VIP-Tree: An Effective Index for Indoor Spatial Queries

Zhou Shao†, Muhammad Aamir Cheema†, David Taniar†, Hua Lu‡
†*Faculty of Information Technology, Monash University, Australia*
‡*Department of Computer Science, Aalborg University, Denmark*
{joe.shao,aamir.cheema,david.taniar}@monash.edu, luhua@cs.aau.dk

ABSTRACT

Due to the growing popularity of indoor location-based services, indoor data management has received significant research attention in the past few years. However, we observe that the existing indexing and query processing techniques for the indoor space do not fully exploit the properties of the indoor space. Consequently, they provide below par performance which makes them unsuitable for large indoor venues with high query workloads. In this paper, we propose two novel indexes called Indoor Partitioning Tree (IP-Tree) and Vivid IP-Tree (VIP-Tree) that are carefully designed by utilizing the properties of indoor venues. The proposed indexes are lightweight, have small pre-processing cost and provide near-optimal performance for shortest distance and shortest path queries. We also present efficient algorithms for other spatial queries such as *k* nearest neighbors queries and range queries. Our extensive experimental study on real and synthetic data sets demonstrates that our proposed indexes outperform the existing algorithms by several orders of magnitude.

1. INTRODUCTION

1.1 Motivation

Research shows that human beings spend more than 85% of their daily lives in indoor spaces [15] such as office buildings, shopping centers, libraries, and transportation facilities (e.g., metro stations and airports). Due to this important fact, the recent breakthroughs in indoor positioning technologies (see [20], and its references), and the widespread use of smart phones, indoor location-based services (LBSs) are expected to boom in the coming years [23, 1, 2] and some reports suggest that indoor LBSs would have an even bigger impact than their outdoor counterparts [3].

Indoor LBSs can be very valuable in many different domains such as emergency services, health care, location-based marketing, asset management, and in-store navigation, to name a few. In such indoor LBSs and many others, indoor distances play a critical role in improving the service quality. For example, in an emergency, an indoor LBS can guide people to the near by exit doors. Similarly, a passenger may want to find the shortest path to the boarding gate in an airport, a disabled person may issue a query to find accessible toilets within 100 meters in a shopping mall, or a student may issue a query to find the nearest photocopier in a university campus.

Proceedings of the VLDB Endowment, Vol. 10, No. 4

Driven by recent advances in indoor location technology and popularity of indoor LBSs, there is a huge demand for efficient and scalable spatial query-processing systems for indoor location data. Unfortunately, as we explain next, the outdoor techniques provide below par performance for indoor spaces and the existing indoor techniques fail to fully utilize the unique properties of indoor venues resulting in poor performance.

1.2 Limitations of Existing Techniques

1.2.1 Outdoor techniques

Techniques for outdoor LBSs cannot be directly applied for indoor LBSs due to the specific characteristics in indoor settings. Referring to the aforementioned examples, briefly speaking, we need to not only represent the spaces (airport, shopping center) in proper data model but also manage all the indoor features (lifts, escalators, stairs) and locations of interest (boarding gates, exit doors, and shops) such that search can be conducted efficiently. Indoor spaces are characterized by indoor entities such as walls, doors, rooms, hallways, etc. Such entities constrain as well as enable indoor movements, resulting in unique indoor topologies. Therefore, outdoor techniques cannot be directly applied on indoor venues.

One possible approach for indoor data management is to first model the indoor space to a graph using existing indoor data modelling techniques [21, 8] and then applying existing graph algorithms to process spatial queries on the indoor graph. However, as we demonstrate in our experimental study, this approach lacks efficiency and scalability – the state-of-the-art outdoor techniques ROAD [19] and G-tree [30] may take more than one second to answer a single shortest distance query. This is mainly because the existing outdoor techniques rely on the properties of road networks and fail to exploit the properties specific to indoor space. For example, the indoor graphs have a much higher average out-degree (up to 400) as compared to the road networks that have average out-degree of 2 to 4. Consequently, the size of the indoor graphs is much larger relative to the actual area it covers. For example, we use the buildings in Clayton campus of Monash University as a data set in our experiments and the corresponding indoor graph has around 6.7 Million edges and around $41,000$ vertices. Compared to this, the road network corresponding to California and Nevada states consists of around 4.6 Million edges and 1.9 Million vertices [9]. Thus, specialized techniques are required that carefully exploit the properties of indoor space to provide efficient results.

1.2.2 Indoor techniques

Adopting the idea of mapping the indoor space to a graph and applying graph algorithms, existing techniques use door-to-door graph [27] and/or accessibility base graph [21] to process various indoor spatial queries.

Door-to-door (D2D) graph [27]. In a D2D graph, each door in the indoor space is represented as a graph vertex. A weighted edge is created between two doors d_i and d_j if they are connected to the

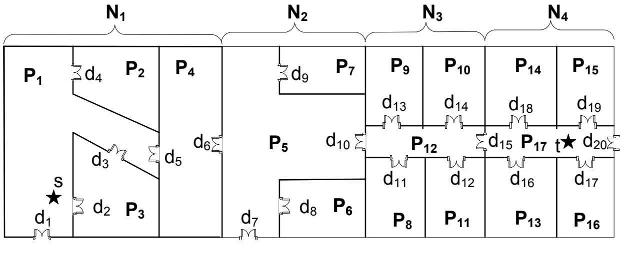
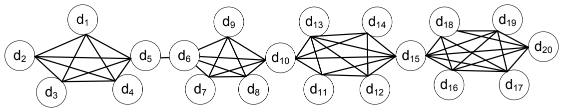
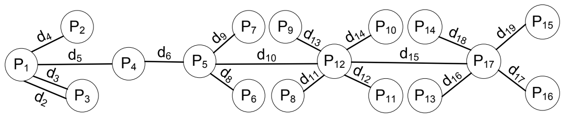


Figure 1: An indoor venue containing 17 partitions and 20 doors



(a) Door-to-Door Graph



(b) Accessibility Base Graph

Figure 2: Indexing Indoor Space

same indoor partition (e.g., room, hallway), where the edge weight is the indoor distance between the two doors. Fig. 1 shows an example of an indoor space that contains 17 indoor partitions (P_1 to P_{17}) and 20 doors (d_1 to d_{20}). The corresponding D2D graph is shown in Fig. 2(a) where edge weights are not displayed for simplicity. The doors d_1 to d_5 are all connected to each other by edges because they are associated to the same partition P_1.

Accessibility base (AB) graph [21]. In an AB graph, each indoor partition is mapped to a graph vertex, and each door is represented as an edge between the two partitions it connects. Fig. 2(b) shows the AB graph for the indoor space shown in Fig. 1. Since partitions P_1 and P_2 are connected by door d_4, an edge labeled as d_4 is created between P_1 and P_2 in the AB graph. Partitions P_1 and P_3 are connected by two doors d_2 and d_3, and thus two labeled edges are created between P_1 and P_3. Although an AB graph captures the connectivity information, it does not support indoor distances.

Distance matrix (DM) [21]. A distance matrix can also be used to facilitate shortest distance/path queries. A distance matrix stores the distances between all pairs of doors in the indoor space. Although this allows optimally retrieving the distance between any two doors (i.e., in $O(1)$), it requires huge pre-processing cost and quadratic storage which makes it unattractive for large indoor venues. Furthermore, the distance matrix cannot be used to answer k nearest neighbors (kNN) and range queries without utilizing other structures such as D2D graph and AB graph.

The existing techniques apply graph algorithms on a D2D graph and/or AB-graph to answer spatial queries. For instance, the state-of-the-art indoor spatial query processing technique [21] computes the shortest distance between a source point s and a target point t (shown as stars in Fig. 1) using Dijkstra's like expansion on a D2D graph or AB-graph. Although several optimizations are employed in [21], these techniques essentially rely on a Dijsktra's like expansion over the entire graph which is computationally quite expensive. Consequently, the state-of-the-art indoor query processing takes more than 100 seconds to answer a single shortest path query on the Clayton campus data set used in our experiments.

1.3 Contributions

In this paper, we propose two novel indoor indexes called Indoor Partitioning tree (IP-Tree) and Vivid IP-Tree (VIP-Tree) that optimize the indexing by exploiting the properties of indoor spaces. The basic observation is that the shortest path from a point in one indoor region to a point in another region passes through a small subset of doors (called access doors). For example, the shortest path between two points located on different floors of a building must pass one of the stairs/lifts connecting the two floors. The proposed indexes take into account this observation in their design and have the following attractive features.

Near-optimal efficiency. Our experimental study on real and synthetic data sets demonstrates that IP-Tree and VIP-Tree outperform the state-of-the-art techniques for indoor space [21] and road networks [30, 19] by several orders of magnitude. In comparison with the distance matrix, that allows constant time retrieval of distance between any two doors at the cost of expensive pre-computing and quadratic storage, our VIP-Tree also achieves comparable, near-optimal performance for shortest distance and path queries.

Low indexing cost. VIP-Tree and IP-Tree have small construction cost and low storage requirement. For example, for the largest data set used in our experiments that consists of around 83,000 rooms (around 13.4 Million edges), VIP-Tree and IP-Tree consume around 600 MB and can be constructed in less than 2 minutes. In contrast, it took almost 14 hours to construct the distance matrix for a much smaller building consisting of around 2,700 rooms (around 110,000 edges).

Low theoretical complexities. Our proposed indexes do not only provide practical efficiency but also have low storage and computational complexities. Table 1 compares the storage complexity and shortest distance/path computation cost of our proposed approach with the distance matrix which has near-optimal computational complexity. For the data sets used in our experiments, the average values of ρ and f are less than 4. For our proposed trees, M is the number of leaf nodes which is bounded by the number of doors D. Note that VIP-Tree has a significantly low storage cost compared to the distance matrix but has the same computational complexity.

	Storage	Shortest Dist.	Shortest Path
IP-Tree	$O(\rho^2 f^2 M + \rho D)$	$O(\rho^2 \log_f M)$	$O((\rho^2 + w) \log_f M)$
VIP-Tree	$O(\rho^2 f^2 M + \rho D \log_f M)$	$O(\rho^2)$	$O(\rho^2 + w)$
DM	$O(D^2)$	$O(\rho^2)$	$O(\rho^2 + w)$

Table 1: *Comparison of computational complexities. ρ: average # of access doors, f: average number of children in a node, M: # of leaf nodes, D: # of doors, w: # of edges on shortest path*

High adaptability. Similar to popular outdoor indexes (such as R-tree, Quad-tree, G-tree), our proposed indexes follow a branch-and-bound structure that can be easily adapted to answer various other indoor queries not covered in this paper. For example, the proposed indexes can be used to answer spatial keyword queries in indoor space by integrating the inverted lists with the nodes of the tree, e.g., in a way similar to how R-tree is extended to IR-tree [11] to support spatial keyword queries in outdoor space.

2. INDEXING INDOOR SPACE

First, we define some terminology and the data model used in this paper. An indoor partition that has only one door is called a *no-through* partition (e.g., partitions P_2, P_9 and P_{10} in Fig. 1) because no shortest path can pass through this partition. A partition which has more than γ doors is called a *hallway* partition. γ is a system parameter and is a small value (e.g., in this paper, we choose $\gamma = 4$). In Fig. 1, partitions P_1, P_5, P_{12} and P_{17} are the hallway partitions. All other partitions are called general partitions. A special indoor entity such as a staircase or an escalator connecting two floors is considered as a general partition with two doors at its connecting floors. Similarly, a lift connecting n floors is divided into $n - 1$ general partitions where each partition connects two consecutive floors.

Similar to existing work, we use a door-to-door graph [27] to model the indoor space. The distances between the doors can be set appropriately, e.g., set to zero for a lift/escalator if the distance corresponds to the *walking* distance or to a non-zero value if the distance is the travel time. We remark that such indoor data models can capture all spatial features of indoor space. If more details of geometric features are required (e.g., texture, color, shape of indoor objects), then the CityGML [7] data objects can be embedded in each partition. The results generated by our spatial query processing algorithms can be passed to other applications (e.g., [12, 14]) to provide visual/landmark-based navigation to the users. Next, we present the details of our indexes.

2.1 Indoor Partitioning Tree (IP-Tree)

2.1.1 Overview

The basic idea is to combine adjacent indoor partitions (e.g., rooms, hallways, stairs) to form leaf nodes and then iteratively combining adjacent leaf nodes until all nodes are combined into a single root node. Fig. 3 shows an IP-Tree of the indoor venue shown in Fig. 1 where the indoor space is first converted into four leaf nodes (N_1 to N_4). Each leaf node consists of several indoor partitions. Specifically, $N_1 = \{P_1, \cdots, P_4\}$, $N_2 = \{P_5, \cdots, P_7\}$, $N_3 = \{P_8, \cdots, P_{12}\}$, and $N_4 = \{P_{13}, \cdots, P_{17}\}$. The leaf nodes are iteratively merged until root node is formed, e.g., N_1 and N_2 are merged to form N_5 whereas N_3 and N_4 are merged to form N_6.

Definition 1. **Access door.** A door d is called an access door of a node N if d connects it to the space outside of N (i.e., one can enter or leave N via d). The set of access doors of a node N are denoted as $AD(N)$.

In Fig. 1, the access doors of N_1 are d_1 and d_6. IP-Tree stores the access doors for each node in the tree. Fig. 3 shows the access doors of each node in the boxes below the nodes, e.g., $AD(N_1) = \{d_1, d_6\}$ and $AD(N_5) = \{d_1, d_7, d_{10}\}$. Note that the shortest path to/from a point s in N_1 to/from a point t outside of N_1 must pass through one of its access doors d_1 and d_6.

	d_1	d_6	d_7	d_{10}
d_1	0	9	13, d_6	15, d_6
d_6	9	0	4	6
d_7	13, d_6	4	0	7
d_{10}	15, d_6	6	7	0

Distance Matrix for N_5

	d_1	d_7	d_{10}	d_{20}
d_1	0	13	15	25, d_{10}
d_7	13	0	7	17, d_{10}
d_{10}	15	7	0	10
d_{20}	25, d_{10}	17, d_{10}	10	0

Distance Matrix for N_7

	d_1	d_2	d_3	d_4	d_5	d_6
d_1	0	2	5, d_2	6	7, d_3	9, d_5
d_6	9, d_2	7, d_3	4, d_5	7, d_5	2	0

Distance Matrix for N_1

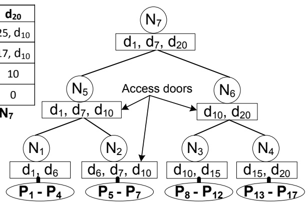


Figure 3: Indoor Partitioning Tree

To efficiently compute shortest distance/path between indoor locations, the IP-Tree stores distance matrices for leaf nodes and non-leaf nodes. Below, we provide the details.

Distance matrices for leaf nodes. For each leaf node N, the distance matrix stores distances between every door $d_i \in N$ to every access door $d_j \in AD(N)$. Fig. 3 shows an example of the distance matrix for the node N_1 where the distances between every door $d_i \in N_1$ (i.e., d_1 to d_6) and every access door $d_j \in AD(N_1)$ (i.e., d_1 and d_6) are stored.

To support the shortest *path* queries, the distance matrix also stores some additional information. Specifically, for a leaf node N, in addition to the shortest distance between $d_i \in N$ and $d_j \in AD(N)$, the distance matrix also stores a door d_k on the shortest path from d_i to d_j. d_k is called the next-hop door for the entry corresponding to d_i and d_j. Specifically, if the shortest path from d_i to d_j lies entirely inside the node N then d_k corresponds to the first door on the shortest path from d_i to d_j. In Fig. 1, the next-hop door on the shortest path from d_1 to d_6 is d_2. Therefore, in the distance matrix of N_1 (see Fig. 3), d_2 is the next-hop door for the entry of d_1 in the row corresponding to d_6. Similarly, d_3 is the next-hop door for the entry corresponding to d_2 and d_6 because d_3 is the first door on the shortest path from d_2 to d_6.

If the shortest path from d_i to d_j passes outside of N then d_k corresponds to the first door on the shortest path that is an access door of at least one leaf node in the tree. Although this scenario is not common (and Fig. 1 does not have an example of it), this is critical to efficiently retrieve the shortest path between two points. We give a detailed example and reasoning of this later in Section 3.2. Finally, if the shortest path between d_i and d_j does not involve any other door (e.g., d_5 to d_6), the next-hop door is set as `NULL`. For better readability, the matrices in Fig. 3 show only non-null values.

Distance matrices for non-leaf nodes. Consider a non-leaf node N that has f children $N_1, N_2, \cdots, N_f$. The distance matrix of N stores distances between every access door of its children, i.e., it stores distances between all doors in $\cup_{i=1}^{f} AD(N_i)$. For example, in Fig. 3, the distance matrix of the node N_7 stores the distances between $AD(N_5)$ and $AD(N_6)$, i.e., d_1, d_7, d_{10} and d_{20}. Furthermore, for each entry d_i and d_j in the distance matrix of N, we also store the first door $d_k \in \cup_{i=1}^{f} AD(N_i)$ on the shortest path from d_i to d_j (called next-hop door as stated earlier). Note that d_k in this case is an access door of the children of N and is not any arbitrary door.

In Fig. 3, the entry in the distance matrix of N_7 corresponding to d_1 and d_{20} stores d_{10}. Note that the first door on the shortest path from d_1 to d_{20} ($d_1 \rightarrow d_2 \rightarrow d_3 \rightarrow d_5 \rightarrow d_6 \rightarrow d_{10} \rightarrow d_{15} \rightarrow d_{20}$) is d_2 but we maintain d_{10} in the distance matrix because it is the first door among the access doors of the children of N_7 that is on the shortest path from d_1 to d_{20}. The entry corresponding to d_1 and d_7 has `NULL` because the shortest path from d_1 to d_7 does not contain any access door of the children of N_7.

2.1.2 Constructing IP-Tree

The IP-tree is constructed in a bottom-up manner in four steps: 1) the indoor partitions are combined to create leaf nodes (also called level 1 nodes); 2) the nodes at each level l are merged to form the nodes at level $l + 1$. This is iteratively repeated until we only have one node at the next level; 3) the distance matrices for leaf nodes are constructed; 4) the distance matrices of non-leaf nodes are created. Next, we describe the details of each step.

1. Creating leaf nodes. Two partitions are called *adjacent* partitions if they have at least one common door (e.g., P_1 and P_2). We iteratively merge adjacent partitions and construct the leaf nodes by considering the following two simple two rules.

i. If a general partition has more than one adjacent hallways, it is merged with the hallway with greater number of common doors with the general partition. Ties are broken by preferring the hallway that is on the same floor. If the general partition occupies more than one floors (e.g., it is a staircase) or if both hallways are on the same floor, the tie is broken arbitrarily.

ii. Merging of a partition with a leaf node is not allowed if the merging will result in leaf node having more than one hallways. This is because the shortest distance/path queries between points in different hallways is more expensive. This rule ensures that all hallways are in different leaf nodes which allows us to fully leverage the tree structure to efficiently process the queries. The algorithm terminates when no further merging is possible, i.e., every possible merging will result in the violation of this rule.

EXAMPLE 1 : In Fig. 1, the partitions P_2 and P_3 are combined with the hallway partition P_1. The partition P_4 could be combined with either P_5 or P_1 because both P_1 and P_5 have exactly 1 common door with P_4 and are on the same floor. We assume that it is combined with P_1. Thus, P_1 to P_4 are combined to form the leaf node N_1. Note that the hallway P_5 cannot be included in the leaf node N_1 because doing so would violate the rule ii. The partitions P_6 and P_7 are combined with P_5 to form a leaf node N_2. Similarly,

P_8 to P_{12} are combined to form the node N_3 and P_{13} to P_{17} are combined to construct the leaf node N_4. The algorithm stops because no further merging is possible without violating rule ii. ■

2. Merging nodes of the IP-Tree. Let t be the minimum degree of the IP-Tree denoting the minimum number of children in each non-root node. Algorithm 1 shows the details of merging the nodes at level l (denoted as $\mathcal{N}_l$) to create the nodes at level $l + 1$ (denoted as $\mathcal{N}_{l+1}$) such that each node has at least t children. Alorithm 1 is iteratively called until $\mathcal{N}_{l+1}$ contains at most t nodes in which case all these nodes are merged to form the root node. Below, are the details of the algorithm.

We define degree of a node N_i at level $l + 1$ to be the number of level l nodes contained in N_i. A min-heap H is initialized by inserting all nodes in $\mathcal{N}_l$ and the key for each node is set to its degree initialized to one because no level l nodes are merged yet (line 1). If two nodes have the same degree, the heap prefers the node which has smaller number of adjacent nodes. This is because some nodes can only be merged with exactly one other node and such nodes should be given preferences in merging, e.g., in Fig. 1 and Fig. 3, N_1 is merged with N_2 and N_4 is merged with N_3 because both N_1 and N_4 can only be merged with exactly one other node.

Algorithm 1: createNextLevel($\mathcal{N}_l$, t)

Input : $\mathcal{N}_l$: nodes at the current level l, t: minimum degree
Output : $\mathcal{N}_{l+1}$: nodes at the next level $l + 1$
1 insert each $N_i \in \mathcal{N}_l$ in a min-heap H with key set to $N_i.degree = 1$;
2 **while** $H.top().degree < t$ **do**
3 deheap a node N_i from H;
4 $N_j \leftarrow$ node with highest number of common access doors with N_i;
5 remove N_j from H and merge N_i and N_j into a new node N_k;
6 insert N_k in H with key $N_i.degree + N_j.degree$;
7 move nodes from H to $\mathcal{N}_{l+1}$;

The nodes are iteratively de-heap from the heap and merged with one of the adjacent nodes with a goal to minimize the total number of access doors of the nodes at the parent level. Let $|AD(N_i)|$ denote the number of access doors of a node N_i and $|AD(N_i) \cap AD(N_j)|$ denote the number of *common* access doors in nodes N_i and N_j. If the two nodes N_i and N_j are merged into a parent node N, the number of access doors in the parent node N is $|AD(N_i)|+|AD(N_j)|-2 \times |AD(N_i) \cap AD(N_j)|$. Thus, the nodes that have a greater number of common access doors are given higher priority to be merged together (line 4). After a node N_i and N_j are merged to form a node N_k, the node N_k is inserted in the heap (line 6). The algorithm stops when the top node in the heap has a degree of at least t (line 2). This implies that every node in the heap contains at least t level l nodes, i.e., at least t children.

3. Constructing distance matrices for leaf nodes. Recall that the distance matrix for a leaf node N stores the distance and the next-hop door on the shortest path between every door $d_i \in N$ to every access door $d_j \in AD(N)$. We compute these distances and the next-hop doors using Dijkstra's search on the D2D graph. Specifically, for each access door d_j of a leaf node N, we issue a Dijkstra's search until all doors in the node N are reached. Since the doors of the leaf nodes are close to each other, this Dijkstra's search is quite cheap as only the nearby nodes in the D2D graph are visited.

EXAMPLE 2 : To create the distance matrix of leaf node N_1 that contains doors d_1 to d_6, we first issue a Dijkstra's search starting at d_1 on the graph shown in Fig. 2(a) and expand the search until all doors d_1 to d_6 are reached. The distances and next-hop doors are populated in the distance matrix row corresponding to the door d_1. The same process is repeated for the other access door d_6. ■

4. Constructing distance matrices for non-leaf nodes. Let leaf nodes be on level 1 of the tree (the lowest level) and root node be at the highest level of the tree. We construct the distance matrices of the nodes in a bottom-up fashion, i.e., the distance matrices of all the nodes at level l are created before the distance matrices of the nodes at level $m > l$. We construct the distance matrices of nodes at level $l > 1$ of the IP-Tree using a graph called *level-l graph* denoted as $\mathcal{G}_l$.

Level-l graph ($\mathcal{G}_l$). The vertices of $\mathcal{G}_l$ correspond to the access doors of the nodes at $(l-1)$-th level of the tree. An edge between two doors d_i and d_j is created in $\mathcal{G}_l$ if both d_i and d_j are the access doors of the same node at $(l-1)$-th level. The weight of the edge is $dist(d_i, d_j)$ which has already been computed when the distance matrices of $(l-1)$-th level were computed. Note that $\mathcal{G}_l$ is a connected graph because, at every level l, all nodes in the indoor space are connected through common access doors.

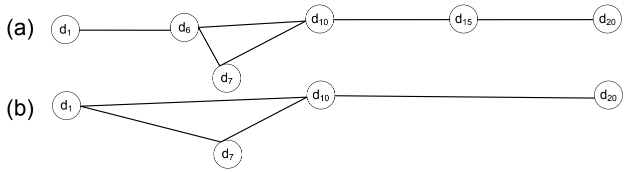


Figure 4: (a) $\mathcal{G}_2$: level-2 graph; (b) $\mathcal{G}_3$: level-3 graph

Fig. 4 shows level-2 and level-3 graphs for our running example. To construct the distance matrices of level 2 nodes of the tree shown in Fig. 3, we use the graph in Fig. 4(a) where the vertices correspond to the access doors of the nodes at level 1 (i.e., leaf nodes) of the tree (e.g., d_1, d_6, d_7, d_{10}, d_{15}, d_{20}). In $\mathcal{G}_2$ shown in Fig. 4(a), edges are created between d_6, d_7 and d_{10} because these are the access doors in the same leaf node (see Fig. 3). Similarly, to construct the distance matrices of level 3 nodes, we use the graph shown in Fig. 4(b) where the vertices of the graph are the access doors of level 2 nodes.

The distance matrix of a node N at level l of the tree is then computed using a Dijkstra's like expansion on $\mathcal{G}_l$ for each door d_i until all other doors d_j in N have been reached. This operation is quite efficient because i) the graph is significantly smaller than the original D2D graph and ii) the Dijkstra's expansion is not expensive because the relevant doors are close to each other in $\mathcal{G}_l$.

EXAMPLE 3 : To construct the distance matrix of node N_5, the graph shown in Fig. 4(a) is used. The distance matrix for N_5 contains the entries for doors d_1, d_5, d_7 and d_{10}. To populate the column corresponding to d_1, a Dijkstra's like expansion is conducted at d_1 on the graph shown in Fig. 4(a) until all other doors (i.e., d_5, d_7 and d_{10}) are reached. The entries for other doors are populated in the same way. ■

2.1.3 Storage Complexity

In addition to IP-tree, our algorithms also require the D2D graph to compute the shortest distance/path between two points located in the same leaf node of the IP-tree. D2D graph is a standard data structure used by all previous algorithms. In this section, we analyse the storage complexity of IP-Tree.

Let D and P denote the total number of doors and partitions in the indoor space, respectively. Let M be the number of leaf nodes where $M \leq P$. Let ρ be the average number of access doors in a node. The total size of *all* leaf node matrices is $O(\rho D)$. This is because the distance matrix for a leaf node N stores the distance between each door in N to every access door of the node. Note that each door can belong to at most two leaf nodes because each door is connected to at most two indoor partitions. Since the average number of access doors is $O(\rho)$, the total storage cost for all leaf node distance matrices is $O(\rho D)$.

Let f be the average number of children for a non-leaf node. Then, the average size of a non-leaf distance matrix is $O(\rho^2 f^2)$. Since each node is merged with at least one other node at the same level, the total number of nodes at a level l are at most half of the total number of nodes at level $l-1$. Hence, the total number of non-leaf nodes in IP-tree is $O(M)$ (bounded by the total number of leaf nodes). Hence, the total size of all distance matrices of non-leaf nodes is $O(\rho^2 f^2 M)$. Therefore, the total storage complexity[1] of IP-Tree is $O(\rho^2 f^2 M + \rho D)$. Note that IP-Tree also needs to store, for each partition P_i, the leaf nodes that contain P_i and the doors connected to it. The total cost of this is $O(D + P)$. Since $P \leq D$ (each indoor partition has at least one door), the total complexity of IP-Tree is $O(\rho^2 f^2 M + \rho D)$.

2.2 Vivid IP-Tree (VIP-Tree)

Vivid IP-Tree (VIP-Tree) is very similar to IP-tree except that it stores, for each door d_i in the indoor space, the following additional information. Let N be the leaf node that contains the door d_i. For every door d_j that is an access door in one of the ancestor nodes of N, VIP-tree stores $dist(d_i, d_j)$ as well the next-hop door d_k on the shortest path from d_i to d_j. This information can be efficiently computed by our efficient shortest distance/path algorithms using IP-tree.

As stated earlier, each door d_i can belong to at most two leaf nodes. Since the height of the tree is $O(\log_f M)$ and the average number of access doors in a node is ρ, VIP-Tree takes an additional $O(\rho \log_f M)$ space for each door d_i. Hence, the total additional cost for all doors is $O(\rho D \log_f M)$. Therefore, the total storage complexity of VIP-Tree is $O(\rho^2 f^2 M + \rho D \log_f M)$ as compared to $O(\rho^2 f^2 M + \rho D)$ cost of IP-Tree.

3. INDOOR QUERY PROCESSING

In this section, we propose our query processing algorithms for shortest distance queries, shortest path queries, *k*NN queries and range queries.

3.1 Shortest Distance Queries

3.1.1 Shortest Distance Using IP-Tree

In this section, we present algorithms to compute the indoor shortest distance $dist(s, t)$ between a source point s and a target point t. When both s and t are located in the same leaf node, $dist(s, t)$ can be computed using D2D graph (similar to existing approaches). Since s are t are close to each in D2D graph, the distance computation is not expensive. Next, we show how to compute $dist(s, t)$ when both s and t are in different leaf nodes.

Given a point p in the indoor space, we use `Partition(p)` and `Leaf(p)` to denote the partition and the leaf node that contains the point p, respectively. First, we describe how to compute the shortest distance between s and an access door d of the leaf node that contains s, i.e., $d \in AD($`Leaf(s)`$)$. Although $dist(s, d)$ in this case can be computed using D2D graph, we may improve the performance by utilizing the distance matrices stored in the leaf nodes. Below, we describe the details.

Shortest distance between s and an access door $d \in$ AD(`Leaf(s)`). In this paper, an access door d of `Leaf(s)` that is also a door of `Partition(s)` is called a local access door of `Partition(s)`. If the access door d is not a door of `Partition(s)`, it is called a global access door for `Partition(s)`. Fig. 5(a) shows the leaf node N_1 which has two access doors d_1 and d_6. d_1 is a local access door of P_1 and d_6 is a global access door of P_1.

If d is a local access door of `Partition(s)` then $dist(s, d)$ can be trivially computed. If d is a global access door, $dist(s, d)$ can be computed as follows.

$$dist(s, d) = min_{\forall d_i \in \texttt{Partition(s)}} dist(s, d_i) + dist(d_i, d) \qquad (1)$$

Since d is an access door of `Leaf(s)`, $dist(d_i, d)$ can be retrieved from its distance matrix in $O(1)$. However, the total cost may still be high if the number of doors in `Partition(s)` is large. We address this issue by using the concepts of *inferior* and *superior* doors of a partition.

Definition 2. **Superior door:** Let P be a partition and `Leaf(P)` be the leaf node containing the partition P. A door $d_i \in P$ is called a superior door of P if either i) d_i is a local access door of P or ii) there exists a global access door d_j such that the shortest path from d_i to d_j does not pass through any other door of the partition P.

The doors that are not superior are called inferior doors. Consider the example of Fig. 5(a) that shows a leaf node containing partitions P_1 to P_4. The access doors of the node are d_1 and d_6 where d_1 is the local access door of P_1 and d_6 is its global access door. The superior doors of the partition P_1 are d_1 and d_5. d_1 is the superior door because it is a local access door of the partition. d_5 is a superior door because the shortest path from d_5 to the global access door d_6 does not pass through any other door. The doors d_2, d_3 and d_4 are the inferior doors for partition P_1. For example, the door d_2 is an inferior door because the shortest path from d_2 to the global access door d_6 passes through at least one other door of the partition P_1.

Intuitively, the shortest path from any point $s \in P$ to any global access door d_j must pass through one of the superior doors of P. Therefore, we only consider the superior doors in Eq. 1. In the example of Fig. 5(a), the shortest path from $s \in P$ to d_6 must pass through one of its superior doors (d_1 or d_5). Hence, $dist(s, d_6) = min(dist(s, d_1) + dist(d_1, d_6), dist(s, d_5) + dist(d_5, d_6))$.

This significantly improves the cost of computing $dist(s, d)$ because the number of superior doors is significantly smaller than the total number of doors especially for hallways that contain many doors. Our experiments demonstrate that the maximum number of superior doors is 4 for all data sets even for the hallways that contain more than a hundred doors.

Shortest distance between s and all access doors of an ancestor of `Leaf(s)`. Let N be an ancestor node of `Leaf(s)`. We present an algorithm to compute the distances between s and *all* access doors of N. This is a key algorithm used in computing $dist(s, t)$ for two arbitrary points s and t located in different leaf nodes.

Algorithm 2 shows the details of computing $dist(s, d)$ for every $d \in AD(N)$ where N is an ancestor node of `Leaf(s)`. The basic idea is to first compute the distances from s to all access doors in `Leaf(s)` using the superior doors as described above. Then, the algorithm iteratively retrieves the parent node and computes distances to the access doors of the parent node until the ancestor node N is reached. Next lemma shows that $dist(s, d)$ for an access door d in N can be computed using the distances from s to the access doors of its child node.

Lemma 1. Let N_{parent} be the current node being processed and N_{child} be its child node. Let d be an access door of N_{parent}. The shortest path for a point $s \in N_{child}$ to d must pass through at least one access door of N_{child}.

PROOF. Note that an access door d of a parent node N_{parent} must be an access door of at least one of its children nodes. If d is the access door of N_{child} then the shortest path from s to d must end at d (which proves the lemma). If d is not an access door of N_{child}, then d must be a door outside of N_{child}. Hence, the shortest path from s (which is inside N_{child}) to d (which is outside N_{child}) must pass through at least one access door of N_{child}. □

If $dist(s, d_i)$ for every $d_i \in AD(N_{child})$ is known, then $dist(s, d)$ for a door $d \in AD(N_{parent})$ can be computed as follows.

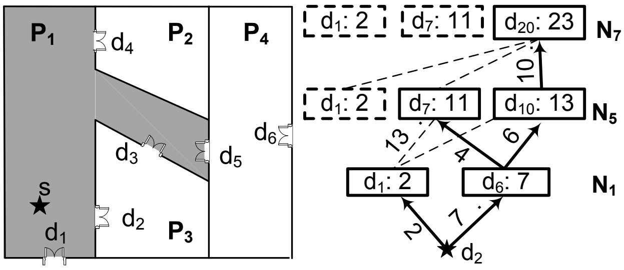
$$dist(s, d) = min_{\forall d_i \in AD(N_{child})} dist(s, d_i) + dist(d_i, d) \qquad (2)$$

Note that $dist(d_i, d)$ is stored in the distance matrix of the node N_{parent} because both d_i and d are the access doors of the children of N_{parent}. Hence, $dist(d_i, d_j)$ can be retrieved in $O(1)$.

[1]Our experiments on three real data sets demonstrate that f and ρ are small in practice (less than 4 for all real data sets).

Algorithm 2: getDistances (s, N)

Input : s: source, N: an ancestor node of $Leaf(s)$
Output : $Distances$: shortest distance between s and every $d \in AD(N)$
1 Initialize N_{parent} to be the parent node of $Leaf(s)$;
2 Initialize N_{child} to $Leaf(s)$;
3 **while** N_{child} is not the same as N **do**
4   **for** each unmarked $d \in AD(N_{parent})$ **do**
5         $dist(s, d) = min_{\forall d_i \in AD(N_{child})} dist(s, d_i) + dist(d_i, d)$;
6   mark d and then insert $dist(s, d)$ in $Distances$ if $d \in AD(N)$;
7     $N_{child} \leftarrow N_{parent}$;
8     $N_{parent} \leftarrow$ parent node of N_{parent};



(a) Superior doors   (b) Illustration of Algorithm 2

Figure 5: Shortest distance computation

Although Algorithm 2 is self explanatory, we elaborate it with an example.

EXAMPLE 4 : Consider the example of Fig. 1 and Fig. 3 and assume that we want to compute the shortest distances between d_2 and every access door of the root node N_7 (i.e., d_1, d_7 and d_{20}). `Leaf(`d_2`)` is the node N_1. The algorithm assumes that $dist(s, d)$ for every access door d of `Leaf(s)` has been computed as described above. For example, $dist(d_2, d_1) = 2$ and $dist(d_2, d_6) = 7$ have been computed (see Fig. 5(b)).

N_{parent} is initialized to be the parent node of N_1 (i.e., N_{parent} is N_5). The shortest distance to each access door in N_5 (e.g., d_1, d_7 and d_{10}) is then computed based on the distances from d_2 to the access doors in N_1. For instance, $dist(d_2, d_7) = min(dist(d_2, d_1) + dist(d_1, d_7), dist(d_2, d_6) + dist(d_6, d_7)) = min(2 + 13, 7 + 4) = 11$. Fig. 5(b) illustrates the processing of the algorithm where the incoming edges (thick arrows and broken lines) to a door demonstrate a possible path to the door and the thick arrows show the path that lead to minimum distance, e.g., d_7 has two incoming edges: one from d_1 and the other from d_6. The shortest distance is $dist(d_2, d_6) + dist(d_6, d_7) = 7 + 4 = 11$ and the edge between d_6 and d_7 is shown using a solid arrow. Similarly, $dist(d_2, d_{10}) = min(dist(d_2, d_1) + dist(d_1, d_{10}), dist(d_2, d_6) + dist(d_6, d_{10})) = 13$.

After $dist(s, d)$ is computed for every access door d of N_{parent}, the algorithm iteratively retrieves the parent node of N_{parent} to compute distances from s to its access doors (see lines 7 and 8). In Fig. 5(b), N_7 becomes N_{parent} and N_5 becomes N_{child} and the distances to the access doors of N_7 are computed using the previously computed distances to the access doors of N_5. For example, $dist(d_2, d_{20})$ is the the minimum of $dist(d_2, d_1) + dist(d_1, d_{20})$, $dist(d_2, d_7) + dist(d_7, d_{20})$ and $dist(d_2, d_{10}) + dist(d_{10}, d_{20})$. The thick arrows show the shortest path from d_2 to each access door.

If $dist(s, d)$ for a door d in N_{parent} is already known because d is also an access door for N_{child}, its distance is not needed to be recomputed. Fig. 5(b) shows such doors in a rectangle drawn in broken lines, e.g., $dist(d_2, d_1)$ is computed at node N_1 and it does not need to be recomputed when nodes N_5 and N_7 are accessed. In Algorithm 2, we mark each door d for which $dist(s, d)$ has been computed (line 6) and only compute the distances from s to the doors that are not marked (line 4). ■

Shortest distance between two arbitrary points s and t. Now, we are ready to describe how to compute $dist(s, t)$ for two arbitrary points s and t located in different leaf nodes `Leaf(s)` and `Leaf(t)`.

Lemma 2. Let $LCA(s, t)$ be the lowest common ancestor node of `Leaf(s)` and `Leaf(t)`. Let N_s (resp. N_t) be the child of $LCA(s, t)$ which is an ancestor of `Leaf(s)` (resp. `Leaf(t)`). The shortest path from s to t must path through at least one access door of N_s and at least one access door of N_t.

PROOF. We first show that t lies outside N_s. We prove this by contradiction. Assume that t is inside N_s. If t is inside N_s then N_s must be a common ancestor of the leaf nodes containing s and t. However, N_s is the child of the *lowest* common ancestor of `Leaf(s)` and `Leaf(t)`. Hence, N_s cannot be a common ancestor which contradicts the assumption that t lies inside N_s.

Since t lies outside N_s and s lies inside N_s, the shortest path from s to t must pass through an access door of N_s (by definition of access doors). Following the same reasoning, the shortest path from s to t must also pass through an access door of N_t. □

Consider the example of Fig. 1 and Fig. 3 where s is in N_1 and t is in N_4, $LCA(s, t)$ is the node N_7, N_s is N_5 and N_t is N_6. The shortest path between s to t must pass through an access door of N_5 and an access door of N_6, e.g., the shortest path in Fig. 1 passes through d_{10} which is an access door for both N_5 and N_6.

By using the above lemma, $dist(s, t)$ can be computed as follows.

$$dist(s, t) = min_{\forall d_i \in AD(N_s), \forall d_j \in AD(N_t)} dist(s, d_i) + dist(d_i, d_j) + dist(d_j, t) \quad (3)$$

Note that $dist(d_i, d_j)$ is stored in the distance matrix of $LCA(s, t)$ because N_s and N_t are the child nodes of $LCA(s, t)$ and d_i and d_j are the access doors of N_s and N_t, respectively. $dist(s, d_i)$ for every $d_i \in AD(N_s)$ and $dist(d_j, t)$ for every $d_j \in AD(N_t)$ can be computed using Algorithm 2. Algorithm 3 shows the details of computing $dist(s, t)$ when s and t are in different leaf nodes.

Algorithm 3: $dist(s, t)$ when s and t are in different leaf nodes

1 $N_s \leftarrow$ ancestor of Leaf(s) and a child of LCA(Leaf(s), Leaf(t));
2 $N_t \leftarrow$ ancestor of Leaf(t) and a child of LCA(Leaf(s), Leaf(t));
3 getDistances(s, N_s); `/* Algorithm 2 */;`
4 getDistances(t, N_t); `/* Algorithm 2 */;`
5 **return** $min_{\forall d_i \in N_s, \forall d_j \in N_t} dist(s, d_i) + dist(d_i, d_j) + dist(d_j, t)$

Complexity Analysis. First, we evaluate the cost of Algorithm 2. Let ρ be the average number of access doors in a node. To compute the distance from s to a door d in a node N_{parent}, the algorithm considers paths through all access door in the child node N_{child} (see Eq. (2)). Hence, the cost to compute the distance of one door at node N_{parent} is $O(\rho)$ assuming that distances to every access door in N_{child} are known. Hence, the total cost to compute distances from s to all doors in a node N_{parent} is $O(\rho^2)$. Let h be the number of nodes between `Leaf(s)` and the node N. The total cost for computing distances from s to every $d \in AD(N)$ is $O(h\rho^2)$.

Recall that Algorithm 2 also requires computing distances between s and every access door of `Leaf(s)`. Let α be the average number of superior doors in a partition. The cost to compute distances from s to every access door in `Leaf(s)` is $O(\alpha\rho)$. Hence, the total cost of Algorithm 2 is $O(h\rho^2 + \alpha\rho)$.

Now, we evaluate the total cost of Algorithm 3. The cost of line 5 of the algorithm is $O(\rho^2)$ because each of N_s and N_t has $O(\rho)$ access doors. Also, the algorithm makes two calls to Algorithm 2. Therefore, the total cost of the algorithm is the same as that of Algorithm 2, i.e., $O(h\rho^2 + \alpha\rho)$. Since α and ρ both are very small values and $\alpha \approx \rho$, we simplify the complexity to $O(h\rho^2)$. Note that h is bounded by the height of the tree which is $O(\log_f M)$ where M is the number of leaf nodes in the tree.

3.1.2 Shortest Distance Using VIP-Tree

The shortest distance computation using VIP-tree is similar except that we modify Algorithm 2 that computes the distances from s to all access doors of an ancestor node N. Let SUP denote the set of superior doors of `Partition(s)`. Then, $dist(s,d)$ for an access door d of an ancestor node N is $dist(s,d) = min_{\forall d_i \in SUP} dist(s,d_i) + dist(d_i,d)$. Recall that VIP-Tree stores distances between d_i to all access doors of its ancestor nodes. Hence, $dist(d_i,d)$ can be retrieved in $O(1)$.

Let α be the average number of superior doors. The total cost of the modified Algorithm 2 is $O(\alpha\rho)$ as compared to $O(h\rho^2 + \alpha\rho)$ cost of the original Algorithm 2 used by IP-tree. For VIP-Tree, Algorithm 3 uses the modified Algorithm 2 and this reduces the overall cost for VIP-tree to $O(\rho^2 + \alpha\rho)$ from $O(h\rho^2 + \alpha\rho)$. This can be simplified to $O(\rho^2)$ considering that $\alpha \approx \rho$.

3.2 Shortest Path Queries

3.2.1 Shortest Path Using IP-Tree

As described earlier, if both s and t are in the same leaf node we use an expansion similar to Dijkstra's algorithm on the D2D graph to compute $dist(s,t)$. Thus, the actual shortest path can be easily maintained during the computation of $dist(s,t)$. Next, we describe how to recover shortest path when s and t are in different leaf nodes.

During the shortest distance computation (Algorithm 3), we maintain the intermediate doors on the path accessed by the algorithm. This gives a partial shortest path. For example, in the example of Fig. 5(b), the partial shortest path from d_2 to d_{20} is $d_2 \to d_6 \to d_{10} \to d_{20}$ (see thick arrows). Next, we describe how to decompose these edges to recover the complete shortest path.

An edge $d_i \to d_j$ is called a final edge if the shortest path from d_i to d_j does not contain any other door. Otherwise, the edge $d_i \to d_j$ is called a partial edge. We recursively decompose each partial edge $d_i \to d_j$ on the partial shortest path until each decomposed edge is a final edge. In this section, when we say a door d_i is an access door without referring to any specific node, it means that d_i is an access door of at least one node in the tree. Algorithm 4 describes how to decompose an edge $d_i \to d_j$.

Algorithm 4: Decompose($d_i \to d_j$)

1 **if** d_i and d_j both are non-access doors **then**
2 $d_i \to d_j$ is a final edge; `/* Lemmas 4 and 6 */;`
3 **else**
4 **if** d_i and d_j both are access doors **then**
5 $N \leftarrow$ the lowest common ancestor of d_i and d_j;
6 **else** `// only one of` d_i `and` d_j `is access door`
7 $N \leftarrow$ leaf node containing d_i & d_j; `/* Lemmas 4 and 7 */;`
8 Let d_k be the next-hop door of d_i and d_j in the distance matrix of N;
9 **if** d_k is NULL **then**
10 $d_i \to d_j$ is a final edge; `/* Lemma 3 */;`
11 **else**
12 Return $d_i \to d_k \to d_j$;

If both d_i and d_j are non-access doors then it can be proved that $d_i \to d_j$ is a final edge (Lemmas 4 and 6 in Section 3.2.2). Note that $d_i \to d_j$ is either an edge returned by Algorithm 3 or an edge resulting from decomposition of another edge by Algorithm 4. The proof is non-trivial and is given in the next section.

If both d_i and d_j are the access doors (line 4 of Algorithm 4), we will use the distance matrix of the lowest common ancestor node N of `Leaf(`d_i`)` and `Leaf(`d_j`)`. Otherwise, if only one of d_i and d_j is an access door, we will use the distance matrix of the leaf node N that contains both d_i and d_j. Lemmas 4 and 7 in the next section prove that, for each such edge $d_i \to d_j$ considered by Algorithm 4, we can always find both d_i and d_j in the same leaf node N.

Let N be the node as described above. We look up the distance matrix of N and retrieve the next-hop door d_k for the entry corresponding to d_i and d_j. The shortest path $d_i \to d_j$ is then decomposed to $d_i \to d_k \to d_j$. If d_k is `NULL` then $d_i \to d_j$ is a final edge and does not need to be decomposed (as we prove later in Lemma 3).

EXAMPLE 5 : Suppose we want to decompose $d_{10} \to d_{20}$. The lowest common ancestor of d_{10} and d_{20} is N_6 (see Fig. 3). The next-hop door for d_{10} and d_{20} in the distance matrix of N_6 is d_{15}. Therefore, $d_{10} \to d_{20}$ is decomposed into $d_{10} \to d_{15} \to d_{20}$. The algorithm then tries to decompose $d_{10} \to d_{15}$ using the lowest common ancestor N_3 of d_{10} and d_{15}. The next-hop door of d_{10} and d_{15} in the distance matrix of N_3 is `NULL`. Therefore, $d_{10} \to d_{15}$ is a final edge. Similarly, $d_{15} \to d_{20}$ is also a final edge.

Now, assume we want to decompose $d_2 \to d_6$. Since only d_6 is an access door, we find the leaf node N_1 that contains both d_2 and d_6. The next-hop door from d_2 to d_6 in the distance matrix of N_1 is d_3. Hence, $d_2 \to d_6$ is decomposed to $d_2 \to d_3 \to d_6$. $d_2 \to d_3$ is a final edge because both d_2 and d_3 are non-access doors. We decompose $d_3 \to d_6$ to $d_3 \to d_5 \to d_6$ in a similar way using the distance matrix of N_1. $d_3 \to d_5$ is a final edge because both d_3 and d_5 are non-access doors. $d_5 \to d_6$ is a final edge because the next-hop door for d_5 and d_6 in the distance matrix of N_1 is `NULL`. Hence, $d_2 \to d_6$ is decomposed to $d_2 \to d_3 \to d_5 \to d_6$. ■

A key property of Algorithm 4 is that if only one of d_i and d_j is an access door (see line 6) then there always exists a leaf node N that contains both d_i and d_j. We prove this later in Section 3.2.2. This property is made possible due to the special way we store next-hop door d_k for leaf nodes. Specifically, recall that if the shortest path from d_i to d_j passes outside of the leaf node N then next-hop door d_k is not any ordinary first door on the shortest path from d_i to d_j but d_k is the first access door on the shortest path from d_i to d_j. As shown in the next example, the above property cannot be ensured if d_k is not selected this way.

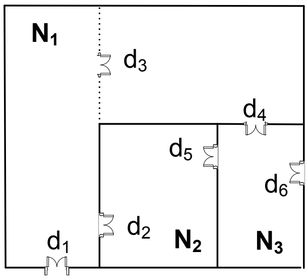


Figure 6: Choosing next-hop door for leaf nodes

EXAMPLE 6 : Consider the example of Fig. 6 that shows three leaf nodes N_1, N_2 and N_3. Suppose that we are creating the distance matrix of leaf node N_2 that contains two access doors d_2 and d_5. Assume that the shortest path from d_2 to d_5 is $d_2 \to d_3 \to d_4 \to d_5$. Note that d_3 is the first door on the shortest path. If we choose d_3 as the next-hop door, the edge $d_2 \to d_5$ will be decomposed into $d_2 \to d_3 \to d_5$. Now, if we try to decompose $d_3 \to d_5$, there does not exist any leaf node that contains both d_3 and d_5 (d_3 is a non-access door and d_5 is an access door). Hence, Algorithm 4 will fail to decompose it. To address this, we choose d_4 as the next-hop door which is the first access door on the shortest path. Hence, $d_2 \to d_5$ is decomposed to $d_2 \to d_4 \to d_5$. Note that d_2, d_4 and d_5 all are access doors and each edge can be further decomposed using the distance matrix of the least common ancestor node (at line 4). ■

3.2.2 Proof of correctness

In this section, we prove the correctness of Algorithm 4. First we show that $d_i \to d_j$ is a final edge if d_k is `NULL` (line 10).

Lemma 3. The next-hop door d_k for d_i and d_j in the distance matrix of N can only be `NULL` if $d_i \rightarrow d_j$ is a final edge.

PROOF. If N is a leaf level node and d_k is `NULL` then there does not exist any other door on the shortest path from d_i to d_j because the distance matrices for the leaf nodes are computed using the original D2D graph. Hence, $d_i \rightarrow d_j$ is a final edge. Next, we show that d_k cannot be `NULL` if N is a non-leaf node.

We prove this by contradiction. Let the lowest common ancestor node N be a non-leaf node at level $l > 1$ of the tree. Recall that the distance matrix of a node N at level l is computed using the level-l graph $\mathcal{G}_l$. The vertices in $\mathcal{G}_l$ are the access doors of the level $l-1$ nodes and an edge is created between two doors d_i and d_j if both doors are the access doors of the same node at level $l-1$. Note that d_k can only be `NULL` if there exists an edge between d_i and d_j in $\mathcal{G}_l$. This implies that both d_i and d_j are the access doors of the same node N' at level $l-1$ of the tree. However, if this is the case then N cannot be a lowest common ancestor because N' is also a common ancestor at a lower level. □

Next, we need to show that, for each edge $d_i \rightarrow d_j$ considered by Algorithm 4, the following two conditions hold: (1) $d_i \rightarrow d_j$ is a final edge if both d_i and d_j are non-access doors (line 2); (2) d_i and d_j can both be found in the same leaf node N if only one of d_i and d_j is an access door (line 7). Note that the edges considered by Algorithm 4 are either the edges on the partial shortest path maintained during the execution of Algorithm 3 or the edges decomposed earlier by Algorithm 4 itself. First, we prove the above two conditions for each edge on the partial shortest path maintained by Algorithm 3.

Lemma 4. Let $d_i \rightarrow d_j$ be an edge returned by Algorithm 3. (1) $d_i \rightarrow d_j$ is a final edge if both d_i and d_j are non-access doors; (2) d_i and d_j can both be found in the same leaf node N if only one of the d_i and d_j is an access door.

PROOF. Each of d_i and d_j at line 5 of Algorithm 3 is an access door, e.g., $d_i \in AD(N_s)$ and $d_j \in AD(N_t)$. Similarly, Algorithm 2 (which is called by Algorithm 3) also considers only the access doors along the path except when the distance from s (resp. t) to the access doors of `Leaf(s)` (resp. `Leaf(t)`) is to be computed. Hence, the lemma is only applicable for the case when the distances from s (resp. t) to every access door d_j of `Leaf(s)` (resp. `Leaf(t)`) are computed. This is because both doors are access doors for each other edge. We prove the lemma for the case when distance from s to $d_j \in AD($`Leaf(s)`$)$ is computed. The proof for the distance from d_j to t is similar.

Note that the shortest path from s to d_j is $s \rightarrow d_i \rightarrow d_j$ where d_i is a door in `Partition(s)` (see Eq. (1)). The edge $d_i \rightarrow d_j$ contains one access door (d_j) and it is easy to see that both d_i and d_j are in the same leaf node `Leaf(s)` - this proves (2). Now, we prove (1) by showing that every edge on the shortest path from s to d_i is a final edge. Recall that we compute the shortest path between two points in the same leaf node using a Dijkstra's like expansion on the original D2D graph. Hence, every edge on the shortest path from $s \rightarrow d_i$ is a final edge. □

Next, we prove the two conditions for the edges that are obtained as a result of decomposing another edge by Algorithm 4. First, we show that the two conditions are only applicable to an edge if it was decomposed by Algorithm 4 using a leaf node N at line 8.

Lemma 5. Assume we decompose $d_i \rightarrow d_j$ into $d_i \rightarrow d_k \rightarrow d_j$ as described in Algorithm 4. If N is a non-leaf node then d_i, d_k and d_j all are access doors.

PROOF. Assume that the lowest common ancestor node N of d_i and d_j is at level $l > 1$ of the tree. Recall that the distance matrix of nodes at level $l > 1$ is created using a graph $\mathcal{G}_l$ that contains the access doors of nodes at level $l-1$. Hence, d_k is an access door of a node at level $l-1$. Note that N can only be a non-leaf node if both d_i and d_j are access doors. Hence, d_i, d_j and d_k all are access doors. □

Next, we prove the condition (1) for each edge decomposed by Algorithm 4.

Lemma 6. Assume we decompose $d_i \rightarrow d_j$ into $d_i \rightarrow d_k \rightarrow d_j$ as described in Algorithm 4. Each edge in $d_i \rightarrow d_k \rightarrow d_j$ satisfies the following: if both doors in the edge are non-access doors then the edge is a final edge.

PROOF. As stated in Lemma 5, if N is a non-leaf node then d_i, d_k and d_j all are access doors and this lemma is not applicable. Therefore, this lemma only applies when N is a leaf node.

Since at least one of d_i and d_j is an access door for each *partial* edge $d_i \rightarrow d_j$ considered by Algorithm 4 (Lemma 4), this lemma is only applicable to either $d_i \rightarrow d_k$ (assuming d_i is a non-access door) or $d_k \rightarrow d_j$ (assuming d_j is a non-access door). Without loss of generality, assume that d_i is a non-access door. The lemma is not applicable to $d_k \rightarrow d_j$ because d_j is an access door. We prove the lemma for $d_i \rightarrow d_k$.

Since d_i is a non-access door and d_j is an access door, Algorithm 4 decomposes $d_i \rightarrow d_j$ by retrieving the next-hop door d_k from the distance matrix of the leaf node N that contains both d_i and d_j. If d_k is an access door (e.g., shortest path from d_i to d_j passes outside of N) then the lemma is not applicable on $d_i \rightarrow d_k$ because at least one door is an access door. If d_k is not an access door then it is the next-hop door computed using the original D2D graph for the leaf node N. Hence, $d_i \rightarrow d_k$ is a final edge. □

The nex lemma proves the condition (2) for each edge decomposed by Algorithm 4.

Lemma 7. Assume we decompose $d_i \rightarrow d_j$ into $d_i \rightarrow d_k \rightarrow d_j$ as described in Algorithm 4. Each edge in $d_i \rightarrow d_k \rightarrow d_j$ satisfies the following: if only one of the doors is an access door then both doors can be found in the same leaf node.

PROOF. As stated in Lemma 5, if N is a non-leaf node then d_i, d_k and d_j all are access doors and this lemma is not applicable. Therefore, this lemma only applies when N is a leaf node.

If the shortest path from d_i to d_j lies entirely inside N then d_k is always inside N. This implies that d_i, d_j and d_k all are inside the same leaf node N. If the shortest path from d_i to d_j passes outside of N then, as stated earlier, d_k is always chosen to be an access door. Since at least one of d_i and d_j is an access door, the lemma is only applicable to one of $d_i \rightarrow d_k$ and $d_k \rightarrow d_j$ (because both doors in the other edge are access doors). Without loss of generality, assume that d_i is a non-access door. We prove the lemma for $d_i \rightarrow d_k$. Since d_i is a non-access door of the leaf node N then d_k must be an access door of N because the shortest path from d_i which is inside N cannot go out of N without passing through an access door of N. Hence, both d_i and d_k can be found in the leaf node N. □

3.2.3 Complexity Analysis

Let w be the number of doors on the shortest path from s to t. The algorithm needs to find the lowest common ancestor for $O(w)$ pairs of doors. Finding the lowest common ancestor for a single pair of doors takes at most $O(\log_f M)$ - the height of the IP-tree. Hence, the algorithm takes $O(w \log_f M)$ in addition to the cost of shortest distance query. Therefore, the total cost of the shortest path query is $O(w \log_f M + \rho^2 \log_f M)$.

3.3 Shortest Path Using VIP-Tree

Recall that VIP-Tree stores, for each door d_i in indoor space, its distance and next-hop door to every access door d_j of each of its ancestor node N. Similar to the leaf node distance matrices, the next-hop door d_k is the first access door on the shortest path from d_i to d_j if the shortest path from d_i to d_j passes outside of N. In this case, $d_i \rightarrow d_j$ is decomposed to $d_i \rightarrow d_k \rightarrow d_j$ and these edges are further decomposed in a way similar to IP-Tree.

If the shortest path from d_i to d_j lies entirely inside N then d_k is the first door on the shortest path from d_i to d_j. In this case, $d_i \rightarrow d_j$ is decomposed to $d_i \rightarrow d_k \rightarrow d_j$ where $d_i \rightarrow d_k$ is a final edge and $d_k \rightarrow d_j$ can be further decomposed. Note that d_k is inside the node N and the next-hop door for $d_k \rightarrow d_j$ can be found because d_j is an access door of an ancestor of `Leaf(`d_k`)`.

The worst case cost of the shortest path recovery is $O(w \log_f M)$ assuming that for each edge $d_i \rightarrow d_j$, the shortest path passes outside of N. This is because in this case the algorithm needs to find the lowest common ancestor for the decomposed edge $d_k \rightarrow d_j$. However, we remark that this worst case scenario is very rare in practice and, in almost all cases, the shortest path passes within N. Hence, the expected complexity of shortest path recovery is $O(w)$. The total expected cost for shortest path algorithm using VIP-Tree is then $O(\rho^2 + w)$.

3.4 Querying Indoor Objects

Indexing Indoor Objects. Given a set of objects O, we embed it with IP-Tree and VIP-Tree as follows. For each object $o \in O$ located in a partition P, we record a pointer to the leaf node of the tree that contains the partition P. Furthermore, for each access door d_i of a leaf node N, we maintain the list of objects located in N sorted on their distances from d_i. This allows efficient computation of distances from a given query point to the objects in a leaf node.

k **Nearest Neighbors (*k*NN) Queries**. Algorithm 5 presents the details of computing *k*NNs using our proposed index structures. It is a standard best-first search algorithm widely used on various branch and bound structures such as R-tree, Quad-tree etc.

Algorithm 5: *k* Nearest Neighbors

```
Input  : q: query point, k
Output : kNNs
1  d^k = ∞;          /* d^k is distance to current k^th NN */;
2  getDistances(q,root);           /* Algorithm 2 */;
3  Initialize a heap H with root of the tree;
4  while H is not empty do
5      de-heap a node N from heap;
6      if mindist(q, e) > d^k then
7          return kNN;
8      if N is a non-leaf node then
9          for each child N' of N do
10             if N' contains objects then
11                 insert N' in heap with mindist(q, N');
12     else
13         Use objects in N to update kNN and d^k;
```

The algorithm requires computing $mindist(q, N)$ for different nodes in the tree. $mindist(q, N)$ is the minimum distance from the query q to any point in the node N. $mindist(q, N)$ is zero if q is in a partition contained in the sub-tree of the node N. If N does not contain q, then $mindist(q, N)$ is the minimum distance from q to an access door of the node N, i.e., $mindist(q, N) = min_{\forall d \in AD(N)} dist(q, d)$. A straightforward way to compute $mindist(q, N)$ is to use Algorithm 3. Next, we show that we could optimize $mindist(q, N)$ for branch and bound algorithms because these algorithms access the nodes in a particular order.

Lemma 8. Let N_1 and N_2 be the two sibling nodes. If N_1 contains q then $dist(q, d_i)$ for any access door $d_i \in AD(N_2)$ is $min_{\forall d_j \in AD(N_1)} dist(q, d_j) + dist(d_i, d_j)$.

PROOF. Note that the only common points between two sibling nodes may be the common access doors. If q is located at a common access door d_i then the proof is obvious. If q is not located at a common access door then it must be located outside N_2 (because q is inside N_1). Hence, the shortest path from from q to any access door d_i of N_2 must pass through at least one access door of N_1. Hence, $dist(q, d_i) = min_{\forall d_j \in AD(N_1)} dist(q, d_j) + dist(d_i, d_j)$. □

Note that $dist(d_i, d_j)$ can be retrieved from the distance matrix of the parent node of N_1 and N_2.

Lemma 9. If N_1 does not contain q and N_2 is a child of N_1 then $dist(q, d_i)$ for any access door $d_i \in AD(N_2)$ is $min_{\forall d_j \in AD(N_1)} dist(q, d_j) + dist(d_i, d_j)$.

PROOF. Since q is outside N_1 and N_2 is inside N_1, the shortest path from q to a door $d_i \in N_2$ must pass through at least one access door d_j of N_1. Hence, $dist(q, d_i) = min_{\forall d_j \in AD(N_1)} dist(q, d_j) + dist(d_i, d_j)$. □

Note that if $dist(q, d_j)$ for every access door d_j of N_1 is already known, then $mindist(q, N_2)$ can be computed in $O(\rho^2)$ using Lemma 8 or Lemma 9. Below are the details.

At line 2 of Algorithm 5, we compute distance from q to each access door of the root node by calling Algorithm 2. Note that Algorithm 2 computes distances from q to all access doors of each ancestor node of `Leaf(q)` in the process. We maintain these distances for each ancestor node of `Leaf(q)`. Now, when a child node N' of a node N is to be inserted in the heap at line 11, $mindist(q, N')$ can be computed using either Lemma 8 or Lemma 8.

Specifically, if N contains q then this implies that at least one sibling N_{sib} of N' contains q. Since we already know distances from q to every access door of N_{sib} (because it is an ancestor of `Leaf(q)`), Lemma 8 can be applied to compute $mindist(q, N')$. On the other hand, if N does not contain q then Lemma 9 is applied. Hence, $mindist(q, N')$ can be easily computed in $O(\rho^2)$ for each node accessed by the algorithm.

Range Queries Given a range r, a range query returns every object $o \in O$ for which $dist(q, o) \leq r$. The algorithm to process range queries is very similar to Algorithm 5 except that d^k is set to r and all objects in a node N are returned if the furthest object in the node is within the range r. We omit the details due to the space limitations.

4. EXPERIMENTS

4.1 Experimental Settings

Indoor Space. We use three real data sets: Melbourne Central [4], Menzies building [5] and Clayton Campus [6]. Melbourne Central is a major shopping centre in Melbourne and consists of 297 rooms spread over 7 levels (including ground and lower ground levels). Menzies building is the tallest building at Clayton campus of Monash University consisting of 14 levels (including basement and ground floor) and 1306 rooms. The Clayton data set corresponds to 71 buildings (including multilevel car parks) in Clayton campus of Monash University. We obtained the floor plans of all buildings and manually converted them to machine readable indoor venues. Coordinates of the buildings are obtained by using OpenStreetMap and the sizes of indoor partitions (e.g., rooms, hallways) are determined. A three dimensional coordinate system is used where the first two represent x and y coordinates of indoor entities (e.g., rooms, doors) and the third represents the floor number. For Clayton data set, the D2D graph also contains edges between the entry/exit doors of different buildings where the weight corresponds to the outdoor distance between the doors.

To evaluate the algorithms on even larger data sets, we extend Melbourne Central (denoted as MC), Menzies building (denoted as Men) and Clayton (denoted as CL) by replication. Table 2 gives details of the real indoor venues and the larger replicated venues. For example, MC-2 indicates that a replica of Melbourne Central is placed on top of the original building. CL-2 denotes that each building in the Clayton campus has been replicated to increase its size by two. The replicas are connected with the original buildings by stairs. The number of edges shown in Table 2 corresponds to the total number of edges in the D2D graph for each indoor space. The distance matrix used by the state-of-the-art indoor technique cannot be built on the venues larger than Men-2.

Datasets	Description	# doors	# rooms	# edges
MC	Melbourne Central	299	297	8,466
MC-2	2 times MC	600	597	16,933
Men	Menzies building	1,368	1,306	56,035
Men-2	2 times Men	2,738	2,613	112,114
CL	Clayton Campus	41,392	41,100	6,700,272
CL-2	2 times CL	83,138	82,540	13,400,884

Table 2: Indoor venues used in experiments

Competitors. All algorithms are implemented in C++ on a PC with 8GB RAM and Intel Core I5 CPU running 64-bit Ubuntu. We compare our proposed indexes (IP-Tree and VIP-Tree) with the following competitors.

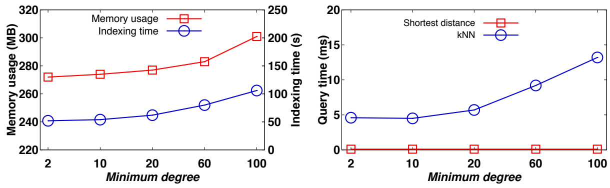
Distance Matrix (DistMx). As described earlier, the shortest distance and shortest path queries can be efficiently computed using a distance matrix that materializes distances between all pairs of doors in the space.

Distance-aware model (DistAw) [21]. We also compare our algorithm with the state-of-the-art indoor query processing index called distance-aware model (shown as DistAw). For shortest distance/path queries, DistAw uses only the accessibility base graph and D2D graph. For kNN and range queries, DistAw model also proposes to use DistMx to speed up the query processing. In the experiments, we use DistAw++ to denote the algorithm that exploits DistMx (requiring an additional $O(D^2)$ space). We use DistAw to denote the algorithm that does not required DistMx.

ROAD [19] and G-tree [30]. We also compare our algorithm with the state-of-the-art indexes for spatial query processing on road networks (G-tree and ROAD). These indices are constructed by passing the D2D graph as input and the query processing algorithms are adapted to suit indoor query processing. For each indoor venue, we experimentally choose the best value for the parameter τ used by G-tree.

Queries and Objects. To evaluate the performance for shortest distance/path queries, $10,000$ pairs of source and target points are randomly generated in the indoor space. To evaluate kNN and range queries, $10,000$ query points are randomly generated in the indoor space. We use washrooms in the buildings as the objects (e.g., the query is to find the nearest washroom). The number of washrooms in Men-2 is 50. We also generate synthetic object sets consisting of 10, 50, 100 and 500 objects - 50 is the default value. We choose a small set of objects because the kNN queries are more challenging for smaller object sets (as also reported in existing work on road networks [9]). This is because a larger area is to be explored to compute the kNNs when the number of objects is small. Furthermore, we believe that the real world scenarios for kNN queries contain a small number of objects, e.g., ATM machines, washrooms, charging-kiosks etc. k is varied from 1 to 10 and the default value of k is 5. The range is varied from 50 to 1000 meters and the default value is 100 meters. The figures report average query processing cost for each algorithm.

Choosing t for IP-Tree and VIP-Tree. We evaluated the effect of the minimum degree t (see Algorithm 1) on our indexes and found that the best performance is achieved for $t = 2$. Fig. 7 shows the index construction cost and query time of VIP-tree on Clayton data set. The construction time and construction cost increases as



(a) Construction cost (b) Querying cost

Figure 7: Effect of minimum degree t on VIP-Tree

t increases mainly because the size of distance matrices increases which requires more storage and more computation time to materialize the distances. The size of t does not affect the query time for shortest distance queries mainly because the cost is independent of the height of the tree – recall that VIP-tree computes shortest distance in $O(\rho^2)$ and ρ is not affected by t. The cost of kNN query increases with t mainly because fewer nodes can be pruned when t is large which requires the algorithm to access a larger number of nodes. The trend for IP-tree are similar. In the rest of the experiments, we use $t = 2$ for our indices. Although the results are not shown due to the space limitations, we also found that the average number of access doors and superior doors is less than 4 for all data sets and the maximum number is around 8. This provides an insight on why our indices perform exceptionally well for indoor spaces.

4.2 Indexing Cost

Construction time. Fig. 8(a) compares the time it takes to construct each index using the accessibility base graph and D2D graph. Since DistAw only uses the accessibility base graph and D2D graph, its index construction is not shown. Note that DistAw++ does use DistMx and its construction cost is the same as DistMx. To construct DistMx, for each door, we use a Dijkstra's like expansion until all other doors in the graph have been marked. This requires $O(D)$ expansions on the D2D graph which is quite expensive. Consequently, DistMx has a very high construction cost and it took almost 14 hours to construct DistMx for Men-2 consisting of $2,738$ doors requiring computing almost 7.5 million shortest distances/paths.

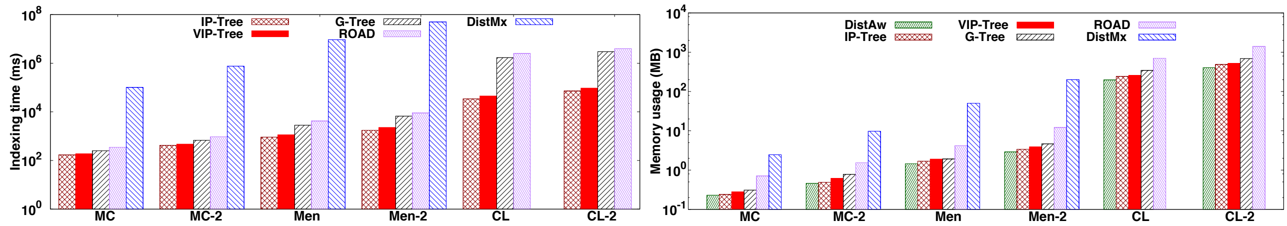
The construction cost for IP-Tree and VIP-Tree is less than 90 seconds even for the largest data set (CL-2) that consists of more than $83,000$ doors and around 13.4 Million edges in the D2D graph. As expected VIP-Tree takes more time than IP-Tree because it needs to compute and store the distances between each door d_i to every access door in the ancestor nodes of d_i. G-tree and ROAD take around one hour to build the index for CL-2 data set.

Index size. Fig. 8(b) compares the size of different indexes. As expected, DistMx is the largest index. DistAw has the smallest index size because it only needs the accessibility base graph and D2D graph. IP-Tree, VIP-Tree and G-tree have sizes comparable to DistAw index. The storage cost of VIP-Tree is slightly higher than IP-Tree which demonstrates that materializing the distances to the access doors of all ancestors nodes does not increase the storage cost dramatically but significantly improves the query processing cost as we show later. G-tree and ROAD consume more space than IP-Tree and VIP-Tree mainly because these were designed for road networks having a small average outdegree (2 to 4) as compared to the D2D graph which has a much higher out-degree (up to 400). This results in a larger number of *border* nodes and hence consuming more space.

4.3 Query Performance

4.3.1 *Shortest distance queries*

In Fig. 9 we evaluate the algorithms for shortest distance queries on different indoor data sets. First, we present a simple optimization to improve the performance of DistMx. A straightforward ap-

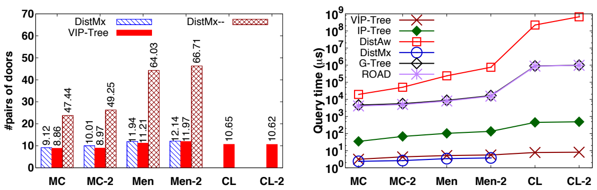


(a) Construction time (b) Index Size

Figure 8: Indexing Cost

proach to compute the distance from s to t is to use DistMx to calculate distances between every door d_i in `Partition(s)` and every door d_j in `Partition(t)` and picking the pair d_i and d_j that minimizes $dist(s,d_i) + dist(d_i,d_j) + dist(d_j,t)$. Let D_s and D_t be the number of doors in `Partition(s)` and `Partition(t)`, respectively. This requires checking $D_s \times D_t$ pairs of doors to retrieve the shortest distance and the cost may be high if $D_s \times D_t$ is large. A simple optimization is to ignore the doors in `Partition(s)` and `Partition(t)` that lead to no-through partitions.

The above optimization significantly reduces the pairs of doors that need to be considered. Fig. 9(a) shows the effect of this optimization where DistMx uses this optimization and DistMx- - does not use this optimization. The numbers on top of bars correspond to the number of pairs needed to be considered by each algorithm. As can be seen, this simple optimization significantly reduces the number of pairs and improves the performance of DistMx by up to several times. In the rest of the experiments, we use this optimization for DistMx. The numbers for VIP-Tree correspond to the pair of superior doors to be considered. This number is slightly smaller than the number of pairs considered by DistMx but the cost is slightly higher because VIP-Tree needs to first compute distances from s and t to the access doors of the children of lowest common ancestor which requires more computation.



(a) Optimizing DistMx (b) Comparing all algorithms

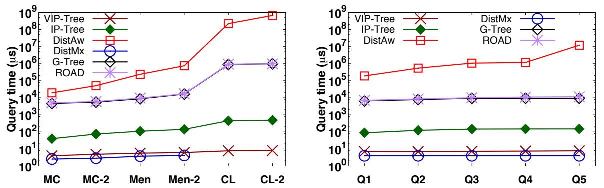
Figure 9: Shortest Distance Queries

Fig. 9(b) compares the performance of all techniques for shortest distance queries. Since DistMx returns distance between any two doors in the graph in $O(1)$, it gives the best performance. However, VIP-Tree provides a comparable performance. Note that DistMx has quadratic storage cost and huge construction cost. Recall that we were not able to construct DistMx for indoor venues larger than Men-2. VIP-Tree significantly outperforms IP-Tree at the expense of a slightly higher storage cost. Both VIP-Tree and IP-Tree outperform the other three techniques by several order of magnitude, e.g., for CL-2 data set, VIP-Tree processes a shortest distance query in around 10 microseconds as compared to ROAD and G-tree that take almost one second to answer a single shortest path query.

4.3.2 *Shortest path queries*

Fig. 10 compares the techniques for shortest path queries. We note that the overhead of recovering shortest paths is negligible, i.e., for each algorithm, the cost of shortest distance queries is similar to the cost of shortest path queries (compare Fig. 9(b) and Fig. 10(a)).

Next, we evaluate the effect of the distance between s and t on the performance of different algorithms for the shortest path queries. We use Men-2 to demonstrate the results because this is the largest data set for which DistMx works. Let d_{max} be the maximum distance between any two points in Men-2 building. We divide the distance range $[0, d_{max}]$ into five intervals ($Q1$ to $Q5$) of equal length $l = d_{max}/5$, e.g., $Q1 = [0, l]$, $Q2 = [l, 2l], \ldots, Q5 = [4l, 5l]$. We then randomly generate source and target points and allocate them to relevant Qi based on the distances between them. Hence, the pairs of source and target points corresponding to $Q1$ have the smallest distances (within range $[0, l]$) and the pairs in $Q5$ have largest distances $[4l, 5l]$.



(a) Different buildings (b) Effect of distance b/w s and t

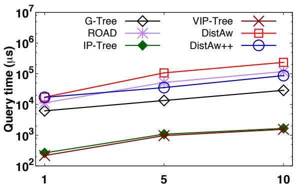
Figure 10: Shortest Path Queries

Fig. 10(b) shows the effect of distances on the performance of different algorithms. The cost of DistAw increases by almost two orders of magnitude as the distance increases. The cost for IP-Tree slightly increases from $Q1$ to $Q3$ because the lowest common ancestor is at a higher level when source and target are further from each other. This requires visiting more levels of the tree resulting in an increased cost. However, the cost does not increase further for $Q4$ and $Q5$ because, in most of the cases for $Q3$, the lowest common ancestor is already the root node. A similar behavior can be observed for G-tree and ROAD. The effect of distance is negligible on DistMx and VIP-Tree because these algorithms require retrieving relevant entries from the distance matrices which is independent on the distances between the source and target points. A similar trend was observed for shortest distance queries.

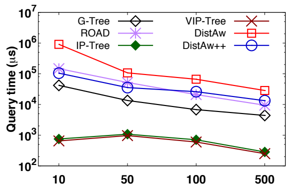
4.3.3 *Querying Indoor Objects*

***k*NN Queries**. Fig. 11(a), Fig. 11(b) and Fig. 11(c) evaluate different algorithms by varying k, the number of objects, and the indoor buildings, respectively. VIP-Tree and IP-Tree perform equally well. This is because IP-tree computes $mindist(q,N)$ for a node N with the same complexity as that of VIP-Tree due to the optimizations presented in Section 3.4. Both VIP-Tree and IP-Tree outperform the other algorithms by several orders of magnitude. Note that DistAw++ is the existing method that utilizes DistMx to speed up the query processing. Nevertheless, it is outperformed by our proposed techniques.

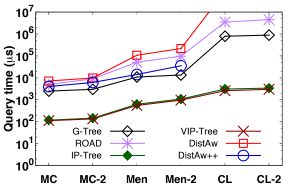
Fig. 11(b) shows that the cost of all algorithms decreases as the number of objects increases. This is because kNNs can be found closer to the query point as the number of objects increases. Hence, the algorithms require exploring a smaller area. On the other hand, the query processing cost increases for all algorithms as the value of k or the data set size increases.



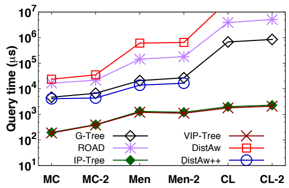
(a) Effect of k (kNN query)



(b) # of objects (kNN query)



(c) Indoor venues (kNN query)



(d) Indoor venues (range query)

Figure 11: kNN and Range Queries

Range Queries. Fig. 11(d) evaluates the performance of different techniques for range queries. The cost of all algorithms increases with for larger venues mainly because the sizes of the indexes increase. VIP-Tree and IP-Tree both perform equally well and outperform the other competitors by several orders of magnitude.

5. RELATED WORK

Data modelling for indoor space is fundamental for querying indoor space. In [18], a 3D model is proposed for indoor space but it fails to support indoor distance computations. CityGML [7] and IndoorGML [8] are XML based methods to model and exchange the indoor space. As stated in Section 1, the distance-aware model [21] introduced an accessibility base graph and D2D graph that enable indoor distance computations between two indoor positions.

Indoor positioning data received from RFID is cleaned using spatio-temporal constraints. Graph based methods [10] take advantages of indoor constraints to fix cross and missing readings in the raw RFID data. These constraints are also applied to construct probabilistic trajectories [13] from raw RFID data.

RTR-tree and TP^{2}R-tree [16] are two indoor structures extended from R-tree which index trajectories of indoor moving objects. In terms of indoor partitions like rooms and hallways, *ind*R-tree [24] constructs a composite index that indexes indoor entities into different layers with indoor moving objects stored in the leaf level. For querying indoor data, shortest distance/path, kNN and range queries are studied under various settings [22, 25, 26]. The most notable techniques [21, 28] have already been discussed in Section 1 (e.g., D2D graph, AB graph and distance matrix).

Since an indoor space can be converted into a D2D graph, techniques in spatial road networks can also be applied. G-tree [30, 29] is the state-of-the-art technique for outdoor query processing. Although our proposed indexes are inspired by G-tree, there are some fundamental differences as our indexes carefully exploit the properties specific to indoor space. Specifically, G-tree uses an existing multilevel graph partitioning algorithm [17] for graph decomposition whereas we design a new algorithm that carefully exploits the properties of the indoor space to minimize the total number of access doors. Also, the smaller number of access doors in our nodes allows us to use materialization in the VIP-tree which proves to be a much more efficient strategy but is not feasible for G-tree. Furthermore, our algorithms to process shortest path queries, range queries and kNN queries are also entirely different

6. CONCLUSION

In this paper, we propose two novel indexes, IP-Tree and VIP-Tree, for efficiently processing indoor spatial queries. We also present efficient algorithms to answer shortest path queries, shortest distance queries, k nearest neighbors queries and range queries. IP-Tree and VIP-Tree have low storage requirement, small preprocessing cost and are highly efficient. Our extensive experimental study on real and synthetic data sets demonstrates that the proposed indexes outperform the existing techniques by several orders of magnitude.

7. REFERENCES

[1] https://www.abiresearch.com/press/over-800-million-smartphones-using-indoor-location.

[2] forbes.com/sites/forrester/2013/01/23/indoor-venues-are-the-next-frontier-for-location-based-services.

[3] http://venturebeat.com/2012/12/31/trends/.

[4] http://www.melbournecentral.com.au/.

[5] http://lostoncampus.com.au/15641.

[6] https://www.monash.edu/pubs/maps/3-Claytoncolour.pdf.

[7] http://www.citygml.org/.

[8] http://www.opengeospatial.org/projects/groups/indoorgmlswg.

[9] T. Abeywickrama, M. A. Cheema, and D. Taniar. k-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *PVLDB*, 2016.

[10] A. I. Baba, H. Lu, T. B. Pedersen, and X. Xie. Handling false negatives in indoor RFID data. In *MDM*, pages 117–126, 2014.

[11] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. In *PVLDB*, 2013.

[12] Z. Fang, Q. Li, X. Zhang, and S. Shaw. A GIS data model for landmark-based pedestrian navigation. *International Journal of Geographical Information Science*, 26(5):817–838, 2012.

[13] B. Fazzinga, S. Flesca, F. Furfaro, and F. Parisi. Cleaning trajectory data of rfid-monitored objects through conditioning under integrity constraints. In *EDBT*, pages 379–390, 2014.

[14] H. Hile, R. Grzeszczuk, A. L. Liu, R. Vedantham, J. Kosecka, and G. Borriello. Landmark-based pedestrian navigation with enhanced spatial reasoning. In *Pervasive Computing*, pages 59–76, 2009.

[15] P. L. Jenkins, T. J. Phillips, E. J. Mulberg, and S. P. Hui. Activity patterns of Californians: Use of and proximity to indoor pollutant sources. *Atmospheric Environment. Part A. General Topics*, 1992.

[16] C. S. Jensen, H. Lu, and B. Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *SSTD*, 2009.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998.

[18] J. Lee. A spatial access-oriented implementation of a 3-d gis topological data model for urban entities. *GeoInformatica*, 2004.

[19] K. C. K. Lee, W. Lee, B. Zheng, and Y. Tian. ROAD: A new spatial object search framework for road networks. *IEEE TKDE*, 2012.

[20] F. Li, C. Zhao, G. Ding, J. Gong, C. Liu, and F. Zhao. A reliable and accurate indoor localization method using phone inertial sensors. In *UbiComp*, 2012.

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

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

[23] M. Werner. *Indoor Location-Based Services: Prerequisites and Foundations*. Springer, 2014.

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

[25] X. Xie, H. Lu, and T. B. Pedersen. Distance-aware join for indoor moving objects. *IEEE TKDE*, 2015.

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

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

[28] W. Yuan and M. Schneider. Supporting continuous range queries in indoor space. In *MDM*, 2010.

[29] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE TKDE*, 27(8):2175–2189, Aug 2015.

[30] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *CIKM*, 2013.