Efficiently Processing Spatial and Keyword Queries in Indoor Venues

Zhou Shao, Muhammad Aamir Cheema, David Taniar, Hua Lu, and Shiyu Yang

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 first 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 are also the first to study spatial keyword queries in indoor venues. We propose a novel data structure called Keyword Partitioning Tree (KP-Tree) that indexes objects in an indoor partition. We propose an efficient algorithm based on VIP-Tree and KP-Trees to efficiently answer spatial keyword queries. Our extensive experimental study on real and synthetic data sets demonstrates that our proposed indexes outperform the existing solutions by several orders of magnitude.

Index Terms—Indoor query processing, Indoor index, Spatial keyword query

1 Introduction

1.1 Motivation

Due to the recent breakthroughs in indoor positioning technologies (see [17], and its references), and the widespread use of smart phones, indoor location-based services (LBSs) are becoming increasingly popular [5]. 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 nearby 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.

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.

One possible approach for indoor data management is to first model the indoor space to a graph using existing indoor data modelling techniques [2], [19] 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 [16] and G-tree [37] 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 [4]. Our experimental study shows that these outdoor techniques take around 1 second to answer a single indoor shortest distance query, in contrast, our specialized techniques that carefully exploit the properties of indoor space can process the same query in around 10 microseconds.

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 [33] and/or accessibility base graph [19] to process various indoor spatial queries.

Door-to-door (D2D) graph [33]. 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 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

- *Zhou Shao, Muhammad Aamir Cheema and David Taniar are at Faculty of Information Technology, Monash University, Australia E-mail: {joe.shao, aamir.cheema, david.taniar}@monash.edu*
- *Hua Lu is at Department of Computer Science, Aalborg University, Denmark. E-mail: luhua@cs.aau.dk*
- *Shiyu Yang is at School of Software Engineering, East China Normal University, China. E-mail: syyang@sei.ecnu.edu.cn*

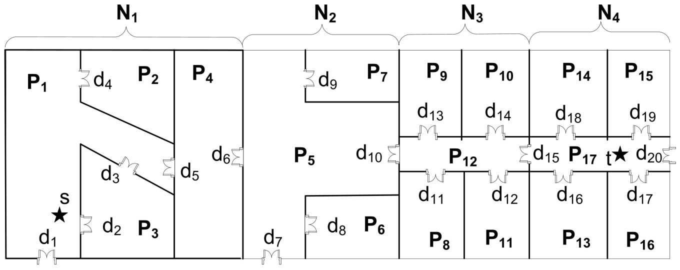


Fig. 1: An indoor venue containing 17 partitions and 20 doors

(P_1 to P_{17}) and 20 doors (d_1 to d_{20}). The corresponding D2D graph is shown in Fig. 2a 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 [19]. 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. 2b 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 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.

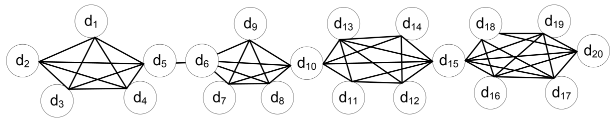
Distance matrix (DM) [19]. A distance matrix can also be used to facilitate shortest distance/path queries. A distance matrix stores the distances between all pairs of doors in the indoor space. Although this allows optimally retrieving the distance between any two doors (i.e., in $O(1)$), it requires 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 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 [19] 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 [19], 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 (whereas our proposed technique processes the same query in around 10 microseconds).

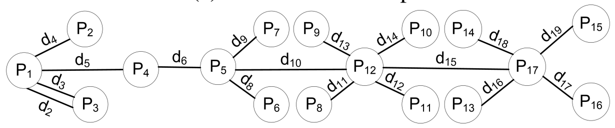
1.3 Contributions

To handle fundamental spatial queries, 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 [19] and road networks [16], [37] by several orders of magnitude. In comparison



(a) Door-to-Door Graph



(b) Accessibility Base Graph

Fig. 2: Indexing Indoor Space

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 (and 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. A detailed theoretical analysis is provided in the conference version [25] of this paper.

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*

	Storage	**Shortest Distance**	**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	$O(D^2)$	$O(\rho^2)$	$O(\rho^2 + w)$

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 such as spatial keyword queries. This paper is an extended version of our earlier paper [25] and, in this paper, we show that the VIP-Tree can be easily extended to answer spatial keyword queries on objects in indoor venues that are associated with textual information. We also propose a novel data structure called Keyword Partitioning Tree (KP-Tree) to effectively index the objects in a single indoor partition (e.g., products in a supermarket, medicines in a pharmacy etc.). Our experimental study on real world data sets demonstrates that the technique that utilizes VIP-Tree and KP-Trees significantly outperforms the competitors.

The rest of the paper is organized as follows. In Section 2, we present the related work. Our proposed indexes IP-Tree and VIP-Tree are discussed in Section 3. Section 4 presents the detailed algorithms for indoor spatial queries based on the proposed indexes. Section 5 presents the details of our proposed Keyword Partitioning Tree and spatial keyword query processing algorithm. A detailed experimental evaluation is presented in Section 6 followed by conclusions in Section 7.

2 Related Work

Indoor Data Modelling Data modelling for indoor space is fundamental for querying indoor space. In [15], a 3D model is proposed for indoor space but it fails to support indoor distance computations. CityGML [1] and IndoorGML [2] are XML based methods to model and exchange the indoor space data. As stated in Section 1, the distance-aware model [19] introduces an extended graph based on an accessibility base graph and D2D graph that enables indoor distance computations between two indoor positions.

Spatial Queries in Indoor Space. Several indoor spatial queries such as shortest distance queries, shortest path queries, kNN queries and range queries have been studied under various settings [20], [31], [32], [35]. The most notable techniques [19], [33] have already been discussed in Section 1. Existing research also studies various other indoor queries such as trip planning queries and multi-criteria route planning queries [23], [24]. Indexing and querying moving indoor objects have also been studied in the past [11], [30].

Spatial Queries in Outdoor Space. Query processing in Euclidean space and road networks [6], [10], [26], [34] is very well studied. Since an indoor space can be converted into a D2D graph, various techniques [27], [36], [37] in spatial road networks can also be applied. G-tree [36], [37] is the state-of-the-art technique for processing a variety of spatial queries on road networks. Although our proposed indexes, IP-Tree and VIP-Tree, are inspired by G-tree, there are some fundamental differences. Specifically, G-tree uses an existing multilevel graph partitioning algorithm [14] 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, kNN queries and spatial keyword queries are also entirely different.

Spatial Keyword Queries in Outdoor Space. Spatial keyword queries have been extensively studied in Euclidean space [18], [22], [29]. For example, the Inverted R-tree [38] is proposed that creates, for each keyword t, an R-tree based on the objects that contains the keyword t. Inverted R-trees are efficient when the number of query keywords is small because the query needs to process only a few R-trees. Information retrieval R-tree (IR^2-tree) [9] aims to address this problem by utilizing *signatures* that summarize the keywords contained in the descendent entries of a node. Information R-tree (IR-tree) [8], [28] utilizes inverted files for each node that maintains the keywords information in the node. WIR-tree [29] is similar to IR-tree, but it partitions the objects according to keyword frequencies instead of spatial locations. A detailed experimental evaluation comparing different spatial keyword approaches is presented in [7].

Various spatial keyword queries have also been studied on road networks [12], [18], [21], [36]. For example, DESKS [18] considers the direction constraints for keyword queries, while [12] is designed for large graphs. G-tree [36] has also been utilized to answer spatial keyword queries. Recently, in [3], the authors propose a framework, K-SPIN, to answer a variety of spatial keyword queries on road networks. To the best of our knowledge, we are the first to focus on spatial keyword queries in indoor space.

3 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 [33] 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 [1] data objects can be embedded in each partition. Although our techniques can be immediately applied for directional entities (e.g., doors/lifts/stairs), for the sake of simplicity, we assume that all such entities are bidirectional.

3.1 Indoor Partitioning Tree (IP-Tree)

3.1.1 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 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.

Definition 3.1. 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 from/to 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.

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.

	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

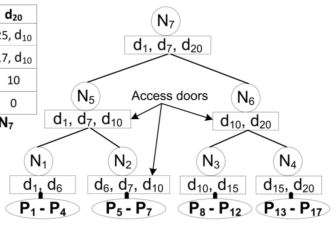


Fig. 3: Indoor Partitioning Tree

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 and correctly retrieve the shortest path between two points (see [25] for details). 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 the 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.

3.1.2 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 (i.e., root 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 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 is not allowed if it will result in a leaf node having more than one hallways. This is because the shortest distance/path queries between points in different hallways are 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 3.1. 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. 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 the 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 a 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 preference 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 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
    $N_i.degree = 1$;
2 while $H.top().degree < t$ do
3     deheap a node $N_i$ from $H$;
4     $N_j \leftarrow$ node with highest # of common doors with $N_i$;
5     remove $N_j$ from $H$ and merge $N_i$ and $N_j$ to form $N_k$;
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-heaped 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 at least t (line 2). This implies that every node in the heap contains 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 settled. 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 3.2. 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. 2a and expand the search until all doors d_1 to d_6 are settled. 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. 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 are 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.

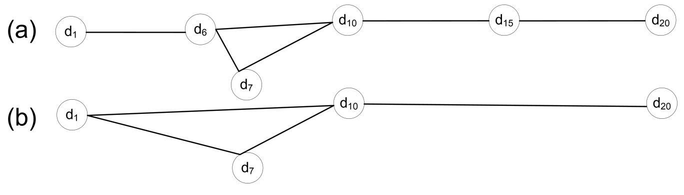


Fig. 4: (a) $\mathcal{G}_2$: level-2 graph; (b) $\mathcal{G}_3$: level-3 graph

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 3.3. 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.

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. The total space complexity of the IP-Tree is $O(\rho^2 f^2 M + \rho D)$ where D is the total number of doors in the indoor space, ρ is the average number of access doors in a node, f is the average number of children for a non-leaf node and M is the total number of leaf nodes which is bounded by P (the total number of partitions in the indoor space). Our experiments on three real data sets demonstrate that f and ρ are small in practice (less than 4 for all real data sets). For a detailed theoretical analysis, please see [25].

3.2 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 as 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 shown in [25], the space complexity of VIP-Tree is $O(\rho^2 f^2 M + \rho D \log_f M)$.

4 Spatial Query Processing

Our proposed indexes, IP-Tree and VIP-Tree, can be used to efficiently answer various spatial queries such as shortest distance, shortest path, kNN and range queries. Since the focus of this extended paper is on processing spatial keyword queries (Section 5), due to the space limitation, this section only presents how to answer shortest distance queries using VIP-Tree. The interested readers are referred to the conference version [25] for the details of other spatial queries mentioned above.

Now, we present our algorithm to compute the indoor shortest distance $dist(s, t)$ between a source point s and a target point t using VIP-Tree. When both s and t are located in the same leaf node, $dist(s, t)$ can be computed using D2D graph (similar to the existing approaches). Since s and t are close to each other in D2D graph, the distance computation using Dijkstra's like expansion is not expensive in this case. Next, we show how to compute $dist(s, t)$ when both s and t are in different leaf nodes.

Let `Leaf(p)` denote the leaf node of the VIP-Tree that contains the point p. The following lemma is a key to compute $dist(s, t)$ for two arbitrary points s and t located in different leaf nodes `Leaf(s)` and `Leaf(t)`.

Lemma 4.1. 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.

$$\min_{\forall d_i \in AD(N_s), \forall d_j \in AD(N_t)} dist(s,d_i) + dist(d_i,d_j) + dist(d_j,t) \tag{1}$$

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 $d_i \in AD(N_s)$ can also be efficiently computed because VIP-tree materializes distances between each door d_s of the partition containing s and each $d_i \in AD(N_s)$. Similarly, $dist(d_j,t)$ can also be efficiently computed using VIP-Tree. We further improve the computation cost of $dist(s,d_i)$ and $dist(d_j,t)$ using only the *superior doors* [25] of the partitions containing s and t. The interested readers are referred to [25] for the details of how superior doors are used to efficiently compute the distances and a detailed theoretical analysis of our shortest distance algorithms.

5 Spatial Keyword Query Processing

Spatial keyword queries have been extensively studied in outdoor space. Arguably, the two most popular and well studied spatial keyword queries are Boolean kNN queries and Top-k kNN queries (see [17] for details). To the best of our knowledge, we are the first to study spatial keyword queries in indoor space. In this paper, we focus on processing *indoor boolean kNN spatial keyword (iBkNN-SK) queries*. However, our proposed ideas can be extended to answer a variety of other spatial keyword queries.

5.1 Problem Definition

We represent an indoor *spatio-textual* object o as a spatial point located in an indoor venue and a set of keywords (terms) from a vocabulary $\mathcal{V}$, represented by $o.loc$ and $o.\mathcal{T}$, respectively. We often use o to refer to $o.loc$ when there is no ambiguity.

Definition 5.1. iBkNN-SK Query. Given a set $\mathcal{O}$ of *spatio-textual* objects, a query object q where $q.loc$ is its indoor location and $q.\mathcal{T}$ is the set of query keywords, an iBkNN-SK is to find k closest objects to $q.loc$ that contain every keyword in $q.\mathcal{T}$.

Example 5.1. Take Fig. 5 as an example that shows a set of *spatio-textual* objects $\mathcal{O} = \{o_1, o_2, ..., o_{12}\}$ in an indoor venue. Assume that a user located at query point q wants to find the nearest object (i.e., k=1) which contains keywords t_1 and t_2 ($q.\mathcal{T}$=$\{t_1,t_2\}$). The object o_5 is returned as the result because it is the closest object to q containing both t_1 and t_2.

5.2 Some possible solutions

In this section, we briefly discuss how to extend existing techniques to answer spatial keyword queries in indoor environment.

Extending Distance-aware Model (DistAw) [19]. The distance aware model is the state-of-the-art algorithm for indoor query

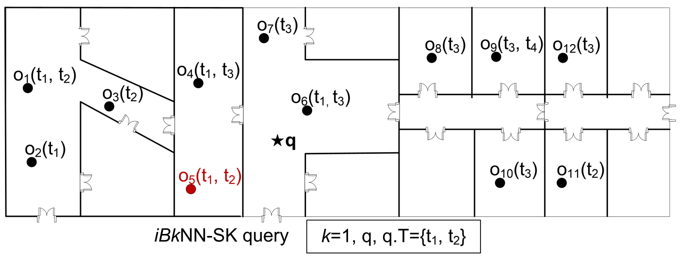


Fig. 5: Example of iBoolean-kNN Query

processing. To solve iBkNN queries, we embed the keyword information with each indoor partition. Specifically, for each indoor partition containing at least one object, the keyword set of the partition is the union of the keywords of the objects in the partition. During search process, DistAw uses the accessibility base graph and the keyword set for each partition to prune the un-necessary partitions.

DistAw++. In DistAw, indoor distances are computed during the expansion process. To improve the efficiency, the authors proposed to utilize a distance matrix that materializes indoor distances between all pairs of doors. This results in significant improvements in query processing time at the cost of quadratic storage requirement. We use DistAw++ to indicate a version of DistAw that uses a distance matrix to accelerate the query processing.

Extending G-tree [36]. As discussed earlier, to adopt outdoor techniques like G-tree, the indoor space is converted into a D2D graph. G-tree index is then built on this D2D graph. An inverted list is added for each node of the G-tree to efficiently prune unnecessary nodes during the query processing.

Extending VIP-Tree. We extend VIP-Tree by adding an inverted list for each node of the VIP-Tree in a way similar to existing spatial keyword indexes for outdoor techniques such as IR-tree [8], [28], i.e., for each VIP-Tree node, we store a set of all keywords in its sub-tree. The modified VIP-Tree is called *inverted VIP-Tree* (IVIP-Tree).

Our experimental results (see Fig. 11 in Section 6.2.2) show that IVIP-tree is up to an order of magnitude better than all other approaches discussed above. This shows the effectiveness and adaptability of VIP-tree for different settings.

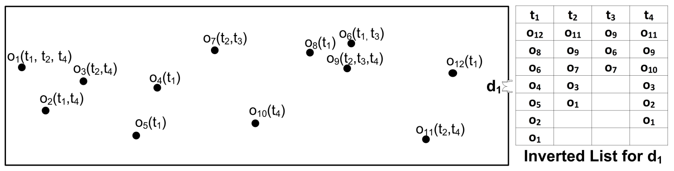
Next, we propose another novel data structure called Keyword-Partitioning tree (KP-tree) specifically designed to handle spatio-textual objects in indoor partitions such as products in supermarkets and books in a library etc. A KP-tree is created for each indoor partition and allows efficient retrieval of relevant objects when the search reaches a particular partition. Each indoor partition in the IVIP-tree is linked to its KP-tree. Our experimental study shows that the KP-Trees further improve the performance of IVIP-Tree by up to an order of magnitude.

5.3 Keyword-Partitioning Tree (KP-Tree)

5.3.1 Motivation

Recall that the existing techniques map an indoor venue to a graph where a node represents a door or a partition. The indoor graph is then traversed to answer the queries and when the search reaches a partition, the objects in it are retrieved to process the query. Typically, an indoor partition contains a reasonably large number of objects such as products in a supermarket, books in a library or medicines in a pharmacy etc. For example, in the real world data set that we use in our experimental study, a single JB Hi-Fi store (an entertainment retailer in Australia) contains around $30,000$ different products. To efficiently answer the queries, once the search reaches an indoor partition, specialized indexes should be employed to efficiently retrieve the relevant objects in the partition. One possible solution is to use one of the existing approaches (e.g., inverted lists, IR-tree) to index the spatio-textual objects in a partition. However, we note that these techniques have certain limitations for indoor datasets as explained next.

Inverted lists can be utilized to retrieve relevant objects in a partition. Specifically, for each door d_i of the indoor partition and for each keyword t_j, an inverted list is created which stores the objects containing t_j in ascending order of their distances from the door d_i. Fig. 6 shows an example where the indoor partition contains 12 objects and has only one door d_1. For each unique keyword (t_1 to t_4), an inverted list is created that stores the relevant objects in ascending order of their distances from d_1. These lists can be used to prune some irrelevant objects. Assume that a query q is located outside the partition where $q.\mathcal{T}$=$\{t_1, t_4\}$. Once the search reaches this partition, the inverted lists of t_1 or t_4 can be accessed to find the nearest objects containing both of the keywords. However, many objects in the inverted lists may not contain all query keywords resulting in sub par performance. For example, the closest object containing both keywords is o_2 but this object is located at the end of the inverted lists t_1 and t_4. In other words, the algorithm needs to access many irrelevant objects before finding the answer.



t_1	t_2	t_3	t_4
o_{12}	o_{11}	o_9	o_{11}
o_8	o_9	o_6	o_9
o_6	o_7	o_7	o_{10}
o_4	o_3		o_3
o_5	o_1		o_2
o_2			o_1
o_1			

Fig. 6: Inverted List

Another possible approach is to use spatial keyword indexes like IR-Tree [8] for each partition. These indexes typically group spatially close objects into nodes which are further hierarchically grouped into parent nodes until a root node is formed. Each node in the tree contains a summary of all keywords contained in the subtree rooted at this node. During query processing, a node may be pruned if its summary does not contain all query keywords. Since the objects are mainly grouped based on their spatial closeness, the keyword summaries may not be very useful in pruning. This is especially problematic for indoor venues where density of the objects is quite high (a small shelf may have hundreds of different products). In Fig. 6, assume that a node groups the objects o_6, o_8 and o_9. The keyword summary of this node would contain all unique keywords for the partition (i.e., t_1 to t_4) and, as a result, this node (and all of its ancestor nodes) lose pruning effectiveness. We also remark that, in real world data sets, the number of objects that satisfy query keywords is typically small. Therefore, techniques that can efficiently filter the objects satisfying keyword criteria are expected to perform better than the approaches that mainly focus on spatial filtering.

There exist some indexing techniques such as WIR-Tree [29] that aim to index objects based on the keywords similarity instead of spatial closeness of the objects. However, these techniques are adversely affected by an object that contains many keywords. Assume that there exists an object that contains all of the keywords. When this object is grouped with some other objects in a leaf node, the keyword summary of this leaf node (and each of its ancestor node) would contain all of the keywords thus losing the pruning ability for the whole branch.

For the sake of only this example, assume that o_1 in Fig. 6 contains all the keywords t_1, t_2, t_3 and t_4. Fig. 7(b) shows the corresponding WIR-tree. The object o_1 is grouped with o_2 in the node W_1 which contains all query keywords. Consequently, the node W_1 and all its ancestor nodes (W_3 and W_9) lose pruning ability, i.e., every query would need to traverse these three nodes.

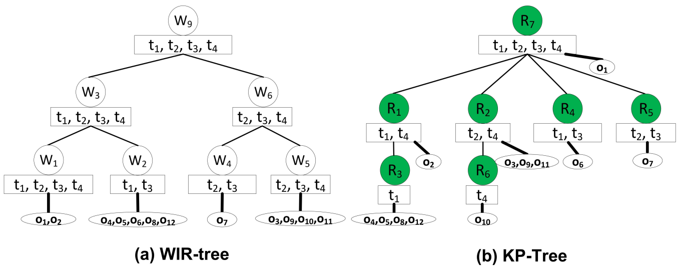


Fig. 7: WIR-tree and KP-Tree for the objects in Fig. 6 except that we assume o_1 contains all four keywords

In this paper, we propose a new index called *Keyword Partitioning Tree* (KP-Tree) to address the limitations described above. The proposed index has two distinct features that helps addressing the limitations: 1) objects are grouped mainly based on their keywords; and 2) unlike most of the existing indexes, objects in KP-Tree are not necessarily indexed at the leaf nodes. Instead, objects having more keywords are likely to be indexed at intermediate nodes higher in the tree structure which addresses the problem with indexes like WIR-Tree. For example, Fig. 7(a) shows KP-Tree for the same example for which WIR-Tree was shown. In the KP-Tree, the object o_1 is indexed at the root node, and as a result, its children nodes do not lose pruning capabilities. We present more details of KP-Tree in the next section. Note that, for the rest of the paper, we use the objects in Fig. 6 as represented and do not assume that o_1 contains all keywords.

5.3.2 Overview of KP-Tree

We give a brief overview of KP-Tree and some of its properties before formally describing its construction in next section. Fig. 8 is used as an example to illustrate KP-Tree for the objects in Fig. 6. Each node R in KP-Tree consists of a list of keywords represented as $R.\mathcal{T}$. Specifically, for every node R, $R.\mathcal{T}$ is the union of the keywords contained in its children. For example, $R_9.\mathcal{T} = R_7.\mathcal{T} \cup R_8.\mathcal{T} = \{t_1, t_2, t_3, t_4\}$. In KP-tree, each object o is attached with a node R if $o.\mathcal{T} = R.\mathcal{T}$. For example, the object o_1 is associated with the node R_2 because $R_2.\mathcal{T} = o_1.\mathcal{T} = \{t_1, t_2, t_4\}$. Similarly, the objects o_2 is associated with R_1 because it contains t_1 and t_4. Note that KP-Tree is different from most of the existing tree structures in the sense that the objects may be associated with non-leaf nodes. KP-Tree has two kinds of nodes: *fruitful nodes* (shaded nodes) and *fruitless nodes* (white nodes). A fruitful node is a node that has at least one object attached to it. In contrast, a fruitless node does not have any object attached to it. In Fig. 8, R_9 and R_8 are fruitless nodes and all other nodes are fruitful nodes.

A node in KP-Tree is also linked to its pre-computed object and node matrices. An object matrix records distance from each door d of the partition to each object o attached with the node, e.g., see the object matrix for R_3. A node matrix for a node R records the *minimum* distance from each door d to each child node R_i of R. The minimum distance $mindist(d, R_i)$ is the minimum distance from d to any object contained in the sub-tree rooted at R_i. Consider the node matrix for R_7 in Fig. 8. The minimum distance from d_1 to R_1 is 5 because the minimum distance from d_1 to the objects in the sub-tree rooted at R_1 (i.e., o_2, o_4, o_5, o_8 and o_{12}) is $dist(d_1, o_{12}) = 5$.

The query is processed in a traditional best-first manner using a heap that stores entries according to their minimum distances from q where distances are obtained utilizing the distance matrices. An entry e is pruned if $q.\mathcal{T} \not\subseteq e.\mathcal{T}$. For each node retrieved from the

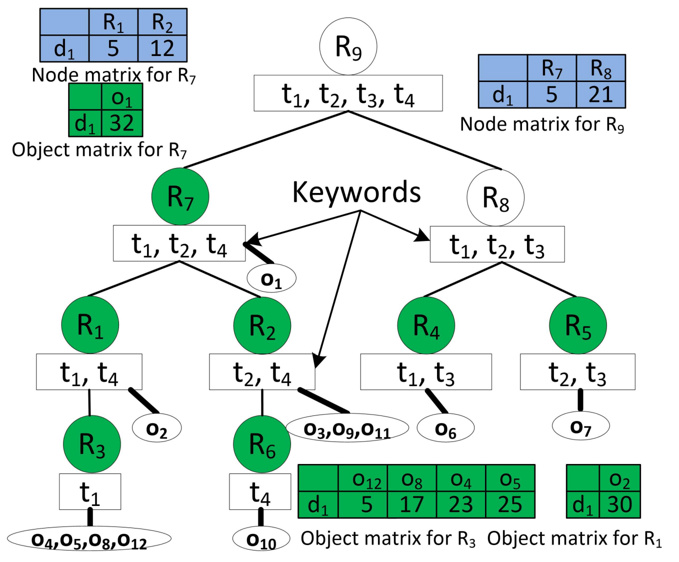


Fig. 8: KP-Tree for the objects in Fig. 6

heap, its children that contain all query keywords are inserted in the heap. Furthermore, if the node is fruitful, the objects associated with it are also inserted in the heap. Consider the query q in our running example located on the door d where $q.\mathcal{T} = \{t_1, t_4\}$ and $k = 1$. First, the root node R_9 is accessed and its child R_7 is inserted in the heap whereas R_8 is ignored because it does not contain all query keywords. Next, R_7 is accessed and its child R_1 is inserted in the heap with key 5 whereas R_2 is ignored. Furthermore, since R_7 is a fruitful node, its object o_1 is also inserted in the heap with key 32. Next R_1 is accessed and its object o_2 is inserted in the heap with key 30. Its child R_3 is ignored because it does not contain all query keywords. Finally, the object o_2 is retrieved from the heap and is reported as answer.

The above example illustrates how to process a query considering objects in a single partition. In Section 5.4, we present the details of how the VIP-Tree and KP-Tree are utilized to process queries in an indoor venue containing many paritions.

5.3.3 Constructing KP-Tree

A KP-Tree is constructed in 4 steps: 1) fruitful nodes are created by grouping the objects having exactly the same keywords; 2) *fruitful subtrees* are constructed using the fruitful nodes; 3) the KP-Tree is constructed using the fruitful subtrees built in the previous step and a *keyword graph*; 4) the distance matrices are constructed for each node. Next, we describe the details of each step.

1) Constructing fruitful nodes. In this step, the objects that have exactly the same set of keywords are grouped together to form fruitful nodes. For example, objects o_3, o_9 and o_{11} have two keywords t_2 and t_4 and they are combined to construct a fruitful node R_2. Note that there may be fruitful nodes that have exactly one object. E.g., o_1 is the only object containing t_1, t_2 and t_4 and a fruitful node R_7 is constructed that contains o_1. In Fig. 8, the shaded nodes are the fruitful nodes.

2) Constructing fruitful subtrees. In this step, the fruitful nodes are hierarchically arranged to form possibly more than one subtrees. A fruitful subtree satisfies the property that, for each node R and its parent node R_p, R_p contains all keywords of R, i.e., $R.\mathcal{T} \subset R_p.\mathcal{T}$. Note that $R.\mathcal{T} \neq R_p.\mathcal{T}$ because each fruitful node constructed at the previous step is associated with a unique set of keywords. For a node R, there may be more than one fruitful nodes containing all keywords of R. These nodes are called potential parents for node R. Among these potential parents, we choose a node R_p to be the parent of R that has the smallest number of keywords. If two potential parents have the same number of keywords, the node with the smaller number of children is chosen to be the parent. If two nodes have the same number of keywords and children, ties are broken arbitrarily.

In Fig. 8, the potential parents for R_3 are R_1, R_4 and R_7. The node R_7 has more keywords than R_1 and R_4 and is not considered to be the parent of R_3. R_1 and R_4 both have exactly two keywords and currently have no child so an arbitrary decision is made and R_1 is chosen to be the parent of R_3.

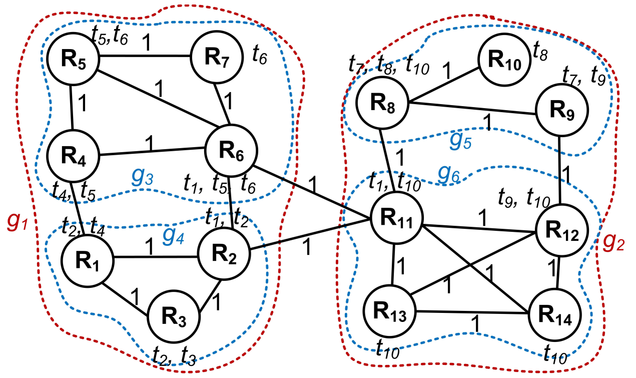
Algorithm 2: Constructing sub-trees

```
  Input    : $\mathcal{R}$: a set of fruitful nodes
1 for each node $R \in \mathcal{R}$ in ascending order of # of keywords do
2     choose a parent node $R_p$;
3     if $R_p$ is NULL then
4         Set $R$ as the root of its subtree;
5     else
6         set $R_p$ as the parent of $R$ in its subtree;
```

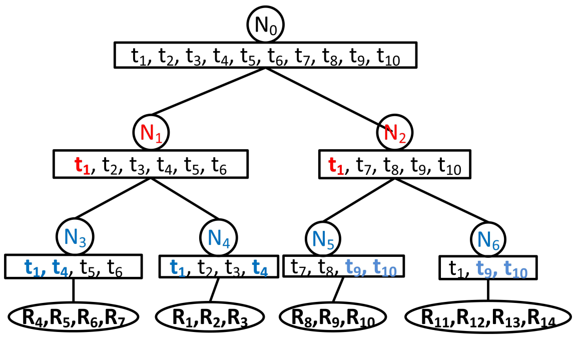
Algorithm 2 shows the details of constructing fruitful subtrees using a set of fruitful nodes $\mathcal{R}$. The nodes are accessed in ascending order of their number of keywords, i.e., the subtrees are constructed in a bottom-up approach. If there is no potential parent for a node R, it indicates that this node is the root node for a fruitful subtree. For example, in Fig. 8, there are no potential parents for the nodes R_4, R_5 and R_7 and these nodes correspond to the root nodes for three fruitful nodes. The fruitful subtree rooted at R_7 contains the nodes R_1, R_2, R_3 and R_6. Note that some fruitful nodes consist of only one node (e.g., R_4 and R_5).

Remark. The height of the tallest fruitful subtree is bounded by the length of the longest *superset chain* in the data set. A superset chain is a chain of objects $\{o_1, \cdots, o_m\}$ such that, for each $1 < i \leq m$, $o_i.\mathcal{T} \supset o_{i-1}.\mathcal{T}$, i.e., keyword set of i-th object is a superset of the keywords of $(i-1)$-th object. In Fig. 6, $\{o_{10}, o_3, o_1\}$ is a superset chain (see objects under nodes R_6, R_2 and R_7 in Fig. 8). It is easy to see that the maximum height of any fruitful subtree is equal to the number of objects in the longest *superset chain* in the data set (e.g., 3 in Fig. 8). In the worst case, the length of the longest superset chain is equal to the number of objects in the data set. However, in real-world scenarios, the height of the fruitful subtree is small because the real world data sets do not have long superset chains (an implication of Zipf's law [3]). We confirmed this by using our real world data sets and found that the maximum height of any fruitful subtree is 5.

3) Constructing KP-Tree using keyword graph. In this step, the root nodes of the subtrees constructed in the previous step are used to construct KP-tree (e.g., the nodes R_4, R_5 and R_7 are taken as input and a KP-Tree is constructed). KP-Tree is constructed in a top-down approach where each node is split into children such that the overlap (the number of common keywords) between its child nodes is minimized. Let X be the number of common keywords between two child nodes. Each query that contains these common keywords will need to traverse through both of the child nodes (and possibly branches below them). Thus, a smaller X is more effective as fewer queries would require traversing through more than one branch. Thus, we aim to minimize the overlap of keywords among children. As noted in [29], optimally dividing the objects in to nodes while minimizing the overlap of the keywords is NP-hard. While there may be other intuitive heuristics possible, in this paper, we use an intuitive and effective heuristic approach based on a *keyword graph* and a graph partitioning algorithm to guide the KP-Tree construction. Next, we describe the details – we use node to refer to an entity in the KP-Tree and vertex to refer to an entity in the keyword graph.



(a) Graph partition



(b) KP-Tree

Fig. 9: Constructing KP-Tree using a keyword graph

Each root node of the subtrees constructed in the previous step forms a vertex of the keyword graph. Every pair of vertices that have at least one common keyword are connected to each other by an edge where the edge weight is the number of common keywords between the two vertices. If the keyword graph is disconnected, we arbitrarily add edges with weight zero (between disconnected components) to obtain a connected graph. Considering the example Fig. 8 where the root nodes of the subtrees are R_4, R_5 and R_7 and these correspond to three vertices in the keyword graph. The vertices corresponding to R_4 and R_5 are connected to each other via an edge with weight 1 because the number of common keywords between R_4 and R_7 is 1. Next, we describe how the keyword graph is used to construct the KP-Tree.

Initially, a root node of the KP-Tree is created which contains all the keywords. A graph partitioning algorithm is used that *cuts* the keyword graph into f disconnected components where f is the maximum number of children for each intermediate node of the KP-Tree. Each disconnected component of the keyword graph corresponds to one child node which is associated with all the keywords in this disconnected component. Since the goal is to minimize the overlap of keywords among the child nodes, the graph partitioning algorithm aims at minimizing the total weight of the edges that connect the disconnected components. Each node of the KP-Tree is recursively decomposed using the above procedure until it contains at most α vertices. Since optimal graph partitioning is NP-Hard, we adopt a famous heuristics algorithm, called the multilevel partitioning algorithm [13] for graph partitioning.

Example 5.2. We illustrate the algorithm using an example assuming that the root nodes of the fruitful subtrees at the previous step are $\{R_1, R_2, \cdots, R_{14}\}$. Fig. 9(a) shows a sample keyword graph. To avoid mixup between nodes in KP-Tree and vertices in the keyword graph, we refer to a node of KP-Tree as N_i and a vertex in keyword graph as R_i. The root node N_0 contains all keywords $(t_1, \cdots, t_{10})$. Assuming $f = 2$, the graph is partitioned into two graphs g_1 and g_2 as shown in Fig. 9(a) minimizing the total weight of the edges connecting g_1 and g_2. The children of N_0 in KP-tree are two nodes (N_1 and N_2) obtained using the disconnected components, i.e., the node N_1 corresponds to g_1 and contains all keywords contained in g_1 (keywords $t_1, \cdots, t_6$) and the node N_2 corresponds to g_2 and consists of all keywords in g_2 (t_1, t_7, t_8, t_9 and t_{10}) – the common keyword t_1 is shown in bold. Next, the children of N_1 are computed by recursively partitioning the graph g_1 into g_3 and g_4. Similarly, the graph g_2 is partitioned into g_5 and g_6 to obtain the children nodes of N_2. Note that each R_i in Fig. 9(b) corresponds to the root node of a fruitful subtree constructed in the previous step which implies that R_i is not necessarily a leaf node in KP-Tree.

4) Constructing object and node matrices. For each fruitful node R in KP-Tree, an object matrix is created to store the distances between every door d of the partition P and every object o of the partition. Consider the indoor partition in Fig. 6 which only has one door d_1. For node R_3 in Fig. 8, the object matrix stores the distances between d_1 and the objects o_4, o_5, o_8 and o_{12}.

For each non-leaf fruitful and fruitless node R of the KP-Tree, we also create a node matrix. Specifically, for each non-leaf node R, the node matrix stores *minimum distance* $mindist(d, R_i)$ between every door d of the partition and each child R_i of R where $mindist(d, R_i)$ corresponds to the minimum distance from d to any object in the subtree rooted at R_i. In Fig. 8, the object matrix for R_7 stores $mindist(d_1, R_1) = 5$ because the objects in the tree rooted at R_1 are o_2, o_4, o_5, o_8 and o_{12} and $dist(d_1, o_{12}) = 5$ is the smallest distance from d to these objects. Similarly, $mindist(d_1, R_2) = 12$ is also stored in the node matrix. We construct the object and node matrices in a bottom-up manner. Thus, the minimum distances from d to a node R_i can be efficiently computed using the object and node matrices of the children.

5.3.4 *Handling object updates*

In this section, we discuss how to update KP-Tree when the objects change their locations (i.e., products are moved to different aisles) or when objects are deleted/added from/to a partition. If the keywords of an object change, we treat it as deleting the object and then adding a new object with updated keywords.

Handling location update If the location of an object o is changed, we update the distance matrix of the fruitful node associated with the object. Specifically, we compute new distances of o from each door d_j of the partition and update the relevant entry in the distance matrix for the door d_j. Then, we iteratively access its parent node and update the distance matrix if the minimum distance has changed. We continue this bottom-up update until the distance matrix of the parent node does not require to be changed (i.e., its minimum distance is not affected by the update).

Handling object deletion. To delete an object o with keywords $\mathcal{T}$, we first locate the fruitful node that contains o. This can be achieved in $O(1)$ by maintaining an array that records, for each object, the fruitful node that contains it. The object is then deleted from this node and the distance matrix of the node is updated by removing the corresponding entry. Then, the distance matrices of the parent nodes are iteratively updated as required using a bottom-up traversal. If the object is the only object at the fruitful node, then the fruitful node is deleted and its children are assigned to be the children of its parent.

Handling object insertion. To insert an object o with keywords $o.\mathcal{T}$, if a node in KP-Tree with exactly the same keywords exists, we simply add the object to this node and update the distance matrices of this node and its parent nodes as required. Otherwise, we handle the object insertion as follows. We start a top-down traversal from the root of the KP-Tree and continue traversing down a node R that contains all keywords of the object, i.e., $o.\mathcal{T} \subseteq R.\mathcal{T}$. If there are more than one such nodes R, we choose a node R with a smaller subtree below it. The traversal stops at a node R when none of its child R_c contains all of the object's keywords, i.e., for each child R_c of R, $o.\mathcal{T} \nsubseteq R_c.\mathcal{T}$. A fruitful node R_i is created with keywords $R_i.\mathcal{T}$ set to $o.\mathcal{T}$ and R_i becomes a child node of R (recall R contains all keywords of R_i). Finally, distance matrix of R_i is computed and the distance matrices of its parent nodes are iteratively updated if required.

5.4 Query Processing

We index the indoor venue using IVIP-Tree and, for each indoor partition P, we create a KP-Tree that indexes the objects inside it. One possible approach to answer an indoor boolean kNN spatial keyword (iBkNN-SK) query is to use an algorithm very similar to the traditional branch-and-bound kNN algorithm except that the nodes of the IVIP-Tree that do not contain all query keywords are ignored and, when the search reaches a partition P, its KP-Tree is traversed to efficiently retrieve the objects containing all query keywords and updating kNNs accordingly. However, this approach may be sub optimal as explained below.

Assume a nearest neighbor query q and two partitions P_1 and P_2 such that $mindist(q, P_1) < mindist(q, P_2)$. In this case, the algorithm will first traverse the KP-Tree of P_1 to retrieve the relevant objects from P_1. Suppose P_1 contains numerous relevant objects but the actual nearest neighbor is in the partition P_2. The algorithm will first retrieve all relevant objects from P_1 before accessing the partition P_2 and finding the actual nearest neighbor. In this case, a complete traversal of the KP-Tree of P_1 may be un-necessary and traversing it only partially may improve the performance. To achieve this, we propose to use a single min-heap that stores the entries from IVIP-Tree as well as the entries from different KP-Trees to avoid un-necessarily accessing all objects from a partition. We present the details below.

Algorithm 3 shows our proposed algorithm to answer iBkNN-SK queries. Similar to the traditional kNN algorithms, d^k which refers to the distance of current k^{th}NN is initialized to infinity. A min-heap H is used to allow accessing the entries of the IVIP-Tree and the KP-Trees in ascending order of their minimum distances from q. If the de-heaped entry N is a non-leaf node of the IVIP-Tree, the algorithm inserts every child N' of N in the min-heap that contains all query keywords. If the de-heaped entry N is a leaf node of the IVIP-Tree, for each partition P_i of this leaf node that contains all query keywords, the algorithm inserts the root R_i of the KP-Tree of P_i in the min-heap with $mindist(q, R_i)$ where $mindist(q, R_i)$ can be efficiently obtained using node matrices of the node. If the de-heaped entry N is a node of the KP-Tree for a partition, the algorithm inserts in the min-heap every child N' of N that contains all query keywords. Furthermore, if N is a fruitful node, all the objects associated with N that contain all query keywords are also inserted in the min-heap. Finally, if the de-heaped entry N refers to an object, this object is added to the answer set and d^k is updated accordingly. The algorithm terminates when $mindist(q, N)$ for a de-heaped node N is not smaller than d^k.

Algorithm 3: iBkNN-SK query processing algorithm

```
1  $d^k = \infty$;   /* distance to current $k^{th}$NN */;
2  Initialize a heap $H$ with root of the IVIP-tree;
3  while $H$ is not empty do
4      de-heap an entry $N$ from heap;
5      if $mindist(q, N) \geq d^k$ then
6          return $k$NN;
7      if $N$ is a non-leaf node of IVIP-tree then
8          for each child $N'$ of $N$ do
9              if $q.\mathcal{T} \subseteq N'.\mathcal{T}$ then
10                 insert $N'$ in heap with $mindist(q, N')$;
11     if $N$ is a leaf node of IVIP-tree then
12         for each partition $P_i$ in $N$ do
13             $R_i \leftarrow$ root node of KP-Tree of $P_i$;
14             if $q.\mathcal{T} \subseteq R_i.\mathcal{T}$ then
15                 insert $R_i$ in heap with $mindist(q, R_i)$;
16     if $N$ is a node of KP-Tree then
17         for each child $N'$ of $N$ do
18             if $c.\mathcal{T} \subseteq N'.\mathcal{T}$ then
19                 insert $N'$ in heap with $mindist(N', c)$;
20         if $N$ is a fruitful node then
21             for each object $o$ associated with it do
22                 if $q.\mathcal{T} \subseteq o.\mathcal{T}$ then
23                     insert $o$ in heap with $dist(q, o)$;
24     if $N$ is an object then
25         add the object to $kNN$ and update $d^k$;
```

6 EXPERIMENTS

All algorithms are implemented in C++ on a PC with 8GB RAM and Intel Core I5 CPU running 64-bit Ubuntu. First, we present the results for shortest distance queries in Section 6.1. Then, in Section 6.2, we evaluate the algorithms for spatial keyword queries. Readers are referred to [25] for the experimental evaluation of other spatial queries such as shortest path, kNN and range queries.

6.1 Shortest Distance Queries

6.1.1 Experimental Settings

Indoor Space. We use three real data sets: Melbourne Central Shopping Center, Menzies library building and Clayton Campus. 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.

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.

Competitors. We compare our proposed indexes (IP-Tree and VIP-Tree) with the following competitors.

TABLE 2: Indoor venues used in experiments

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

Distance Matrix (DistMx). This technique uses a distance matrix that materializes distances between all pairs of doors requiring $O(D^2)$ space.

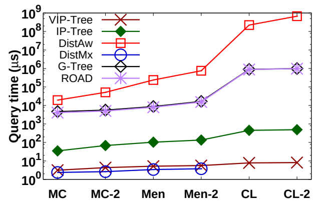
Distance-aware model (DistAw) [19]. Distance-aware is the state-of-the-art indoor query processing index which uses an extended graph based on the accessibility base graph.

ROAD [16] and G-tree [37]. We also compare our algorithms with the state-of-the-art indexes for spatial query processing on road networks (G-tree and ROAD). These indexes 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 in G-tree.

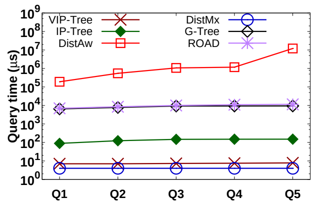
IP-Tree and VIP-Tree. We evaluate shortest distance algorithms using both IP-Tree and VIP-Tree. For both indexes, we experimentally choose the value of minimum degree t (see Algorithm 1) and find that the best performance is achieved for $t = 2$. Although the results are not shown due to the space limitations, we also found that the average number of access doors is less than 4 for all data sets and the maximum number of access doors is around 8. This provides an insight on why our indexes perform exceptionally well for indoor spaces.

6.1.2 Query Performance

Fig. 10a compares the performance of all techniques for shortest distance queries. $10,000$ pairs of source and target points are randomly generated in the indoor space. The results show the average cost for a single shortest distance query. 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 despite its significantly smaller index size. VIP-Tree significantly outperforms IP-Tree at the expense of a slightly higher indexing cost. Both VIP-Tree and IP-Tree outperform the other three techniques by several orders 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.



(a) Different indoor datasets



(b) Effect of distance b/w s and t

Fig. 10: Shortest Distance Queries

Next, we evaluate the effect of the distance between s and t on the performance of different algorithms. We use Men-2 to demonstrate the results because this is the largest data set for which DistMx can be built. 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 $[4l, 5l]$.

Fig. 10b 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.

6.2 Indoor Boolean kNN Spatial Keword Queries

6.2.1 Experimental Settings

Indoor Venue and Keyword Datasets. We use Chadstone Shopping Centre as the indoor venue. Chadstone Shopping Centre is the largest shopping centre in Australia with total retail floor area over $200,000$ m^2 and consists of around 530 stores across 4 levels. We obtained the floor plans of Chadstone Shopping Centre and manually converted them to machine readable indoor venues. To get the object datasets, we choose 11 stores (2 technology stores, 2 supermarkets, 3 home accessories stores, 2 pharmacies and 2 liquor stores) and extract the keywords related to the products from their websites. The details for the object sets for each store are shown in Table 3.

TABLE 3: Details of Stores

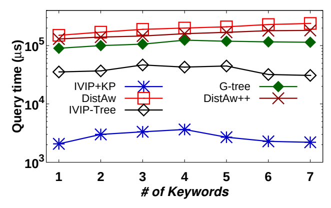
Category	Store Name	# unique products	# unique keywords
Technology	EBGames (EB)	12,848	8,432
	JB Hi-Fi (JB)	28,980	22,551
Supermarket	Woolworths (WO)	11,632	8,641
	Coles (CO)	19,079	9,991
Home Accessaries	Target (TA)	5,866	5,285
	Harris Scarf (HA)	5,307	6,793
	BigW (BI)	21,682	16,329
Liquor	Liquorland (LI)	1,397	1,382
	Dan Murphy's (DA)	14,364	9,586
Pharmacy	Amcal (AM)	7,603	5,573
	Chemist Warehouse (CH)	11,141	7,707

We use these stores to obtain several real world object data sets. Table 4 gives the details of the object data sets. The capital letters denote the category of stores used in the data set. For example, the data set TS refers to the data set that contains all **T**echnology stores (i.e., EB Games and JB Hi-Fi) and all **S**upermarkets (i.e., Coles and Woolworths). The default data set, TSHLP, is the biggest data set containing all types of stores and consists of around $140,000$ unique products (i.e., objects) across 11 different stores.

TABLE 4: Details of keyword datasets

Dataset	Vocabulary size	# products
TS	35,803	72,539
TSH	50,056	105,394
TSHLP	60,014	139,899

To evaluate our algorithms on larger indoor venues, we use Monash University Clayton Campus as the indoor venue and, for


(a) Chadstone

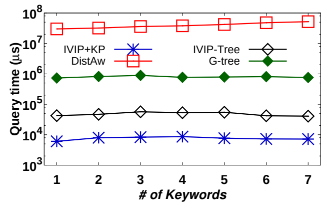
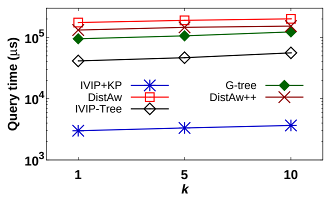

(b) Clayton

Fig. 11: Effect of # keywords


(a) Chadstone

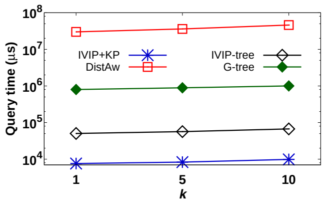

(b) Clayton

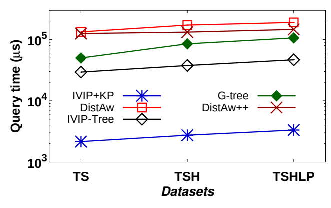
Fig. 12: Effect of k

each of the object datasets in Table 4, the stores are allocated to different indoor partitions in the indoor venues in Clayton campus.
Queries. Queries are generated using the same approach as in [7]. Specifically, we first randomly choose an object from the dataset and treat its location as the query location. Then, we randomly choose a specified number of words from the object as the query keywords. If the total number of objects that contain these query keywords is less than 10, we ignore this query and repeat the process by randomly choosing another object and keywords from it. This is to ensure that each iBkNN-SK query returns at least k objects. The value of k varies from 1 to 10 with the default value set to be 5. The default objects dataset is TSHLP and the default number of keywords is set to be 3. For each experiment, we run 100 queries generated as described above and report the average query processing cost.

The indoor spatial keyword query processing techniques rely on two types of indexes: a venue-level index (e.g., IVIP-Tree) that contains keyword summaries at each node and allows efficient pruning of irrelevant areas of the indoor venue; and a partition-specific index (e.g., KP-Tree) which is built for each indoor partition containing objects and allows efficiently obtaining the relevant objects in the partition. We evaluate our venue-level index and partition-specific index separately to clearly demonstrate the improvement made by each index. Specifically, in Section 6.2.2, we demonstrate superiority of IVIP-Tree compared to other venue-level indexes assuming that all indexes use the same partition-specific indexes. Then, in Section 6.2.3, we compare our partition-specific index, KP-Tree, with other partition-specific indexes assuming that all techniques use the same venue-level index (IVIP-Tree).

6.2.2 Evaluating Venue-Level Indexes

Competitors. In this section, we compare the following venue-level indexing techniques assuming that each index including IVIP-Tree indexes the objects in each indoor partition using inverted lists.
DistAw [19]. As described before, DistAw utilizes the AB graph of the indoor venue and keywords information is embedded for each partition.
DistAw++. This is the same as DistAw except that, to accelerate distance computations, a distance matrix is used to compute the distances between any two doors in the indoor venue.


(a) Chadstone

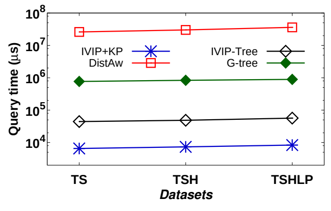

(b) Clayton

Fig. 13: Effect of object data sets

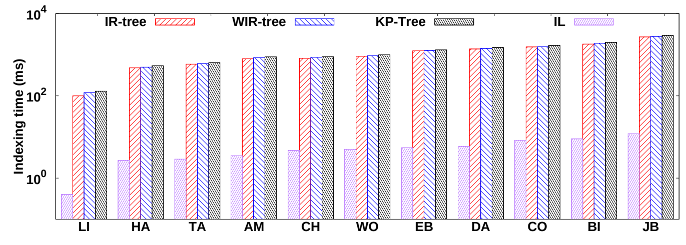
G-tree [36]. We also compare our algorithm with the state-of-the-art technique for query processing in road network (G-tree). G-tree is built on the D2D graph converted from the indoor venue. G-tree is extended to handle spatial keyword queries by storing summaries of keywords with each node.
IVIP-Tree IVIP-Tree is our venue-level index which, like other competitors discussed above, uses inverted lists for each indoor partition.
IVIP-Tree +KP-Tree We also show the performance of IVIP-Tree when it uses KP-Tree to index the objects in each partition. This is shown as IVIP+KP in the figures.

We do not show the results for the construction cost of the venue-level indexes because these are similar to the construction cost shown for spatial queries in the previous section.
Results. Fig. 11, 12 and 13 show the experimental results for different number of keywords, varying k and different object data sets for both indoor venues: Chadstone Shopping Center and Monash University Clayton Campus. Our venue-level index, IVIP-Tree, significantly outperforms other venue level indexes. When KP-Tree is used for indexing the objects in every partition (i.e., IVIP+KP), our technique outperforms all other methods by at least one order of magnitude. This shows the effectiveness of our venue-level index IVIP-Tree as well as our partition-specific index KP-Tree. Note that DistAw++ is only available for smaller indoor venues due to the $O(D^2)$ construction time and storage requirement for the distance matrix. Therefore, results for DistAw++ are not shown for the Clayton data set.

Fig. 11a shows that the querying cost of our techniques increases when the number of keywords is increased from 1 to 4 and the cost decreases when the number of keywords is further increased from 4 to 7. This is because, as the number of keyword increases, more nodes of indexes can be pruned as fewer nodes contain all query keywords. On the other hand, the distance between query to the objects satisfying keyword criteria also increases resulting in an increased cost. Similar behavior was reported in [7] for some spatial keyword query processing techniques in Euclidean space.

6.2.3 Evaluating Partition-Specific Indexes

Partition-specific indexes. To evaluate partition-specific indexes, we use IVIP-Tree to index the indoor venue and each competitor uses, for each indoor partition, a certain partition-specific index to index the objects in it. Specifically, we compare *KP-Tree* with *Inverted Lists (IL)*, *IR-tree [8]* and *WIR-tree [29].* For each approach, we experimentally determined the best values of the parameters used in the index. For KP-Tree, the fanout f is chosen to be 64 and α is set to 32.
Indexing cost. Fig. 14 compares the construction time and index size for each indexing technique for different stores in our data sets (see Table 3 for the details of each abbreviation). The stores on x-axis are listed in increasing order of the total number of unique products in each store. As expected, inverted lists can be constructed significantly more efficiently as compared to


(a) Construction time

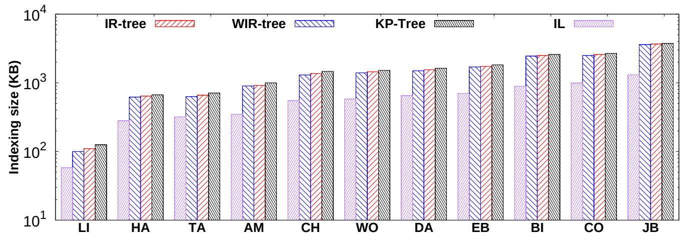
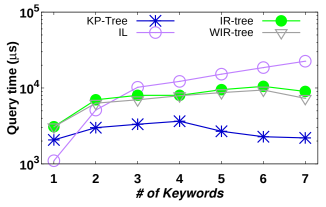

(b) Index size

Fig. 14: Indexing Cost


(a) Chadstone

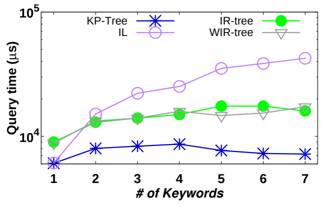
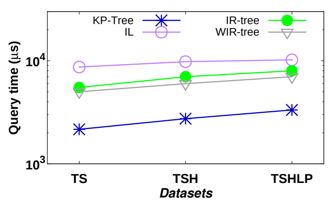

(b) Clayton

Fig. 15: Effect of # keywords


(a) Chadstone

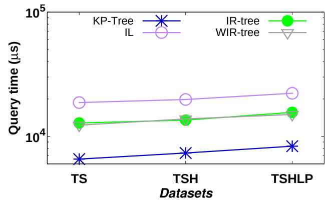
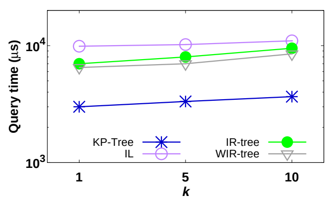

(b) Clayton

Fig. 17: Effect of object sets


(a) Chadstone

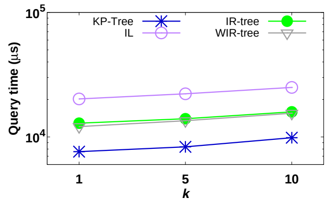

(b) Clayton

Fig. 16: Effect of k

other approaches because the construction cost mainly consists of sorting each list based on distances of the objects from each door in the partition The index size of inverted list is also the smallest. The construction time and index size of KP-tree is comparable with other approaches although a little higher. For the biggest store (JB HiFi) containing around $30,000$ unique products, KP-tree is constructed in about 4 seconds and the index size is around 5 MB.

Handling object updates. We use three largest data sets in terms of the number of unique products and keywords (Coles, BigW and JB Hi-Fi). To generate a location update (resp. object deletion), we randomly choose an object and assign it a new randomly chosen location in the partition (resp. delete the randomly chosen object). For object insertion, we initially create a KP-Tree with 90% of randomly chosen objects in the data set. Then, we insert the remaining 10% objects in it one by one. Table 5 reports the throughput (number of updates that can be handled per second) for each type of update. As expected, location update is the cheapest to handle whereas the object insertion takes more time as it requires finding an appropriate location to insert the object. Note that the number of updates in the real-world applications is typically small and the results show that our techniques can easily handle up to several thousand updates per second.

TABLE 5: Throughput (# of updates handled per second)

	Coles	**BigW**	**JB Hi-Fi**
Location changes	$96,711$	$75,244$	$44,802$
Object deletions	$56,561$	$39,808$	$27,344$
Object insertions	$6,126$	$5,175$	$4,126$

Querying cost. Fig. 15, 16 and 17 show the querying cost of each approach for different number of keywords, varying k and different object data sets for both indoor venues: Chadstone Shopping Center and Monash University Clayton Campus. Our proposed partition-specific index, KP-tree, significantly outperforms other partition-specific approaches for all data sets and settings.

Fig. 15 shows the effect of number of keywords on all algorithms. As anticipated, inverted lists (IL) give the best performance when the query consists of only one keyword. This is because it requires only checking one list which is already sorted on distances. However, the performance of IL significantly deteriorates as the number of query keywords increases. The cost of tree based indexes first increases with the increase in number of keywords and then decreases as the number of keywords is further increased. As explained earlier, this is because the number of nodes that can be pruned increases with the increase in number of keywords but, at the same time, the distances to the k nearest neighbors also increases which requires accessing more nodes of the indexes.

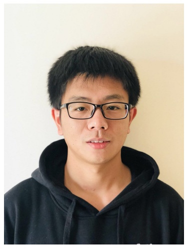
7 Conclusions and Future Work

In this paper, we propose two novel indoor indexes, IP-Tree and VIP-Tree, for efficiently processing shortest distance queries. IP-Tree and VIP-Tree have low storage requirement, small pre-processing 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. For spatial keyword queries, we extended VIP-Tree by embedding keyword information on each node. We also proposed a partition-specific index called KP-Tree that indexes the objects for each indoor partition. The experimental studies demonstrate that our proposed indexes significantly outperform the competitors. An important direction for future work is to study spatial keyword queries considering similarity to the query keywords as well as synonyms and product categories.

Acknowledgments. Muhammad Aamir Cheema is supported by Australian Research Council FT180100140 and DP180103411. Hua Lu is partly supported by Independent Research Fund Denmark (No. 8022-00366B). Shiyu Yang is supported by NSFC61802127 and Shanghai Sailing Program 18YF1406700.

References

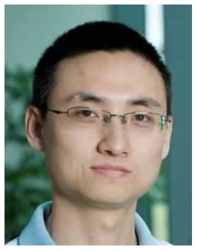
[1] http://www.citygml.org/.
[2] http://www.opengeospatial.org/projects/groups/indoorgmlswg.
[3] T. Abeywickrama, M. A. Cheema, and A. Khan. K-SPIN: Efficiently processing spatial keyword queries on road networks. *IEEE Transactions on Knowledge and Data Engineering*, 2019.
[4] T. Abeywickrama, M. A. Cheema, and D. Taniar. k-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *PVLDB*, 2016.
[5] M. A. Cheema. Indoor location-based services: challenges and opportunities. *SIGSPATIAL Special*, 10(2):10–17, 2018.
[6] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *VLDB J.*, 21(1):69–95, 2012.
[7] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*.
[8] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.
[9] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.
[10] A. Hidayat, S. Yang, M. A. Cheema, and D. Taniar. Reverse approximate nearest neighbor queries. *IEEE Trans. Knowl. Data Eng.*, 30(2):339–352, 2018.
[11] C. S. Jensen, H. Lu, and B. Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *SSTD*, 2009.
[12] M. Jiang, A. W. Fu, and R. C. Wong. Exact top-k nearest keyword search in large networks. In *ACM SIGMOD*, pages 393–404, 2015.
[13] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing*, 1995.
[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998.
[15] J. Lee. A spatial access-oriented implementation of a 3-d gis topological data model for urban entities. *GeoInformatica*, 2004.
[16] K. C. K. Lee, W. Lee, B. Zheng, and Y. Tian. ROAD: A new spatial object search framework for road networks. *IEEE TKDE*, 2012.
[17] 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.
[18] G. Li, J. Feng, and J. Xu. DESKS: direction-aware spatial keyword search. In *IEEE ICDE*, pages 474–485, 2012.
[19] H. Lu, X. Cao, and C. S. Jensen. A foundation for efficient indoor distance-aware query processing. In *ICDE*, 2012.
[20] H. Lu, B. Yang, and C. S. Jensen. Spatio-temporal joins on symbolic indoor tracking data. In *ICDE*, 2011.
[21] J. B. Rocha-Junior and K. Nørvg. Top-k spatial keyword queries on road networks. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 168–179, 2012.
[22] S. B. Roy and K. Chakrabarti. Location-aware type ahead search on spatial databases: semantics and efficiency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 361–372, 2011.
[23] C. Salgado, M. A. Cheema, and D. Taniar. An efficient approximation algorithm for multi-criteria indoor route planning queries. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018*, pages 448–451, 2018.
[24] Z. Shao, M. A. Cheema, and D. Taniar. Trip planning queries in indoor venues. *The Computer Journal*, pages 1–18, 2017.
[25] Z. Shao, M. A. Cheema, D. Taniar, and H. Lu. VIP-Tree: An effective index for indoor spatial queries. *PVLDB*, 10(4):325–336, 2016.
[26] Z. Shao and D. Taniar. Enhanced range search with objects outside query range. *World Wide Web*, 18(6):1631–1653, 2015.
[27] Z. Shao, D. Taniar, and K. M. Adhinugraha. Voronoi-based range-knn search with map grid in a mobile environment. *Future Generation Comp. Syst.*, 67:305–314, 2017.
[28] D. Wu, G. Cong, and C. S. Jensen. A framework for efficient spatial web object retrieval. *VLDB J.*, 21(6):797–822, 2012.
[29] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.*, 24(10):1889–1903, 2012.
[30] X. Xie, H. Lu, and T. B. Pedersen. Efficient distance-aware query evaluation on indoor moving objects. In *ICDE*, 2013.
[31] X. Xie, H. Lu, and T. B. Pedersen. Distance-aware join for indoor moving objects. *IEEE TKDE*, 2015.
[32] B. Yang, H. Lu, and C. S. Jensen. Scalable continuous range monitoring of moving objects in symbolic indoor space. In *CIKM*, 2009.
[33] B. Yang, H. Lu, and C. S. Jensen. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In *EDBT*, 2010.
[34] S. Yang, M. A. Cheema, X. Lin, Y. Zhang, and W. Zhang. Reverse k nearest neighbors queries and spatial reverse top-k queries. *VLDB J.*, 26(2):151–176, 2017.
[35] W. Yuan and M. Schneider. Supporting continuous range queries in indoor space. In *MDM*, 2010.
[36] 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.
[37] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *CIKM*, 2013.
[38] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.



Zhou Shao is a Teaching Associate at Faculty of Information Technology, Monash University, Australia. He Obtained his PhD from Monash University Australia in 2018. He received his Master degree in Monash University Australia in 2014 and Bachelor degree in Nanjing University of Aeronautics and Astronautics in 2012 both in Information Technology. His research interests include spatial database, especially in indoor data management.

Muhammad Aamir Cheema is an Associate Professor at Faculty of Information Technology, Monash University, Australia. He obtained his PhD from UNSW Australia in 2011. He is the recipient of 2012 Malcolm Chaikin Prize for Research Excellence in Engineering, 2013 Discovery Early Career Researcher Award, 2014 Dean's Award for Excellence in Research by an Early Career Researcher, 2018 Future Fellowship, 2018 MSA Teaching Award and 2019 Young Tall Poppy Science Award. He has won two CiSRA best research paper of the year awards, two invited papers in the special issue of IEEE TKDE on the best papers of ICDE (2010 and 2012), and two best paper awards at WISE 2013 and ADC 2010, respectively. He is a senior member of the IEEE.

David Taniar received BSc, MSc, and PhD, all in Computer Science, specialising in Databases. His research is mainly in parallel database, and spatial/mobile query processing. He has published extensively in these areas, including a book in High Performance Parallel Database Processing (Wiley, 2008). He is the Founding Editor in-Chief of two SCIE journals (Data Warehousing and Mining, and Web and Grid Services). He is currently an Associate Professor at the Faculty of Information Technology, Monash University.



Hua Lu is a professor MSO in the Department of Computer Science, Aalborg University, Denmark. He received the BSc and MSc degrees from Peking University, China, and the PhD degree in computer science from National University of Singapore. His research interests include data management, geographic information systems, and mobile computing. He has served as PC co-chair or vice chair for NDBC 2019, MDM 2012, ISA 2011 and MUE 2011, PhD forum cochair for MDM 2016, and demo chair for SSDBM 2014. He has served on the program committees for conferences such as VLDB, ICDE, KDD, WWW, CIKM, DASFAA, ACM SIGSPATIAL, SSTD, MDM, PAKDD and APWeb. He is a senior member of the IEEE.

Shiyu Yang is an Associate Professor at School of Software Engineering, East China Normal University. He received the BS and MS degrees from the Dalian University of Technology, China, and the PhD degree from the University of New South Wales, Australia. His research interests include spatial databases and location-based services. He has published papers in conferences and journals including ICDE, PVLDB and VLDB Journal.