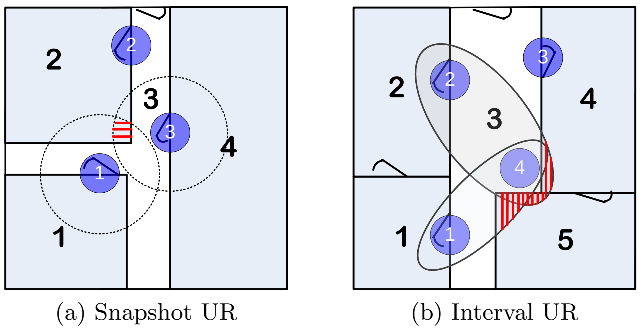


Figure 8: Indoor Topology Check

In Figure 8(b), an object o is detected by devices 1, 4, and then 2. Due to the maximum speed constraint, the object cannot have entered the shaded regions (in rooms 4 and 5). Therefore, these regions should be excluded from object o's interval uncertainty region $UR(o, [t_s, t_e])$. Otherwise, the flows of rooms 4 and 5 would be incorrectly increased and they may enter the top-k result as false positives.

In our framework, we do such indoor topology checks to capture the uncertainty regions for indoor moving objects better. Specifically, after an uncertainty region UR is obtained, we divide it into several disconnected parts according to the indoor topology. For each such part, we check its indoor distance from the involved devices. If the indoor distance exceeds the corresponding maximum Euclidean distance, the part is excluded. For simplicity, we use $UR(o, t)$ and $UR(o, [t_s, t_e])$ to refer to the object's uncertainty region also after the topology check in the following.

Remark For the sake of conciseness in our presentation, we implicitly assume that the detection ranges of proximity detection devices do not overlap. Overlapping detection ranges can be accommodated by making only slight changes in the corresponding low-level geometric computations; no changes are needed to the overall process of deriving the uncertainty regions for objects. Therefore, the uncertainty analysis presented in this section and the algorithms to be presented in the next section are able to accommodate overlapping detection ranges. We omit the low-level details here in order to keep our discussion concentrated and concise.

4. TOP-k INDOOR POI ALGORITHMS

We proceed to design query processing algorithms by making use of the uncertainty regions derived in Section 3. Section 4.1 presents the indexes we use for query processing.

Sections 4.2 and 4.3 detail the algorithms for snapshot and interval queries, respectively.

4.1 Indexes

We use two R-tree [4] based indexes for the data to facilitate query processing—one for the tracking data and one for the indoor POIs.

The object tracking table (OTT) (refer to Table 2) is indexed by an augmented 1D R-tree named A1R-tree [17] on the temporal attributes as follows. Let rd_c be a tracking record for object o in OTT. Such a record rd_c is indexed by an A1R-tree leaf entry of the form $(t^{\vdash}, t^{\dashv}, Ptr_p, Ptr_c)$. Specifically, Ptr_c is a pointer to record rd_c, and Ptr_p points to record rd_p that is o's previous record in OTT, i.e., rd_p is rd_c's predecessor for the same object. In addition, we set $t^{\vdash} = rd_p.t_e$ and $t^{\dashv} = rd_c.t_e$. We call $(t^{\vdash}, t^{\dashv}]$ (i.e., $(rd_p.t_e, rd_c.t_e]$) an augmented tracking time interval. A non-leaf entry in an A1R-tree is of the form $(t^{\vdash}, t^{\dashv}, cp)$, where cp is a pointer to a child node and $[t^{\vdash}, t^{\dashv}]$ is the minimum bounding interval that contains all time intervals in the child node.

With an A1R-tree, we are able to efficiently obtain all the tracking records relevant to the uncertainty region determination described in Section 2. For a snapshot uncertainty region at query time t, a point query with t as the parameter via the A1R-tree will return a leaf entry $le = (t^{\vdash}, t^{\dashv}, Ptr_p, Ptr_c)$ where $(t^{\vdash}, t^{\dashv}]$ covers t. If $le.Ptr_p.t_e < t < le.Ptr_c.t_s$, the object is inactive at time t, and $le.Ptr_p$ ($le.Ptr_c$) points to rd_{pre} (rd_{suc}) (see Figures 1 and 2(b)). Otherwise, the object is active at time t ($le.Ptr_c.t_s \leq t \leq le.Ptr_c.t_e$), and $le.Ptr_c$ points to the rd_{cov} (see Figures 1 and 2(a)).

For the uncertainty region in a query time interval $[t_s, t_e]$, a range query with $[t_s, t_e]$ as the parameter via the A1R-tree returns a series of leaf entries such that the first leaf entry's augmented tracking time interval covers t_s and that of the last leaf entry covers t_e. Furthermore, the first (last) leaf entry contains the pointer to the particular first (last) tracking record needed for the four cases presented in Section 3.2, depending on the case encountered. All in-between tracking records, if any, are accordingly obtained through those in-between leaf entries.

The set of indoor POIs is indexed by another R-tree, called R_P. For simplicity, we consider one floor and therefore use a 2D R-tree to index all indoor POIs in our implementation. Nevertheless, our analysis of uncertainty regions as well as the query processing techniques can be extended to multi-floor cases.

4.2 Snapshot Query Algorithms

4.2.1 Iterative Algorithm

A straightforward approach to compute the snapshot query (Problem 1) is to compute the snapshot flow value for each POI p in the query and then return the k POIs with the highest flow values. This iterative algorithm is formalized in Algorithm 1.

The iterative algorithm uses a hash table *flows* to keep flow values for all POIs (lines 1–2). Using a point query on the A1R-tree R_O on the OTT (line 3), the algorithm obtains all the relevant leaf entries whose augmented tracking time interval contains the query time t, as described in Section 4.1. For each object o thus obtained from the OTT (lines 4–5), the algorithm derives the uncertainty region $UR(o,t)$ for either an inactive state (lines 6–9) or an active state (line 11). Then all POIs that intersect $UR(o,[t_s,t_e])$ are found (line 12). For each such POI p, its flow value is increased by the current object o's presence in p (lines 13–14). Finally, the top-k POIs are returned (line 15).

Algorithm 1 iterativeSnapshot(R-tree R_P for indoor POIs, A1R-tree R_O for OTT, time point t, integer k)

```
 1: initialize a hash table $flows : \{POI\} \rightarrow [0, +\infty]$
 2: for each POI $p$ do $flows[p] \leftarrow 0$
 3: LeafEntrySet $les \leftarrow R_O$.PointQuery($t$)
 4: for each leaf entry $le \in les$ do
 5:     $o \leftarrow le.Ptr_c.objectID$
 6:     $ring_1 \leftarrow Ring(le.Ptr_p.deviceID, V_{max} \cdot (t - le.Ptr_p.t_e))$
 7:     if $le.Ptr_p.t_e < t < le.Ptr_c.t_s$ then  ▷ The object is in an inactive state
 8:         $ring_2 \leftarrow Ring(le.Ptr_c.deviceID, V_{max} \cdot (le.Ptr_c.t_s - t))$
 9:         $UR(o,t) \leftarrow ring_1 \cap ring_2$
10:     else                          ▷ The object is in an active state
11:         $UR(o,t) \leftarrow ring_1 \cap le.Ptr_c.deviceID.Range$
12:     $ps \leftarrow R_P.IntersectionQuery(UR(o,t))$
13:     for each POI $p \in ps$ do
14:         $flows[p] \leftarrow flows[p] + \frac{Area(UR(o,t) \cap p)}{Area(p)}$
15: return the top-$k$ from $flows.keys$ with the highest values
```

4.2.2 Join Algorithm

The iterative algorithm suffers from two limitations. It iterates over all objects and relevant POIs, which may be inefficient. Also, it needs to compute a considerable number of uncertainty regions. This may not pay off, as some POIs may finally get very low overall flow values while having incurred complex uncertainty region computations. Motivated by these observations, we design a more efficient join based method that is formalized in Algorithm 2.

The join algorithm consists of three phases. The first (lines 1–11) builds an in-memory aggregate R-tree R_I for all objects whose augmented tracking interval covers query time t. These objects are again obtained by a point query on the A1R-tree (line 2). If an object o is inactive at t (line 5), we obtain the minimum bounding rectangles (MBRs) of the two (*pre* and *suc*) proximity detection devices' detection ranges, and extend each of them by the corresponding maximum possible distance (lines 6–7). The two extended MBRs are then merged to form the object's MBR (line 8). Otherwise, the MBR of device dev_{cov}'s range is used for the active state (lines 10). After the MBR is determined, the object o is inserted into tree R_I, where each node entry is augmented with a field *count* that is the number of all objects in the corresponding sub-tree.

The second phase (lines 12–18) is the initialization for joining the POI R-tree R_P and the aggregate object R-tree R_I. Here, the algorithm initializes a priority queue Q that gives higher priority to R_P node entries (groups of POIs) that potentially have higher flow values. In particular, each R_P entry e_P is associated with a join list of R_I entries whose MBRs overlap e_P's MBR. Note that for any POI p in e_P's sub-tree, those objects that can contribute to p's flow value can only come from such R_I entries. When the two tree roots are joined (line 13–18) initially, the *count* values in the R_I entries are used to upper bound the flow values as an object's presence in any POI never exceeds 1 (Definition 1).

The third phase carries out the join (lines 19–48). The

Algorithm 2 joinSnapshot(R-tree R_P for indoor POIs, A1R-tree R_O for OTT, time t, integer k)

```
 1: initialize an in-memory aggregate R-tree $R_I$
 2: LeafEntrySet $les \leftarrow R_O$.PointQuery($t$)
 3: for each leaf entry $le \in les$ do
 4:     $o \leftarrow le.Ptr_c.objectID$
 5:     if $le.Ptr_p.t_e < t < le.Ptr_c.t_s$ then          ▷ Inactive state
 6:         $mbr_1 \leftarrow$ extend MBR($le.Ptr_p.deviceID.Range$) by $V_{max} \cdot (t - le.Ptr_p.t_e)$
 7:         $mbr_2 \leftarrow$ extend MBR($le.Ptr_c.deviceID.Range$) by $V_{max} \cdot (le.Ptr_c.t_s - t)$
 8:         $mbr \leftarrow$ MBR($mbr_1, mbr_2$)
 9:     else
10:         $mbr \leftarrow$ MBR($le.Ptr_c.deviceID.Range$)
11:     insert $(o, mbr)$ into $R_I$
12: initialize a priority queue $Q$
13: for each entry $e_P$ in $R_P.root$ do
14:     $ubFlow \leftarrow 0$; $list \leftarrow \varnothing$
15:     for each entry $e_I$ in $R_I.root$ do
16:         if $e_P.mbr$ intersects $e_I.mbr$ then
17:             $ubFlow \leftarrow ubFlow + e_I.count$;  $list \leftarrow list \cup \{e_I\}$
18:     $Q.enqueue(\langle e_P, list, ubFlow\rangle)$
19: $result \leftarrow \varnothing$;  initialize a hash table $H_U$
20: while $Q$ is not empty do
21:     $\langle e_P, list\rangle \leftarrow Q.dequeue()$
22:     if $e_P$ is a leaf entry then
23:         if $list$ is null then
24:             add POI $e_P.object$ to $result$
25:             if $result = k$ then return $result$
26:         else
27:             if $list$ contain leaf entries then
28:                 $flow \leftarrow 0$
29:                 for each entry $e_I \in list$ do
30:                     if $H_U[e_I.object] = \varnothing$ then
31:                         $H_U[e_I.object] \leftarrow UR(e_I.object, t)$
32:                     $flow \leftarrow flow + \phi_{t,e_P.object}(e_I.object)$
33:                 if $flow \neq 0$ then $Q.enqueue(\langle e_P, null, flow\rangle)$
34:             else
35:                 expandList($e_P, list$)
36:     else
37:         if $list$ contain leaf entries then
38:             for each sub-entry $e'_P$ in $e_P.node$ do
39:                 $ubFlow \leftarrow 0$; $list2 \leftarrow \varnothing$
40:                 for each entry $e'_I \in list$ do
41:                     if $e'_P.mbr$ intersects $e'_I.mbr$ then
42:                         $ubFlow \leftarrow ubFlow + 1$
43:                         $list2 \leftarrow list2 \cup \{e'_I\}$
44:                 if $list2 \neq \varnothing$ then
45:                     $Q.enqueue(\langle e'_P, list2, ubFlow\rangle)$
46:         else
47:             for each sub-entry $e'_P$ in $e_P.node$ do
48:                 expandList($e'_P, list$)
```

join order is controlled by priority queue Q (lines 20–21). If the current R_P entry e_P is a leaf entry, it is processed as follows. If e_P's join list is empty, which means its concrete flow value has been calculated and the value is higher than those yet to be calculated, it is added to the result (line 24), and if the result contains k POIs, the algorithm terminates (line 25). Otherwise, the join list may contain leaf entries or non-leaf entries from R_I. In the former case (line 27), the flow value of e_P is calculated by deriving the uncertainty region and presence for each object in the join list (lines 28–32). If the flow value is non-zero, the POI in e_P is pushed back to Q with an empty join list (line 33). To avoid deriving uncertainty regions repeatedly for objects that appear in multiple join lists, we use a hash table H_U (lines 19 and 30–31) to store the uncertainty regions for objects. If e_P's join list contains non-leaf entries, procedure expandList (Algorithm 3) is called to join e_P with sub-entries from the join list. The procedure ensures that e_P is only associated with those R_I entries whose MBRs intersect e_P's (line 4), and it uses the *count* values in the R_I entries to estimate the upper bound of e_P's flow value (line 5).

Algorithm 3 expandList(Entry e_P in R-tree R_P for indoor POIs, List $list$ of entries in R-tree R_I)

```
1: $ubFlow \leftarrow 0$; $list2 \leftarrow \varnothing$
2: for each entry $e_I \in list$ do
3:     for each sub-entry $e'_I$ in $e_I.node$ do
4:         if $e_P.mbr$ intersects $e'_I.mbr$ then
5:             $ubFlow \leftarrow ubFlow + e'_I.count$
6:             $list2 \leftarrow list2 \cup \{e'_I\}$
7: if $list2 \neq \varnothing$ then
8:     $Q.enqueue(\langle e_P, list2, ubFlow\rangle)$
```

If the current R_P entry e_P is a non-leaf entry, it is processed as follows. If the join list contains leaf entries (line 37), the join algorithm overestimates the flow value for each of e_P's sub-entries when joining them with the relevant entries in the join list (lines 38–43). Only sub-entries with a non-empty join list are pushed back into the priority queue (lines 44–45). Otherwise, procedure expandList is called for each of e_P's sub-entries (lines 47–48).

4.3 Interval Query Algorithms

4.3.1 Iterative Algorithm

Algorithm 4 offers a straightforward way of computing the interval query (Problem 2). Overall, it follows the same iterative paradigm as does Algorithm 1 for the snapshot query.

Algorithm 4 iterativeInterval(R-tree R_P for indoor POIs, A1R-tree R_O for OTT, time interval $[t_s, t_e]$, integer k)

```
 1: initialize a hash table $flows : \{POI\} \rightarrow [0, +\infty]$
 2: for each POI $p$ do $flows[p] \leftarrow 0$
 3: LeafEntrySet $les \leftarrow R_O$.RangeQuery($[t_s, t_e]$)
 4: initialize a hash table $H$
 5: for each leaf entry $le \in les$ do
 6:     append $le.S$ to $H[le.objectID]$
 7: for each key $objectID \in H.keys$ do
 8:     get $(rd_s, \ldots, rd_e)$ from $H[objectID]$
 9:     calculate $UR(objectID, [t_s, t_e])$ from $(rd_s, \ldots, rd_e)$
10:     $ps \leftarrow R_P.IntersectionQuery(UR(objectID, [t_s, t_e]))$
11:     for each POI $p \in ps$ do
12:         $flows[p] \leftarrow flows[p] + \frac{Area(UR(o,[t_s,t_e]) \cap p)}{Area(p)}$
13: return the top-$k$ from $flows.keys$ with the highest values
```

It uses a range query on the A1R-tree to obtain the relevant leaf entries and objects whose augmented tracking time interval overlap the query time interval $[t_s, t_e]$ (line 3), as described in Section 4.1. An interval query may involve a sequence of tracking records of an object, and therefore the algorithm uses a hash table to form the sequences for all objects obtained (lines 4–6). The uncertainty region of each object is calculated (lines 8–9) according to the discussion in Section 3.2.

4.3.2 Join Based Algorithm

Basic framework. Like for snapshot query processing, we have a join based algorithm for interval query processing. As formalized in Algorithm 5, it also contains three phases: aggregate object R-tree R_I construction (lines 1–9), join initialization (lines 10–16), and join processing (lines 17–46). The differences here lie mainly in the first phase and in the details of deriving the uncertainty regions for objects in a join list in the third phase.

Algorithm 5 joinInterval(R-tree R_P for indoor POIs, A1R-tree R_O for OTT, time interval $[t_s, t_e]$, integer k)

```
1: initialize a hash table $H$
2: LeafEntrySet $les \leftarrow R_O$.RangeQuery($[t_s, t_e]$)
3: for each leaf entry $le \in les$ do
4:     append $le.S$ to $H[le.objectID]$
5: initialize an in-memory aggregate R-tree $R_I$
6: for each key $objectID \in H.keys$ do
7:     get $(rd_s, \dots, rd_e)$ from $H[objectID]$
8:     $mbr \leftarrow$ MBR($objectID, [t_s, t_e]$)
9:     insert $(objectID, mbr)$ into $R_I$
10: initialize a priority queue $Q$
11: for each entry $e_P$ in $R_P.root$ do
12:     $ubFlow \leftarrow 0$; $list \leftarrow \varnothing$
13:     for each entry $e_I$ in $R_I.root$ do
14:         if $e_P.mbr$ intersects $e_I.mbr$ then
15:             $ubFlow \leftarrow ubFlow + e_I.count$;  $list \leftarrow list \cup \{e_I\}$
16:     $Q.enqueue(\langle e_P, list, ubFlow\rangle)$
17: $result \leftarrow \varnothing$;  initialize a hash table $H_U$
18: while $Q$ is not empty do
19:     $\langle e_P, list\rangle \leftarrow Q.dequeue()$
20:     if $e_P$ is a leaf entry then
21:         if $list$ is null then
22:             add POI $e_P.object$ to $result$
23:             if $result = k$ then return $result$
24:         else
25:             if $list$ contain leaf entries then
26:                 $flow \leftarrow 0$
27:                 for each entry $e_I \in list$ do
28:                     if $H_U[e_I.object] = \varnothing$ then
29:                         $H_U[e_I.object] \leftarrow UR(e_I.object, [t_s, t_e])$
30:                     $flow \leftarrow flow + \phi_{t_s, t_e, e_P.object}(e_I.object)$
31:                 if $flow \neq 0$ then $Q.enqueue(\langle e_P, null, flow\rangle)$
32:             else
33:                 expandList($e_P, list$)
34:     else
35:         if $list$ contain leaf entries then
36:             for each sub-entry $e'_P$ in $e_P.node$ do
37:                 $ubFlow \leftarrow 0$; $list2 \leftarrow \varnothing$
38:                 for each entry $e'_I \in list$ do
39:                     if $e'_P.mbr$ intersects $e'_I.mbr$ then
40:                         $ubFlow \leftarrow ubFlow + 1$
41:                         $list2 \leftarrow list2 \cup \{e'_I\}$
42:                 if $list2 \neq \varnothing$ then
43:                     $Q.enqueue(\langle e'_P, list2, ubFlow\rangle)$
44:         else
45:             for each sub-entry $e'_P$ in $e_P.node$ do
46:                 expandList($e'_P, list$)
```

Although this basic framework still works for interval query processing, preliminary experimentation suggests that it can be improved significantly. In the following, we identify the performance bottleneck and present improvements to the design of the join based algorithm.

Improvements. The uncertainty regions for an interval query are much larger than those for a snapshot query. If a single MBR is used for an interval uncertainty region, the MBR can cover considerable dead space. This is the case in Algorithm 5 that uses a coarse MBR estimation, especially when an overall MBR is created for all tracking records of an object during the query interval (line 8). To alleviate this problem, we introduce several improvements.

Instead of using a single, large MBR to represent an object's trajectory during $[t_s, t_e]$, we use a series of much tighter MBRs, each of which is created based on a pair of consecutive tracking records. Suppose that an object o's relevant tracking records during $[t_s, t_e]$ are $\langle rd_1, \dots, rd_m\rangle$. For each pair (rd_i, rd_{i+1}), we create a small MBR mbr_i for the extended ellipse (Section 3.1.3) defined by the two tracking records.

After we create all such $m - 1$ smaller MBR mbr_is, we create the overall MBR mbr for all of them. When we insert mbr into the aggregate R-tree R_I, we create additional information at the leaf node level for all such smaller mbr_is for mbr. In particular, we insert a pointer from the entry for mbr to the list of mbr_is, such that we can easily access these when we visit mbr's node.

Next, we modify the procedure that expands the join list (Algorithm 3). Instead of simply checking whether two MBRs intersect, we do additional checks when a leaf node entry e'_I is taken from R-tree R_I (line 3 in Algorithm 3). In particular, if e'_I is a leaf node entry and its MBR intersects e_P's (line 4), we continue to check e_P's MBR against the smaller MBRs covered by e'_I as described above. We include e'_I into the join list only if at least one such smaller MBR intersects e_P's MBR. These finer-grained checks are expected to eliminate many false positives in the join list that would otherwise result from the large, dead space-dominant MBR of e'_I. This arrangement reduces the join list size for e_P and reduces also the subsequent join cost.

In similar fashion, we apply the additional MBR intersection checks to Algorithm 5 when it process leaf entries from R_I (line 39). This is also expected to reduce the join list and the join cost.

An example of the improvements is shown in Figure 9. Object o's overall MBR, whose dead space is indicated by

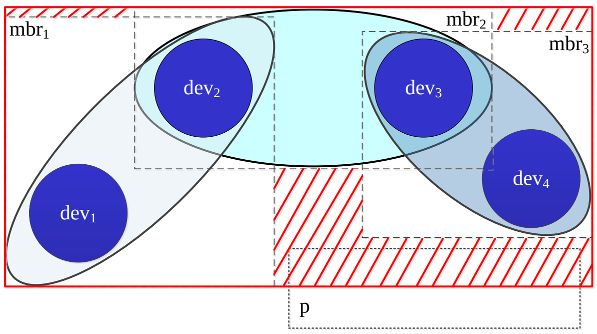


Figure 9: Less Coarse MBR Checks

the shaded parts, overlaps a POI p, and therefore o is included in p's join list initially. Later, o's complex uncertainty region $UR(o)$ is derived to calculate its presence in p. It turns out that the complex calculation does not contribute to the query result because $UR(o)$ does not intersect with p. Specifically, object o's single, large MBR can be replaced by three smaller MBRs. As shown in the figure, each of the three smaller MBRs bounds the ellipse corresponding to a pair of consecutive tracking records. Using the smaller MBRs, object o will be excluded from p's join list, and the calculation of $UR(o)$ will be skipped since none of the three smaller MBRs intersects with p. We only implement the improved framework in the experimental comparisons.

5. EXPERIMENTAL STUDIES

This section reports on the experimental studies of our research. Section 5.1 presents the experimental settings. Sections 5.2 and 5.3 cover the experiments on synthetic and real data, respectively.

5.1 Experimental Settings

All algorithms are implemented in Java and run on a computer with Windows 7 Enterprise edition, an Intel Core i7-2620M 2.70GHz CPU, and 8.0GB main memory.

Synthetic data set: We use a floor plan with about 100 rooms that are all connected by doors to a hallway. We place a total of 143 RFID readers by doors and along the hallways. We generate object movements using the random waypoint model [11]. All objects move with a fixed speed of 1.1 m/s, which is also used as the maximum speed V_{max}.

We vary multiple parameters when generating the data, as shown in Table 4 where default values are in bold. We vary $|O|$, i.e., the number of objects in the OTT from 10 K to 50 K. We vary the RFID detection range, the radius of the circular region covered by an RFID reader, from 1m to 2.5m. For the complete synthetic data set, the number of the OTT tracking records falls in the range from 140 K to 2,000 K.

Parameters	Settings
$\|O\|$	10K, 20K, ..., **50K**
Detection Range (meter)	1, **1.5**, 2, 2.5
$\|P\|$ (% of all indoor POIs)	2%, **4%**, 6%, 8%, 10%
k	1, 2, 3, 4, **5**, ..., 10, ... 15
$t_e - t_s$ (minutes)	10, 20, **30**, ..., 60

Table 4: Parameter Settings in Experiments

Real-world data set: We use a real data set collected from Copenhagen International Airport (CPH) where passengers with Bluetooth-enabled mobile phones are tracked by deployed Bluetooth radios. We extract the data for a period of 7 months with the most tracking records. Our OTT contains approximately 600K records for about 10K passengers. We do not vary the detection range in the experiments on the real data set because we have no access to change the configurations in the real deployment. We use the same V_{max} as in the synthetic data.

Query parameters: We also vary query parameters according to Table 4. Specifically, k (the number of top ranked indoor POIs to be returned) is varied from 1 to 15. The query POI set is determined as follows. For both synthetic and real data, 375 POIs are determined in the indoor space at distinctive locations and with different areas. Multiple POIs may come from the same large room that is divided into multiple uses. We control the number of query POIs ($|P|$) as a percentage (2% to 10%) of the total number of indoor POIs. Given a percent, the query POI set is determined as a random subset of the total 375 POIs. We start from 2% to make sure there are sufficient query POIs for returning the top-k results. On the other hand, we do not include more than 10% of all POIs in a query because, intuitively, not all indoor POIs are interesting and frequently visited, so query issuers like building officers may query a subset of all indoor POIs. Furthermore, $t_e - t_s$ is the query time interval used in the interval top-k indoor POI query. We vary it from 10 to 60 minutes.

5.2 Experiments on Synthetic Data

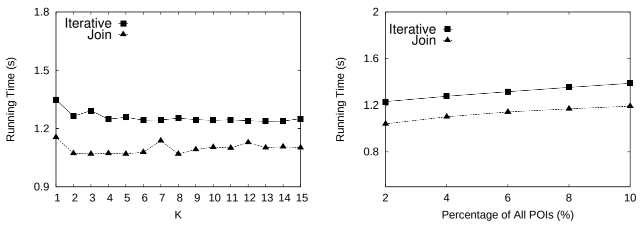
5.2.1 Results for Snapshot Queries

We first evaluate the performance of the snapshot query by varying parameters k, $|P|$, and the detection range. We do not change parameter $|O|$ for the snapshot query because the number of moving objects retrieved at a given time point is fairly random, and is smaller than and independent of $|O|$.

The results of varying k, $|P|$, and the detection range are presented in Figures 10 and 11(a). As a general observation, the join algorithm outperforms the iterative algorithm. This is because the former is able to prune more objects when their uncertainty regions do not overlap with the regions of POIs.

The effect of varying k is shown in Figure 10(a). As can be observed, varying k has only little influence on both algorithms. This indicates that both algorithms are stable with respect to query parameter k. The relatively high cost for both algorithms at $k = 1$ is due to the intensive initial computation of the uncertainty regions for many objects, which, however, does not pay off for a simple top-1 query.

The effect of varying $|P|$ is shown in Figure 10(b). As the number of query POIs increases, the running time increases slightly. This is because more query POIs occupy larger areas, and thus more moving objects are present in POIs. Thus, more object uncertainty regions intersect POIs, which yields longer processing times.



(a) Effect of k (b) Effect of $|P|$

Figure 10: Snapshot Query on Synthetic Data Set

Figure 11(a) presents the effect of varying the detection range of RFID readers. When the detection range increases, the uncertainty region increases as well. Therefore, more computation time is needed to estimate the areas of uncertainty regions. Hence, the running time is the largest when the detection range is set to 2.5m. The slight decrease at 2 meters is attributed to the fluctuation of the OTT size.

5.2.2 Results for Interval Queries

The performance of the interval query algorithms is reported in Figures 11(b) and 12. In this experiment, we vary all five parameters listed in Table 4. Similar to the experiments for snapshot queries, the join algorithm runs faster than the iterative algorithm in almost all settings, which we attribute to its effective pruning strategy.

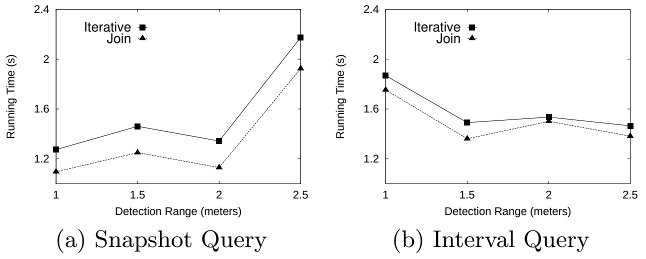
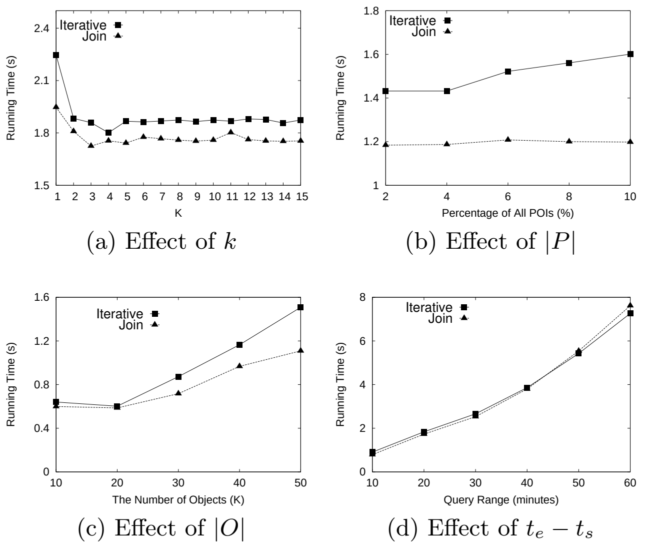

(a) Snapshot Query (b) Interval Query

Figure 11: Effect of Detection Range

As shown in Figure 12(a), varying k does not significantly impact the performance of the algorithms except for $k = 1$. Overall, both algorithms are stable with respect to query result size except when $k = 1$. The longer running time for $k = 1$ occurs because the considerable computations on the uncertainty regions do not pay off for the very small top-1 query results. The performance improves when k increases because neither algorithm works in an incremental manner with respect to k. The relatively high cost at $k = 1$ is not observed in the counterpart experiments on the real data set (see Figure 14(a)), which is attributed to the considerably smaller size of the real data.

The effect of varying $|P|$ is presented in Figure 12(b). When increasing $|P|$, the running time of the iterative algorithm increases, while the join algorithm stays stable. Both algorithms have longer running times when compared to their snapshot query counterparts. This is understandable, as uncertainty regions in the snapshot setting are much smaller than those in the interval setting.


(a) Effect of k (b) Effect of $|P|$

(c) Effect of $|O|$ (d) Effect of $t_e - t_s$

Figure 12: Interval Query on Synthetic Data Set

Figure 12(c) shows that the running time of both algorithms increases as $|O|$ increases. Nevertheless, the join algorithm remains more efficient. This indicates that the join algorithm is most scalable.

As the query time interval $t_e - t_s$ increases, the running time of the two algorithms increases accordingly, as presented in Figure 12(d). Longer query intervals tend to yield larger and more complex uncertainty regions that require more time to process.

As presented in Figure 11(b), when the detection range increases, the running time of both algorithms tends to decrease. This trend is opposite to the observation in the experiment for snapshot queries where the detection range is changed. In the interval setting, the algorithms compute on moving objects trajectories in a given time interval and thus involve a series of tracking records and devices. When the detection ranges of the devices increase, an object's uncertainty regions between pairs of consecutive devices shrink. As a result, objects' overall uncertainty regions tend to be smaller and thus take less time to process. At the outlier of 2 meters, the OTT is small, and the tracking data is sparse, which offsets the benefit of a larger detection range.

5.3 Experiments on Real Data

Results for snapshot queries are reported in Figure 13. Clearly the join algorithm outperforms the iterative algorithm. Both algorithms are fairly stable with respect to varying k, as shown in Figure 13(a). When the number of query POIs is increased, both algorithms' running times increase moderately and almost linearly, as shown in Figure 13(b). These results indicate that our designs are stable and scalable for snapshot queries in real indoor settings.

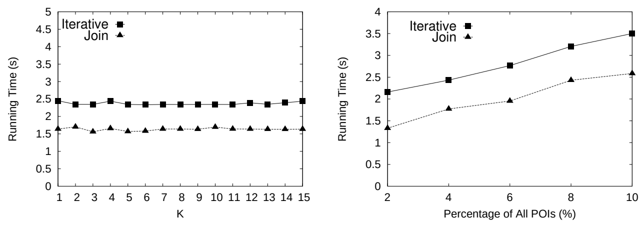

(a) Effect of k (b) Effect of $|P|$

Figure 13: Snapshot Query on CPH Data Set

Interval query performance results are reported in Figure 14. It is clear that the join algorithm outperforms the iterative algorithm for all settings in these experiments.

The effect of varying k is shown in Figure 14(a). The join algorithm is more efficient and slightly more stable when varying k. This indicates that our strategy of finer MBRs for interval uncertainty regions pays off.

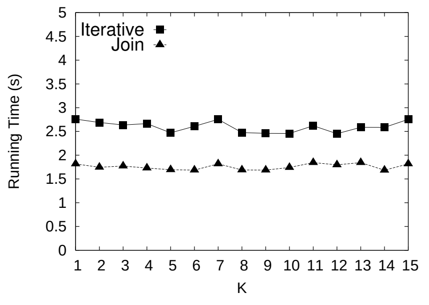
The effect of varying the number of query POIs is shown in Figure 14(b). The more stable performance of the join algorithm is again attributed to the use of finer MBRs for interval uncertainty regions. Smaller MBRs can help prune objects (and their uncertainty regions) more effectively even when there are more POIs and those POIs collectively occupy larger areas.

The effect of varying the query interval length is shown in Figure 14(c). Longer intervals yield longer object trajectories and larger uncertainty regions. This explains the increase in the running times of both algorithms. Nevertheless, the join algorithm is still more efficient thanks to the use of finer MBRs.

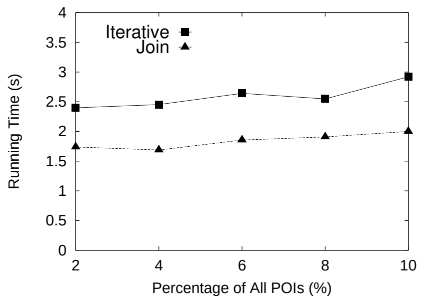
6. RELATED WORK

We briefly review related work in this section. Section 6.1 covers the research on indoor-space moving objects, and Section 6.2 covers the density queries in outdoor spaces.

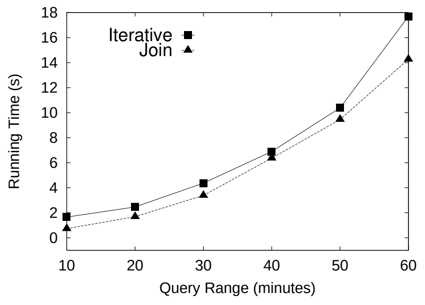
6.1 Indoor-Space Moving Objects



(a) Effect of k



(b) Effect of $|P|$



(c) Effect of $t_e - t_s$

Figure 14: Interval Query on CPH Data Set

Due in part to their different topology, indoor spaces are modeled [10, 13, 16, 21, 22] differently than outdoor spaces. Moreover, indoor moving objects are tracked by means different than outdoor GPS and thus generate different tracking data.

Assuming a generic setting of symbolic indoor tracking, Yang et al. study continuous range monitoring queries [25] and probabilistic k nearest neighbor queries [26] on indoor moving objects. Uncertain query results are returned as objects' locations are unknown when they are outside any detection range. Yu et al. [28] propose a particle filter based method to infer undetected locations under RFID tracking, which thus improves query result quality. Unlike these works that concern the current locations of indoor moving objects, this paper works on historical indoor tracking data.

Lu et al. [17] define spatio-temporal joins over indoor moving objects whose historical locations are captured in the same format as the symbolic tracking data assumed in this paper. However, while we compute flows for static indoor POIs according to historical tracking data, Lu et al. compute pairs of indoor moving objects that were in the same location according to historical data.

Xie et al. [23,24] study indoor distance-aware spatial queries over online indoor moving objects whose positions are reported as probabilistic samples rather than using RFID detection ranges. Therefore, the proposed techniques are unsuitable for the problems we consider in this paper.

Recently, Ahmed et al. [2] define indoor density queries based on historical RFID indoor tracking data. Their density definition differs from our definition of flow. Moreover, we analyze the inherent uncertainties in the data and design uncertainty-aware solutions, whereas Ahmed et al. do not consider uncertainty.

6.2 Density Queries in Outdoor Spaces

Most existing research works on the querying of object flow and density are for outdoor Euclidean spaces or spatial road networks.

Tao et al. [20] use an aggregate RB-tree (aRB-tree) for indexing spatio-temporal objects in a Euclidean space and propose algorithms to count objects in a given spatial window during a given time interval. The proposed solution considers no location uncertainty and the aRB-tree cannot handle the complex indoor topology. Therefore, that solution cannot be used for the problems addressed in this paper.

Employing micro-clustering [29], Li et al. [15] propose techniques that cluster moving objects and dynamically update the clusters as the objects move linearly. The proposed techniques do not apply in our setting, where indoor object movements can not be captured by linear models but are reported based on discrete detection ranges. Yiu and Mamoulis [27] propose density based and hierarchical methods to partition and cluster static objects on a spatial network. Their proposal uses network distances between static positions and therefore does not solve our problems on indoor moving objects.

Hadjieleftheriou et al. [5] study threshold density region queries that find outdoor regions with more objects than a given threshold. Their solution assumes that the objects move according to known linear functions and are indexed by uniform space-time grids. Jensen et al. [9] study snapshot dense region queries in a Euclidean plane where objects move linearly and are indexed by a B^x-tree [8]. With the same linear motion assumption, Hao et al. [6] study continuous density queries by using a quad-tree to index moving objects. To improve density query results, Ni and Ravishankar [18] redefine the density and use small square neighborhoods to approximate arbitrary outdoor regions. These proposals [5, 6, 9, 18] are unsuitable for our problems, a key reason being that objects in indoor spaces cannot be modeled well by linear functions.

Huang and Lu [7] define online density region queries on moving objects that are observed by sensors at fixed positions in a geographical area. The proposed solution assumes that object locations are captured as certain points in a 2D Euclidean space. This assumption renders the solution unsuitable for our problems where indoor moving object locations are captured as uncertainty regions.

Li et al. [14] devise techniques to return traffic density-based hot routes from historical trajectories in a road network. Cai et al. [12] design a clustering based technique for continuously monitoring dense road segments. Different from these works, we target indoor spaces where objects' historical movements are captured as detection ranges associated with time intervals.

7. CONCLUSION AND FUTURE WORK

In this paper, we study how to find the top-k frequently visited indoor Points of Interest (POIs) using symbolic indoor tracking data that captures object movements indoors. We define two types of queries in this regard. A snapshot query finds those indoor POIs that were visited by the most tracked objects (e.g., people) at a given time point, whereas an interval query finds such POIs for a given time interval. As symbolic indoor tracking data can only capture trajectories with a considerable degree of uncertainty, we define appropriate ways to quantify how frequently an indoor POI is visited by probabilistically counting objects' presences in the POI. Subsequently, we conduct a detailed analysis on the uncertainty regions of objects in the settings of the two types of queries. Based on the uncertainty analysis, we design algorithms for both query types. We use both synthetic and real data sets to evaluate the query processing algorithms, and the performance results show that our proposed join-based algorithms are capable of significantly outperforming straightforward baselines and are much more scalable in terms of data set size and query interval length.

Several directions exist for future research. First, it is relevant to consider an object's dwell time when calculating its interval presence in a POI. The interval-related definitions in the paper can be extended for that purpose. In particular, an object's interval uncertainty region can be extended to also reflect the temporal aspect in addition to the spatial aspect. Second, it is of interest to extend the uncertainty analysis to support multiple floors. The new challenge is to track object movement between floors appropriately and to derive the uncertainty regions accordingly. Third, it is of interest to investigate how to solve the problems addressed in the paper using other types of indoor tracking data. To this end, it can be considered whether the proposed techniques can be applied to, or adapted for, other data types. Fourth, it is relevant to develop techniques for finding the currently crowded indoor POIs by using tracking data. Fifth, it is of interest to evaluate the query results against real indoor POIs in order to see how effective the proposed query types are at finding frequently visited indoor POIs. For that purpose, ground truth data on popular indoor POIs is needed.

Acknowledgments

The work is partly supported by the NILTEK project, funded by European Regional Development Fund.

8. REFERENCES

[1] InLocation Alliance. `http://www.in-location-alliance.com/`, 2016 (accessed January 2016).

[2] T. Ahmed, T. B. Pedersen, and H. Lu. Finding dense locations in indoor tracking data. In *MDM*, pages 189–194, 2014.

[3] X. Cao, G. Cong, and C. S. Jensen. Mining significant semantic locations from GPS data. *PVLDB*, 3(1):1009–1020, 2010.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[5] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *SSTD*, pages 306–324, 2003.

[6] X. Hao, X. Meng, and J. Xu. Continuous density queries for moving objects. In *MobiDE*, pages 1–7, 2008.

[7] X. Huang and H. Lu. Snapshot density queries on location sensors. In *MobiDE*, pages 75–78, 2007.

[8] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B$^+$-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.

[9] C. S. Jensen, D. Lin, B. C. Ooi, and R. Zhang. Effective density queries on continuouslymoving objects. In *ICDE*, page 71, 2006.

[10] C. S. Jensen, H. Lu, and B. Yang. Graph model based indoor tracking. In *MDM*, pages 122–131, 2009.

[11] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[12] C. Lai, L. Wang, J. Chen, X. Meng, and K. Zeitouni. Effective density queries for moving objects in road networks. In *WAIM*, pages 200–211, 2007.

[13] J. Lee. A spatial access-oriented implementation of a 3-D GIS topological data model for urban entities. *GeoInformatica*, 8(3):237–264, 2004.

[14] X. Li, J. Han, J. Lee, and H. Gonzalez. Traffic density-based discovery of hot routes in road networks. In *SSTD*, pages 441–459, 2007.

[15] Y. Li, J. Han, and J. Yang. Clustering moving objects. In *SIGKDD*, pages 617–622, 2004.

[16] H. Lu, X. Cao, and C. S. Jensen. A foundation for efficient indoor distance-aware query processing. In *ICDE*, pages 438–449, 2012.

[17] H. Lu, B. Yang, and C. S. Jensen. Spatio-temporal joins on symbolic indoor tracking data. In *ICDE*, pages 816–827, 2011.

[18] J. Ni and C. V. Ravishankar. Pointwise-dense region queries in spatio-temporal databases. In *ICDE*, pages 1066–1075, 2007.

[19] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-object representations. In *SSD*, pages 111–132, 1999.

[20] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–225, 2004.

[21] E. Whiting, J. Battat, and S. Teller. Topology of Urban Environments. *CAADFutures*, pages 114–128, 2007.

[22] M. F. Worboys. Modeling indoor space. In *ISA*, pages 1–6, 2011.

[23] X. Xie, H. Lu, and T. B. Pedersen. Efficient distance-aware query evaluation on indoor moving objects. In *ICDE*, pages 434–445, 2013.

[24] X. Xie, H. Lu, and T. B. Pedersen. Distance-aware join for indoor moving objects. *IEEE Trans. Knowl. Data Eng.*, 27(2):428–442, 2015.

[25] B. Yang, H. Lu, and C. S. Jensen. Scalable continuous range monitoring of moving objects in symbolic indoor space. In *CIKM*, pages 671–680, 2009.

[26] B. Yang, H. Lu, and C. S. Jensen. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In *EDBT*, pages 335–346, 2010.

[27] M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In *SIGMOD*, pages 443–454, 2004.

[28] J. Yu, W. Ku, M. Sun, and H. Lu. An RFID and particle filter-based indoor spatial query evaluation system. In *EDBT*, pages 263–274, 2013.

[29] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *SIGMOD*, pages 103–114, 1996.