2012 IEEE 28th International Conference on Data Engineering

A Foundation for Efficient Indoor Distance-Aware Query Processing

Hua Lu[1], Xin Cao[2], Christian S. Jensen[3]

[1]*Department of Computer Science, Aalborg University, Denmark*
luhua@cs.aau.dk
[2]*School of Computer Engineering, Nanyang Technological University, Singapore*
xcao1@e.ntu.edu.sg
[3]*Department of Computer Science, Aarhus University, Denmark*
csj@cs.au.dk

***Abstract*—Indoor spaces accommodate large numbers of spatial objects, e.g., points of interest (POIs), and moving populations. A variety of services, e.g., location-based services and security control, are relevant to indoor spaces. Such services can be improved substantially if they are capable of utilizing indoor distances. However, existing indoor space models do not account well for indoor distances.**

To address this shortcoming, we propose a data management infrastructure that captures indoor distance and facilitates distance-aware query processing. In particular, we propose a distance-aware indoor space model that integrates indoor distance seamlessly. To enable the use of the model as a foundation for query processing, we develop accompanying, efficient algorithms that compute indoor distances for different indoor entities like doors as well as locations. We also propose an indexing framework that accommodates indoor distances that are pre-computed using the proposed algorithms. On top of this foundation, we develop efficient algorithms for typical indoor, distance-aware queries. The results of an extensive experimental evaluation demonstrate the efficacy of the proposals.

I. Introduction

People spend large amounts of their daily lives in indoor spaces such as office buildings, shopping centers, and transportation infrastructures, e.g., airports and metro stations. A variety of technologies that use, e.g., Wi-Fi, Bluetooth, and RFID, enable positioning in indoor settings [3], [6]–[8]. A variety of location-based services (LBSs) that use such positioning are relevant to people in indoor spaces. For example, a museum service can guide visitors through an interesting yet complex exhibition. As another example, a boarding reminder service in an airport can remind air passengers, especially those far away from their gates, of their departures.

Such indoor LBSs will benefit from the availability of accurate indoor distances. Referring to the boarding reminder service, a naive approach is to broadcast reminders to all passengers that have checked in for a particular flight. However, this may well annoy those passengers already at the gate and in the process of boarding. It is attractive to target instead only passengers far from their boarding gates, and to appropriately direct them to their gates.

To achieve this, it is necessary to know the indoor distance from passengers' current positions to their corresponding boarding gates. In the museum tour guide service example, indoor distance awareness also offers tourist desirable convenience of shortest indoor walking paths.

Distances play an important role in a variety of indoor service scenarios, just as they do in outdoor settings. Shortest indoor paths are critical in emergency response, e.g., in case of a fire in an office building. Also, shortest indoor paths often imply optimal routes in computer games [1] that have complex indoor environments as (parts of) their settings.

Indoor spaces are characterized by various entities, notably doors and rooms. Such entities enable as well as constrain indoor movement, and thus render traditional space models for Euclidean spaces and spatial network spaces unsuitable for indoor environments.

An example indoor floor plan is shown in Figure 1. Each

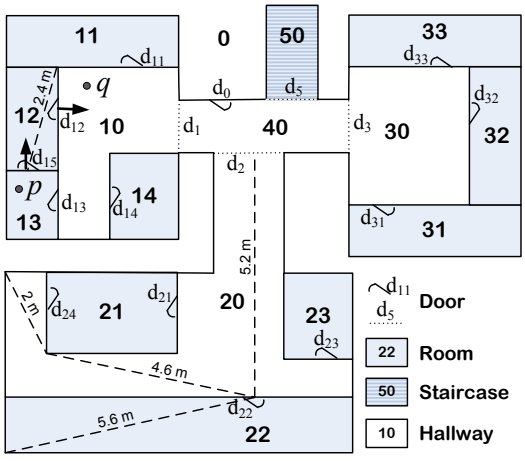


Fig. 1. Floor Plan

door is identified by a corresponding d_i, and an arrow attached to a door implies directionality of the door. For example, doors d_{12} and d_{15} are unidirectional. One can walk through door d_{15} only from room 13 to room 12.

The Euclidean distance between two indoor positions such as p and q carries little meaning because it goes through walls between rooms. Rather, one has to go through door d_{13}, or doors d_{15} and d_{12}, to reach position q from position p.

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.44

On the other hand, existing indoor space models [4], [11], [12], [14] pay little attention to indoor distances. For example, Li and Lee [11] count the number of doors along indoor routes to identify indoor nearest neighbors. This arrangement often fails to reflect the actual indoor distance one needs to walk to reach an indoor destination.

In the example shown in Figure 1, the shortest indoor path from position p to position q ($p \rightarrow d_{15} \rightarrow d_{12} \rightarrow q$) goes through doors d_{15} and d_{12}. The aforementioned model [11] takes the clearly longer path that goes through door d_{13} ($p \rightarrow d_{13} \rightarrow q$) since this path only involves one door.

To improve on existing models, we propose a versatile distance-aware data management infrastructure for indoor spaces. The infrastructure provides a graph-based indoor space model that captures indoor topology information and integrates basic indoor distances. Such a distance-aware model supports efficient computation of different types of indoor distances, e.g., door-to-door distance as well as position-to-position distance. Furthermore, the infrastructure provides an indexing framework that indexes pre-computed door-to-door distances and thus facilitates indoor distance-aware query processing.

In summary, we make the following contributions. First, we propose a distance-aware indoor space model that integrates indoor distances. Second, we develop efficient algorithms that compute indoor distances accordingly. Third, we design an indexing framework for indoor distances as well as objects with the intention to facilitate the processing of indoor distance-aware queries. Fourth, on top of the foundation thus formed, we develop efficient algorithms for different types of indoor distance-aware queries. Finally, we conduct an extensive empirical study to evaluate our proposals.

The rest of this paper is organized as follows. Section II reviews related work. Section III details our distance-aware indoor space model and the algorithms for indoor distance computation. Section IV presents the indexing framework for indoor objects. Section V discusses indoor distance-aware query processing that utilizes the indexing framework. Section VI reports the experimental findings and Section VII concludes the paper and discusses directions for future work.

II. Related Work

Symbolic indoor space models are often preferred over geometric coordinate models because the former are able to capture the semantics associated with indoor entities [4] and the movements enabled (or disabled) by such entities.

An existing 3D Geometric Network Model [10] treats the vertical and horizontal connectivity relationship among 3D spatial cells separately. A related 3D Indoor Geo-Coding technique employs the 3D *Poincaré Duality* [13] transformation to map 3D spatial cells from primal space to dual space. This technique is used for the planning of an evacuation network in a multi-level building [9]. A 3D metrical-topological model [17] describes both the shapes and connectivity of spatial cells for navigation purposes. Another 3D model [14] combines space partitions with possible events in a dual space, to enable navigation in multi-layered buildings. Such 3D models focus on topological relationships rather than quantitative indoor distances. In other words, they do not support calculations of indoor distances and distance-aware queries, which is what we consider in this paper.

A lattice-based semantic location model [11] maintains semantic relationships and distances, e.g., the nearest neighbor relationship among indoor entities. This model, however, defines the "length" of an indoor path by the number of doors it goes through, rather than the actual walking distance considered in this paper. As discussed in Section I, this model is likely to fall short in many practical scenarios.

Different ways of transforming a floor plan into a graph also exist [5], [16]. Our previous work on a graph-based model for indoor space is proposed to support efficient indoor tracking [8], but lacks support for indoor distance. The relevant previous work [18] employs the metric of minimum indoor walking distance, but it does not elaborate on how indoor distances can be incorporated into a space model.

The recent iNav proposal [20] models doors as nodes and rooms as edges in a graph but it is unable to capture door directionality. A brief discussion of this aspect is given in Section III-C2. Further, iNav does not include the arbitrary indoor position-to-position distances. The LEGO representation [19] involves only so-called connector-to-connector indoor distances, and it covers neither how to compute such distances nor how to compute more complex indoor distances.

The indoor space model we propose also differs from those ones used for path planning in robotics applications [2], [15]. Such robotics models focus on connectivity and reachability rather than on distances, and they target applications in relation to robots with very limited spatial awareness. In contrast, our model aims to support very different indoor location-based services that involve humans equipped with mobile devices like smartphones.

Last but not least, obstacles and obstructed distances are not the focus of this paper. Our model is able to handle general indoor topologies and distances. Obstructed distances form a small, local part in our model only if an indoor partition has obstacles. Reversely, it is unclear whether or how an existing approach [21] for spaces with obstacles can be used to model complex indoor topologies. Extending the existing approach to support indoor distances and distance-aware queries requires substantial efforts and does not appear to be feasible. Given a pair of source and destination positions, a complex visibility graph must be created on the fly by regarding all indoor entities (rooms, doors, walls, etc.) as obstacles before any further operations can be applied. In contrast, our model creates, for the entire indoor space, a single general graph, on top of which all indoor distances are computed efficiently.

III. Distance-Aware Modeling of Indoor Spaces

This section presents the details of the distance-aware indoor space model. Section III-A proposes mapping structures that capture indoor topology information needed for measuring indoor distances. Section III-B briefly introduces a base graph model that does not accommodate indoor distances.

Section III-C presents the indoor distance-aware model that substantially extends the base graph model. Section III-D proposes relevant indoor distance computation algorithms that make use of the proposed distance-aware model.

For simplicity and due to space limitations, we use the relatively simple floor plan in Figure 1 as a running example. Rooms, a staircase, a hallway, and doors and walls are shown in the figure. We use the term *indoor partition* to indicate a room, a hallway, or a staircase. Semantically, each indoor partition is a smallest piece of independent space that is connected to other partitions by one or more doors. For example, room 11 is a partition in Figure 1, and so is staircase 50. Table I lists the notation used throughout the paper.

Notation	Meaning
$\mathcal{S}_{partition}$	Set of indoor partitions
$\mathcal{S}_{door}$	Set of doors
v, v_i	Indoor partitions
d, d_i	Doors
p, p_i	Indoor positions
G_{accs}	Accessibility graph for an indoor space
G_{dist}	Distance-aware graph for an indoor space

TABLE I
NOTATION

A. Topology Information Mappings

A door usually connects two adjacent partitions in the sense that one can move from one partition to the other through the door. To simplify the presentation, we stipulate that each door always connects exactly two partitions[1]. Accordingly, we use the mapping $D2P$ to map a door d_k to one or two partition pairs (v_i, v_j) such that one can move from partition v_i to partition v_j through door d_k.

$$D2P : \mathcal{S}_{door} \rightarrow 2^{\mathcal{S}_{partition} \times \mathcal{S}_{partition}} \quad (1)$$

Referring to Figure 1, we have $D2P(d_{12}) = \{(v_{12}, v_{10})\}$ for door d_{12}. For door d_{15}, we have $D2P(d_{15}) = \{(v_{13}, v_{12})\}$. In addition, we have $D2P(d_{21}) = \{(v_{20}, v_{21}), (v_{21}, v_{20})\}$ for door d_{21}. Thus, $D2P(d_i)$ captures the directionality of a given door d_i. Specifically, door d_i is unidirectional if $|D2P(d_i)| = 1$ and bidirectional if $|D2P(d_i)| = 2$. In our example, door d_{21} is bidirectional; doors d_{12} and d_{15} are unidirectional.

Given a door d_k, from $\pi_2(D2P(d_k))$[2], we are able to know the partition(s) one can enter through d_k. We call each such partition an *enterable partition* of d_k. For convenience, we use the following derived mapping to denote $\pi_2(D2P(d_k))$.

$$D2P_{\sqsupset} : \mathcal{S}_{door} \rightarrow 2^{\mathcal{S}_{partition}} \quad (2)$$

Similarly, from $\pi_1(D2P(d_k))$ we are able to know the partition(s) one can leave through d_k. We call each such partition a *leaveable partition* of d_k. Also for convenience, we use the following derived mapping to denote $\pi_1(D2P(d_k))$.

$$D2P_{\sqsubset} : \mathcal{S}_{door} \rightarrow 2^{\mathcal{S}_{partition}} \quad (3)$$

[1]We regard all of outdoor space as a special partition. If a door connects more than two partitions, we can convert it to multiple doors, each connecting two partitions.

[2]Here, $\pi_2(\{(v_1^1, v_1^2), \ldots, (v_n^1, v_n^2)\}) = \{v_1^2, \ldots, v_n^2\}$, and π_1 is defined correspondingly.

For door d_{12} in Figure 1, we have $D2P_{\sqsupset}(d_{12}) = \{v_{10}\}$ and $D2P_{\sqsubset}(d_{12}) = \{v_{12}\}$; for door d_{15}, we have $D2P_{\sqsupset}(d_{15}) = \{v_{12}\}$ and $D2P_{\sqsubset}(d_{15}) = \{v_{13}\}$; and for door d_{15}, we have $D2P_{\sqsupset}(d_{21}) = D2P_{\sqsubset}(d_{21}) = \{v_{20}, v_{21}\}$.

Two derived mappings can be reversed to capture relevant topology information in the opposite direction. In particular, the mapping $P2D_{\sqsupset}$ maps a partition v to all the doors through which one can enter v; similarly, the mapping $P2D_{\sqsubset}$ maps a partition v to all the doors through which one can leave v.

$$P2D_{\sqsupset} : \mathcal{S}_{partition} \rightarrow 2^{\mathcal{S}_{door}} \quad (4)$$

$$P2D_{\sqsubset} : \mathcal{S}_{partition} \rightarrow 2^{\mathcal{S}_{door}} \quad (5)$$

Given a partition v, we call each door in $P2D_{\sqsupset}(v)$ an *enterable door* of v. Similarly, a door in $P2D_{\sqsubset}(v)$ is a *leaveable door* of partition v. Refer again to Figure 1. For partition v_{10}, we have $P2D_{\sqsupset}(v_{10}) = \{d_1, d_{11}, d_{12}, d_{13}, d_{14}\}$ and $P2D_{\sqsubset}(v_{10}) = \{d_1, d_{11}, d_{13}, d_{14}\}$. For partition v_{12}, we have $P2D_{\sqsupset}(v_{12}) = \{d_{15}\}$ and $P2D_{\sqsubset}(v_{12}) = \{d_{12}\}$. For partition v_{13}, we have $P2D_{\sqsupset}(v_{13}) = \{d_{13}\}$ and $P2D_{\sqsubset}(v_{13}) = \{d_{13}, d_{15}\}$. Finally, for partition v_{21}, we have $P2D_{\sqsupset}(v_{21}) = P2D_{\sqsubset}(v_{21}) = \{d_{21}, d_{24}\}$.

When there is no need to differentiate the directionality, we use $P2D(v_j)$ to denote $P2D_{\sqsupset}(v_j) \cup P2D_{\sqsubset}(v_j)$. For each door $d_i \in P2D(v_j)$, we say d_i *touches* partition v_j. It can be observed that only the mapping $D2P$ is fundamental, in that all other mappings can be derived from it. We utilize the derived mappings for ease of presentation.

B. Accessibility Base Graph

By capturing the essential connectivity and accessibility, the base graph describes the topology of a floor plan of a possibly complex indoor space. A base graph is constructed from a floor plan based on the *Poincaré Duality* [13]. Each partition in the floor plan, such as a single room, a staircase, or a hallway, is represented as a vertex in the base graph. In addition, the exterior of the indoor space is represented by a singe vertex.

Edges are used for capturing the relationships between vertices. An edge can capture the *connectivity* between two partitions. Each connection in the floor plan, such as a door, a hatch, or an escape window, is then represented by an edge. For example, if two rooms are connected by a door, the two corresponding vertices are connected by an edge. Since several doors may connect two rooms, several edges between the same pair of vertices must be accommodated.

Sometimes, a door may permit movement in only one direction. For example, a security check point in an airport allows only one-way movement. This also applies at entrances and exits in underground stations. In Figure 1, door d_{12} only permits entry into the hallway from room 12, and door d_{15} only permits the movement from room 13 to room 12 but not in the opposite direction. Accordingly, the *accessibility graph*, a labeled, directed graph, is constructed to represent the movement permitted by doors or connections. The accessibility graph G_{accs} is given by the triple (V, E_a, L), where:

1) $V = \mathcal{S}_{partition}$ is the set of the vertices.

2) $E_a = \{(v_i, v_j, d_k) \mid (v_i, v_j) \in D2P(d_k)\}$ is the set of labeled, directed edges.
3) $L = \mathcal{S}_{door}$ is the set of edge labels.

The accessibility graph for our example is shown in Figure 2. The direction of an edge indicates the movement direction permitted by the corresponding connection.

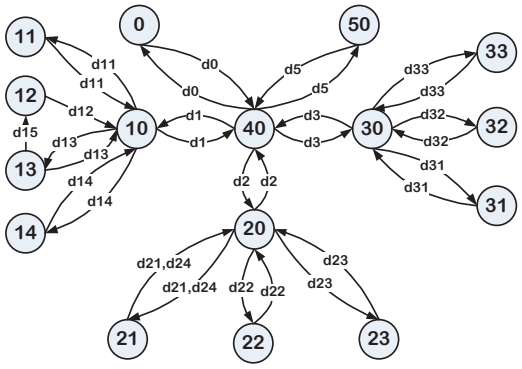


Fig. 2. Accessibility Base Graph

C. Distance-Aware Model

The accessibility graph does not capture indoor distance information. It is also noteworthy that we cannot plug indoor distances into the base graph by simply adding costs to edges. Edges represent doors that in themselves do not define any indoor distances. In Figure 2, the edge between partitions 12 and 10 represents door d_{12}. However, no distance information is associated with door d_{12} (and the corresponding edge). This argument also applies to the case that involves the outdoor space, e.g., partition 0 in Figure 2.

This important particularity calls for novel model design for indoor distances. Our objective is to integrate indoor distances into the base graph such that distance-aware queries can be supported efficiently with the same structure. Next, we propose an extended graph model with specific constructs that incorporate relevant kinds of indoor distances.

1) Extended Graph Model: Our distance-aware model integrates indoor distances into the accessibility graph in a seamless way such that both topology and distance information are captured in the same structure. We are concerned with the minimum indoor walking distance [18], the shortest distance one has to move in an indoor space to reach a destination, e.g., to reach the nearest heart starter in an office building.

The distance-aware model is a graph structure G_{dist} that is given by the 5-tuple $(V, E_a, L, f_{dv}, f_{d2d})$, where:

1) $V = \mathcal{S}_{partition}$ is the set of vertices.
2) $E_a = G_{accs}.E_a$ is the set of directed edges, where G_{accs} is the accessibility graph as defined in Section III-B.
3) $L = \mathcal{S}_{door}$ is the set of edge labels.
4) $f_{dv} : \mathcal{S}_{door} \times V \rightarrow \mathcal{R} \cup \{\infty\}$ maps an edge to a distance value. Specifically, given a door d_i and a partition v_j,
$$f_{dv}(d_i, v_j) = \begin{cases} max_{p \in v_j} \|d_i, p\|, & \text{if } v_j \in D2P_{\sqsupset}(d_i); \\ \infty, & \text{otherwise.} \end{cases}$$
In other words, if v_j is an enterable partition of door d_i , $f_{dv}(d_i, v_j)$ returns the longest distance one can reach within v_j from d_i; otherwise, $f_{dv}(d_i, v_j)$ returns ∞.
5) $f_{d2d} : V \times \mathcal{S}_{door} \times \mathcal{S}_{door} \rightarrow \mathcal{R} \cup \{\infty\}$ maps a 3-tuple to a distance value. Given a partition v_k and two different doors d_i and d_j,
$$f_{d2d}(v_k, d_i, d_j) = \begin{cases} \|d_i, d_j\|_{v_k}, & \text{if } d_i \in P2D_{\sqsupset}(v_k) \text{ and } d_j \in P2D_{\sqsubset}(v_k); \\ 0, & \text{if } d_i = d_j \text{ and } d_i, d_j \in P2D(v_k); \\ \infty, & \text{otherwise.} \end{cases}$$
In other words, if doors d_i and d_j both touch partition v_k, $f_{d2d}(v_k, d_i, d_j)$ returns the distance between d_i and d_j within v_k, which is termed $\|d_i, d_j\|_{v_k}$. If $d_i = d_j$ and both are in $P2D(v_k)$, we stipulate $f_{d2d}(v_k, d_i, d_j) = 0$. Otherwise, $f_{d2d}(v_k, d_i, d_j)$ returns ∞ to indicate that one cannot go from d_i to d_j (or the other way around) by only moving within v_k.

Note that distance $\|d_i, d_j\|_{v_k}$ above is not necessarily a Euclidean distance because there may be entities in the line of sight between two doors. This is exemplified by the distance between doors d_{22} and d_{24} in Figure 1. Likewise, exhibition stands in a large museum may be arranged in such a way that they represent obstacles that block straight Euclidean movements. The model proposed in this paper is able to accommodate possible obstacles and to support obstructed distances in indoor spaces. Nevertheless, obstructed distances are not the focus of the paper as they are relevant only if an indoor partition has obstacles. Computation of such obstructed, local distances is covered elsewhere [21].

We refer to the running example in Figure 1 to explain the proposed model. Door d_{22} has two edges in G_{dist}: the one from partition v_{20} to partition v_{22}, and the one from v_{22} to v_{20}. The longest distance one can go within v_{22} from door d_{22} is $f_{dv}(d_{22}, v_{22}) = 5.6$ meters, and the similar distance within v_{20} is $f_{dv}(d_{22}, v_{20}) = 6.6$ meters[3]. For partition v_{20} and its doors d_2 and d_{22}, we have $f_{d2d}(v_{20}, d_2, d_{22}) = 5.2$ meters, the shortest distance between d_2 and d_{22} within v_{20}. For partition v_{12} and its two doors d_{12} and d_{15}, we have $f_{d2d}(v_{12}, d_{12}, d_{15}) = \infty$ because one cannot go from d_{12} to d_{15} within v_{12}. Accordingly, we have $f_{d2d}(v_{12}, d_{15}, d_{12}) = 1.6$ meters, the Euclidean distance between d_{12} and d_{15}.

2) Discussion: Consistent with the accessibility base graph, the distance-aware model represents indoor partitions as graph vertices and doors as edges. This design decision is justified by the following reasons.

Doors can be unidirectional, i.e., only permitting movement in one direction. For example, this occurs in settings where security is a concern, such as in airports. When representing doors by directed graph edges, this directionality is captured naturally. In contrast, it is not obvious how to capture directionality if doors are modeled as vertices.

Doors can also be open only at certain times. This occurs in a variety of real-world settings, e.g., in office buildings, shopping centers, and airports. Modeling doors as edges enables

[3]We use the midpoints of doors when measuring door-related distances.

easy extension of our proposal to accommodate such temporal variations, simply by adding temporal information to edges. In contrast, if modeling doors as vertices, the closing of a door invalidates corresponding vertices, which may invalidate additional edges; and the opening of a door also necessitates similar, complicated operations on the graph.

D. Indoor Distance Computation

In this section, we present how different kinds of indoor distances can be computed efficiently using the distance-aware model. Given a start and an end, we are interested in the minimum indoor walking distance between them. We give algorithms for different start and end types: a door or a position in the indoor space of interest.

1) Door-to-Door Minimum Walking Distance: Given a source door d_s and a destination door d_t, Algorithm d2dDistance(d_s, d_t) finds the minimum walking distance from d_s to d_t by searching the distance-aware graph G_{dist}. The pseudo code is shown in Algorithm 1. The idea is to expand the search from d_s in a similar way as does Dijkstra's algorithm, keeping all unvisited doors in a priority queue.

Two arrays, namely $dist[.]$ and $prev[.]$, are used to store the indoor path information during the execution of the algorithm. Specifically, $dist[d_j]$ stores the current shortest path distance from source d_s to a door d_j; $prev[d_j]$ stores the corresponding previous partition and door pair (v, d_i) through which the algorithm visits the current door d_j.

Algorithm 1 d2dDistance(Source door d_s, destination door d_t)

```
1: initialize a min-heap H
2: for each door d_i ∈ S_door do
3:     if d_i ≠ d_s then
4:         dist[d_i] ← ∞
5:     else
6:         dist[d_i] ← 0
7:     enheap(H, ⟨d_i, dist[d_i]⟩)
8:     prev[d_i] ← null
9: while H is not empty do
10:    ⟨d_i, dist[d_i]⟩ ← deheap(H)
11:    if d_i = d_t then
12:        return dist[d_i]
13:    mark door d_i as visited
14:    parts ← D2P_□(d_i)
15:    for each partition v ∈ parts do
16:        for each unvisited door d_j ∈ P2D_□(v) do
17:            if dist[d_i] + G_dist.f_d2d(v, d_i, d_j) < dist[d_j] then
18:                dist[d_j] ← dist[d_i] + G_dist.f_d2d(v, d_i, d_j)
19:                replace d_j's element in H by ⟨d_j, dist[d_j]⟩
20:                prev[d_j] ← (v, d_i)
```

Two characteristics distinguish algorithm d2dDistance from Dijkstra's algorithm. First, algorithm d2dDistance works on complex structures encompassing graph elements and additional constructs that are necessary for computing indoor distances. In contrast, Dijkstra's algorithm assumes that graph edges are directly associated with distances (or costs). Second, algorithm d2dDistance visits each graph edge (corresponding to a door) at most once by keeping them in the priority queue. Whereas Dijkstra's algorithm visits graph vertices which in our case correspond to indoor partitions.

Array $prev[.]$ can be omitted if we are only interested in the shortest distance and do not care about the concrete shortest path. In other words, array $prev[.]$ can be used to reconstruct the concrete shortest path, in terms of indoor partitions and doors, from source door d_s to destination door d_t.

2) Position-to-Position Minimum Walking Distance: Given an indoor position p, function getHostPartition(p) returns the partition that contains p. This function can be implemented as a point query using a spatial access method (e.g., an R-tree) that indexes all partitions in the indoor space of interest.

We need a function $dist_V$ that captures, for an indoor position p and a door d, the distance(s) that involves only one pertinent partition v, the one that contains p and is touched by d. In other words, such a distance is obtained within one single partition v only. This is formalized as follows:

$$dist_V : P \times \mathcal{S}_{door} \rightarrow \mathcal{R} \cup \{\infty\} \quad (6)$$

Here, P denotes the set of all positions in the indoor space of interest. Given a position p and a door d that touches the partition obtained in getHostPartition(p), $dist_V(p, d)$ returns the shortest intra-partition distance between p and d. This is the minimum distance one must walk to get from position p to door d without leaving p's host partition. For the sake of simplicity, we stipulate that $dist_V(p, d)$ returns ∞ if d does not touch p's host partition.

Now we are ready to give the algorithms that compute the minimum walking distance from a position p_s to another position p_t. The basic algorithm (shown in Algorithm 2) first locates the source partition v_s that hosts p_s and the destination partition v_t that hosts p_t. The set of doors that allow one to leave v_s is obtained in set $P2D_{\sqsubset}(v_s)$; the set of doors that allow one to enter v_t is obtained in set $P2D_{\sqsupset}(v_t)$.

The minimum door-to-door distance from each door d_s in $P2D_{\sqsubset}(v_s)$ to each door d_t in $P2D_{\sqsupset}(v_t)$ is computed, and the intra-partition distances $dist_V(p_s, d_s)$ and $dist_V(p_t, d_t)$ are added to that distance to get one possible position-to-position distance from d_s to d_t. All such possible position-to-position distances are computed in the same way, and the minimum one is returned as the result.

Algorithm 2 pt2ptDistance(Source indoor position p_s, destination indoor position p_t)

```
1: v_s ← getHostPartition(p_s)
2: v_t ← getHostPartition(p_t)
3: dist ← ∞
4: for each door d_s ∈ P2D_□(v_s) do
5:     dist_1 ← dist_V(p_s, d_s)
6:     for each door d_t ∈ P2D_⊐(v_t) do
7:         dist_2 ← dist_V(p_t, d_t)
8:         if dist > dist_1 + d2dDistance(d_s, d_t) + dist_2 then
9:             dist ← dist_1 + d2dDistance(d_s, d_t) + dist_2
10: return dist
```

Algorithm 2 calls d2dDistance(d_s, d_t) blindly for each pair of source and destination doors. The algorithm can be refined

to avoid this. Specifically, common portions of doors, partitions, and paths may exist when all possible door-to-door distances are computed for different d_s's and d_t's, and it is possible to make use of such overlapping to reduce the computational cost in position-to-position distance computation.

A refined version is shown in Algorithm 3. Variable $doors_s$ is initialized to contain all leaving doors (line 3) of the source partition v_s. Variable $doors_t$ is initialized to contain all entering doors (line 4) of the destination partition v_t. A door d_s in $doors_s$ is excluded if it leads to a non-destination partition that has d_s as its sole leaving door (lines 5–8). In such a case, one cannot reach the destination by entering d_s's enterable partition. The remainder of the algorithm uses computations in previous iterations for source-destination pairs (lines 9–37).

Algorithm 3 pt2ptDistance2(Source indoor position p_s, destination indoor position p_t)

```
1: v_s ← getHostPartition(p_s)
2: v_t ← getHostPartition(p_t)
3: doors_s ← P2D_□(v_s)
4: doors_t ← P2D_⊐(v_t)
5: for each door d_s ∈ doors_s do
6:     np ← the partition in D2P_□(d_s) \ {v_s}
7:     if P2D_□(np) = {d_s} and np ≠ v_t then
8:         remove d_s from doors_s
9: dist_m ← ∞
10: for each door d_s ∈ doors_s do
11:     doors ← ∅
12:     for each door d_t ∈ doors_t do
13:         if dist_V(p_s, d_s) + dist_V(p_t, d_t) < dist_m then
14:             add d_t to doors
15:     initialize a min-heap H
16:     for each door d_i ∈ S_door do
17:         if d_i ≠ d_s then
18:             dist[d_i] ← ∞
19:         else
20:             dist[d_i] ← 0
21:         enheap(H, ⟨d_i, dist[d_i]⟩)
22:     while H is not empty do
23:         ⟨d_i, dist[d_i]⟩ ← deheap(H)
24:         if d_i ∈ doors then
25:             doors ← doors \ {d_i}
26:             if dist_m > dist_V(p_s, d_s) + dist[d_i] + dist_V(p_t, d_i)
                then
27:                 dist_m ← dist_V(p_s, d_s) + dist[d_i] + dist_V(p_t, d_i)
28:             if doors = ∅ then
29:                 break
30:         mark door d_i as visited
31:         parts ← D2P_□(d_i)
32:         for each partition v ∈ parts do
33:             for each unvisited door d_j ∈ P2D(v) do
34:                 if d_j ∈ P2D_□(v) then
35:                     if dist[d_i] + G_dist.f_d2d(v, d_i, d_j) < dist[d_j] then
36:                         dist[d_j] ← dist[d_i] + G_dist.f_d2d(v, d_i, d_j)
37: return dist_m
```

For each source door d_s, all destination doors to be considered are used to initialize a set $doors$, excluding any destination door d_t that is too far away compared to the current shortest indoor distance $dist_m$ (lines 11–14). After that, an expansion following the spirit of Dijkstra's algorithm is carried out for d_s and all doors in set $doors$. When expanding the search to further away doors, the algorithm only visits those doors that allow objects to move out (lines 34). Also, the algorithm updates the shortest distance when a shorter path is found (lines 24–27). The current expansion for a door d_s terminates when set $doors$ becomes empty (lines 25, 28–29).

Unlike Algorithm 1, Algorithm 3 does not use array $prev[.]$ to store the concrete indoor shortest path information. Such an array can be employed to further optimize the computation of position-to-position minimum indoor walking distance. Specifically, the stored shortest indoor paths and distances that have been computed in previous iterations can be used to further eliminate unnecessary distance computations. The corresponding further optimized version is shown in Algorithm 4.

It initializes in the same way as does Algorithm 3, except that a two-dimensional array $dists[d_i][d_j]$ is employed to store the currently shortest indoor distance from source door d_i to destination door d_j (lines 9–10). The algorithm uses array $dists[d_i][d_j]$ in two ways. When a destination door d_i is popped from priority queue H and processed (lines 26–30), its previous door d_j on the shortest path is obtained (line 31). If this door d_j is a source door and its identifier is larger than that of the current source door d_s (line 33),[4] the shortest indoor distance from source door d_j to destination door d_i, stored in $dists[d_j][d_i]$, is exactly the difference between $dist[d_i] - dist[d_j]$. This, together with the check in line 15 to see whether $dists[d_j][d_i]$ is infinity, saves computations that are otherwise invoked for d_j in a future for-loop iteration starting in line 12. Accordingly, the shortest indoor distance from the source position to the destination position is updated if necessary (lines 35–36), by using the distance from door d_j to door d_i. This backward optimization is continued until the current source door d_s is reached (line 32).

A similar optimization also applies to the forward direction in the algorithm (lines 40–45). In this case, d_i, popped from priority queue H, is a source door and its identifier is less than that of the current source door d_s (line 40). This implies that d_i has been processed before by the for-loop starting in line 12. Therefore, the shortest path distances from d_i to all destination doors in set $doors$ is available in array $dists[.][.]$. As a result, we do not continue the for-loop to compute the shortest path distance from d_s to each destination door d_j in set $doors$. Instead, we make use of the known shortest distance from d_i to each d_j (line 41–42), since the iterations so far indicate that the shortest path from d_s to d_j must go through d_i. Accordingly, the shortest indoor distance from source position to destination position is updated if necessary (lines 43–44), by using the distance from door d_s to door d_j.

In summary, we have developed three means of computing position-to-position distances in indoor spaces. Algorithm 2 iteratively makes blind use of door-to-door distances (Algorithm 1) to obtain the minimum indoor position-to-position distance. In contrast, Algorithms 3 and 4 utilize specialized optimizations and reuse the door-to-door distances that have

[4]We assume that all doors in set $doors_s$ are accessed in ascending order of their identifiers in the for-loop in line 10. Therefore, it is implied that d_j has not been processed by the for-loop as a source door.

Algorithm 4 pt2ptDistance3(Source indoor position p_s, destination indoor position p_t)

```
1: v_s ← getHostPartition(p_s)
2: v_t ← getHostPartition(p_t)
3: doors_s ← P2D_□(v_s)
4: doors_t ← P2D_⊐(v_t)
5: for each door d_s ∈ doors_s do
6:    np ← the partition in D2P_□(d_s) \ {v_s}
7:    if P2D_□(np) = {d_s} and np ≠ v_t then
8:       remove d_s from doors_s
9:    for each door d_t ∈ doors_t do
10:      dists[d_s][d_t] ← ∞
11: dist_m ← ∞
12: for each door d_s ∈ doors_s do
13:    doors ← ∅
14:    for each door d_t ∈ doors_t do
15:       if dists[d_s][d_t] = ∞ and dist_V(p_s, d_s) + dist_V(p_t, d_t) <
          dist_m then
16:          add d_t to doors
17:    initialize a min-heap H
18:    for each door d_i ∈ Σ_door do
19:       if d_i ≠ d_s then
20:          dist[d_i] ← ∞
21:       else
22:          dist[d_i] ← 0
23:       enheap(H, ⟨d_i, dist[d_i]⟩)
24:       prev[d_i] ← null
25:    while H is not empty do
26:       ⟨d_i, dist[d_i]⟩ ← deheap(H)
27:       if d_i ∈ doors then
28:          doors ← doors \ {d_i}
29:          if dist_m > dist_V(p_s, d_s) + dist[d_i] + dist_V(p_t, d_i)
             then
30:             dist_m ← dist_V(p_s, d_s) + dist[d_i] + dist_V(p_t, d_i)
31:          (v, d_j) ← prev[d_i]
32:          while d_j ≠ d_s do
33:             if d_j ∈ doors_s and d_j > d_s then
34:                dists[d_j][d_i] ← dist[d_i] − dist[d_j]
35:                if dist_m > dist_V(p_s, d_j) + dists[d_j][d_i] +
                   dist_V(p_t, d_i) then
36:                   dist_m ← dist_V(p_s, d_j) + dists[d_j][d_i] +
                      dist_V(p_t, d_i)
37:             (v, d_j) ← prev[d_j]
38:          if doors = ∅ then
39:             break
40:       else if d_i ∈ doors_s and d_i < d_s then
41:          for each door d_j ∈ doors do
42:             dists[d_s][d_j] ← dist[d_i] + dists[d_i][d_j]
43:             if dist_m > dist_V(p_s, d_s) + dists[d_s][d_j] +
                dist_V(p_t, d_j) then
44:                dist_m ← dist_V(p_s, d_s) + dists[d_s][d_j] +
                   dist_V(p_t, d_j)
45:          break
46:       mark door d_i as visited
47:       parts ← D2P_□(d_i)
48:       for each partition v ∈ parts do
49:          for each unvisited door d_j ∈ P2D(v) do
50:             if d_j ∈ P2D_□(v) then
51:                if dist[d_i] + G_dist.f_d2d(v, d_i, d_j) < dist[d_j] then
52:                   dist[d_j] ← dist[d_i] + G_dist.f_d2d(v, d_i, d_j)
53:                   prev[d_j] ← (v, d_i)
54: return dist_m
```

been computed so far. We experimentally compare all three versions in Section VI-A.

IV. Indoor Distance-Aware Indexes

We propose an indexing framework that is aware of the notion of indoor distance presented in Section III. In Section IV-A, we present the base indexing structure that is derived from the indoor topology and indoor distances. In Section IV-B, we show how to index indoor objects on top of the base indexing structure.

A. Base Indexing Structure

To support distance based queries, we need to access indoor distances, especially door-to-door distances. Therefore, we store all door-to-door distances, computed as defined in Section III-D1, in a **Door-to-Door Distance Matrix**, denoted as M_{d2d}. It is an N-by-N matrix, where $N = |\mathcal{S}_{doors}|$ is the total number of doors. Let d_i and d_j be two door identifiers. Without loss of generality, we suppose $1 \leq d_i < d_j \leq |\mathcal{S}_{doors}|$. Regarding M_{d2d}, we have the following: (1) $M_{d2d}[d_i, d_i] = 0$; (2) $M_{d2d}[d_i, d_j] = \text{d2dDistance}(d_i, d_j)$; and (3) $M_{d2d}[d_i, d_j]$ may differ from $M_{d2d}[d_j, d_i]$.

The top-left part of Figure 1 has six doors, namely d_1, d_{11}, d_{12}, d_{13}, d_{14}, and d_{15}. The corresponding door-to-door distance matrix is shown in Figure 3. It is clear that this matrix is not symmetric, e.g., $M_{d2d}[d_{11}, d_{15}] \neq M_{d2d}[d_{15}, d_{11}]$. This is due to the existence of directed doors. A door-to-door shortest path from door d_i to d_j via a directional door d_k is not identical to the shortest path in the other direction, i.e., from d_j to d_i.

$$\begin{pmatrix} & d_1 & d_{11} & d_{12} & d_{13} & d_{14} & d_{15} \\ d_1 & 0 & 1.7 & 2.7 & 3.2 & 2.6 & 4.3 \\ d_{11} & 1.7 & 0 & 1.9 & 3.4 & 3 & 4.4 \\ d_{12} & 2.7 & 1.9 & 0 & 2 & 2.2 & 3 \\ d_{13} & 3.2 & 3.4 & 2 & 0 & 1.2 & 1 \\ d_{14} & 2.6 & 3 & 2.2 & 1.2 & 0 & 2.2 \\ d_{15} & 3.2 & 3.4 & 1.5 & 3.5 & 3.7 & 0 \end{pmatrix}$$

Fig. 3. Distance Matrix

We also use $M_{d2d}[d_i, *]$ to denote all the distances from door d_i to all (other) doors, which can be regarded as an array of distance values. To further facilitate the processing of distance based queries, we create a sequential index for every distance array $M_{d2d}[d_i, *]$ such that distance retrieval for door d_i can be conducted in a sorted fashion. Specifically, we create for the distance matrix M_{d2d} an N-by-N **Distance Index Matrix**, denoted as M_{idx}, with the following property: Given a door identifier d_i, and two integers $1 \leq j < k \leq |\mathcal{S}_{door}|$, $M_{d2d}[d_i, M_{idx}[d_i, j]] \leq M_{d2d}[d_i, M_{idx}[d_i, k]]$.

Referring again to the top-left part of Figure 1, the distance index matrix corresponding to the door-to-door distance matrix is shown in Figure 4. In this particular example, the first row indicates that the following distance ordering holds: $\text{d2dDistance}(d_1, d_1) \leq \text{d2dDistance}(d_1, d_{11}) \leq \text{d2dDistance}(d_1, d_{14}) \leq \text{d2dDistance}(d_1, d_{12}) \leq \text{d2dDistance}(d_1, d_{13}) \leq \text{d2dDistance}(d_1, d_{15})$.

$$\begin{pmatrix} & 1 & 2 & 3 & 4 & 5 & 6 \\ d_1 & d_1 & d_{11} & d_{14} & d_{12} & d_{13} & d_{15} \\ d_{11} & d_{11} & d_1 & d_{12} & d_{14} & d_{13} & d_{15} \\ d_{12} & d_{12} & d_{11} & d_{13} & d_{14} & d_1 & d_{15} \\ d_{13} & d_{13} & d_{15} & d_{14} & d_{12} & d_1 & d_{11} \\ d_{14} & d_{14} & d_{13} & d_{12} & d_{15} & d_1 & d_{11} \\ d_{15} & d_{15} & d_{12} & d_1 & d_{11} & d_{13} & d_{14} \end{pmatrix}$$

Fig. 4. Distance Index Matrix

In the design of the matrixes M_{d2d} and M_{idx} we implicitly regard all doors as being topologically equal. This was done to achieve a general design. It is possible that a particular door or staircase is topologically more important than others. In such cases, it is of interest to build such knowledge into our proposal so that this can be exploited in indoor distance computation and even query processing. Nevertheless, identifying the different degrees of topological significance of doors and staircases requires extra effort and domain knowledge. In this paper we aim for a general and self-contained design and leave topological significance for future research.

B. Indexing Indoor Objects

Motivated by the fact that any indoor object must be located in some partition, we store objects within the same partition together in an object bucket or several chained buckets. Since each partition is touched by one or more doors through which objects can move in or out, we create pointers from doors to their connected partitions and the associated bucket(s).

We use a linear table **Door-to-Partition Table** (DPT) to store the relationship between doors and partition object buckets. Each record in DPT is a 5-tuple $(d_i, vPtr_1, dist_1, vPtr_2, dist_2)$, where d_i is a door identifier. The remaining fields are relevant to the partitions that d_i touches.

If $D2P(d_i) = \{(v_j, v_k)\}$, which means that door d_i is directional from partition v_j to v_k, then field $vPtr_1$ is a null pointer, $vPtr_2$ is a pointer that points to the object bucket of partition v_k, $dist_1$ is ∞, and $dist_2$ is $G_{dist}.f_{dv}(d_i, v_k)$.

Otherwise, $D2P(d_i) = \{(v_j, v_k), (v_k, v_j)\}$, meaning that door d_i is bidirectional between partition v_j to v_k. Without loss of generality, we assume that two partition identifiers satisfy $v_j < v_k$. Then field $vPtr_1$ is a pointer that points to the object bucket of partition v_j, $vPtr_2$ is a pointer that points to the object bucket of partition v_k, $dist_1$ is $G_{dist}.f_{dv}(d_i, v_j)$, and $dist_2$ is $G_{dist}.f_{dv}(d_i, v_k)$.

Referring to Figure 1 again. Remember that we have $D2P(d_{15}) = \{(v_{13}, v_{12})\}$. As a result, door d_{15}'s corresponding DPT entry is $(d_{15}, \text{null}, \infty, vPtr_2, 2.4)$, where $vPtr_2$ points to partition v_{13}'s object bucket.

As a door touches at most two partitions, there is only one record for each door, and the field of door identifier d_i serves as the primary key of the DPT. To facilitate record retrieval based on door identifiers, we sort the whole DPT on the d_i field. Given a door identifier d_i, we use DPT$[d_i]$ to denote the corresponding record.

V. Processing Indoor Distance-Aware Queries

We proceed to propose algorithms for indoor distance-aware queries. In Section V-A, we describe how distance-aware queries over indoor spatial objects can be processed by exploiting the indexing structures proposed in Section IV. In Section V-B, we describe how to organize spatial objects within an indoor partition in a more fine-grained manner within our proposed indexing framework.

A. Queries

We consider range and nearest neighbor queries over indoor objects. We process each query type by making use of the indexing structures described in Section IV.

1) Range Query: Given an indoor position q and a distance range r, a range query $Q_r(q, r)$ returns those indoor objects that are within distance r of q. By making use of the indexing structures, a range query $Q_r(q, r)$ is processed according to the pseudo code shown in Algorithm 5.

Algorithm 5 range(Position q, distance r)

```
1: v ← getHostPartition(q)
2: R ← rangeSearch(v's bucket, p, r)
3: for each door d_i ∈ P2D_⊏(v) do
4:     r_1 ← r − dist_V(q, d_i)
5:     for j from 1 to |S_door| do
6:         d_j ← M_idx[d_i, j]
7:         if M_d2d[d_i, d_j] > r_1 then
8:             break
9:         else
10:            r_2 ← r_1 − M_d2d[d_i, d_j]
11:            if DPT[d_j].vPtr_1 ≠ null then
12:                if DPT[d_j].dist_1 ≤ r_2 then
13:                    add objects in DPT[d_j].vPtr_1's bucket to R
14:                else
15:                    R ← R ∪ rangeSearch(DPT[d_j].vPtr_1, d_j, r_2)
16:            if DPT[d_j].vPtr_2 ≠ null then
17:                if DPT[d_j].dist_2 ≤ r_2 then
18:                    add objects in DPT[d_j].vPtr_2's bucket to R
19:                else
20:                    R ← R ∪ rangeSearch(DPT[d_j].vPtr_2, d_j, r_2)
21: return R
```

The algorithm first gets the query position q's host partition v (line 1), then searches for possibly qualifying objects within v by utilizing the grid index built on v (line 2). Here, procedure rangeSearch(B_i, q, r) searches an object bucket B_i for all objects within distance r of q. Specifically, bucket B_i corresponds to an indoor partition v_i and contains all the objects currently in v_i. In contrast, q is either a position inside v_i or a door of v_i. More details regarding intra-partition object organization and search are to be addressed in Section V-B.

Subsequently, for each door d_i through which one can leave v, a linear search is done among its distances to all other doors that are stored in $M_{d2d}[d_i, *]$ (lines 3–20). By exploiting the index in $M_{idx}[d_i, *]$, the search is conducted in non-descending order of $M_{d2d}[d_i, d_j]$, where d_j is a door covered by the query distance r from query position q (lines 4–8).

For every such door d_j, the algorithm processes its connected partition(s) through DPT. If a partition is entirely

within the query range, i.e., DPT$[d_j].dist_x + dist_V(q, d_i) + M_{d2d}[d_i, d_j] \leq r$ ($x \in \{1, 2\}$), all objects in the corresponding bucket are added to the result (lines 12–13 and 17–18). Otherwise, a range query is done against the partition with d_j as the query position and $r - dist_V(q, d_i) - M_{d2d}[d_i, d_j]$ as the range distance (lines 15 and 20). The above process is repeated for every door d_j until it is found that a new door d_j is too far away from q, i.e., $M_{d2d}[d_i, d_j] + dist_V(q, d_i) > r$.

Note that DPT$[d_j].vPtr_1$ and DPT$[d_j].vPtr_2$ can be the partitions where q is located, but we still need to do the range search again. This is because obstacles can result in the intra-partition distance between two points not being the shortest. We refer to the example shown in Figure 5. The figure contains four obstacles $O1$ to $O4$ in rooms 1 and 2. For two points in room 2, p near d_6 and q near d_9, intra-room path $p \rightarrow C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow q$ is not the shortest indoor distance path from p to q. Instead, path $p \rightarrow d_7 \rightarrow d_8 \rightarrow q$ is the shortest although it starts from room 2, enters room 1, and finally returns to room 2. This remark also holds for the other query type.

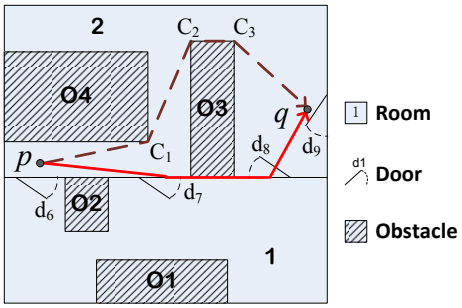


Fig. 5. Indoor Obstacles

2) Nearest Neighbor Query: Given an indoor position q, a nearest neighbor (NN) query $Q_{nn}(q)$ returns the indoor object whose distance from q is the smallest among all objects. By making use of the indexing structures, an NN query $Q_{nn}(q)$ is processed as detailed in Algorithm 6.

The current nearest neighbor and its distance from q are kept in nn and $dist_{nn}$ respectively (line 1). Initially, the algorithm gets the query position q's host partition v and searches v utilizing a grid index built on v (lines 2–3). Here, nnSearch($B_i, q, dist_{nn}$) searches an object bucket B_i to find the nearest neighbor from q, where $dist_{nn}$ is the current nearest neighbor distance that can be used for fast pruning. As mentioned above, bucket B_i contains all the objects in its corresponding indoor partition v_i, and q is either a position inside v_i or a door of v_i. Details regarding the intra-partition object organization and search are given in Section V-B.

Subsequently, for each door d_i through which one can leave v, search is conducted in the similar way as for range query processing (lines 4–19). All other doors are accessed in non-descending order of the distances from d_i by making use of the index in $M_{idx}[d_i, *]$ for the distances stored in $M_{d2d}[d_i, *]$. If a door d_j is within the current nearest distance, i.e., $dist_V(q, d_i) + M_{d2d}[d_i, d_j] \leq dist_{nn}$, d_j's connected partition(s) is searched (lines 11–19). Otherwise, door d_j is outside the current nearest distance, and therefore the search with respect to door d_i stops (lines 8–9).

In order to return k ($k > 1$) nearest neighbors, the algorithm can be extended as follows. Instead of two variables nn and $dist_{nn}$, a k-element array $result$ is used to maintain the k nearest neighbor as well as their distances to query position q. In particular, $result[i].nn$ ($1 \leq i \leq k$) is the i-th nearest neighbor and $result[i].dist$ is the corresponding distance to q. Initially, each $result.nn$ is set to null and each $result[i].dist$ is set to ∞. In addition, function nnSearch(.) is revised accordingly. It takes the entire $result$ array as the third parameter, and updates the array directly during its search rather than returning the result of a single nearest neighbor finally. Given a bucket B_i, nnSearch(.) continues searching B_i for nearest neighbors. When a nearest neighbor nn with distance $dist_{nn}$ to q is found, nnSearch(.) compares $dist_{nn}$ to $result[k].dist$. If $dist_{nn} < result[k].dist$, $(nn, dist_{nn})$ will be inserted into $result$ at the appropriate position so that all distances are in non-descending order and the old k-th element is discarded. Otherwise, i.e., $dist_{nn} \geq result[k].dist$, nnSearch(.) stops searching because no further objects in B_i can be closer to q than the current k nearest neighbors.

Algorithm 6 NN(Position q)

```
1:  nn ← null;  dist_nn ← ∞
2:  v ← getHostPartition(q)
3:  (nn, dist_nn) ← nnSearch(v's bucket, q, dist_nn)
4:  for each door d_i ∈ P2D_□(v) do
5:      r_1 ← dist_V(q, d_i)
6:      for j from 1 to |S_door| do
7:          d_j ← M_idx[d_i, j]
8:          if r_1 + M_d2d[d_i, d_j] > dist_nn then
9:              break
10:         else
11:             r_2 ← r_1 + M_d2d[d_i, d_j]
12:             if DPT[d_j].vPtr_1 ≠ null then
13:                 (obj, dist) ← nnSearch(DPT[d_j].vPtr_1, d_j, dist_nn −
                    r_2)
14:                 if dist + r_2 < dist_nn then
15:                     (nn, dist_nn) ← (obj, dist + r_2)
16:             if DPT[d_j].vPtr_2 ≠ null then
17:                 (obj, dist) ← nnSearch(DPT[d_j].vPtr_2, d_j, dist_nn −
                    r_2)
18:                 if dist + r_2 < dist_nn then
19:                     (nn, dist_nn) ← (obj, dist + r_2)
20: return R
```

B. Intra-Partition Object Index and Search

To process the two types of queries more efficiently, we build a grid index for spatial objects in each indoor partition. We apply a uniform grid to each partition, where each grid cell captures the positions of all the objects in it. Accordingly, we organize all spatial objects in an indoor partition v_i using a corresponding bucket B_i, and B_i consists of multiple sub-buckets each of which corresponds to a grid cell.

The grid index for each indoor partition is built statically when the floor plan is given. As a matter of fact, the grid for a partition needs not necessarily be uniform. We can take into account the shape and possible obstacles in a partition when

constructing the grid. For example, we can create a grid in such a way that some cells cover obstacles only and thus have no corresponding object sub-buckets. As the grid configuration is not the focus of this paper, we omit the low-level details.

We use a simple grid to index spatial objects within each indoor partition because a grid is able to accelerate the distance comparison within a partition. This grid configuration information helps us prune non-qualifying sub-buckets and thus their objects. This way, we can further prune the search space within each indoor partition.

For rangeSearch(B_i, q, r), we search only those grid cells (and their sub-buckets in B_i) that overlap the circle centered at q and with radius r. If a cell is fully inside the circle, all the objects in its sub-bucket are included directly.

For nnSearch($B_i, q, dist_{nn}$), similarly, we need to search only those grid cells (and their sub-buckets in B_i) that overlap the circle centered at q and with radius $dist_{nn}$. During the intra-partition search, $dist_{nn}$ is updated dynamically to reduce the search range.

VI. Experimental Evaluation

We experimentally evaluate our proposals in this section. Section VI-A evaluates the indoor distance computation algorithms of the indoor distance model. Section VI-B evaluates the query algorithms. All algorithms are implemented in Java.

A. Performance of Distance Computation

We evaluate the efficiency of three indoor position-to-position distance algorithms, namely Algorithms 2, Algorithm 3, and Algorithm 4 that are proposed in Section III-D2. Both a desktop PC and a mobile phone are used to run the distance algorithms. Specifically, we use a desktop PC with a 2.66GHz Core2 Duo CPU and 3.25GB main memory running Windows XP Pro. In addition, we use a Samsung Nexus S phone with a 1GHz ARM processor and 346MB memory accessible to applications and running Android.

We generate different office building plans. For each floor of a building, we generate 30 rooms and 2 staircases, and all of them are connected by doors to a hallway in a star-like manner. We vary the number of floors and therefore also the number of doors in each simulated building.

Given a simulated building generated as described above, we treat each staircase as a special room with two doors, each of which connects to its corresponding floor. Inside such a virtual room, the door-to-door distance is the actual walking distance when using the corresponding staircase. This way, the entire multi-floor building is "transformed" into a flat one with a single floor. We then apply the distance-aware model proposed in Section III-C and compute the position-to-position distances accordingly. For each algorithm invocation, we generate at random two indoor positions in the floor plan.

On the desktop PC, we run each algorithm 50 times with random indoor positions, measuring the average running time for each algorithm. According to the results reported in Figure 6, the refined Algorithms 3 and 4 clearly outperform the simpler Algorithm 2. This is attributed to the fact that the latter blindly calls the basic door-to-door distance algorithm (Algorithm 1) without any optimizations.

It is also seen that Algorithms 3 and 4 scale well with the number of floors (and doors), indicating that the optimizations of reusing computations are quite effective and that the optimized algorithms are scalable with respect to the size of an indoor space.

Taking a closer look, Algorithm 4 incurs less time than does Algorithm 3, especially when the indoor space size is large. With 40 floors, Algorithm 4 runs faster than Algorithm 3 by up to 0.5 seconds. This indicates that the optimization applied to Algorithm 4 is effective.

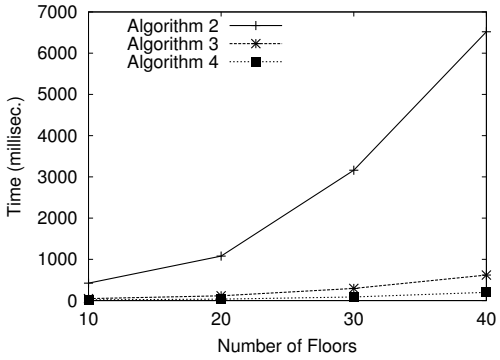


Fig. 6. Distance Algorithms on Desktop PC

On the Android mobile phone, we run each algorithm 10 times with random indoor positions and still measure the average running time for each algorithm. The results are shown in Figure 7, where we disregard Algorithm 2 because it is considerably slower than the two refined versions.

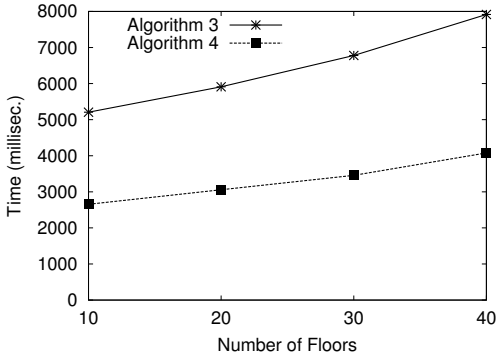


Fig. 7. Distance Algorithms on Android Phone

The results in Figure 7 show that Algorithm 4 runs approximately twice as fast as Algorithm 3 in all settings. This is a significant performance gain in terms of user experience on resource-constrained smartphones.

It is expected that the pt2ptdistance algorithm runs faster if the door-to-door distances are pre-computed and stored for reference. However, such pre-computed results are not always available. Moreover, it can be space-consuming to maintain such pre-computed results, e.g., on a resource-constrained mobile device. These observations, as well as the better performance of Algorithms 3 and 4 as seen in this section, justify the two refined versions of the pt2ptdistance algorithm.

B. Performance of Query Processing

We proceed to evaluate the query processing algorithms proposed in Section V using the desktop PC.

We generate indoor objects (POIs) as follows. Given an indoor space as generated in Section VI-A, a floor is first chosen at random, and then a partition is picked at random on that floor. Subsequently, a random position within the particular indoor partition is chosen as the object's position. In summary, all indoor objects are distributed randomly in the given indoor space.

The indoor partitions used in the experiments do not all have the same size. Nevertheless, the exact size of a partition is not a concern, as all POIs are generated in relative units with respect to a partition. We believe that the star-like manner is a reasonably simplified case that reflects a frequently seen type of indoor spaces.

In all experiments, we issue 100 queries for each query type and report the average response time per query type. For each query type, we look into the effects of three aspects.

First, we investigate how use of the door-to-door distance index matrix proposed in Section IV-A affects query performance. In all experiments relevant to this aspect, we generate spatial objects in a 30-floor building with about 1,000 doors. The number of objects is varied from 1K to 50K, and the corresponding query parameters are set to their default values. When the distance index matrix M_{idx} is not used, each query processing algorithm has to search the entire door-to-door distance matrix M_{d2d} directly.

Second, we investigate how the building size, i.e., the number of floors, affects the query performance. In particular, we set the object density on each floor to 10,000, and we vary the number of floors from 10 to 40. We compare the query performance with and without the distance index matrix.

Third, we investigate how query performance is affected by the corresponding query parameters. In the relevant experiments, we use the same data sets as above, and we process queries by using the distance index matrix.

We store all indexing structures in main memory. The size of the Distance Index Matrix is $|\mathcal{S}_{door}| \times |\mathcal{S}_{door}| \times 4$ bytes, where each matrix element is a door identifier of integer type. The simulated 40-floor building in our experiment has 30 doors plus 2 virtual doors (staircases) at each floor and therefore 32×40 = 1280 doors in total. The corresponding Distance Index Matrix's size is $1280 \times 1280 \times 4$ = 6.25 MB. Each element in DPT is of $4+4+8+4+8$ = 28 bytes. The size of DPT is at most $2 \times |\mathcal{S}_{door}| \times 28 = 56 \times |\mathcal{S}_{door}|$ bytes as each door touches at most two partitions. For the 40-floor building in our experiment, DPT's size is thus 56×1280 = 70 KB.

1) Range Query Performance: For a range query, we randomly pick a floor and generate a random query position on that particular floor. The default range distance is set to 30 meters. We study the effect of using the index matrix when varying the number of objects and the number of floors, and we study the effect of varying the range query size when using the index matrix. The results are reported in Figure 8.

Referring to Figure 8(a), the index matrix improves the query performance only moderately. This is mainly because the query range is not large. The distance index matrix does not help much in pruning when the query range is limited.

According to the results shown in Figure 8(b), the distance index matrix improves the range query performance, especially when the building has more floors and doors.

As the query range increases, according to the results shown in Figure 8(c), the query response time also increases. However, the overall time is still moderate. This indicates that the algorithm using the distance index matrix is effective.

2) kNN Query Performance: For a kNN query, we still randomly pick a floor and then generate a random query position on that particular floor. The default k is set to 100, and the experimental results are reported in Figure 9.

Figure 9(a) shows that the distance index matrix improves the kNN query performance significantly. In particular, the running time is about four times shorter across all object cardinalities. This indicates that the kNN query processing algorithm equipped with the distance index matrix is effective in pruning unpromising parts of the search space. The non-ascending ordering of door-to-door distances indicated by the index matrix enables us to prune indoor partitions drastically whenever the current kNN bound is reduced.

Next, Figure 9(b) again shows that use of the distance index matrix improves kNN query performance significantly. Importantly, the performance gain compared to the cases without the index matrix increases when the number of floors increases. Thus, the matrix renders the kNN query processing scalable with respect to building size.

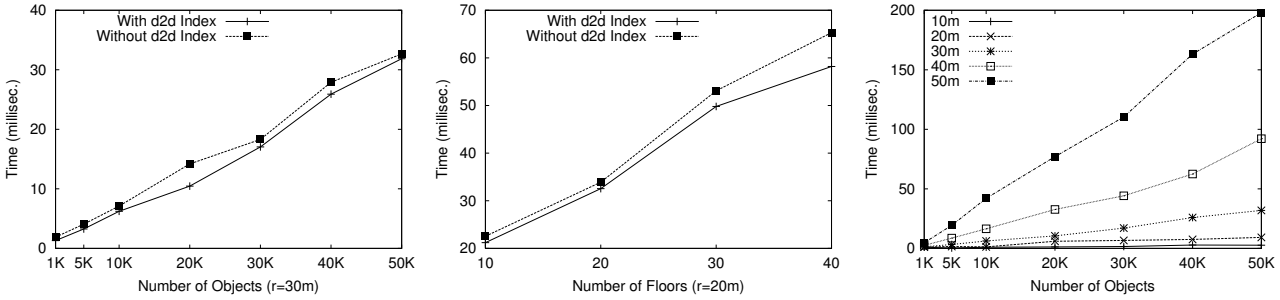
The results in Figure 9(c) show that larger k values incur longer query response times. This is because a larger k value tends to result in more indoor partitions having to be searched for nearest neighbors. Nevertheless, the overall response time of searching for up to 200 indoor nearest neighbors is still as low as 4.5 milliseconds. Thus, matrix-facilitated kNN search is scalable in terms of k.

VII. Conclusion and Future Work

In this paper, we propose a distance-aware indoor space data model that is also equipped with efficient indoor distance computation algorithms. To support efficient distance-aware queries, an indexing framework is designed to organize indoor distances that are computed by the proposed algorithms. On top of the foundation thus formed, specific algorithms are proposed for processing indoor distance-aware queries over indoor spatial objects.

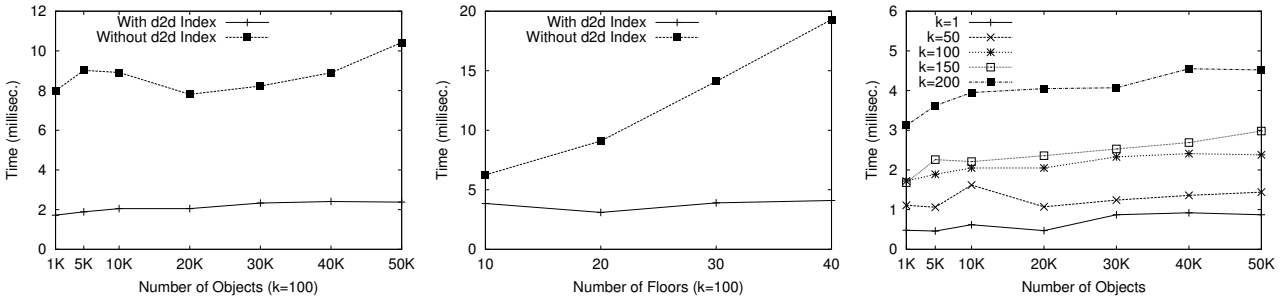
An experimental study is conducted to evaluate the proposals on a desktop PC as well as an Android mobile phone. The results show that the proposals are effective, efficient, and scalable in terms of the indoor space size, indoor object cardinality, as well as the query parameters.

Several directions for future work exist. It is of interest to introduce temporal variations in a distance-aware indoor space model and query processing. For example, some doors in a building may be open only during particular periods of time.



(a) Effect of Object Number (b) Effect of Floor Number (c) Effect of Query Parameter

Fig. 8. Performance of Range Query



(a) Effect of Object Number (b) Effect of Floor Number (c) Effect of Query Parameter

Fig. 9. Performance of kNN Query

Accordingly, an indoor space model must be able to return corresponding indoor distances for different time points.

Further, it is also relevant to consider other types of distance-aware indoor queries that capture practical needs, by using the query types in this paper as building blocks.

Yet another relevant possibility is to propose an integrated distance model for both outdoor and indoor spaces, such that outdoor and indoor location-based services can be supported seamlessly. In such a scenario, the shortest distance path from an outdoor/indoor position to another outdoor/indoor position may involve outdoor and indoor spaces in an interweaved fashion. Consequently, simply applying an outdoor model followed by an indoor model, or the other way around, does not work because it disables the interweaving.

ACKNOWLEDGMENTS

This research was partially supported by the Indoor Spatial Awareness project of the Korean Land Spatialization Group and BK21 program. The authors thank Artur Baniukevic for running the distance algorithms on the Android mobile phone.

REFERENCES

[1] MobyGames. http://www.mobygames.com/.

[2] P. K. Agarwal, B. Aronov, and M. Sharir. Motion Planning for a Convex Polygon in a Polygonal Environment. *Discrete & Computational Geometry*, 22(2):201–221, 1999.

[3] P. Bahl and V. Padmanabhan. RADAR: an in-building RF-based user location and tracking system. In *Proc. of INFOCOM*, pages 775–784, 2000.

[4] C. Becker and F. Dürr. On location models for ubiquitous computing. *Personal and Ubiquitous Computing*, 9(1):20–31, 2005.

[5] G. Franz, H. Mallot, and J. Wiener. Graph-based Models of Space in Architecture and Cognitive Science-a Comparative Analysis. In *Proc. IIAS InterSymp*, pages 30–38, 2005.

[6] M. Hermersdorf. Indoor Positioning with a WLAN Access Point List on a Mobile Device. In *Proc. Workshop on World-Sensor-Web*, 5 pages, 2006.

[7] J. Hightower and G. Borriello. Location Systems for Ubiquitous Computing. *IEEE Computer*, 34(8):57–66, 2001.

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

[9] J. Lee. 3D GIS for geo-coding human activity in micro-scale urban environments. In *Proc. GIScience*, pages 162–178, 2004.

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

[11] D. Li and D. L. Lee. A lattice-based semantic location model for indoor navigation. In *Proc. MDM*, pages 17–24, 2008.

[12] D. Li and D. L. Lee. A topology-based semantic location model for indoor applications. In *Proc. GIS*, page 6, 2008.

[13] J. Munkres. *Elements of algebraic topology*. Addison Wesley Publishing Company, 1993.

[14] C. N. T. Becker and T. H. Becker. A multilayered space-event model for navigation in indoor spaces. *Proc. Workshop on 3D Geo-Info*, pages 61-77, 2008.

[15] J. A. Storer and J. H. Reif. Shortest Paths in the Plane with Polygonal Obstacles. *J. ACM*, 41(5):982–1012, 1994.

[16] C. van Treeck and E. Rank. Analysis of building structure and topology based on graph theory. In *Proc. ICCCBE*, 10 pages, 2004.

[17] E. Whiting, J. Battat, and S. Teller. Topology of Urban Environments: Graph construction from multi-building floor plan data. In *Proc. CAAD Futures*, pages 115–128, 2007.

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

[19] W. Yuan and M. Schneider. Supporting 3D route planning in indoor space based on the LEGO representation. In *Proc. of ISA*, pages 16–23, 2010.

[20] W. Yuan and M. Schneider. iNav: An Indoor Navigation Model Supporting Length-Dependent Optimal Routing. In *Proc. of AGILE*, pages 299–314, 2010.

[21] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu. Spatial queries in the presence of obstacles. In *Proc. EDBT*, pages 366–384, 2004.