Towards Crowd-aware Indoor Path Planning

Liu, Tiantian; Li, Huan; Lu, Hua; Cheema, Muhammad Aamir; Shou, Lidan

Published in:
Proceedings of the VLDB Endowment

DOI (link to publication from Publisher):
10.14778/3457390.3457401

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Towards Crowd-aware Indoor Path Planning

Tiantian Liu[†], Huan Li[†], Hua Lu[‡], Muhammad Aamir Cheema[§], Lidan Shou[♮]
[†]Department of Computer Science, Aalborg University, Denmark
[‡]Department of People and Technology, Roskilde University, Denmark
[§]Faculty of Information Technology, Monash University, Australia
[♮]College of Computer Science, Zhejiang University, China
{liutt,lihuan}@cs.aau.dk, luhua@ruc.dk, aamir.cheema@monash.edu, should@zju.edu.cn

ABSTRACT

Indoor venues accommodate many people who collectively form crowds. Such crowds in turn influence people's routing choices, e.g., people may prefer to avoid crowded rooms when walking from A to B. This paper studies two types of crowd-aware indoor path planning queries. The Indoor Crowd-Aware Fastest Path Query (FPQ) finds a path with the shortest travel time in the presence of crowds, whereas the Indoor Least Crowded Path Query (LCPQ) finds a path encountering the least objects en route. To process the queries, we design a unified framework with three major components. First, an indoor crowd model organizes indoor topology and captures object flows between rooms. Second, a time-evolving population estimator derives room populations for a future timestamp to support crowd-aware routing cost computations in query processing. Third, two exact and two approximate query processing algorithms process each type of query. All algorithms are based on graph traversal over the indoor crowd model and use the same search framework with different strategies of updating the populations during the search process. All proposals are evaluated experimentally on synthetic and real data. The experimental results demonstrate the efficiency and scalability of our framework and query processing algorithms.

PVLDB Reference Format:
Tiantian Liu, Huan Li, Hua Lu, Muhammad Aamir Cheema, Lidan Shou. Towards Crowd-aware Indoor Path Planning. PVLDB, 14(8): 1365-1377, 2021.
doi:10.14778/3457390.3457401

1 INTRODUCTION

Indoor route planning queries are among the most fundamental queries underlying indoor location-based services (LBS) [19, 26, 27, 32, 45]. Such queries can facilitate people in need. For example, in an airport or a train station, passengers prefer to find the fastest path from their current position to the boarding gate. In addition to the shortest or fastest paths, indoor routing supports many variations that meet practical needs. For instance, customers in a shopping mall would like to find a path that can cover some given keywords like a coffee shop and shoes [12]. Meanwhile, indoor venues accommodate many people who collectively form crowds that may in turn influence people's routing choices. For example, crowds may influence one's moving speed, which will have an effect on the travel time of a path. In some places like an airport where passengers are sensitive to travel time, a topologically shortest path may still incur the too long time and result in missing flight if the path fails to consider the effect of crowds. In other scenarios, people en route may prefer to encounter fewer people. For example, during the COVID-19 pandemic, people would like to find a path to avoid human contact as much as possible. As another example, autonomous objects (e.g., driverless cars in an airport) also prefer a path with fewer people en route to mitigate the interference and inconvenience caused by contact with people.

In this paper, we formulate and study two crowd-aware indoor path planning queries. Referring to Figure 1, given a source point p_s, a target point p_t, and a query time t, an Indoor Crowd-Aware Fastest Path Query (FPQ) returns a path with the shortest travel time in the presence of crowds, whereas an Indoor Least Crowded Path Query (LCPQ) returns a path that encounters the least objects en route. As an indoor path is essentially a series of indoor partitions (basic topological units like rooms), FPQ's routing cost is partition-passing time, whereas an LCPQ's is partition-passing contact.

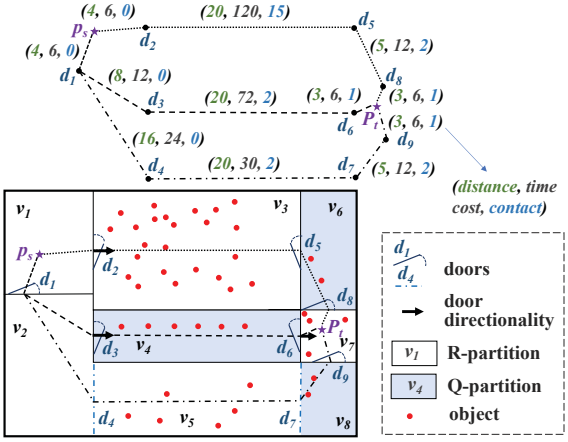

Figure 1: An Example of Floorplan at Query Time t_q

We consider two types of indoor partitions. A *Queue Partition* (Q-partition) requests objects to enter and exit in a line, e.g., a security-check line in an airport or a ticketing entrance in a theater. A *Random Partition* (R-partition) refers to a more general case where there is no restriction on how to pass the partition but one's movement slows down when encountering a crowd. Due to the different topological natures of the Q-partitions and R-partitions,

Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457401

partition-passing time for FPQ and partition-passing contact for LCPQ should be defined differently for the two partition types.

Existing techniques cannot handle the novel FPQ and LCPQ. First, techniques for outdoor route planning [2, 3, 5, 11, 14, 29, 36, 39] do not work for indoor spaces with distinct entities like doors, walls, and rooms, altogether forming a complex topology. Second, the existing indoor route planning methods [12, 26, 31, 32] do not consider the effect of crowds, lacking the modeling foundation for FPQ and LCPQ. Third, some works study indoor flow and density [20, 21], but they do not touch upon route planning.

To solve FPQ and LCPQ efficiently, we design a crowd-aware query processing framework. The framework is composed of three layers. First, the indoor crowd model is the foundation layer of the framework. The model can handle three kinds of information, namely indoor topology, indoor geometry, and crowd-evolution. A time-evolving population estimator derives the future flows and populations for indoor partitions. The estimated values are used as basic routing costs in FPQ and LCPQ. The query processing layer consists of two parts. In part one, two functions, namely partition-passing time function and partition-passing contact function, calculate the routing cost for FPQ and LCPQ, respectively. Again, the differences in routing costs for different partition types are unified into a single computing process. On top of that, part two provides two exact and two approximate path search algorithms that each can process both query types. One of the exact searches uses a global estimator, whereas the other uses an estimator that only estimates a partition's population by looking up its upstream partitions' flows. The two approximate algorithms speed up query processing at the cost of query accuracy. One of them only derives populations for partial partitions, and the other derives populations only when necessary. All proposed techniques are experimentally evaluated on synthetic and real data. The experimental results demonstrate the efficiency and scalability of the proposed framework and query processing algorithms. The results also show the two approximate search algorithms achieve good routing accuracy.

This paper makes the following key contributions.

- We formulate Indoor Crowd-Aware Fastest Path Query (FPQ) and Indoor Least Crowded Path Query (LCPQ), and propose a unified processing framework for these queries (Section 2).
- We design an indoor crowd model that organizes indoor topology and captures indoor partition flows and densities (Section 3).
- We devise a time-evolving population estimator to derive future time-dependent flows and populations for partitions (Section 4).
- We design two exact and two approximate query processing algorithms that each can process both query types (Section 5).
- We conduct extensive experiments on synthetic and real datasets to evaluate our proposals (Section 6).

In addition, Section 7 reviews the related work and Section 8 concludes the paper.

2 PRELIMINARIES

Table 1 lists the notations frequently used in this paper.

2.1 Indoor Crowds

An indoor space is divided by walls and doors into *indoor partitions*. A **Queue Partition** (Q-partition) requests objects to enter and leave *sequentially*, while a **Random Partition** (R-partition) has no such a restriction and objects can enter and leave it *randomly*. Note that the type of partition is usually fixed and will change only when the space layout is redesigned. The issue of topological change is out of the scope of this paper.

Table 1: Notations

Symbol	Meaning
v, d, p	Partition, door, and indoor point
o, O	Object, object set
C	Indoor crowd
$t_{o \triangleright C}$, $t_{C \triangleright o}$	The times o joins and leaves C
$RT(d_i)$	The sequence of d_i's report timestamps
$UTI(v_k)$	The set of v_k's unit (update) time intervals
$\delta_{t_x,t_{x+1}}(v_k)$	v_k's density over $[t_x, t_{x+1}]$
$\rho(v_k, t_c)$	v_k's lagging coefficient at time t_c
$T(d_i, d_j, v_k, t_c)$	The time to pass v_k from d_i to d_j at t_c
$\kappa(d_i, d_j, v_k, t_c)$	The object contact to pass v_k from d_i to d_j at t_c

Within a partition, moving objects (e.g., persons) may form a crowd. Corresponding to the partition types, we formally define an *indoor crowd* as follows.

Definition 1 (Indoor Crowd). *An indoor crowd $C_{t_s,t_e}(v_k)$ [1] is a set of moving objects in a partition v_k during a certain time interval $[t_s, t_e]$. $C.\tau$ denotes the type of crowd C.*

1) *In a **Queue Crowd** (Q-crowd), objects join and leave the crowd in the first-in-first-out (FIFO) manner. Formally, $\forall o_i, o_j \in C_k \wedge C_k.\tau = \mathrm{Q}, t_{o_i \triangleright C_k} \leq t_{o_j \triangleright C_k} \Rightarrow t_{C_k \triangleright o_i} \leq t_{C_k \triangleright o_j}$, where $t_{o_i \triangleright C_k}$ and $t_{C_k \triangleright o_i}$ is the time o_i joins and leaves C_k, respectively.*
2) *In a **Random Crowd** (R-crowd), objects join and leave the crowd without any ordering restrictions. Formally, $\exists o_i, o_j \in C_k \wedge C_k.\tau = \mathrm{R}, t_{o_i \triangleright C_k} < t_{o_j \triangleright C_k}, t_{C_k \triangleright o_i} \geq t_{C_k \triangleright o_j}$.*

A crowd changes as objects join and leave from time to time. In other words, the object population and density of a partition are time-varying, rendering an object's routing cost passing the partition to change as well. Therefore, for crowd-aware routing, it is of fundamental importance to know a crowd's dynamic population or density. This demands dynamic data from a localization system.

However, a localization system may not record the exact trajectory or join/leave time of each individual object due to computing/storage limitations and location privacy concerns. Alternatively, a system may maintain the current number of objects in each partition (or a crowd) and records the number of objects joining and leaving during a time interval. This can be easily achieved, e.g., by installing a counter at a door. In our setting, each door counter reports objects' joining and leaving at a predefined frequency. This means that the object numbers in a crowd are updated at a number of discrete timestamps. Specifically, we use a time-ordered sequence $RT(d_i) = (t_{i1}, \ldots, t_{in})$ to denote the **report timestamps** of the counter at door d_i. As a result, the **update timestamps** relevant to a partition v_k is a time-ordered sequence $UT(v_k) = \bigcup_{d_j \in P2D(v_k)} RT(d_j)$ where $P2D(v_k)$ refers to all doors of partition v_k. Each two consecutive timestamps in the sequence $UT(v_k)$ forms an **unit (update) time interval**. The set of all such intervals from $UT(v_k)$ is denoted by $UTI(v_k)$.

[1] When time is not of particular interest, we use C_k to denote v_k's associated crowd.

At the routing query time, it is necessary to know the flows in the future. However, future exact object numbers from door counters are unavailable at that moment. To this end, we employ door flow functions to model the crowd-evolution (detailed in Section 3.2).

We define a partition's *time-parameterized density* as follows.

Definition 2 (Time-Parameterized Density)**.** *Given a partition v_k and its unit time interval $[t_x, t_{x+1}] \in UTI(v_k)$, its* time-parameterized density *over $[t_x, t_{x+1}]$ is $\delta_{t_x,t_{x+1}}(v_k) = |C_k|/Area(v_k)$, where $|C_k|$ is v_k's population over $[t_x, t_{x+1}]$ and $Area(v_k)$ is v_k's area.*

The population and density in this paper are time-parameterized unless mentioned otherwise. A partition v_k's density at an arbitrary timestamp t_c is estimated with respect to the unit time interval covering t_c. Specifically, we have $\delta_{t_c}(v_k) = \delta_{t_x,t_{x+1}}(v_k)$ where $t_x \le t_c < t_{x+1}, [t_x, t_{x+1}] \in UTI(v_k)$.

2.2 Problem Formulation

In an indoor routing problem, a basic step is to move from one door to another through their in-between partition. To measure the cost to pass a partition, the intra-partition **door-to-door distance** [12] for two doors d_i and d_j is

$$d2d(d_i, d_j) = \begin{cases} |d_i, d_j|_E, & \text{if } D2P_{\sqsupset}(d_i) \cap D2P_{\sqsubset}(d_j) \neq \varnothing; \\ \infty, & \text{otherwise.} \end{cases} \quad (1)$$

where $D2P_{\sqsupset}(d_i)$ gives the set of partitions that one can enter through door d_i and $D2P_{\sqsubset}(d_j)$ gives those that one can leave through door d_j. Therefore, $D2P_{\sqsupset}(d_i) \cap D2P_{\sqsubset}(d_j) \neq \varnothing$ means d_i and d_j share a common partition that one can enter via d_i and leave via d_j. In this case, the Euclidean distance is used between d_i and d_j. Otherwise, the distance between them is set to infinite.

Definition 3 (Indoor Path)**.** *An indoor path from the source p_s to the target p_t is $\phi = (p_s, d_x, \ldots, d_y, p_t)$, where $(d_x, \ldots, d_y)$ is a door sequence, d_x is a leaveable door of p_s's host partition, d_y is an enterable door of p_t's host partiton, and each two consecutive doors d_n, d_{n+1} $(x \le n < y)$ on ϕ have $D2P_{\sqsupset}(d_n) \cap D2P_{\sqsubset}(d_{n+1}) \neq \varnothing$. Each two consecutive path nodes form a* path segment*. The distance of ϕ is computed as $dist_\phi = |p_s, d_x|_E + \sum_{n=x}^{y-1} d2d(d_n, d_{n+1}) + |d_y, p_t|_E$.*

When there is no crowd, the basic time cost of passing an in-between partition v_k from d_i to d_j can be estimated based on the average object moving speed $\bar{s}$, i.e., $T^{(b)}(d_i, d_j) = d2d(d_i, d_j)/\bar{s}$. To reflect a crowd's impact, we use the **lagging coefficient** $\rho(v_k, t_c)$ that takes into account the crowd's density and type as follows.

$$\rho(v_k, t_c) = \begin{cases} 1 + e^{\delta_{t_c}(v_k)/\mathrm{D}_k^{max}}, & \text{if } C_k.\tau = \mathrm{Q}; \\ 1 + e^{(\delta_{t_c}(v_k)/\mathrm{D}_k^{max})^2}, & \text{otherwise.} \end{cases} \quad (2)$$

where $\delta_{t_c}(v_k)$ is v_k's density at time t_c and D_k^{max} corresponds to the maximum density[2] of v_k. For a Q-partition, the ratio $\delta_{t_c}(v_k)/\mathrm{D}_k^{max}$ is applied to reflect the crowding degree. We modify the speed-density model [35] to calculate the lagging coefficient in Equation 2 which reflects real-world scenarios, e.g., in common sense, a crowd usually impacts people's moving speed and results in longer travel time. Equation 2 guarantees that the coefficient is always greater than 1 and it increases monotonically as v_k's density increases. For an R-partition, the square of the ratio is used because R-crowds incur less lagging effect.

Note that other forms of lagging coefficients can be defined and supported within our framework, e.g., lagging can be multiplied by the object number for a queue crowd. Since the lagging coefficient is *not* our research focus, we simply apply Equation 2 in this study.

Using the lagging coefficient, we can calculate our crowd-aware and time-dependent **partition-passing time** as follows.

$$T(d_i, d_j, v_k, t_c) = T^{(b)}(d_i, d_j) \cdot \rho(v_k, t_c) \quad (3)$$

An object needs longer time to pass a more crowded partition.

As a special case, we replace d_i with p_s or replace d_j with p_t in Equation 3, to estimate the cost of a path segment starting with p_s or ending with p_t. Accordingly, v_k is the host partition of p_s or p_t.

With the partition-passing time, we can plan the fastest indoor path for users to avoid undesirable congestion caused by indoor crowds. An indoor path ϕ's *overall travel time* T_ϕ is computed as the sum of the time of passing the partition between each path segment on ϕ. The fastest path query problem is defined as follows.

Problem 1 (Indoor Crowd-Aware Fastest Path Query FPQ)**.** *Given a source p_s and a target p_t, an* indoor crowd-aware fastest path query $\mathsf{FPQ}(p_s, p_t, t)$ *returns a path $\phi(p_s, d_i, \ldots, d_j, p_t)$ such that a) the overall travel time T_ϕ is minimized and b) ϕ is the shortest among all satisfying a). Formally, $\nexists \phi' \neq \phi, T_{\phi'} \le T_\phi \wedge dist_{\phi'} < dist_\phi$.*

Note that the partition-passing time is determined by the time one arrives at that partition, while the arrival time, in turn, is dependent on the partition-passing time of the previous partition. This calls for on-the-fly computation during the search to obtain the overall travel time T_ϕ, which is to be detailed in Section 5.

Example 1. *Figure 1 illustrates an indoor space at time t_q. The query time and crowd-evolution snapshot are considered. We indicate the distance, partition-passing time and object contact on each path segment in the top sketch. We suppose that there are some events in v_7, and v_4, v_6 and v_8 are Q-partitions for ID check before entering v_7. Given a query $\mathsf{FPQ}(p_s, p_t, t_q)$, there are three candidate paths, namely $\phi_1(p_s, d_2, d_5, d_8, p_t)$, $\phi_2(p_s, d_1, d_3, d_6, p_t)$, and $\phi_3(p_s, d_1, d_4, d_7, d_9, p_t)$. Only considering the distance but not the impact from crowds, ϕ_1 is the shortest with a length of 32 meters, while those of ϕ_2 and ϕ_3 are 35 meters and 48 meters, respectively. However, ϕ_1 is not expected to be the fastest path when crowds are concerned. To be specific, ϕ_1 goes through a highly crowded R-partition v_3, incurring a total travel time of 144 seconds. For ϕ_2, the low-populated Q-partition v_4 with a long queue is involved, making the total time cost be 96 seconds. Among all, ϕ_3 is expected to be the fastest with an overall cost of 78 seconds, though it is the longest distance passing 5 partitions.*

Another practically interesting problem is to find the shortest path that contacts the least objects. E.g., it is useful to find a path that avoids human contact as much as possible in the COVID-19 case. Given a path segment (d_i, d_j) that goes through a partition v_k, we calculate the **partition-passing contact** as follows.

$$\kappa(d_i, d_j, v_k, t_c) = \begin{cases} (|d_i, d_j|_E \cdot \mathtt{w}) \cdot \delta_{t_c}(v_k), & \text{if } C_k.\tau = \mathrm{R}; \\ (\mathtt{w}/|d_i, d_j|_E) \cdot (\delta_{t_c}(v_k) \cdot Area(v_k)), & \text{otherwise.} \end{cases} \quad (4)$$

Given a partition v_k, its enterable door d_i, and its leaveable door d_j, for any object reaching d_i at time t_c, the partition-passing contact

[2]The maximum capacity (and therefore the maximum density) of a partition is usually known, such as the room capacity for fire safety.

to pass v_k and reach d_j is defined in terms of the number of objects covered by the buffer of the path segment. The contact to pass an R-partition is the partition density multiplied by the buffer area that is approximated as $|d_i, d_j|_E \cdot \mathtt{w}$ where $\mathtt{w}$ is the buffer width. The contact to pass a Q-partition is the objects within the $\mathtt{w}$ long queue line centered at the user's position, i.e., the proportion $\mathtt{w}/|d_i, d_j|_E$ of the total objects in the queue. This reflects common sense. For example, if we pass a random crowd, the close contacts are those who we meet in the buffer width. If we pass a queue crowd, we only have close distance with those in front of or behind us.

In our implementation, we set $\mathtt{w}$ as the unit distance of 1m. For example, many countries suggest people keep a physical distance of 1m in the COVID-19 pandemic. Similar to the computation of the overall travel time T_ϕ, an indoor path ϕ's *overall contact* κ_ϕ is computed as the sum of the partition-passing contacts of path segments on ϕ. Likewise, Equation 4 applies to the path segment starting with p_s and ending with p_t. Accordingly, we formulate the least crowded path query as follows.

Problem 2 (Indoor Least Crowded Path Query LCPQ)**.** *Given a source p_s and a target p_t, an* indoor least crowded path query $\mathtt{LCPQ}(p_s, p_t, t)$ *returns a path $\phi(p_s, d_i, \ldots, d_j, p_t)$ such that a) the overall contact is the least, and b) ϕ is the shortest among all satisfying a). Formally, $\nexists \phi' \neq \phi, \kappa_{\phi'} \leq \kappa_\phi \wedge dist_{\phi'} < dist_\phi$.*

Example 2. *Consider a query $\mathtt{LCPQ}(p_s, p_t, t_q)$ in Figure 1, the candidates ϕ_1, ϕ_2 and ϕ_3 involve 18, 3 and 5 contacts from the partitions which they pass, respectively. The query returns ϕ_2 since it contacts the fewest objects.*

2.3 Solution Framework

We propose a crowd-aware query processing framework as illustrated in Figure 2.

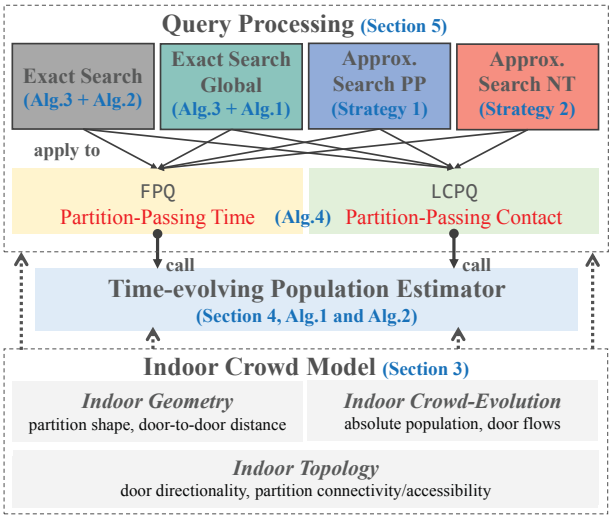

Figure 2: Crowd-Aware Path Planning Framework

In the bottom, an indoor crowd model (cf. Section 3) maintains the following aspects of an indoor space: *Indoor Topology* that captures the directionality of doors and connectivity/accessibility of partitions, *Indoor Geometry* that records the shapes of partitions and walking distances between two doors, and *Indoor Crowd-Evolution* that models the objects joining and leaving the crowds.

Enabled by the indoor crowd model, a *time-evolving population estimator* in the middle layer derives populations (and densities) of partitions at a future time and provides them to the query algorithms. The population estimation process will be detailed in Section 4.

In the top layer, crowd-aware search algorithms process FPQ and LCPQ. Both algorithms are based on graph traversal over the indoor crowd model. To expand to the next path node with the minimum cost, FPQ's algorithm estimates the partition-passing time, while LCPQ search algorithm estimates the partition-passing contact. Both costs are estimated based on the time-evolving populations derived in the middle layer. For both queries, two exact and two approximate search algorithms are proposed. Their main difference lies in the strategy of updating population(s) during the search. All search algorithms will be presented in Section 5. Thanks to modular construction, our framework can be easily extended or reduced. For example, to support regular path planning, we only need the components Indoor Topology and an appropriate query processing algorithm that can be a variant of Algorithm 3.

3 INDOOR CROWD MODEL

3.1 Model Structure

As an extension of the accessibility graph [17], the *indoor crowd model* is a directed, labeled graph $G(V, E, L_V, L_E)$ where

1) V is the set of vertices, each for an indoor partition.
2) E is the set of directed edges such that each edge $e(v_i, v_j, d_k) \in E$ means one can reach v_j from v_i through a door d_k, i.e., $v_i \in D2P_\sqsubset(d_k)$ and $v_j \in D2P_\sqsupset(d_k)$.
3) L_V is the set of vertex labels. Each label in L_V is attached to a partition and captured as a five tuple $[v_i, Area(v_i), \mathsf{M}_{d2d}, \tau, (\mathsf{P}^i_{t_l}, t_l)]$. In particular, v_i identifies the associated partition, $Area(v_i)$ is v_i's area, M_{d2d} is a matrix that stores the intra-partition distance (See Equation 1) between each pair of doors of v_i. In addition, $\tau \in \{\mathsf{R}, \mathsf{Q}\}$ indicates the type of v_i's crowd and $(\mathsf{P}^i_{t_l}, t_l)$ means that v_i's absolute population at a latest timestamp t_l is known as $\mathsf{P}^i_{t_l}$. In practice, the model can record the populations at historical timestamps, though only the latest population is relevant to a query.
4) L_E is the edge label set. For an edge $(v_i, v_j, d_k) \in E$, its label consists of a **door flow function** $f(v_i, v_j, d_k)$ that models the dynamic object flows from v_i to v_j via d_k and a local array $\mathsf{F}[t]$ storing the actual object flows at each update timestamp t.

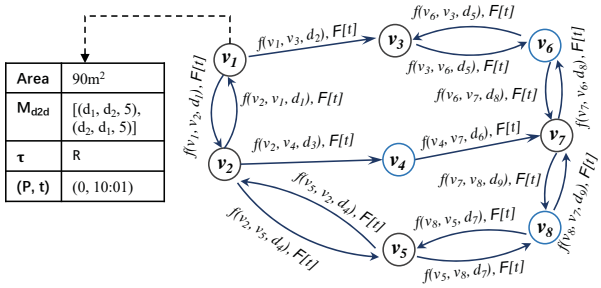

Figure 3: An Example of Indoor Crowd Model

Figure 3 depicts the indoor crowd model corresponding to the space in Figure 1. Unlike a general time-dependent graph (GTG) [11,

44], our model represents doors as edges and partitions as vertices. A GTG may model doors as vertices and partitions as edges and capture time-varying populations or distances as edge weights, but this way falls short in solving our problem. First, GTG's vertices fail to capture the door directionality (e.g., unidirectional security check doors) directly. Referring to Figure 1, d_2 is unidirectional such that one can only go through d_2 from v_1 to v_3. In a GTG, the edges cannot be directed because each edge connects two doors and one can always go from one door to any other door in the same room. E.g., one cannot go through d_2 from v_3 to v_1, but she can go to any door in v_1 from d_2 if she is in v_1. The directionality information can be added in each node, e.g., that for node d_2 can be $\{(v_1, v_3)\}$, and that for node d_1 can be $\{(v_1, v_2), (v_2, v_1)\}$. However, it will result in considerably more space and search costs. Second, a GTG will result in many door-to-door edges for the same partition, which will render the graph-based search inefficient. The experimental comparison with GTG is reported in Section 6.

The time-evolving function $f(v_i, v_j, d_k)$ models the number of objects flowing from v_i to v_j at each report time interval of d_k. In practice, it can be implemented as a time-series prediction model driven by historical data such as ARIMA [8] and LSTM [22], or it can be approximated by a queueing distribution function. For the ease of presentation, in this paper, we employ a specific queueing distribution function to predict the door flows (Section 3.2). Nevertheless, the door flow function can be replaced by other appropriate models or functions, which entails no change to any of the other parts in the overall computation framework (Figure 2).

3.2 Door Flow Function

Following the classic Poisson distribution in queueing theory [7], we design the following door flow function:

$$f(v_i, v_j, d_k) : t \mapsto \mathrm{P}_t, t \in RT(d_k), \mathrm{P}_t \sim Poisson(\lambda) \tag{5}$$

where $t \in RT(d_k)$ is a report timestamp of d_k, P_t is the population that flows from v_i to v_j between t and d_k's next report timestamp, and λ is the expected value of P_t under Poisson distribution.

The door flow function is parameterized by λ and fitted based on a recent period of historical records in a format of $(t', \mathrm{P}_{t'})$. In practice, for each door counter, the most recent timestamps' flows can be accessed from the local array F in the graph edge. An independent thread estimates λ upon such most recent records. Note that the focus of this paper is not to estimate λ based on historical data. For its technical details, we refer readers to a previous work [4]. In our setting, at any query time, an up-to-date door flow function is ready to predict flows for future report timestamps.

4 TIME-EVOLVING POPULATIONS

4.1 Rectifying Door Flows

At a query time t_q, we can access a partition v_k's latest population $\mathrm{P}^k_{t_l}$ at an earlier time $t_l \le t_q$ from the indoor crowd model. To enable the cost estimation for routing, we need to derive v_k's time-evolving population and its future inflows/outflows based on $\mathrm{P}^k_{t_l}$.

Let $[t_0, t_1] \in UTI(v_k)$ be the unit time interval covering t_l. We have $\mathrm{P}^k_{t_0,t_1} = \mathrm{P}^k_{t_l}$, meaning that v_k's population over $[t_0, t_1]$ is equal to $\mathrm{P}^k_{t_l}$. Subsequently, for a future unit time interval $[t_x, t_{x+1}] \in UTI(v_k)$, we compute its population as

$$\mathrm{P}^k_{t_x,t_{x+1}} = \mathrm{P}^k_{t_{x-1},t_x} - out(v_k, t_x) + in(v_k, t_x), \quad x = 1, 2, \ldots \tag{6}$$

where $out(v_k, t_x)$ and $in(v_k, t_x)$ are v_k's estimated outflow and inflow at update timestamp t_x, respectively.

Suppose that all relevant door flow functions are ready at t_q. The inflow and outflow at a future update timestamp can be directly estimated based on the expected values λ. Formally,

$$out(v_k, t_x) = \sum_{d_i \in P2D_{\sqsubset}(v_k) \wedge t_x \in RT(d_i)} \sum_{v_p \in D2P_{\sqcap}(d_i)} f(v_k, v_p, d_i).\lambda$$

$$in(v_k, t_x) = \sum_{d_j \in P2D_{\sqcap}(v_k) \wedge t_x \in RT(d_j)} \sum_{v_q \in D2P_{\sqsubset}(d_j)} f(v_q, v_k, d_j).\lambda$$

where d_i (resp. d_j) is a leaveable (resp. enterable) door updated at time t_x and $v_p \in D2P_{\sqcap}(d_i)$ (resp. $v_q \in D2P_{\sqsubset}(d_i)$) is its enterable (resp. leaveable) partition.

However, the estimated flows may be contrary to the real situation such that a partition's current population ($\mathrm{P}^k_{t_{x-1},t_x}$ in Equation 6) cannot satisfy its outflow ($out(v_k, t_x)$ in Equation 6). In this case, flows at doors should be rectified.

A basic idea is to rectify the expected outflow at each step such that it is not larger than the partition v_k's current population. Meanwhile, v_k's inflow is naturally rectified as it is derived from the outflows of its adjacent partitions at the previous step. In general, a dependency exists between partitions. It demands a suitable way to rectify the relevant outflows at the update timestamps.

An example is depicted in Figure 4, which rectifies the door flows globally. To ease the presentation, at each particular update timestamp t_x we put the absolute populations and door flows in a $|V| \times |V|$ matrix M, where $|V|$ corresponds to the total number of partitions. In particular, $\mathsf{M}[i, i]$ refers to partition v_i's absolute population over unit time interval $[t_{x-1}, t_x]$, while $\mathsf{M}[i, j]$ $(i \neq j)$ means the flow value from partition v_i to v_j over the next unit time interval $[t_x, t_{x+1}]$. Referring to Figure 4(a), partition v_1's population over $[t_{x-1}, t_x]$ is 3 and that of v_2 is 7. Besides, v_1's expected outflows to v_2 and v_3 are 4 and 2, respectively; v_2's inflow from v_1 and v_3 are 4 and 1, respectively. Considering the space efficiency, in the implementation, we store the absolute populations on the graph nodes and the estimated flows on graph edges. That is, the space complexity at each update timestamp is $|V + E|$.

A rectification is then applied to each row of the original matrix as exemplified in Figure 4(b). Specifically, v_1's current population (i.e., 3) is less than the summation of its subsequent outflows (i.e., $4 + 2 = 6$). In this case, we scale down the outflows at all doors to ensure that the actual number of objects outflowing is exactly equal to the current population. That is, $\mathsf{M}'[1, 2] = \mathsf{M}[1, 2] \cdot (3/6) = 2$ and $\mathsf{M}'[1, 3] = \mathsf{M}[1, 3] \cdot (3/6) = 1$, where M and M′ represent the original and rectified matrix, respectively. Note that non-integer values may appear in the rectification. For the computation precision, we use non-integer values in the whole iterative derivation process. Intuitively, the values in the matrix are a probability estimate, i.e., how likely an object will appear in or move to a certain partition.

After the rectification, each partition's population over the next interval $[t_x, t_{x+1}]$ is computed based on Equation 6. In particular, partition v_i's new population is obtained by deducting the overall

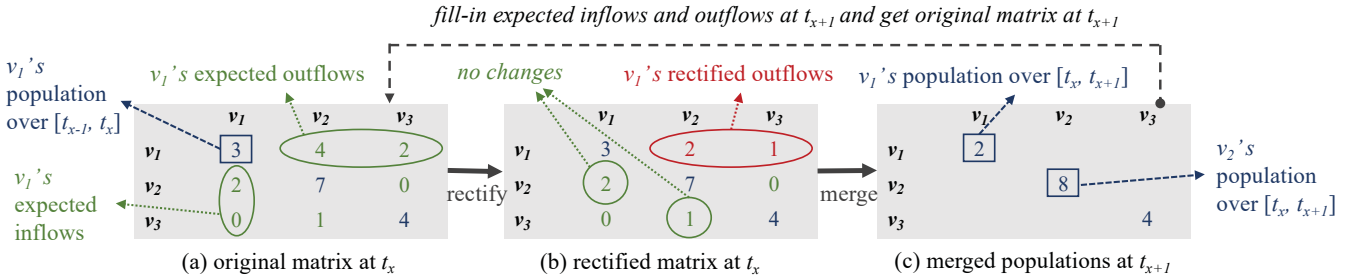

Figure 4: An Example of Rectifying Flows Globally

outflows at the i-th row of M' and then adding the overall inflows at the i-th column of M'. Referring to Figure 4(c), v_1's new population is $3 - 3 + 2 = 2$ while v_2's is $7 - 2 + 3 = 8$. After the merges on each partition, we fill in the matrix inflows and outflows at the timestamp t_{x+1}, and derive the populations iteratively.

4.2 Implementation of Population Estimator

This section presents two versions of population estimators. The *global estimator* estimates all partitions' populations globally (corresponding to the example in Figure 4), whereas the *local estimator* only estimates a relevant partition's population by looking up its upstream partitions flows.

Algorithm 1 PopulationGlobal (future timestamp t^a, indoor crowd model G)

1: get the latest update timestamp t_l^G from G
2: $UT_G \leftarrow \bigcup_{d_j \in G.D} RT(d_j)$
3: A $\leftarrow$ *toArray*($\{t_c \mid t_c \in UT_G \wedge t_c \geq t_l^G \wedge t_c \leq t^a\}$)
4: **for** $t_c \in$ A **do**
5: **for** $e(v_i, v_j, d_k) \in G.E$ **do**
6: **if** $t_c \in RT(d_k)$ **then** $e.\mathsf{F}[t_c] \leftarrow f(v_i, v_j, d_k).\lambda$
7: **else** $e.\mathsf{F}[t_c] \leftarrow 0$
8: **for** $v_i \in V$ **do**
9: **for** $d_k \in P2D_{\sqsubset}(v_i)$ **do**
10: $out_i \leftarrow out_i + (v_i, v_j, d_k).\mathsf{F}[t_c]$
11: get v_i's latest population record (P_t^i, t) from G
12: **if** $\mathsf{P}_t^i - out_i < 0$ **then**
13: **for** $d_k \in P2D_{\sqsubset}(v_i)$ **do**
14: $(v_i, v_j, d_k).\mathsf{F}[t_c] \leftarrow (v_i, v_j, d_k).\mathsf{F}[t_c] \cdot \mathsf{P}_t^i / out_i$
15: **for** $v_i \in V$ **do**
16: **for** $d_k \in P2D_{\sqsubset}(v_i)$ **do**
17: $out_i \leftarrow out_i + (v_i, v_j, d_k).\mathsf{F}[t_c]$
18: **for** $d_k \in P2D_{\sqcap}(v_i)$ **do**
19: $in_i \leftarrow in_i + (v_j, v_i, d_k).\mathsf{F}[t_c]$
20: $\mathsf{P}_{t_c}^i \leftarrow \mathsf{P}_t^i - out_i + in_i$; add $(\mathsf{P}_{t_c}^i, t_c)$ to $G.v_i$

The global estimator (Algorithm 1) takes the indoor crowd model G as input and derives the populations from the latest update timestamp in G to a future timestamp $t^a \geq t_q$. In the beginning, the globally latest update timestamp t_l^G over all partitions is obtained (line 1). Then, the set UT_G of all doors' report timestamps is obtained (line 2) and the period of interest is extracted out of UT_G and organized into an array A (line 3). The algorithm then progressively derives the populations for each timestamp in A (lines 4–20). For each timestamp $t_c \in$ A, the algorithm iterates on each edge $e(v_i, v_j, d_k)$. If the corresponding door d_k updates at t_c, e's flow value at t_c, i.e., $e.\mathsf{F}[t_c]$, is assigned with the estimated flow of the corresponding flow function (line 6). Otherwise, the flow value $e.\mathsf{F}[t_c]$ is assigned with 0 (line 7). Here $e.\mathsf{F}$ is a local array to maintain the rectified flow at each timestamp. Subsequently, the algorithm goes through each partition v_i and aggregates its expected outflow out_i (lines 8–10). If out_i is greater than the population P_t^i at the latest update timestamp, the estimated flows at edges need to be rectified following the example in Figure 4 (lines 11–14). Afterwards, the current timestamp t_c's population for each partition is computed according to Equation 6 and added to G (lines 15–20). Once the process is finished, all partitions' populations at each timestamp can be accessed from the edge nodes of G.

Algorithm 1 needs to update the populations for all partitions. This incurs much unnecessary computation since a path planning search at a particular time only concerns a number of relevant partitions and their populations. If these partitions' populations and flows can be precisely derived without a global update, the overall updating cost can be reduced substantially.

The local estimator is formalized in Algorithm 2. It derives a specific partition's population in a recursive manner. Its preparation (lines 1–3) is almost the same as the counterpart in Algorithm 1, except that the latest update timestamp t_l in line 1 is with respect to the input partition v_i.

Next, the algorithm derives v_i's population in reverse temporal order (lines 4–24). Specifically, the newest update timestamp t_c in A is archived and removed from A (line 5). If t_c just equals t_l, the population is directly obtained from G (lines 6–8). Otherwise, t_c is an earlier timestamp to t_l, and the algorithm recursively calls Algorithm 2 to obtain v_i's population $\mathsf{P}_{t_c}^i$ at t_c (line 9). Once $\mathsf{P}_{t_c}^i$ is derived, the expected flow from each upstream door flow function (see lines 10-12) is compared to v_i's population. Note that the intermediate results are maintained in each edge's local array $\mathsf{F}[t]$ to avoid repeated computations (line 11). If an expected outflow is larger than $\mathsf{P}_{t_c}^i$, a rectification is performed (lines 14–16). In this case, the rectified outflow is assigned with $\mathsf{P}_{t_c}^i$ (line 17). Then, the inflows from all enterable doors are also derived (lines 18-22). For each enterable door d_k, if its inflow has not been derived, Algorithm 2 is recursively called to get the adjacent partition v_j's population at time t_c (line 18-20). Note that the inflow from v_j to v_i will be rectified in this recursion. After that, v_i's overall inflow is obtained (line 21) and its population is computed (line 22). The last two lines of Algorithm 2 return the population nearest to the input time t^a.

Once the partition's population at a particular arrival time is derived, the corresponding partition-passing cost (time or contact) can be computed easily according to Equation 3 or 4. Both global and local population estimators can be utilized by the exact search presented in Section 5.1 (see line 17 in Algorithm 3).

Algorithm 2 PopulationLocal (partition v_i, future timestamp t^a, indoor crowd model G)

1: get v_i's latest update timestamp t_l from G
2: $UT_G \leftarrow \bigcup_{d_j \in G.D} RT(d_j)$
3: A $\leftarrow$ *toArray*($\{t_c \mid t_c \in UT_G \wedge t_c \geq t_l \wedge t_c \leq t^a\}$), $t_{max} \leftarrow$ A.*max*()
4: **while** A is not empty **do**
5: $t_c \leftarrow$ A.*max*(); A $\leftarrow$ A $\setminus$ t_c
6: **if** $t_c = t_l$ **then**
7: get v_i's latest population record ($\mathsf{P}^i_{t_l}, t_l$) from G
8: $\mathsf{P}^i_{t_c} \leftarrow \mathsf{P}^i_{t_l}$
9: **else** $\mathsf{P}^i_{t_c} \leftarrow$ PopulationLocal (v_i, t_c, G)
10: **for** $d_k \in P2D_{\sqsubset}(v_i)$ **do**
11: **if** $(v_i, v_j, d_k).\mathsf{F}[t_c]$ is null **then**
12: $(v_i, v_j, d_k).\mathsf{F}[t_c] \leftarrow f(v_i, v_j, d_k).\lambda$
13: $out_i \leftarrow out_i + (v_i, v_j, d_k).\mathsf{F}[t_c]$
14: **if** $\mathsf{P}^i_{t_c} - out_i < 0$ **then**
15: **for** $d_k \in P2D_{\sqsubset}(v_i)$ **do**
16: $(v_i, v_j, d_k).\mathsf{F}[t_c] \leftarrow (v_i, v_j, d_k).\mathsf{F}[t_c] \cdot \mathsf{P}^i_{t_c}/out_i$
17: $out_i \leftarrow \mathsf{P}^i_{t_c}$
18: **for** $d_k \in P2D_{\sqcap}(v_i)$ **do**
19: **if** $(v_j, v_i, d_k).\mathsf{F}[t_c]$ is null **then**
20: PopulationLocal (v_j, t_c, G)
21: $in_i \leftarrow in_i + (v_j, v_i, d_k).\mathsf{F}[t_c]$
22: $\mathsf{P}^i_{t_c} \leftarrow \mathsf{P}^i_{t_l} - out_i + in_i$
23: $t \leftarrow t_{max}$
24: **return** P^i_t

5 QUERY PROCESSING ALGORITHMS

5.1 Exact Algorithms for FPQ and LCPQ

On top of the indoor crowd model (Section 3), we propose an indoor path search process in Algorithm 3. Following the spirit of Dijkstra's algorithm, our algorithm can handle both FPQ and LCPQ, as long as a cost measure corresponding to a specific query type is set as the routing cost of the graph traversal. Algorithm 3 returns an indoor path ϕ from the source p_s to the target p_t that satisfies the query type QT for a particular query time t_q.

Algorithm 3 Search (source p_s, target p_t, query time t_q, indoor crowd model G, query type QT)

1: initialize a priority queue Q
2: **for** each door $d_i \in G.D$ **do** $prev[d_i] \leftarrow null$
3: $prev[p_s] \leftarrow null$; $prev[p_t] \leftarrow null$
4: $UT_G \leftarrow \bigcup_{d_j \in G.D} RT(d_j)$
5: $t_l \leftarrow \max\{t \in UT_G \mid t \leq t_q\}$; $t^a \leftarrow \emptyset$ ▷ latest update time t_l and arrival time t^a
6: **if** QT = LCPQ **then** $cost_0 \leftarrow (0,0,0)$ **else** $cost_0 \leftarrow (0,0)$ ▷ (distance, time, contact) for LCPQ and (distance, time) for FPQ
7: $S_0 \leftarrow (p_s, cost_0)$ ▷ S(node, cost)
8: $\mathsf{A}_S[p_s] \leftarrow S_0$; Q.*push*($S_0$)
9: **while** Q is not empty **do**
10: $S_i \leftarrow$ Q.*pop*(); $d_i \leftarrow S_i$.node
11: **if** $d_i = p_t$ **then return** GetPath(p_t, *prev*, p_s)
12: **if** $d_i = p_s$ **then** $v \leftarrow host(p_s)$
13: **else** $v \leftarrow D2P_{\sqcap}(d_i) \setminus d_i$'s previous partition
14: mark d_i as visited
15: $t^a \leftarrow \max\{t \in UT_G \mid t \leq t_q + S_i.\text{cost.time}\}$
16: **if** $t^a > t_l$ **then** ▷ further derive populations
17: Population(t^a, G)
18: $t_l \leftarrow t^a$
19: **if** $d_i \in P2D_{\sqcap}(host(p_t))$ **then**
20: Expand(d_i, p_t, G, v, t^a, S_i, QT) ▷ towards target p_t
21: **for** each unvisited door $d_j \in P2D_{\sqsubset}(v)$ **do**
22: Expand(d_i, d_j, G, v, t^a, S_i, QT)

The algorithm starts with initializing a priority queue Q (line 1) whose priority is the minimum travel time for FPQ and the minimum contact for LCPQ. Also, an array *prev* is initialized to record each path node's previous node in the search (lines 2–3). Then, the full set UT_G of the report timestamps over all doors are obtained (line 4). With respect to the query time t_q, the latest update timestamp in UT_G is found and assigned to t_l (line 5). Variable t_l is the latest population derivation time in the search. Next, the cost of the current search is initialized (line 6). The cost for LCPQ consists of overall travel distance, overall travel time, and overall contact value. The cost for FPQ only contains the first two. The source and initial cost are put into a stamp S_0, and S_0 is pushed into Q and maintained in an array A_S as well (lines 7–8).

After the preparation, the algorithm explores the next path node towards p_t in an order controlled by Q (lines 9–22). Specifically, the stamp S_i with the lowest cost is popped from Q, and the corresponding path node d_i is obtained (line 10). If d_i is p_t, i.e., the searched is complete, the algorithm calls a function GetPath(p_t, *prev*, p_s) to return the reverse path from p_t to p_s (line 11). Otherwise, the algorithm explores the next path node as follows.

First, if the current node is the source p_s, the current partition v is obtained as the host of p_s (line 12). Otherwise, v is assigned as d_i's enterable partition[3] (line 13). Then, d_i is marked as visited (line 14). Next, the estimated cost to pass v is obtained as S_i.cost.time and it is added to the query time t_q to get the arrival time t^a to the next path node (line 15). An alignment to the update timestamps in UT_G is needed for t^a. The algorithm then determines if the population needs to be derived to meet the next arrival time t^a (lines 16–18). If the latest derivation time t_l is earlier than t^a, a population estimator is invoked to get v's derived populations up to t^a (line 17). Here, either the global (Algorithm 1) or local estimator (Algorithm 2) can be used. The performance difference of these two ways will be experimentally studied in Section 6.

Afterward, it expands to the next node from the current node d_i. If d_i is an enterable door of p_t's host partition, the expansion goes towards p_t (lines 19-20). Regardless of whether the current path reaches p_t's host partition, the expansion should also reach each unvisited leaveable door of the current partition v (lines 21–22). This ensures that the planned path can leave and re-enter p_t's host partition when the host is currently too crowded.

The function Expand is formalized in Algorithm 4, which expands from the current node p_1 to the next possible node p_2 through partition v. First, it estimates the cost to reach p_2 from p_1 through an inline function Cost (see lines 7–16). In particular, the distance between p_1 and p_2 is obtained as the door-to-door distance if both are doors, or Euclidean distance if either is an indoor point (lines 8–10). Then, the population of the partition v_k to pass is obtained from G (line 11), and the partition-passing time and contact are computed based on Equations 3 and 4, respectively (lines 12 and 14). The corresponding cost is then returned according to the query type QT (lines 13 and 15–16).

Back to Expand in Algorithm 4, once the cost to pass v is obtained, it is added to the current stamp S_i's cost to get the overall cost in the current expansion, i.e., $cost_c$ (line 1). The tuple-form cost is summed in an element-wise way. Next, the estimated cost to reach p_2 so far is obtained from the array A_S (line 2). The algorithm compares the current estimated cost $cost_c$ to the previously recorded cost $cost_{pre}$. If $cost_{pre}$ does not exist or $cost_c$ is lower, a valid expand is performed (lines 3–6). Specifically, a new stamp S' is formed with

[3] To ease presentation, here we only show the case that a door connects two partitions. A complex space can be handled by maintaining a collection of enterable partitions.

the next path node p_2 and the new cost $cost_c$. It is then pushed to Q. If an old stamp exists with the same node, the old stamp is updated by S' (line 5). Then, S' is inserted into A_S and p_2's previous path node is recorded as p_1.

Algorithm 4 EXPAND (start node p_1, end node p_2, indoor crowd model G, partition v, arrival time t^a, stamp S_i, query type QT)

```
1: cost_c ← S_i.cost + COST(p_1, p_2, v, t^a, G)                ▷ element-wise
2: cost_pre ← A_S[p_2].cost
3: if cost_pre is null or cost_c < cost_pre then
4:     S' ← (p_2, cost_c)
5:     Q.push(S')                                               ▷ update if exists
6:     A_S[p_2] ← S'; prev[p_2] ← p_1
7: function COST(p_1, p_2, v_k, t^a, G)
8:     dist ← ∅
9:     if p_1, p_2 are both doors then dist ← v_k.M_d2d(p_1, p_2)
10:    else dist ← |p_1, p_2|_E
11:    get P^k_{t^a} from G
12:    time ← T(p_1, p_2, v_k, t^a)                             ▷ Equation 3
13:    if QT = LCPQ then
14:        contact ← κ(p_1, p_2, v_k, t^a)                      ▷ Equation 4
15:        return (dist, time, contact)
16:    else return (dist, time)
```

In the exact search, the time-evolving populations are rectified and computed rigidly timestamp by timestamp. This may result in a bottleneck in the graph traversal. We intend to reduce the workload for population derivation by approximation. On the one hand, the severity of population derivation can be skipped for those less important partitions, e.g., those far away from the current partition to pass. On the other hand, some of the update timestamps in the iterative derivation can be skipped if the population changes within that iteration period is relatively stable. We proceed to introduce two approximate search algorithms for FPQ and LCPQ.

5.2 Approximate Algorithms for FPQ and LCPQ

We design two strategies to derive approximate populations.

Strategy 1: Population Derivation for Partial Partitions (PP). Recall that the population derivation in Algorithm 2 (see line 20) recursively obtains its adjacent partition's population to ensure the overall derivation is fully precise. This recursion terminates when the outflows of all relevant partitions at all relevant update timestamps have been rectified. In fact, the door flows from a long distance or at a very old timestamp only have a slight impact on a partition's current population. Therefore, one option is to rectify only the outflows of the current relevant partition without strictly processing the outflows of its upstream partitions (i.e., the inflows to the current relevant partition). To this end, only a minor change is made to Algorithm 2: line 20 is modified to $(v_j, v_i, d_k).\mathsf{F}[t_c] \leftarrow f(v_j, v_i, d_k).\lambda$. That is, the outflow of an adjacent partition v_j is directly obtained from the corresponding door flow function.

Strategy 2: Population Derivation at Necessary Timestamps (NT). To further speed up the population derivation for individual partitions, we consider reducing the workload by only calling Algorithm 2 at some necessary timestamps. The general idea is that when we observe that the historical flows of a partition are relatively stable, we skip the iterative population computations and directly estimate its population at the arrival time t^a. Note that here Strategy 2 is used in combination with Strategy 1 to achieve the maximum effect of acceleration.

In particular, when the search visits a partition v_k, the mean μ and standard deviation σ of its flow difference (i.e., inflow deducts outflow) in the historical timestamps are computed as follows.

$$\mu = \Big(\sum\nolimits_{t_x \in \mathsf{UT}_{past}} \big(in(v_k, t_x) - out(v_k, t_x) \big) \Big) / |\mathsf{UT}_{past}|$$

$$\sigma = \Big(\big(\sum\nolimits_{t_x \in \mathsf{UT}_{past}} (in(v_k, t_x) - out(v_k, t_x) - \mu)^2 \big) / |\mathsf{UT}_{past}| \Big)^{1/2}$$

where UT_{past} is a set of the historical update timestamps of v_k. The update timestamps in UT_{past} will be obtained from the local array that we maintain for fitting door flow function in Section 3.2.

If σ is smaller than a pre-defined threshold value η, it indicates that v_k's historical flows change only slightly. Thus, we directly estimate v_k's population according to its historical trend as follows.

$$\mathsf{P}^k_{t^a} = \mathsf{P}^k_{t_l} + \mu \cdot |\{t \in UT(v_k) \mid t_l < t \wedge t \leq t^a\}| \tag{7}$$

where $t_l = \max\{t_x \in UT_G \mid t_x \leq t_q\}$ is the latest population update time as in line 5 of Algorithm 3, μ is the mean of historical flow differences, and $|\{t \in UT(v_k) \mid t_l < t \wedge t \leq t^a\}|$ is the number of skipped update timestamps from t_l to t^a. In our experiment, $\eta = 3$ achieves the best performance approaching exact search results.

Otherwise, the search has to call Algorithm 2 (applied with Strategy 1) to derive population for v_k.

5.3 Complexity Analysis

The main difference of the four algorithms' complexity is related to population derivation. Therefore, we focus on comparing the four ways of population derivation. Assume that we estimate a partition's population at a future timestamp, and the derivation involves k unit time intervals.

The time complexity of the global population estimator (Algorithm 1) is $k|V| \cdot \mathsf{u}$, where u is the unit computational cost for a partition at an update timestamp. For the local estimator (Algorithm 2) which only considers the current partition and its upstream partitions, its time complexity is $\big(k|V| - ((k-1)n^k + (k-2)n^{k-1} + \cdots + n^2)\big) \cdot \mathsf{u}$, where n is the average number of enterable door per partition.

For two approximate strategies, PP rectifies only the outflows of the current certain partition without strictly processing the outflows of its upstream partitions, so the time complexity of PP's population derivation per partition is only $k\mathsf{u}$. NT omits population estimations for some partitions with the relatively stable flow. Thus for a partition in consideration, the time complexity depends on its flow stability. That is, if it is stable, we do not estimate the population; otherwise, the complexity is also $k\mathsf{u}$.

6 EXPERIMENTS

For either FPQ or LCPQ, we implement four search algorithms. Specifically, *PQ is Algorithm 3 calling Algorithm 2, *PQ-G is Algorithm 3 calling Algorithm 1, *PQ-PP is the approximate search using Strategy PP, and *PQ-NT is the approximate search using Strategy NT. All algorithms are implemented in Java and run on a PC with a 2.30GHz Intel i5 CPU and 16 GB memory.

6.1 Results on Synthetic Data

6.1.1 Settings. **Indoor Space.** Using a real-world floorplan[4], we generate a multi-floor indoor space where each floor takes 1368m

[4]https://longaspire.github.io/s/fp.html (Last accessed date: 2021/04/16)

× 1368m. The irregular hallways are decomposed into smaller but regular partitions following the decomposition algorithm in [43]. As a result, we obtain 141 partitions and 216 doors on each floor. We duplicate the floorplan 3, **5**, 7, or 9 times to simulate different indoor spaces. All parameter settings are listed in Table 2 with default values in bold. The four staircases of each two adjacent floors are connected by stairways, each being 20m long. On each floor, we randomly pick 14 out of all those partitions having two doors as the Q-partitions while regarding all others as R-partitions.

Table 2: Parameter Settings

Parameters	Description	Settings
floor	Floor number	3, **5**, 7, 9
$\|o\|$	Partition's maximum object number	300, **600**, 900, 1200, 1500
TI (s)	Time interval	5, **10**, 15, 20
s2t (m)	The shortest distance from p_s to p_t	900, 1100, **1300**, 1500, 1700

Populations and Flows. We generate each partition's population at an initial time randomly from 0 to $|o|$ (see Table 2). We set the max capacity of a partition v as $Area(v) \cdot \beta$ (β is 1 per m^2 in this paper). Note that the initial population will not exceed the max capacity. The parameter λ of each door flow function is varied from 0 to 3 [5]. We use a variable *TI* (5, **10**, 15, or 20 seconds) to control the length of the unit update time interval of partitions. To this end, all doors' initial report timestamps are aligned and they only report the flows in every $n \cdot TI$ seconds ($n = 1, 2, \ldots, 5$ is randomly decided for each door counter).

Query Instances. We use a parameter *s2t* to control the shortest distance from the source point p_s and the target point p_t. First, we randomly select a point p_s from the indoor space. Second, we find a door d whose indoor distance to p_s approximates *s2t*. Then, we expand from d to find a random point p_t whose indoor distance to p_s approaches *s2t*. For each *s2t* value, we generate 100 different pairs to form query instances.

Baseline Methods. We use a general time-dependent graph (GTG) to form a baseline. Each vertex in GTG represents a door and the weight of each edge is the cost between two doors, i.e., the time cost for FPQ or the contact for LCPQ. To be fair, we employ a Dijkstra-based algorithm (*PQ-GTG) without precomputation and combine it with our exact population estimator to process queries. Since GTG fails to represent the door directionality directly, we assume all doors are bidirectional in comparative experiments. Another baseline is the adaptive method based on the indoor crowd model (*PQ-A) that keeps updating and recomputing the optimal route at every node until the user gets to the target point.

Performance Metrics. To compare the efficiency of different search algorithms, we run each query instance ten times and measure their average running time and memory cost. We also look into the accuracy of the four searches. In particular, the query *hit rate* is the fraction of query instances whose search result equals its gold standard result among all 100 instances. The gold result is returned by searching over the detailed simulated trajectories. Moreover, we measure the *relative error* of the estimated routing cost against the gold result. The estimated cost refers to overall travel time T_ϕ for FPQ and overall contact κ_ϕ for LCPQ. Taking FPQ as an example, the relative error is $\gamma = |T_\phi^{(E)} - T_\phi^{(G)}|/T_\phi^{(G)}$ where $T_\phi^{(E)}$ and $T_\phi^{(G)}$ is the overall travel time corresponding to the exact search and gold result, respectively.

6.1.2 Search Performance of FPQ. **Comparison in default setting.** The measures of different FPQ algorithms are reported in Table 3. FPQ-NT performs the best in terms of the running time and memory because it skips the iterative population computations and directly estimates its population at the arrival time in each node. FPQ and FPQ-G perform similarly as two exact searches, implying that the two exact estimators achieve similar efficiency in the default setting. Besides, they are the best in terms of hit rate and relative error. The baseline FPQ-GTG uses the exact estimator that we propose, so its accuracy is the same as FPQ and FPQ-G. However, FPQ-GTG incurs the highest time and memory costs due to the large size of GTG (cf. Section 3). FPQ-PP works as accurately as the exact algorithms, which reflects the effectiveness of Strategy PP. Also, FPQ-PP saves some time and memory. FPQ-NT and FPQ-A perform worse in terms of hit rate and relative error. FPQ-NT skips some intermediate update timestamps, making its population derivation less accurate. FPQ-A expands to next nodes by reevaluation, making its result only optimal locally rather than globally. Note that the running time (and memory) of FPQ-A is the sum of that at all nodes in a path. Indeed, FPQ-A keeps updating until a user gets to the target point, while other methods return the path before departure. We omit FPQ-GTG and FPQ-A in the subsequent experiments as the comparison results show a similar trend to that here.

Effect of *s2t*. We vary the distance *s2t* between p_s and p_t from 900m to 1700m and test the four FPQ algorithms. Referring to Figure 5, all algorithms' running time increases linearly with the source-target distance, since a larger *s2t* involves a larger expansion range as well as more candidate path nodes. Among all algorithms, FPQ-NT runs fastest because it skips the iterative population computation and directly estimates its population at the arrival time in each node. Moreover, the time costs of approximate searches FPQ-PP and FPQ-NT increase very slowly as *s2t* increases. In contrast, the exact searches FPQ and FPQ-G are sensitive to *s2t* because they need to compute more population.

Figure 6 reports on the memory consumption. The memory use of FPQ and FPQ-G grows faster than the others due to an extra cost of rigid population derivation. Contrary to our intuition, FPQ incurs more memory cost than FPQ-G. In our test, the search framework needs to explore a large number of partitions. FPQ-G's global population derivation shares intermediate results across all partitions. For a large *s2t*, FPQ-G may find more shared intermediate results, and thus consumes less memory than FPQ.

Figure 7 reports on the relative errors, for FPQ, FPQ-PP and FPQ-NT.[6] For different *s2t* values, FPQ achieves a lower error. As a sacrifice for search efficiency, FPQ-NT skips some intermediate update timestamps, so its accuracy of population derivation drops more significantly. As *s2t* increases, FPQ-NT deteriorates rapidly while FPQ and FPQ-PP perform quite stably. A larger *s2t* leads to more updated timestamps to derive populations. As a result, FPQ-NT's relatively aggressive strategy of skipping timestamps incurs more estimation errors. In contrast, FPQ and FPQ-PP derive

[5]The value is set according to our analysis of real data. The door flow of a hallway/staircase is relatively more than that of a room.

[6]We exclude FPQ-G as its accuracy is the same with FPQ.

Table 3: Comparison of Algorithms for FPQ and LCPQ on Synthetic Data (best result in bold)

	FPQ	FPQ-G	FPQ-PP	FPQ-NT	FPQ-GTG	FPQ-A	LCPQ	LCPQ-G	LCPQ-PP	LCPQ-NT	LCPQ-GTG	LCPQ-A
Running Time (ms)	584	585	208	**25**	2857	189	446	461	131	**20**	2532	163
Memory (KB)	115	112	111	**12**	278	14	182	192	144	**7**	257	8
Hit Rate (%)	**98**	**98**	**98**	95	**98**	94	83	83	83	60	83	**87**
Relative Error	**4.37E-08**	**4.37E-08**	**4.37E-08**	8.09E-08	**4.37E-08**	0.1233	**0.0128**	**0.0128**	0.0129	0.1113	**0.0128**	0.1256

populations timestamp by timestamp, and so the impact of *s2t* is slight.

Effect of *TI*. According to Figure 8, all four algorithms run faster with a larger update time interval *TI*. Still, FPQ-NT performs best in both measures as it approximates population derivation in both time and space aspects. We omit the result of the memory cost because it has a similar trend with the running time. On the other hand, referring to Figure 9, FPQ-NT's relative error decreases as all doors' *TI* enlarges. This shows that one may consider skipping more timestamps when the flow update at doors is not that frequent.

Effect of *floor*. We vary the floor number to test the scalability of our algorithms. Referring to Figure 10, all algorithms' search time increases steadily with more floors since more candidate path nodes are involved. FPQ-PP and FPQ-NT run faster than FPQ and FPQ-G. Moreover, the running time of the two approximate searches grows more slowly. Figure 11 reports the memory cost of the four algorithms. FPQ-NT needs less memory and is more scalable since it skips some timestamps. We omit the results of relative errors. In the tests, both measures are insensitive to the floor number since the returned path stays unchanged for a given query instance.

We omit the results of varying $|o|$ on FPQ because different initial object numbers have little impact on the search performance.

6.1.3 Search Performance of LCPQ. **Comparison in default setting.** The resulting trend of LCPQ is similar to that of FPQ. As reported in the right part of Table 3, LCPQ-NT is the best in terms of running time and memory due to its skipping strategy, while LCPQ-GTG incurs the highest time and memory costs due to the large graph size. LCPQ-A gets the best hit rate while the exact searches achieve a better result on the relative error. Different from FPQ, LCPQ is highly sensitive to populations. A little error in population derivation can lead to a very different returned path. Hence, the accuracy performance is slightly unstable for the tested algorithms.

Effect of *s2t*. Referring to Figure 12, the running time of each algorithm grows as *s2t* increases. In terms of memory, the results in Figure 13 show that LCPQ and LCPQ-G need more memory than the other two. LCPQ-NT uses the least memory since it skips intermediate timestamps to reduce workload.

Figure 14 reports on the relative errors of exact and approximate searches. Compared to LCPQ and LCPQ-PP, LCPQ-NT incurs significantly higher errors. As we mentioned before, LCPQ query is highly sensitive to the population. A little error in population derivation can lead to a very different returned path. Therefore, LCPQ-NT performs poorly when a larger *s2t* is used.

Effect of *TI*. Referring to Figures 15 and 16, all algorithms incur less time and memory costs as *TI* increases since a larger *TI* leads to fewer callings of population derivation. The approximate approaches LCPQ-PP and LCPQ-NT always perform better in search efficiency. Referring to Figure 17, all algorithms' search effectiveness deteriorates with an increasing *TI*. As less flow information is observed when *TI* becomes larger, the relative errors accumulate. Likewise, LCPQ's search effectiveness is worse than that of FPQ, due to its more stringent requirements on population derivation.

Effect of $|o|$. We test different initial object numbers on LCPQ query processing. As an observation, the running time and memory cost are almost insensitive to $|o|$, since the algorithms do not process each individual object. So we omit the results here. Interestingly, increasing $|o|$ will affect the result accuracy. Referring to Figure 18, as more objects are involved, all methods achieve a lower relative error. We attribute it to that a larger population base is less affected by the flow estimation error and leads to a smaller relative error.

We omit the result about different floor numbers because it exhibits a trend similar to the counterpart of FPQ searches.

6.2 Results on Real Data

We collect a real dataset from a seven-floor, 2700m × 2000m shopping mall in Hangzhou, China. There are ten staircases each being roughly 20m long, and 977 partitions connected by 1613 doors[7]. The max capacity of a partition v is $Area(v) \cdot 1$ per m^2. We collected 1,598 object trajectories with totally more than 90,000 positioning records on 2017/01/05. Nearly 12% of two consecutive locations are not topologically-connected, i.e., not in the same partition or two adjacent partitions. The object movements in-between are uncertain. To count flows against uncertainty, we applied a proven probabilistic method [20] as follows. First, for every two consecutive locations not topologically-connected, a set Φ of valid sub-paths are found. Those sub-paths longer than twice the shortest sub-path are excluded as the object unlikely took them. Second, the probability that the object took sub-path $\phi_i \in \Phi$ is computed as $\mathrm{P}(\phi_i) = \frac{1/length(\phi_i)}{\sum_{\phi_k \in \Phi} 1/length(\phi_k)}$. This way, a shorter sub-path has a higher probability to be taken. Finally, the flow of a door d is the sum of $\mathrm{P}(\phi_i)$s for all ϕ_is through d. On top of the low-level flow computing, we sampled each door's flow every 10 seconds and used the samples to construct our indoor crowd model.

Figure 25 exemplifies a few trajectories, where $m_i(t_j)$ denotes the positioning location of a MAC address m_i at time t_j. For $m_1(t_3)$ and $m_1(t_4)$ that are not topologically-connected, two possible in-between paths are found, namely $\phi_1(m_1(t_3), d_3, d_5), m_1(t_4))$ of 20m long and $\phi_2(m_1(t_3), d_2, d_4), m_1(t_4))$ of 25m long. Their probabilities are $\mathrm{P}(\phi_1) = \frac{1/20}{1/20+1/25} \approx 0.556$ and $\mathrm{P}(\phi_2) = \frac{1/25}{1/20+1/25} \approx 0.444$. We sampled door flows as shown in the right part of Figure 25. E.g., door d_4's flow during $[t'_4, t'_5]$ is $1 + 0.444 = 1.444$ (m_2 with a probability of 1 and m_1 with a probability of 0.444).

Comparison in default setting. We compare different methods for FPQ and LCPQ using real data and report the results in Table 4. In terms of the running time and memory, *PQ-NT performs best while *PQ-GTG is the worst. This is because *PQ-NT skips the iterative population computations while *PQ-GTG uses an exact

[7]We assume that there is no Q-partition in this shopping mall. We varied the fraction of Q-partitions/R-partitions on synthetic data, but it shows little impact on all algorithms.

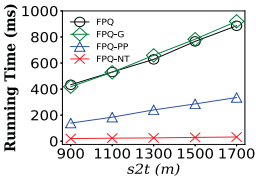

Figure 5: FPQ Time vs. *s2t*

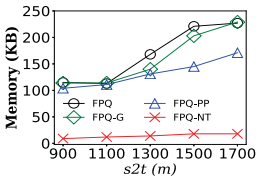

Figure 6: FPQ Mem. vs. *s2t*

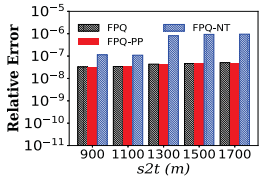

Figure 7: FPQ's γ vs. *s2t*

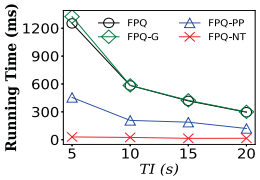

Figure 8: FPQ Time vs. *TI*

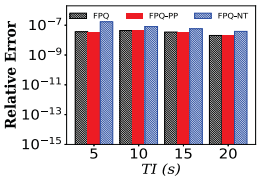

Figure 9: FPQ's γ vs. *TI*

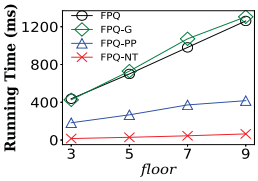

Figure 10: FPQ Time vs. *floor*

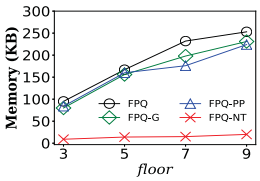

Figure 11: FPQ Mem. vs. *floor*

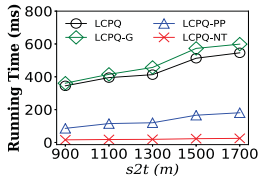

Figure 12: LCPQ Time vs. *s2t*

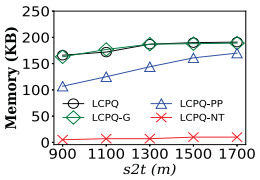

Figure 13: LCPQ Mem. vs. *s2t*

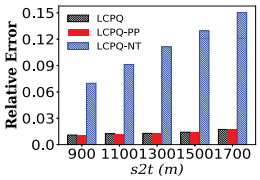

Figure 14: LCPQ's γ vs. *s2t*

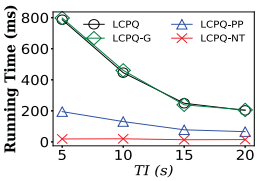

Figure 15: LCPQ Time vs. *TI*

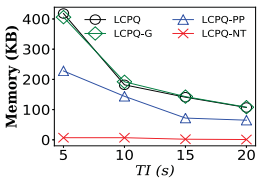

Figure 16: LCPQ Mem. vs. *TI*

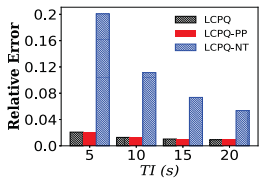

Figure 17: LCPQ's γ vs. *TI*

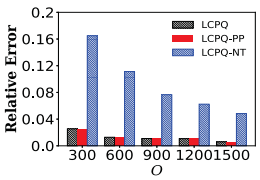

Figure 18: LCPQ's γ vs. *O*

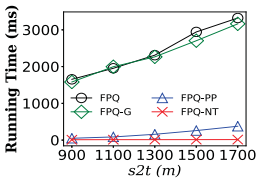

Figure 19: FPQ Time vs. *s2t* (Real)

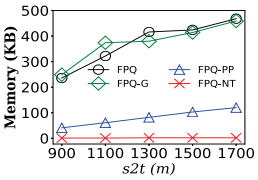

Figure 20: FPQ Mem. vs. *s2t* (Real)

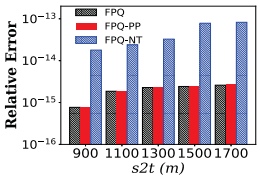

Figure 21: FPQ's γ vs. *s2t* (Real)

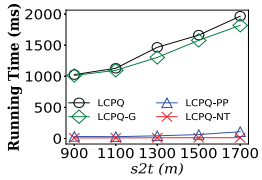

Figure 22: LCPQ Time vs. *s2t* (Real)

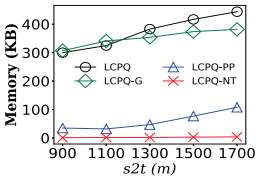

Figure 23: LCPQ Mem. vs. *s2t* (Real)

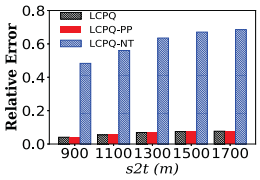

Figure 24: LCPQ's γ vs. *s2t* (Real)

estimator but involves more nodes than does our indoor crowd model. In terms of hit rate and relative error, the results are similar to the counterparts in synthetic data.

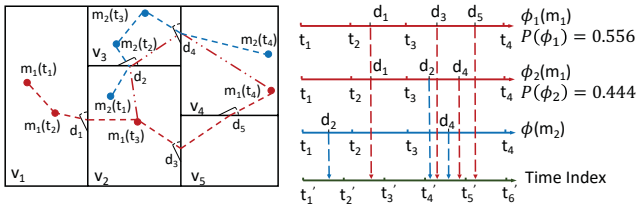

Figure 25: An Example of the Trajectory Data

Effect of *s2t*. Figures 19 and 20 report on running time and memory use of different FPQ searches. All incur more time and memory costs as *s2t* increases. Compared to FPQ-PP and FPQ-NT, FPQ and FPQ-G are more memory- and time-consuming. Compared to the counterparts on synthetic data, the search costs are higher since the shopping mall is of a larger scale. Referring to Figure 21, FPQ-PP/FPQ-NT's search accuracy deteriorates with a larger *s2t*.

The results in Figures 22 and 23 exhibit similar trends as those in Figures 19 and 20. LCPQ-NT's relative error is much higher than that of LCPQ and LCPQ-PP, and it grows faster—more update timestamps involved due to a greater *s2t* lead to less accurate cost estimates. Compared to FPQ, LCPQ has higher relative errors as reported in Figure 24. As defined, partition-passing contact is more sensitive to the derived populations than the partition-passing time.

6.3 Summary of Results

First, in terms of running time and memory, the two approximate searches perform better than the two exact counterparts as workloads reduce. Besides, Strategy NT costs less time and memory than Strategy PP since NT further utilizes historical information to skip timestamps. For hit rate and relative error, PP outperforms NT in that NT skips many timestamps on the basis of PP, further decreasing the accuracy of intermediate results.

Second, the two approximate searches for FPQ perform better than those for LCPQ in terms of hit rate and error rate. The reason is that the partition-passing time is less sensitive to the populations compared to the partition-passing contact.

Third, a larger *s2t* leads to more time and memory consumptions but worse result accuracy, while a larger *TI* exhibits an almost opposite trend. In general, a larger *s2t* or a smaller *TI* means more timestamps for which we need to derive populations. This is critical to the cost estimation. More floors mean more doors/partitions to explore, which lowers the search efficiency. More initial objects render the population derivation spatially more uniform and thus leads to a higher hit rate and lower relative error.

Table 4: Comparison of Algorithms for FPQ and LCPQ on Real Data (best result in bold)

	FPQ	FPQ-G	FPQ-PP	FPQ-NT	FPQ-GTG	FPQ-A	LCPQ	LCPQ-G	LCPQ-PP	LCPQ-NT	LCPQ-GTG	LCPQ-A
Running Time (ms)	1900	1997	67	**11**	25559	53	992	1047	28	**10**	13895	45
Memory (KB)	367	393	61	**1**	669	2	307	341	30	**1**	568	2
Hit Rate (%)	**99**	**99**	**99**	98	**99**	98	88	88	88	67	88	**90**
Relative Error	**1.86E-15**	**1.86E-15**	**1.86E-15**	4.38E-14	**1.86E-15**	0.1492	**0.0546**	**0.0546**	**0.0546**	0.6606	**0.0546**	0.062

Fourth, for the two baselines, *PQ-GTG performs poorly on efficiency because GTG contains more nodes to process. *PQ-A seems good in terms of both efficiency and effectiveness. However, a user of *PQ-A cannot obtain the path before departure because *PQ-A needs to keep updating during expansion.

In general, the results show that the search algorithm with Strategy PP performs best. It costs relatively less time and memory and achieves good query result accuracy. Strategy NT applies well to the cases where door flows are updated frequently. In such a case, skipping some timestamps can improve efficiency without causing excessive errors in population estimates. More experimental results are available in an extended version [25].

7 RELATED WORK

Outdoor Time-Dependent Routing. In this setting, public transportation networks [5, 14, 36] and road networks [2, 3, 11, 29, 39] are modeled as discrete and continuous time-dependent graphs, respectively. Solutions for public transportation networks cannot solve our problem because they are mainly for a time-dependent graph with a static timetable for each station. On the other hand, approaches for road networks do not work for indoor spaces because road network models do not support entities like doors, walls and rooms that together form a complex topology.

From an algorithmic perspective, the solutions for outdoor time-dependent routing are mainly Dijkstra-based algorithms [5, 9, 11, 44], A* algorithms [3, 29], label-based methods [28, 36, 40, 41] (mainly for time-dependent graphs with timetables), and adaptive approaches [2, 3, 13]. Most of these works do not consider crowds that influence people's routing choices.

Traffic-aware Routing. Some works [10, 29, 38] prepare the traffic information by mining historical trajectory data and assume it is known when routing. Shang et al. [30] study traffic-aware fastest path query using a traffic-aware spatial network. Some adaptive approaches [2, 3, 13] can also solve traffic-aware routing problems through continuous reevaluation. Although these works consider the traffic impact, none of them estimates the traffic in the near future when processing a query.

Indoor Routing. Lu et al. [26] propose a distance-aware indoor space model to facilitate indoor shortest path query. To speed up distance-aware indoor pathfinding, Shao et al. [32] design IP/VIP-tree that enable more aggressive pruning. VIP-tree also supports trip planning based on neighbour expansion [31]. Feng et al. [12] study indoor top-k keyword-aware routing query that finds k routes that have optimal ranking scores integrating keyword relevance and distance cost. Other works [18, 23, 24, 27, 45] consider more constraints for indoor routing. However, none of these aforementioned works considers dynamic crowds that are essential to LCPQ and FPQ.

Flows, Crowds and Density. Some existing works study the estimation of flows [33, 34], crowds [6, 37], or dense regions [15, 16] outdoors. However, they all fall short in indoor spaces mainly for two reasons. First, indoor positioning techniques are usually RFID, Wi-Fi, and Bluetooth, which make coarser-grained location data than outdoor GPS data. Second, the indoor topology is so different from the outdoor topology that indoor crowd modeling must consider carefully the connectivity among doors and partitions. Some existing works consider flows and densities in indoor venues. Ahmed et al. [1] propose two graph-based indoor movement models to map raw tracking records into records with object entry and exit times in particular locations. Li et al. propose to find the top-k popular indoor semantic locations [20] with the highest flow values using probabilistic location samples, and the currently top-k indoor dense regions [21] by considering the uncertainty of online positioning reports. However, all these works [1, 20, 21] are different from our work. First, our density analysis is based on coarse-grained flow values reported at door counters, not the point-based localization results count for individual moving objects. Second, our work focuses on path planning in the presence of indoor crowds, while the previous works aim to find interesting location patterns.

8 CONCLUSION AND FUTURE WORK

We study two types of crowd-aware indoor path planning queries. The FPQ returns a path with the shortest travel time in the presence of crowds; the LCPQ returns a path encountering the least objects en route. To solve FPQ and LCPQ, we design a unified framework that consists of 1) an indoor crowd model that organizes indoor topology and captures indoor flows and densities; 2) a time-evolving population estimator that derives future time-dependent flows and populations for relevant partitions; 3) two exact and two approximate query processing algorithms that each can process both query types. We conduct extensive experiments to evaluate our proposals. The results demonstrate the efficiency and scalability of the proposals and disclose the performance differences among all four algorithms.

There exist several directions for future research. First, it is interesting to consider other crowd models, e.g., learning crowd distributions and functions from historical data. Also, it is relevant to further speed up query processing by using an index, e.g., combining the object layer in the composite indoor index [42, 43] with a modified IP/VIP-Tree [32] whose distance matrices are extended with time attributes. Last but not least, it is possible to extend our proposals to support continuous monitoring of the fastest or least crowded paths.

ACKNOWLEDGMENTS

This work was supported by IRFD (No. 8022-00366B), ARC (No. FT180100140 and DP180103411), the Key R&D Program (Zhejiang, China) (No. 2021C009) and NSFC (No. 62050099). Huan Li and Hua Lu are the corresponding authors.

REFERENCES

[1] Tanvir Ahmed, Torben Bach Pedersen, and Hua Lu. 2017. Finding dense locations in symbolic indoor tracking data: modeling, indexing, and processing. *GeoInformatica* 21, 1 (2017), 119–150.

[2] Mostafa K Ardakani and Lu Sun. 2012. Decremental algorithm for adaptive routing incorporating traveler information. *COR* 39, 12 (2012), 3012–3020.

[3] Mostafa K Ardakani and Madjid Tavana. 2015. A decremental approach with the A* algorithm for speeding-up the optimization process in dynamic shortest path problems. *Measurement* 60 (2015), 299–307.

[4] Marllyn Tuley Boswell et al. 1966. Estimating and testing trend in a stochastic process of Poisson type. *AMS* 37, 6 (1966), 1564–1573.

[5] Gerth Stølting Brodal and Riko Jacob. 2004. Time-dependent networks as models to achieve fast exact time-table queries. *ENTCS* 92 (2004), 3–15.

[6] Giovanna Castellano, Ciro Castiello, Corrado Mencar, and Gennaro Vessio. 2020. Crowd detection in aerial images using spatial graphs and fully-convolutional neural networks. *IEEE Access* 8 (2020), 64534–64544.

[7] Prem C Consul and Gaurav C Jain. 1973. A generalization of the Poisson distribution. *Technometrics* 15, 4 (1973), 791–799.

[8] Javier Contreras, Rosario Espinola, Francisco J Nogales, and Antonio J Conejo. 2003. ARIMA models to predict next-day electricity prices. *T-PWRS* 18, 3 (2003), 1014–1020.

[9] Kenneth L Cooke and Eric Halsey. 1966. The shortest route through a network with time-dependent internodal transit times. *JMAA* 14, 3 (1966), 493–498.

[10] Ugur Demiryurek, Farnoush Banaei-Kashani, Cyrus Shahabi, and Anand Ranganathan. 2011. Online computation of fastest path in time-dependent spatial networks. In *SSTD*. 92–111.

[11] Bolin Ding, Jeffrey Xu Yu, and Lu Qin. 2008. Finding time-dependent shortest paths over large graphs. In *EDBT*. 205–216.

[12] Zijin Feng, Tiantian Liu, Huan Li, Hua Lu, Lidan Shou, and Jianliang Xu. 2020. Indoor top-k keyword-aware routing query. In *ICDE*. 1213–1224.

[13] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. 2007. Adaptive fastest path computation on a road network: a traffic mining approach. In *VLDB*. 794–805.

[14] Randolph W Hall. 1986. The fastest path through a network with random time-dependent travel times. *Transp. Sci.* 20, 3 (1986), 182–188.

[15] Xing Hao, Xiaofeng Meng, and Jianliang Xu. 2008. Continuous density queries for moving objects. In *MobiDE*. 1–7.

[16] Xuegang Huang and Hua Lu. 2007. Snapshot density queries on location sensors. In *MobiDE*. 75–78.

[17] Christian S Jensen, Hua Lu, and Bin Yang. 2009. Graph model based indoor tracking. In *MDM*. 122–131.

[18] Dae-Ho Kim, Beakcheol Jang, and Jong Wook Kim. 2018. Privacy-preserving top-k route computation in indoor environments. *IEEE Access* 6 (2018), 56109–56121.

[19] Huan Li, Hua Lu, Feichao Shi, Gang Chen, Ke Chen, and Lidan Shou. 2018. TRIPS: A system for translating raw indoor positioning data into visual mobility semantics. *PVLDB* 11, 12 (2018), 1918–1921.

[20] Huan Li, Hua Lu, Lidan Shou, Gang Chen, and Ke Chen. 2018. Finding most popular indoor semantic locations using uncertain mobility data. *TKDE* 31, 11 (2018), 2108–2123.

[21] Huan Li, Hua Lu, Lidan Shou, Gang Chen, and Ke Chen. 2018. In search of indoor dense regions: An approach using indoor positioning data. *TKDE* 30, 8 (2018), 1481–1495.

[22] Yuxuan Liang, Songyu Ke, Junbo Zhang, Xiuwen Yi, and Yu Zheng. 2018. Geoman: Multi-level attention networks for geo-sensory time series prediction. In *IJCAI*. 3428–3434.

[23] Liu Liu, Sisi Zlatanova, Bofeng Li, Peter van Oosterom, Hua Liu, and Jack Barton. 2019. Indoor routing on logical network using space semantics. *ISPRS INT J GEO-INF* 8, 3 (2019), 126.

[24] Tiantian Liu, Zijin Feng, Huan Li, Hua Lu, Muhammad Aamir Cheema, Hong Cheng, and Jianliang Xu. 2020. Shortest path queries for indoor venues with temporal variations. In *ICDE*. 2014–2017.

[25] Tiantian Liu, Huan Li, Hua Lu, Muhammad Aamir Cheema, and Lidan Shou. 2021. Towards crowd-aware indoor path planning (extended version). *arXiv preprint arXiv:2104.05480* (2021).

[26] Hua Lu, Xin Cao, and Christian S Jensen. 2012. A foundation for efficient indoor distance-aware query processing. In *ICDE*. 438–449.

[27] Wenyi Luo, Peiquan Jin, and Lihua Yue. 2016. Time-constrained sequenced route query in indoor spaces. In *APWeb*. 129–140.

[28] Karl Nachtigall. 1995. Time depending shortest-path problems with applications to railway networks. *EJOR* 83, 1 (1995), 154–166.

[29] Giacomo Nannicini, Daniel Delling, Dominik Schultes, and Leo Liberti. 2012. Bidirectional A* search on time-dependent road networks. *Networks* 59, 2 (2012), 240–251.

[30] Shuo Shang, Hua Lu, Torben Bach Pedersen, and Xike Xie. 2013. Finding traffic-aware fastest paths in spatial networks. In *SSTD*. 128–145.

[31] Zhou Shao, Muhammad Aamir Cheema, and David Taniar. 2018. Trip planning queries in indoor venues. *Comput. J.* 61, 3 (2018), 409–426.

[32] Zhou Shao, Muhammad Aamir Cheema, David Taniar, and Hua Lu. 2016. VIP-tree: an effective index for indoor spatial queries. *PVLDB* 10, 4 (2016), 325–336.

[33] Cong Tang, Jingru Sun, Yichuang Sun, Mu Peng, and Nianfei Gan. 2020. A general traffic flow prediction approach based on spatial-temporal graph attention. *IEEE Access* 8 (2020), 153731–153741.

[34] Yufei Tao, George Kollios, Jeffrey Considine, Feifei Li, and Dimitris Papadias. 2004. Spatio-temporal aggregation using sketches. In *ICDE*. 214–225.

[35] Mark R Virkler and Sathish Elayadath. 1994. *Pedestrian speed-flow-density relationships*. Number HS-042 012.

[36] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient route planning on public transportation networks: A labelling approach. In *SIGMOD*. 967–982.

[37] Shunzhou Wang, Yao Lu, Tianfei Zhou, Huijun Di, Lihua Lu, and Lin Zhang. 2020. SCLNet: Spatial context learning network for congested crowd counting. *Neurocomputing* 404 (2020), 227–239.

[38] Yong Wang, Guoliang Li, and Nan Tang. 2019. Querying shortest paths on time dependent road networks. *PVLDB* 12, 11 (2019), 1249–1261.

[39] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-intact oracle for fastest path and time queries on dynamic spatial networks. In *SIGMOD*. 1841–1856.

[40] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *PVLDB* 7, 9 (2014), 721–732.

[41] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *ICDE*. 145–156.

[42] Xike Xie, Hua Lu, and Torben Bach Pedersen. 2013. Efficient distance-aware query evaluation on indoor moving objects. In *ICDE*. 434–445.

[43] Xike Xie, Hua Lu, and Torben Bach Pedersen. 2014. Distance-aware join for indoor moving objects. *TKDE* 27, 2 (2014), 428–442.

[44] Ye Yuan, Xiang Lian, Guoren Wang, Yuliang Ma, and Yishu Wang. 2019. Constrained shortest path query in a large time-dependent graph. *PVLDB* 12, 10 (2019), 1058–1070.

[45] Yan Zhou, Hong Chen, Yueying Huang, Yunxin Luo, Yeting Zhang, and Xiao Xie. 2018. An indoor route planning method with environment awareness. In *IGARSS*. 2906–2909.