Efficient Query Processing in Indoor Venues

by

Zhou Shao

Thesis

Submitted by Zhou Shao

for fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisor: Assoc. Prof. David Taniar,

Dr. Muhammad Aamir Cheema

Faculty of Information Technology
Monash University

August, 2018

Copyright notice

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: Zhou Shao

Date: 10/05/2018

Acknowledgments

I would like to thank everyone who helped to make this possible. It has been an incredible journey in the past three years. At first, I wish to thank my supervisor, Associate Professor David Taniar. He was my supervisor when I did my honours degree and he did a great help for me to apply my PhD to Monash University. He taught me everything that is required to be a good research student. Whenever I sent him email to ask some questions, he always replied me quickly, sometimes, it was even in mid night.

I would like to thank my co-supervisor, Dr Muhammad Aamir Cheema. I still remember that we worked throughout the night before the due date of VLDB 2017. You taught me how to do a good research and all the details we have to care about. Whenever I knocked the door of your office, you always gave valuable feedbacks to me.

Special thanks to all the nice people in our research groups. It was really enjoyable to have group meetings that we can share our ideas. I have been extremely luck to be in the same office with Agnes, Ammar, Chaluka and Lingxiao. It was always a pleasure to have a chat with you, especially for Agnes since we were in the same group from our honours year.

I would especially like to thank all the staff of Clayton School of Information Technology at Monash University. All of you have been there to support me whenever I needed any help during my Ph.D. program.

At last, a special thanks to my wife. She was always with me to help me get out of some depression moments. Special thanks to my parents who came to Melbourne to help me look after my little baby so that I can focus on my research. The last but not the least, I would like to thank my little daughter, thanks for all the special moments you gave me.

Zhou Shao

Monash University

August 2018

Contents

List of Tables

List of Figures

LIST OF PUBLICATIONS

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

[2] Zhou Shao, Muhammad Aamir Cheema, and David Taniar. Trip planning queries in indoor venues. *The Computer Journal*, 61(3):409-426, 2018

[3] Zhou Shao, Joon Bum Lee, David Taniar, and Yang Bo. A real time system for indoor shortest path query with indexed indoor datasets. In *Databases Theory and Applications - 27th Australasian Database Conference, ADC 2016, Sydney, NSW, September 28-29, 2016, Proceedings*, pages 440-443. Springer, Berlin Heidelberg, 2016

[4] Zhou Shao, Muhammad Aamir Cheema, and David Taniar. VIP-Tree and KP-Tree: Effective Indexes for Spatial and Keyword Queries in Indoor Venues. (under review for VLDBJ)

Efficient Query Processing in Indoor Venues

Zhou Shao
[redacted]
Monash University, 2018

Supervisor: Assoc. Prof. David Taniar,
Dr. Muhammad Aamir Cheema
[redacted]

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 thesis, 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 also present efficient algorithms for some fundamental spatial queries such as k nearest neighbors queries and range queries.

Apart from the fundamental spatial queries, we study a new type of indoor queries, called the *Indoor Trip Planning Query* (*i*TPQ). So far, no specific solutions have been proposed for *i*TPQ. Even if outdoor techniques are revised for *i*TPQ, they fail to process *i*TPQ efficiently. In this thesis, we propose an indoor-specific technique, based on the indoor *VIP-Tree*, called the *VIP-Tree Neighbor Expansion* (*VNE*) method, that also includes new pruning techniques in both pre-processing and query processing phases. Our experimental results show that our proposed method *VNE* outperforms other indoor and outdoor algorithms by several orders of magnitude in terms of processing time with low indexing cost.

Furthermore, 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 algorithms by several orders of magnitude.

Chapter 1

Introduction

1.1 Overview

Recently spatial databases have received increasing research interests due to its importance in people's daily life. Spatial databases are database systems that store the spatial objects like hospitals, schools and parks and provide the support for querying these spatial objects. In a mobile environment, to query the spatial objects, the crucial part is to identify the locations of mobile users. Hence, the global positioning system (GPS) [31, 41, 65] is a fundamental part that locates user locations accurately. According to the user locations, spatial queries take affect to satisfy the requirements of mobile users such as find the shortest route from the user location to his home. The widely used spatial queries are finding the spatial objects according to user's interests. For example, a person would like to find out the nearby restaurants at lunch time, or a taxi driver wants to find the nearest petrol station since the car is about to run out of the fuel. A typical example of spatial queries is illustrated in Fig. 1.1 using the map from Google Maps [2]. There exist a number of cafes around the area of Monash University Clayton Campus. A mobile user u_1 located at the query point q would like to buy a coffee, however, u_1 wants to find the nearby cafes that are not more than 500 meters far away from his current location. Therefore, a spatial query, specifically range query, is invoked that forms a circular area indicated by the black circle. As the result, three cafes are found.

However, research shows that human beings spend more than 85% of their daily lives in indoor spaces [43] such as office buildings, shopping centers, libraries, and transportation facilities (e.g., metro stations and airports). In fact, not many researches have been done in indoor space. In the outdoor space, the fundamental part of query processing is that locations of mobile users

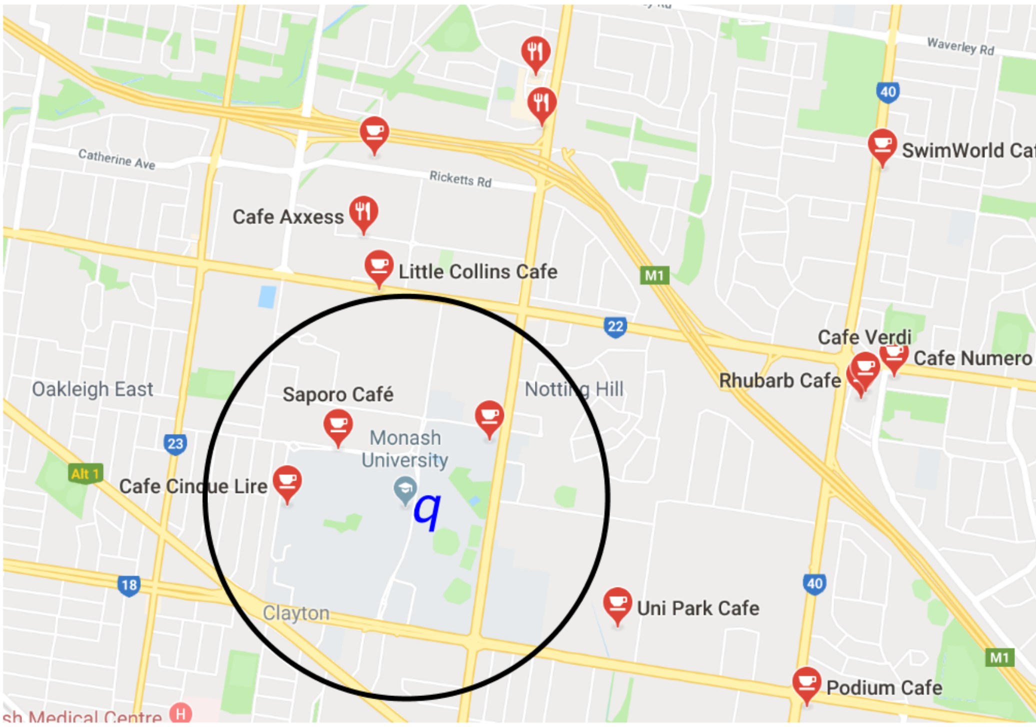


Figure 1.1: An example of spatial queries

are detected by the global positioning system accurately. While in the counterpart indoor space, breakthroughs of the indoor positioning technologies (see [58], and its references) have been made in recent years. Hence, indoor location-based services (LBSs) are expected to boom in the coming years [3, 4, 82] and some reports suggest that indoor LBSs would have an even bigger impact than their outdoor counterparts [5].

Fig. 1.2 shows an indoor complex which is formed by a subway station together with more than 20 buildings like shopping malls and office buildings. The figure shows the floor plan of the subway station while the exists are numbered. The buildings connect through these exists are not shown. In spatial networks, this subway station is modelled as a spatial object. However, for a tourist who wants to find the specific stores like a restaurant and a gym in the indoor space, results of outdoor spatial queries return the indoor space as a spatial object only while the details inside the indoor space are omitted. In fact, the indoor space has a large number of unique properties such as rooms, doors and hallways. With spatial queries in outdoor space, the indoor space is located, but not the specific room is found. Hence, driven by the limitations of the current outdoor techniques, there is a huge demand for efficient and scalable spatial query-processing systems for indoor location data.

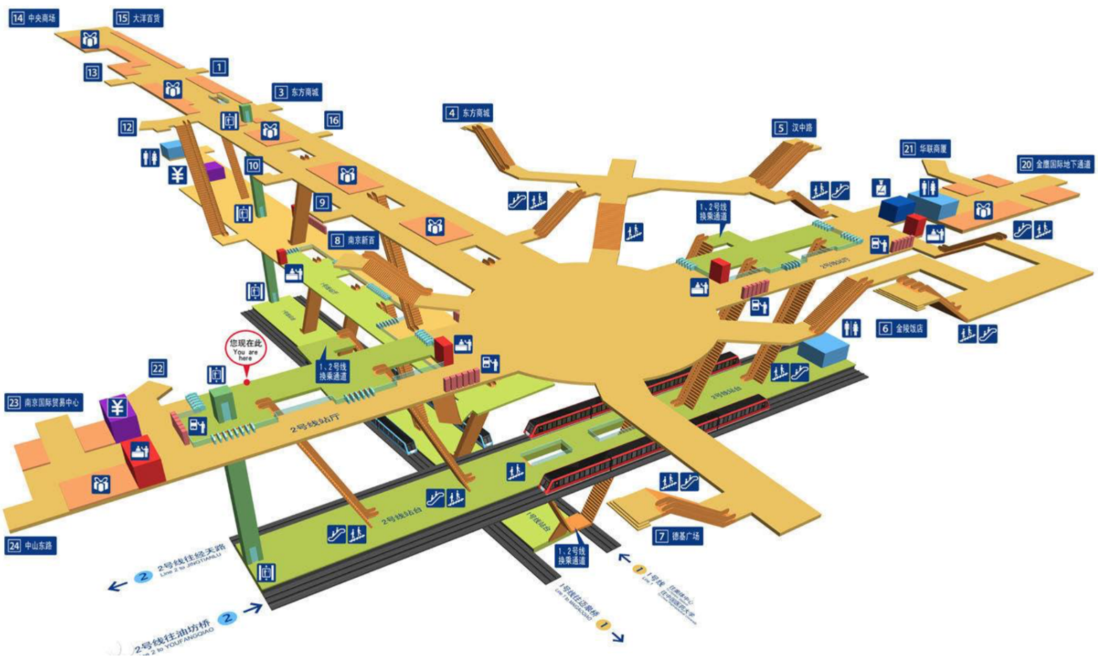
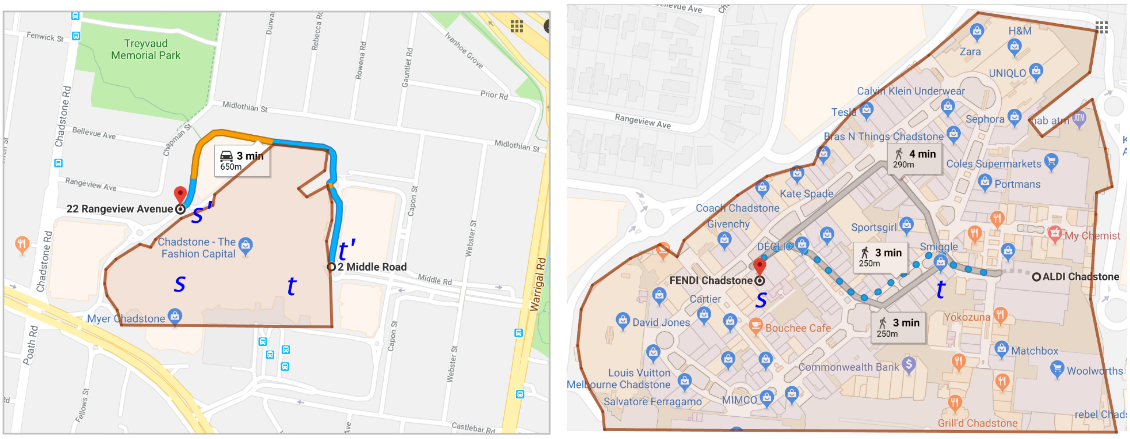


Figure 1.2: An indoor complex [1]

A detailed example in Fig. 1.3 is utilized to explain the major differences between indoor and outdoor spaces, as well as the spatial queries. The area of Chadstone Shopping Centre is shown in both Fig. 1.3(a) and 1.3(b). Fig. 1.3(a) is considered as outdoor space, while Fig.1.3(b) is considered as indoor space. The rooms are represented by polygons or irregular shapes and doors are not indicated on the map. The shaded areas in Fig. 1.3(b) is the indoor map of that in Fig. 1.3(a) (Fig. 1.4 and Fig. 1.5 follow the same style as Fig. 1.3). Assume the user is located at point s (Fendi Chadstone shown in Fig. 1.3(b)) and he wants to go to point t (ALDI Chadstone shown in Fig. 1.3(b)), a shortest path query is required here. For the outdoor shortest path query in Fig. 1.3(a), instead of the two points s and t, the alternative points s' and t' are identified on spatial networks. This is because both s and t are located in the indoor venue, hence, the nearby points on spatial networks are used alternatively. Consequently, the shortest path is shown following the actual roads. On the other hand, the indoor shortest path query is shown in Fig. 1.3(b). The actual shortest path shown in blue dashed line passes through the hallways inside the indoor venue. From these two figures, it clearly shows that spatial queries in spatial networks fails to work properly in indoor space. Hence, specific query techniques are required for indoor space which is the main goal in this thesis.

This section is organized as follows. Section 1.2 gives a introduction of a few useful spatial queries in indoor space. In Section 1.3, we briefly describe the major challenges in this PhD project followed by the main objectives in Section 1.3. The contributions that we have been made



(a) Spatial networks (b) Indoor space

Figure 1.3: Difference between indoor and outdoor spaces

to address these challenges are presented in Section 1.5. At last, we present the organizations of this PhD thesis in Section 1.6.

1.2 Some Useful Indoor Spatial Queries

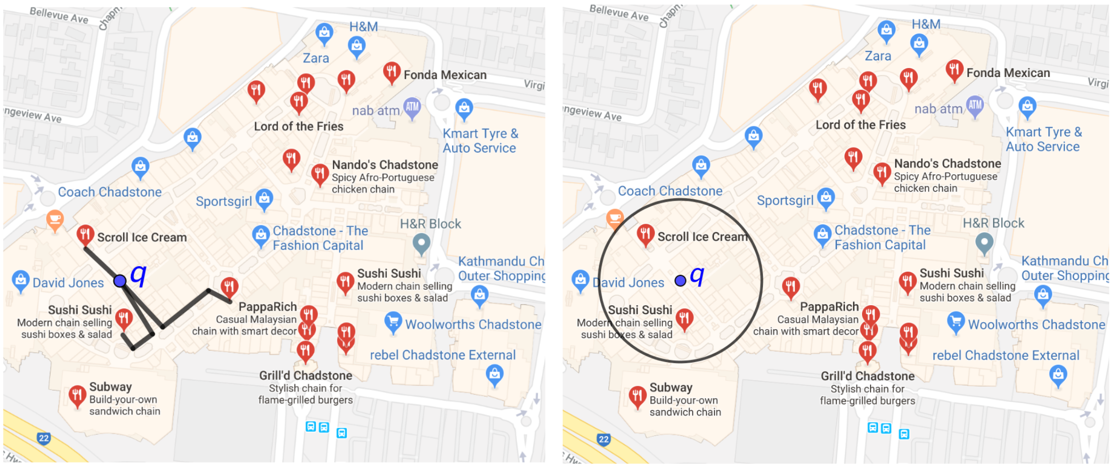
First, we give the details of some basic queries in indoor space in Section 1.2.1. Then, in Section 1.2.2, we present some more advanced queries.

1.2.1 Basic Indoor Spatial Queries

Indoor shortest distance/path queries. Given two indoor points s and t, an indoor shortest distance/path query is to find the shortest distance/path between these two points following the indoor floor plans. Take the same example in Fig. 1.3(b), the user wants to go to ALDI Chadstone from his current location Fendi Chadstone. The shortest path is shown as the blue dash line with the distance 250 meters.

Indoor k nearest neighbor queries. Given the query point q and a set of spatial objects, an indoor k nearest neighbor query is to find the k closest objects based on their distances to q. For example, the user is located in Chadstone Shopping Centre shown in Fig. 1.4(a) indicated by point q, the restaurants nearby are indicated by the red labels. He wants to find 3 nearest restaurants during lunch time. An indoor k nearest neighbor query is invoked that return 3 nearest restaurants: Sushi Sushi, Scroll Ice Cream and PappaRich (ordered by their distances to point q).

Indoor range queries. Given the query point q, a specified radius r and a set of spatial objects, an indoor range query is to find the spatial objects that the distances to q are less than r. Using the same query point q as Fig. 1.4(a), the user would like to find the restaurants that are within 100



(a) Indoor k nearest neighbor query (b) Indoor range query

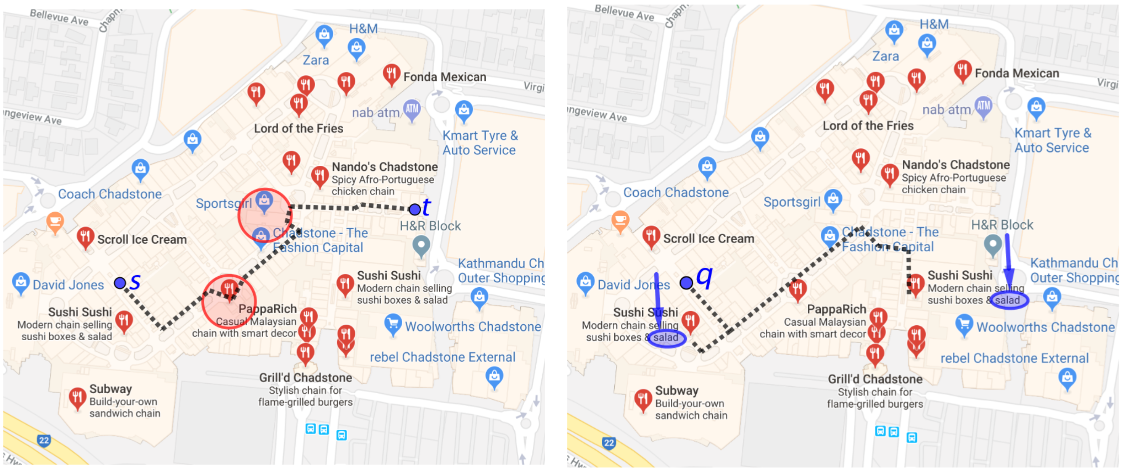
Figure 1.4: Examples of indoor spatial queries

meters. Hence, two restaurants (Sushi Sushi and Scroll Ice Cream) are returned. Note that in real case, indoor distance is utilized instead of Euclidean distance shown here.

1.2.2 Advanced Indoor Spatial Queries

Indoor trip planning queries. Given a starting point s, an ending point t and a set of spatial objects that have been categorized. An indoor trip planning query is to find the shortest path that starts at s, passes through at least one spatial object in each category and reaches to t. A detailed example is shown in Fig. 1.5(a). The spatial objects are divided into two categories: restaurants marked as red labels and clothes shops represented as blue labels. The user located at s wants to have a lunch first and then goes to a clothes shop before he meets his friend at t. An indoor trip planning query is to find the shortest path that passes through two spatial objects marked by the red shaded areas: PappaRich (restaurant) and Sportsgirl (clothes shop).

Indoor keyword queries. Given a query point q and a set of objects with a few keywords, the indoor keyword queries find the spatial objects that satisfy both spatial and keyword constraints. In Fig. 1.5(b), the query point is represented by point q. The spatial objects are tagged with keywords, for example, PappaRich is tagged with "Casual Malaysian chain with smart decor". The user located at point q wants to find a salad store, here, salad is the query keyword. Hence, two Sushi Sushi stores are returned since they contain the keyword "salad" marked by the blue shaded areas. For PappaRich, though it is closer to point q compared to the Sushi Sushi on the right side, it is not the result since it does not have keyword "salad".



(a) Indoor trip planning query (b) Indoor boolean keyword query

Figure 1.5: Examples of indoor spatial queries

1.3 Major Challenges

Due to the major differences between indoor and outdoor spaces together with the examples of some popular indoor spatial queries discussed previously, this PhD project has the following challenges.

1.3.1 Different Representation of Indoor Space

For Euclidean distance, it is represented by a two dimensional space, hence, the distance metric is the length of the straight line between two points. In spatial networks, a graph is utilized and the shortest path has to follow the edges in spatial networks. In indoor space, one possible representation is the door-to-door (D2D) graph [89] that is proposed to model the indoor venue as a graph. Each node in the graph represents a door, while the edge is created the two doors are inside the same partition. Fig. 1.6 shows an indoor venue containing 16 indoor partitions and 20 doors.

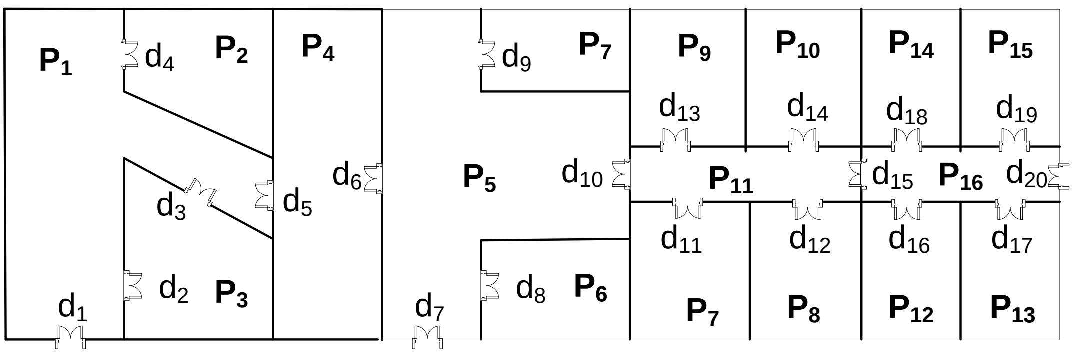


Figure 1.6: An indoor venue

The corresponding D2D graph is shown in Fig. 1.7. Edge weight is omitted for visualization. The problem is that this graph supports the distance calculations between any two nodes (doors). In fact, people located in indoor space are always inside the rooms which is known as an indoor point. However, such indoor points cannot be shown on the graph. Extra techniques are required to enable the D2D graph to support shortest distance computations between two indoor points (shortest distance computations between any two indoor points are the fundamental part for spatial query processing in indoor venues).

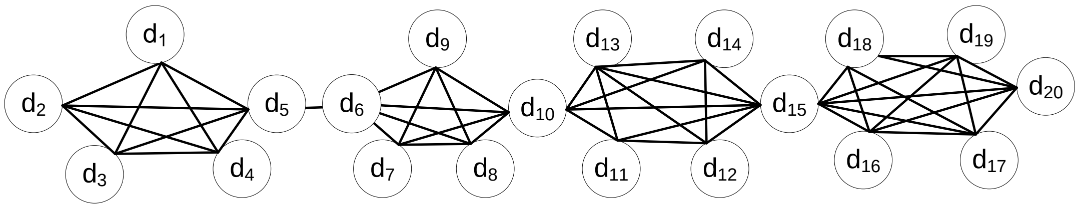


Figure 1.7: D2D graph of the indoor venue

Another representation of indoor space is distance-aware model [62]. The most important part of distance-aware model is the accessibility graph that considers each indoor partition as a node and an edge is generated if two indoor partitions share a common doors. Fig. 1.8 shows the corresponding accessibility graph of the indoor venue in Fig. 1.6. The problem is that the distance information are not embedded in the graph. Meanwhile, the distance computation is based on the expansion on the graph, therefore, it faces the problem of scalability. With the increasing size of indoor venues, the query performance goes down dramatically that has been proved in our experimental evaluation. Hence, the major difficulty here is that how we are going to represent and index the indoor space to support efficient query processing.

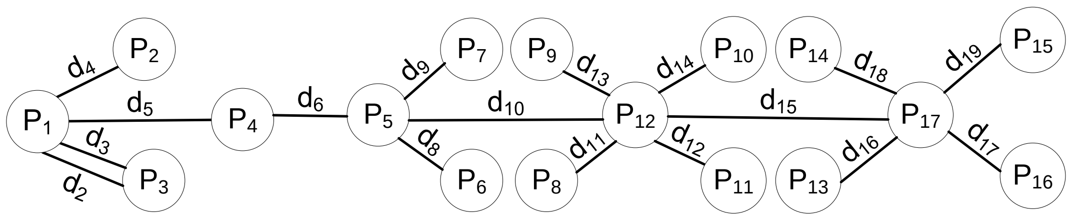


Figure 1.8: Accessibility graph of the indoor venue

1.3.2 Requiring Specific Techniques for Querying Indoor Objects

Once we can index the indoor venue properly, to query indoor objects, specific techniques are required. In Fig. 1.7, It clearly shows that the nodes in the D2D graph have very high out degrees that makes it different from that in spatial networks. In spatial networks, the average out degree of a node is about 2.5. However, in indoor space, the out degree can be huge like more than 100 if one hallway is connected by more than 100 rooms which is normal in some indoor venues like shopping centres. Hence, the state-of-the-art indexes in spatial networks like ROAD [56] and G-tree [93] achieve poor performance on the D2D graph.

Apart from the fundamental queries such as shortest distance/path, k nearest neighbor and range queries, trip planning queries is another important spatial query that is studied in spatial networks. Trip planning query is proved to be a NP-hard problem, hence, most of the existing techniques are focusing on heuristic solutions that solve trip planning query in a reasonable time. For indoor space, as we mentioned earlier, it can be transferred into a D2D graph such that existing outdoor techniques can be applied. The same problem exists for the techniques handling trip planning queries in outdoor space due to the unique properties of indoor spaces. Hence, an efficient algorithm is required to solve trip planning queries in indoor space.

Spatial keyword queries are originally discussed in Euclidean space due to the efficient distance computations. A few techniques are proposed to solve keyword queries such as the inverted index and IR-tree. However, the problem is that the indexes can not be utilized indoor space due to different distance metrics. On the other hand, for the techniques proposed in outdoor space, they can be revised to solve keyword queries in indoor space according to the D2D graph. The problem is that they are not originally designed for indoor space so that they are not efficient (The experimental results show that the revised outdoor techniques are much slower compared to our proposed technique). Hence, an efficient solution has to be provided to solve the keyword queries in indoor space.

1.4 Objectives

In this section, we describe the objectives we are going to achieve in this thesis.

1.4.1 Propose an Effective Index Method in Indoor Space

Due to the nature of indoor space, the exisiting outdoor techniques are not efficient for indoor spatial queries. Hence, in this thesis, we are going to build an unique index for indoor space that carefully exploit the indoor properties. According to the proposed index, the query processing algorithms have to be introduced to solve the indoor spatial queries like shortest distance/path, k nearest neighbors and range queries.

1.4.2 Propose an Efficient Algorithm for Indoor Trip Planning Queries

To the best of our knowledge, there is no existing techniques that are specifically designed to handle trip planning queries in indoor space. Though the techniques utilized in outdoor space can be extended in indoor space based on the *D2D* graph, the query time is not efficient at all. Hence, we are going to proposed an efficient algorithm to solve trip planning queries in indoor space.

1.4.3 Propose an Efficient Algorithm for Indoor Keyword Queries

Similar to trip planning queries in indoor space, no work has been done to solve spatial keyword queries in indoor space. In this thesis, we are going to solve indoor keyword queries by proposing an effective index for objects in the indoor partition. The number of objects in one indoor partition may be large, e.g. around 30,000 objects exists in JB Hifi, hence, index is needed for querying these objects efficiently.

1.5 Contributions

In this section, we briefly discussed the contributions made in this thesis. We proposed novel indexes that index the indoor venue and the objects inside. Meanwhile, efficient techniques are introduced to solve several spatial queries in indoor space.

1.5.1 Spatial-only Queries

To handle spatial queries in indoor space, we carefully exploit the properties of indoor space and proposed an indoor partitioning tree (IP-Tree) that indexing the indoor venue using a tree structure. To further improve the query performance, extra storage cost is utilized that formulates vivid IP-Tree (VIP-Tree). A wide range of spatial queries are supported by both IP-Tree and VIP-Tree such shortest distance/path, k nearest neighbors and range queries. For both IP-Tree and VIP-Tree, they require low indexing cost, but achieve much better query performance compared to the existing state-of-the-art indexes in both outdoor and indoor spaces. VIP-Tree achieves near-optimal efficiency for shortest distance and path queries compared to the distance matrix method (distances between any two doors are pre-computed, hence, distance computation between two indoor points are optimal).

As mentioned before, indoor space is not well studies, hence, no datasets are available for experimental evaluation. We manually built the datasets for over 70 buildings in two indoor venues:

Melbourne Central Shopping Centre [6] and Monash University Clayton Campus [7]. We transferred the floor plan images to machine readable files. Over 40,000 rooms and 40,000 doors have been indexed for our experiments.

This work was published in *Proceedings of the VLDB Endowment* (**PVLDB**) 2017.

1.5.2 Indoor Trip Planning Queries

We are the first one to study Trip Planning Query in indoor space and an exact algorithm is proposed to solve indoor Trip Planning Query (*i*TPQ) efficiently. An expansion-based method is proposed with a few pruning techniques. The pruning techniques are discussed in both pre-processing and query processing phase. A large number of unnecessary candidate routes can be pruned according to the prune techniques that are proved to be efficient in the experimental section. On the other hand, the distance computation is based on the proposed index VIP-Tree that ensures the low indexing cost and efficiency.

For the datasets utilized in the experimental section, the previous indoor venues are used. We manually added the points of interest into the existing indoor venues and put them into different categories.

This research appeared in *The Computer Journal*.

1.5.3 Indoor Keyword Queries

No existing techniques has been proposed for keyword queries in indoor space, hence, we are the first one to study indoor keyword queries. First, we extend the previous index VIP-Tree to inverted VIP-Tree (IVIP-Tree) by embedding keyword information on the nodes and using inverted list to index the objects for the leaf nodes. The experimental evaluation shows that the simple extension of VIP-Tree outperforms the existing techniques that have been revised to solve indoor keyword queries. Furthermore, Keyword partitioning tree (KP-Tree) is proposed to index the objects in one indoor partition. The query processing algorithm is based on IVIP-Tree and KP-Tree. In the experimental evaluation, KP-Tree is proved to much efficient compared to the existing techniques such as IR-tree and WIR-tree with a comparable storage cost.

To get the real dataset for the experimental evaluation, we manually built an index for Chadstone Shopping Centre [8] that over 400 rooms and 400 doors. 11 real stores belonging to 4 categories are utilized to form the keywords datasets. Nearly 140,000 objects in these 11 stores are indexed by storing their product information (keywords) that are used later in our experimental evaluation.

This work is current under review in *International Journal on Very Large Data Bases* (**VLDBJ**).

1.6 Organizations

This dissertation is organized as follows:

- Chapter 2 gives a brief description of the existing techniques that are widely used in spatial databases.
- Chapters 3, 4, 5 present our research on the indexing and query processing techniques in indoor space
  - Chapter 3 discusses our proposed index, as well as the detailed techniques to answer shortest distance/path, k nearest neighbors and range queries in indoor space
  - Chapter 4 studies a more advanced query called indoor trip planning queries.
  - Chapter 5 covers our proposed techniques to solve spatial keyword queries in indoor space
- Chapter 6 concludes our research, provides several possible directions for future work.

Chapter 2

Literature Review

2.1 Overview

In this chapter, we provide a brief overview of the related work for each type of queries that we studied in this thesis. In Section 2.2, we discuss the spatial-only queries such as shortest distance/path queries in different settings like Euclidean space, spatial networks and indoor space. Trip Planning Queries is studied in Section 2.3. Finally, existing techniques for keyword queries are studied in Section 2.4.

2.2 Spatial-only Queries

First, we discuss the spatial-only queries in Euclidean space in Section 2.2.1. After that, the existing techniques in spatial networks are reviewed in Section 2.2.2. At last, Section 2.2.3 provides a brief description of the spatial-only query processing algorithms in indoor space. Table 2.1 is the comparisons of the existing techniques for spatial-only queries that are reviewed in this section. Note that only static queries are shown in the table.

2.2.1 Spatial-only Queries in Euclidean Space

In Euclidean space, an spatial object is modelled as a two-dimensional object. R-tree [38] is the fundamental work that is proposed to index and query the objects in Euclidean space. Given an example in Fig. 2.1, there are 9 objects in the Euclidean space. An object can be a point, a rectangle, or any shape like o_1. For an irregular shape like o_1, it is represented by a Minimum Bounding Rectangle (*MBR*). After that, objects are sorted and processed one by one to form the

Techniques	Euclidean Space		Spatial Networks			Indoor Space		
	*k*NN	Range	Shortest distance /path	*k*NN	Range	Shortest distance /path	*k*NN	Range
R-tree [38]	✓	✓						
SR-Tree [51]	✓	✓						
PK-Tree [80]	✓	✓						
PK+Tree [81]	✓	✓						
Dijkstra [29]			✓					
A* [37]			✓					
Labelling-based algorithms [14, 15, 16, 25, 26, 27]			✓					
Hierachy-based techniques [19, 35, 78]			✓					
INE [66]				✓	✓			
IER [66]			✓	✓	✓			
SILC [73]			✓	✓	✓			
ROAD [55, 56]			✓	✓	✓			
G-Tree [93, 94]			✓	✓	✓			
Voronoi-based approaches [53, 72, 87, 91, 92]			✓	✓	✓			
Distance-aware model [62]						✓	✓	✓

Table 2.1: Comparisons of existing techniques for spatial-only queries

R-tree shown in Fig. 2.2. A branch-and-bound searching algorithm [70] is employed to query the objects indexed by R-tree.

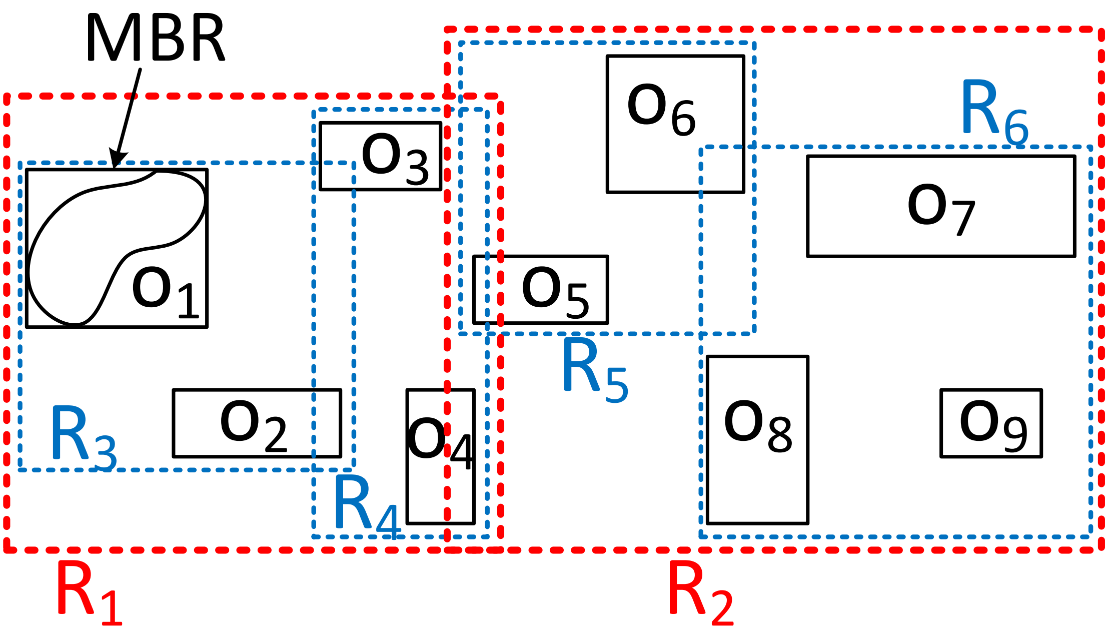


Figure 2.1: Points in Euclidean space

As we can see from Fig.2.2, tree nodes R_3 and R_4 intersect with each other that causes the searching process goes to both of the nodes. This is because the query point has the same distance to R_3 and R_4 when it is located in the intersection between the MBRs of R_3 and R_4. To improve the efficiency of the searching algorithm, a number of variants [18, 48, 51, 74, 80, 81]. SR-Tree [51] is the enhancement of R-Tree. PK-Tree [80] is proposed according to quadtree, while PK+Tree [81] is the enhancement of PK-Tree.

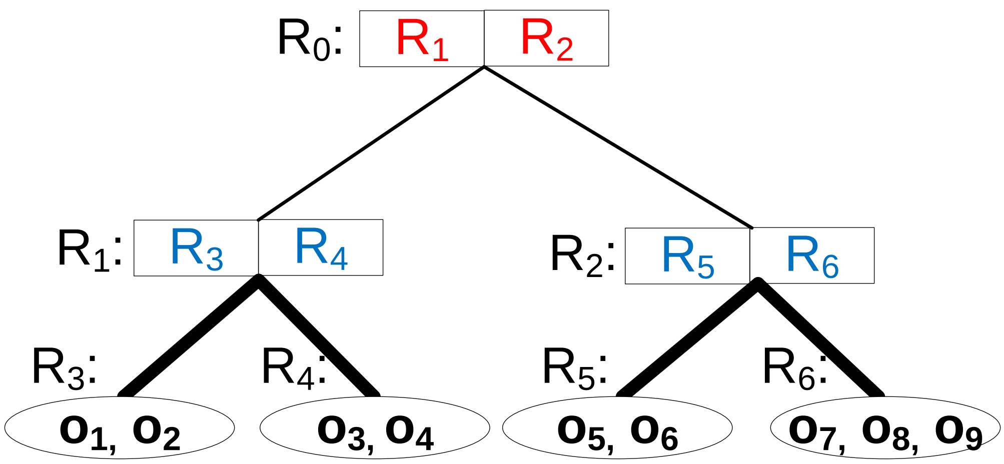


Figure 2.2: R-tree

2.2.2 Spatial-only Queries in Spatial Networks

For spatial-only queries in spatial road networks, they are divided into three categories: *shortest path query, kNN query and range query*. The fundamental work for query processing in road networks is Dijkstra search algorithm [29], which has been extended to a large number of different techniques.

For shortest path queries, [68] extended Dijkstra search to a bidirectional version. In his work, the search process started from both starting and ending points of the query. Once a common node is reached, a candidate shortest path is generated. Although it improve the performance of the original Dijkstra search, it is not efficient on a large spatial network. Apart from this, [37] proposed an A* search algorithm, which introduced a heuristic function to estimate the lower bound of a candidate path for further pruning. A simple heuristic method is using Euclidean distance. For the current expanding node, compute the Euclidean distance between current node and the destination. The node that has the minimum estimated distance to the destination is picked before reaching to the destination. The performance of A* algorithm is highly relying on the heuristic function. For these methods, they are both index free methods, which means no pre-processing is needed. However, with the increasing size of spatial networks, the query performance becomes the bottleneck.

Labelling-based algorithms [14, 15, 16, 25, 26, 27] are then proposed to handle shortest distance queries in large networks. The main idea is to store shortest information for each node in pre-processing phase. During the query time, the shortest path between two nodes can be quickly retrieved based on the shortest path information. However, since for each node in the spatial networks, a label set is generated, hence, the size of the index is an issue.

Hierarchy is widely used to solve shortest distance/path queries [19, 35, 78]. The main idea is to extract the graph into different level while details of each level is different. During search process, high level graphs can be utilized to compute the distances quickly by avoiding processing the details in the lower level graphs. Contraction hierarchy [35] is an efficient technique that is proposed to solve shortest distance queries. For every node in the graph, an order is created according to the importance of the nodes. Based on the node ordering, every node is processed sequentially to generate the shortcuts. By adding the shortcuts, the query processing can bypass a large number of intermediate nodes when the source and destination points are far away.

The previous techniques are designed to solve shortest distance/path queries. Next, we are going to present the existing techniques that solve k nearest neighbors and range queries as well.

Based on Dijkstra algorithm, Incremental Network Expansion (INE) [66] has been proposed which is quite similar to Dijkstra algorithm. It is an expansion based method. Starting from a query point q, the adjacent nodes are processed. During the expansion, objects are accessed and added to a candidate set until kth nearest object is closer to the query point compared to the current nearest node that has not been processed. Because of the size of spatial networks, INE does not scale well. To improve the performance of INE, Incremental Euclidean Restriction (IER) [66] was proposed that kept pruning the nodes that leaded to a longer distance. These Dijkstra based algorithms do not require any index. Next, we present the techniques that have unique indexes built for the spatial networks.

Spatially Induced Link- age Cognizance ($SILC$) [73] is such an algorithm which built an index for every single node in spatial road networks based on their shortest path information. According to the index of the current processing node and the destination, the next node on the shortest path can be determined. Therefore, it incrementally added the next node on the shortest path such that query processing is quite efficient. The problem is that building the index takes a long time and index size is a problem for large networks. After that, Route Overlay and Association Directory (ROAD) [55, 56] have been proposed. The main idea is to bypass regions that do not contain objects by using search space pruning. The graph is partitioned into a number of sub-graphs. Each sub-graph contains a list of borders that connect with the nodes outside the sub-graph. Distances between borders are pre-computed. During query processing, if no object exists in the sub-graph, the sub-graph is bypassed. G-Tree [93, 94] is another index based method, which utilised a graph partition algorithm to partition spatial road networks. A similar way to partition the spatial networks like ROAD is utilized while the input graph is partitioned into $f > 2$ sub-graphs. Each sub-graph is then recursively partitioned until each sub-graph contains no more than $\tau > 1$ vertices. Since the spatial network is a sparse graph, the number of borders for each sub-graph is limited. Hence, the query processing based on G-tree is efficient.

A sample network graph is shown in Fig. 2.3 containing 9 vertices $\{v_1, v_2, .., v_9\}$ and a set of edges with edge weights on them. A graph partition method is employed to partition the graph into a number of subgraphs such as g_1 to g_6. However, optimal graph partition is proved to be a NP-hard problem [34], hence, a heuristic method [50] is utilized to perform the graph partition on the graph.

Based on Fig. 2.3, the corresponding G-tree is shown in Fig. 2.4. The root node g_0 indicates the whole graph, while g_1 and g_2 represents the subgraphs partitioned from g_0. The vertices

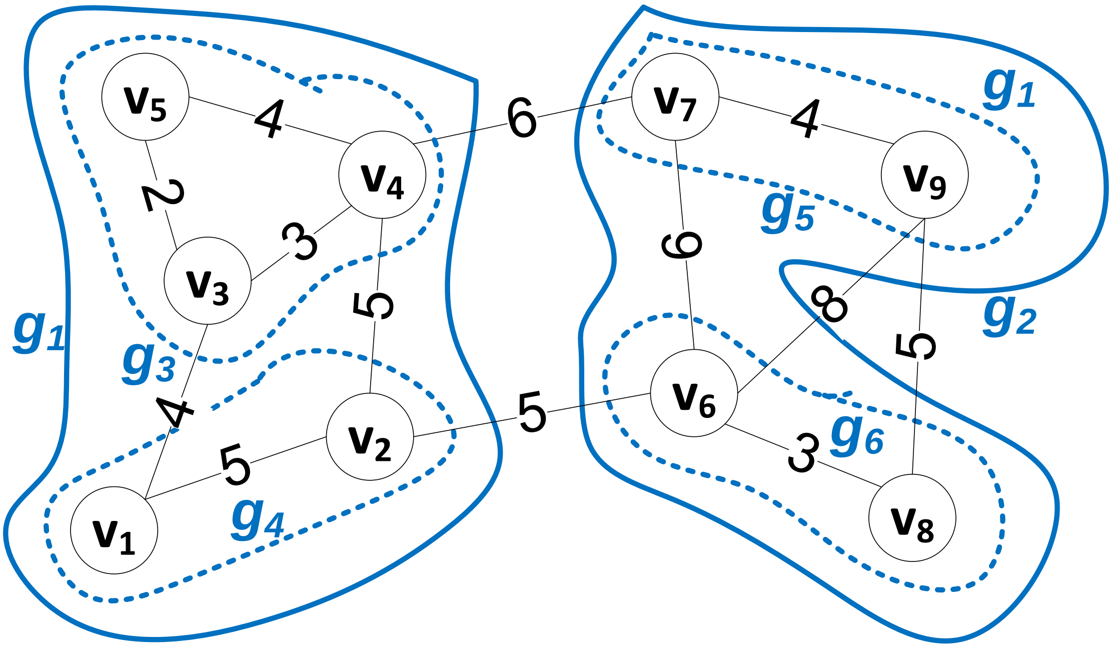
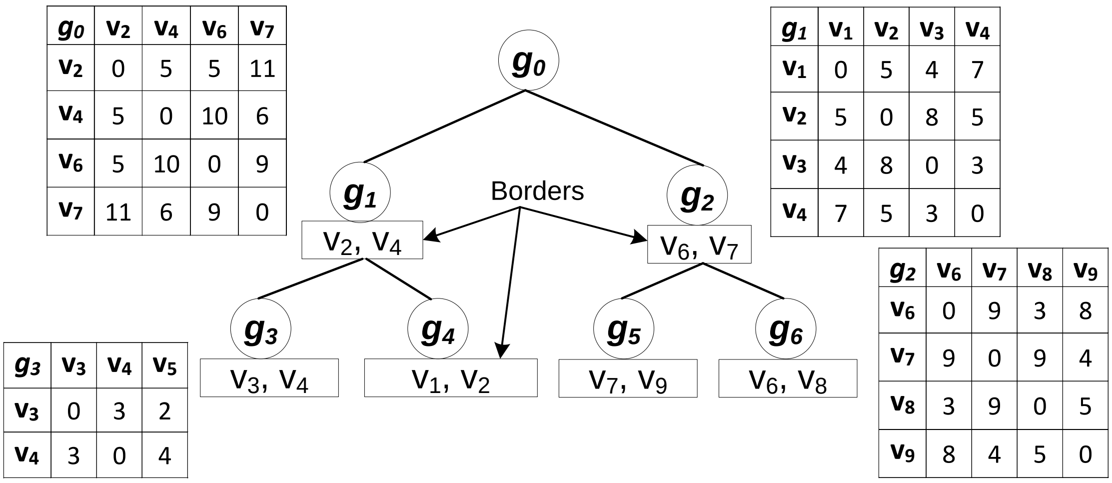


Figure 2.3: Graph partition

shown at the bottom of each node are the borders that connect the subgraph with the vertices outside. For each node, a distance matrix is pre-computed to accelerate the query processing algorithms. According to the G-tree index, a shortest distance query can be solved in n steps where the maximum n equals to 2 times the height of the tree.



g_0	v_2	v_4	v_6	v_7
v_2	0	5	5	11
v_4	5	0	10	6
v_6	5	10	0	9
v_7	11	6	9	0

g_1	v_1	v_2	v_3	v_4
v_1	0	5	4	7
v_2	5	0	8	5
v_3	4	8	0	3
v_4	7	5	3	0

g_2	v_6	v_7	v_8	v_9
v_6	0	9	3	8
v_7	9	0	9	4
v_8	3	9	0	5
v_9	8	4	5	0

g_3	v_3	v_4	v_5
v_3	0	3	2
v_4	3	0	4

Figure 2.4: G-tree

Another branch of techniques for queries in spatial net works are Voronoi-based approaches [53, 72, 87, 91, 92]. For Voronoi-based approaches, spatial road networks were pre-processed based on the objects, hence, retrieving objects became much faster than the original network. Take Fig. 2.5 as an example. A voronoi diagram is generate according to the 9 objects $\{p_1, p_2, ..., p_9\}$ that partitions the space into 9 areas known as voronoi cells, each of which contains an object. This object is called generator point for the voronoi cell. For any query point q located in the corresponding area, the first nearest neighbor is the generate point. Hence, the first nearest neighbor can be retrieved in $O(1)$ time. An expansion method is utilized to find the k nearest neighbors based on the voronoi diagram. The problem is that the pre-processing and storage cost is large. Meanwhile, once the objects set changes, it will result in the re-generation of the voronoi diagram.

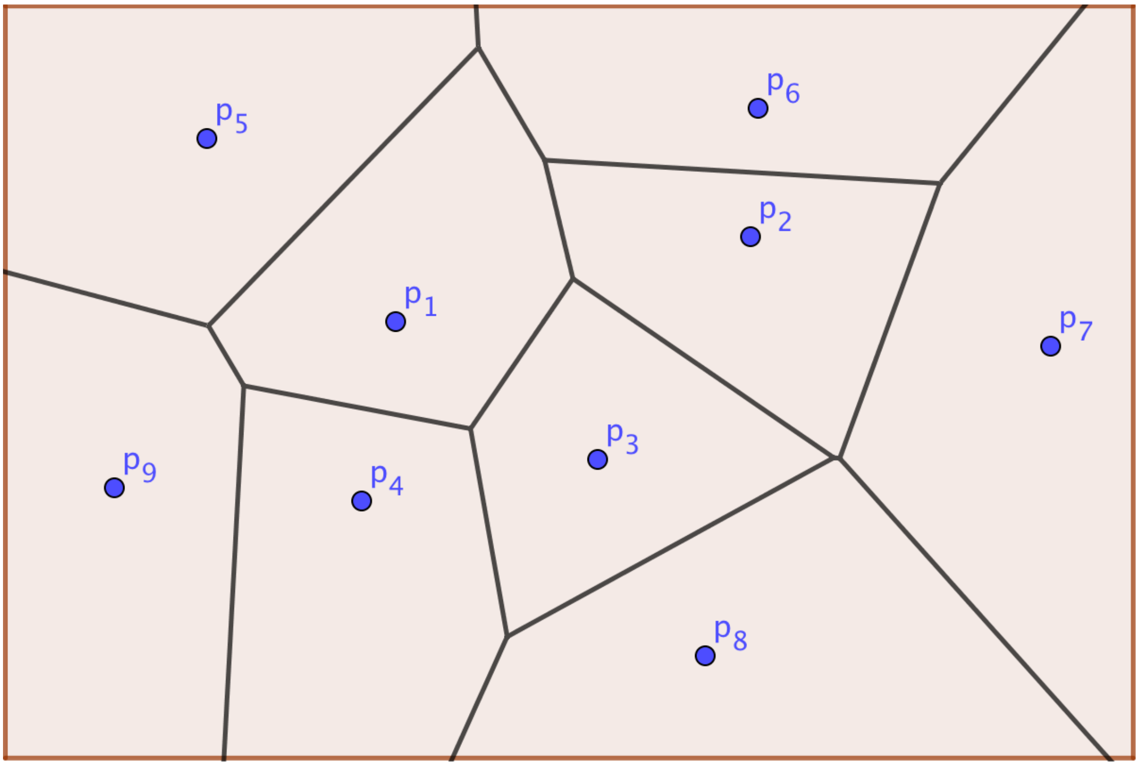


Figure 2.5: Voronoi Diagram

2.2.3 Spatial-only Queries in Indoor Space

Data modelling for indoor space is fundamental for querying indoor space. In [54], a 3D model is proposed for indoor space but it fails to support indoor distance computations. CityGML [11] and IndoorGML [12] are XML based methods to model and exchange the indoor space data. Distance-aware model [62] introduces an extended graph based on an accessibility base (AB) graph that enables indoor distance computations between two indoor positions. An AB graph considers each indoor partition as a node, while a common door between two rooms are indicated by an edge. The direction of the door can be considered as well by using the directed edges on the graph. However, different from the graph in spatial networks, in the AB graph, distances cannot be presented. Door-to-door (D2D) graph is then proposed in [85]. In a D2D graph, each node represents a door in indoor space and an edge is generated if two doors are in the same partition. The edge weights are the distances between two doors.

Since GPS technology cannot be applied in indoor space, indoor positioning technology (see [58], and its references) is developed in recent years to retrieve the accurate user locations. The main purpose of this thesis is to index and query the indoor space and indoor positioning is another indoor research area, hence, for the rest of the thesis, we assume that all the index and query algorithms are built on the accurate indoor positions. Indoor positioning data received from RFID is cleaned using spatio-temporal constraints. Graph based methods [17] take advantages of indoor constraints to fix cross and missing readings in the raw RFID data. These constraints are also applied to construct probabilistic trajectories [32] from raw RFID data.

RTR-tree and TP^{2}R-tree [45] are two indoor structures extended from R-tree which index trajectories of indoor moving objects. In terms of indoor partitions like rooms and hallways, *ind*R-tree [85] constructs a composite index that indexes indoor entities into different layers with indoor moving objects stored in the leaf level. For querying indoor data, shortest distance/path, *k*NN and range queries are studied under various settings [64, 86, 88, 90]. The most notable techniques have been discussed in [62, 89]. We present the details since these techniques are highly related to this thesis.

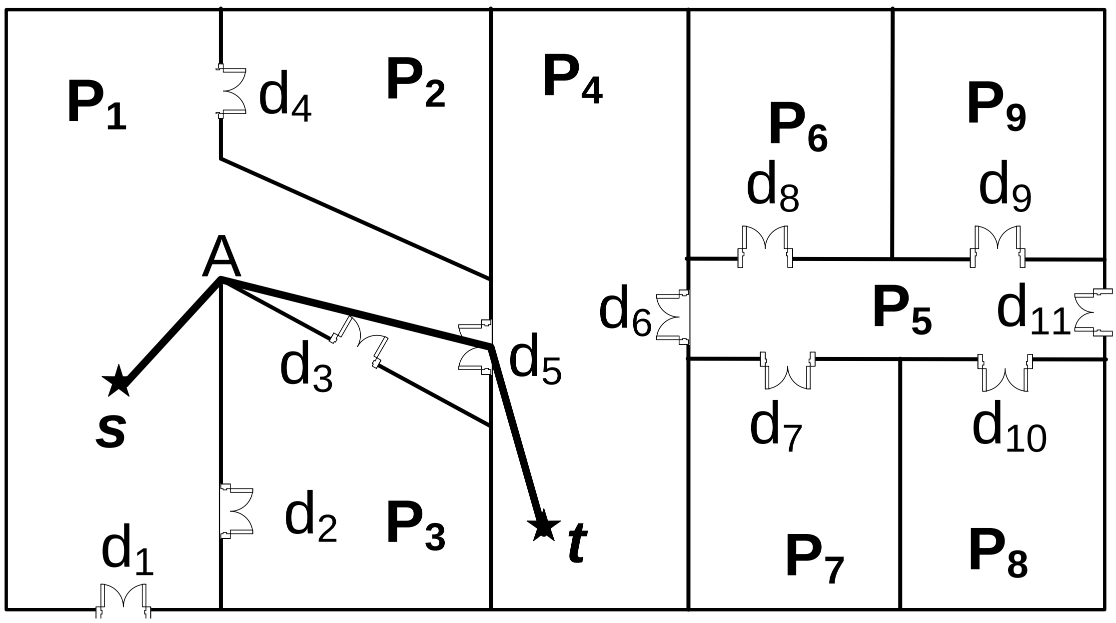


Figure 2.6: An indoor venue

Given an indoor venue shown in Fig. 2.6, there are 9 partitions and 11 doors. Different from the Euclidean distance, the distance between two points in indoor space cannot be the straight line because people cannot pass through walls. Meanwhile, for two points in the same room and there are no obstacles between them, Euclidean distance can be used, which makes it different from network distance. Hence, indoor minimum walking distance [62] is proposed as the distance metric. Given two indoor points s and t shown in Fig. 2.6, the distance between these two points are represented by the bold line that passes point A and door d_5.

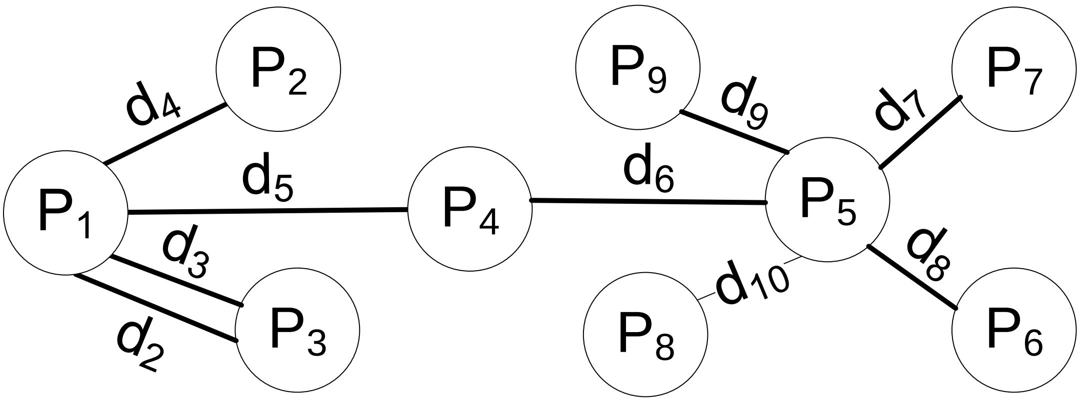


Figure 2.7: Accessability graph

According to the indoor venue in Fig. 2.6, accessibility graph (AB graph) is shown in Fig. 2.7. Each node indicates an indoor partition, while each edge is created if a door exists in both partitions. For some indoor partitions that contain more than one door like P_1 and P_3, every door is shown in the graph. For distance computation based on AB graph, extra computation or storage cost is needed since the distance are not embedded in the graph. To handle this problem, distance matrix [62] is proposed.

Fig. 2.8 illustrates how the distance matrix works. It computes the distances between every two doors in the indoor space, which is d_1, d_2, ..., d_{11}. To further improve the efficiency, for each door d_i, the other doors are sorted based on the distances to d_i. Since distance matrix requires $O(D^2)$ pre-computation and storage costs where D is the number of doors in the indoor venue, it cannot scale well with the increasing size of indoor venues.

$$
\begin{array}{c c}
 & \begin{array}{cccccc} d_1 & d_2 & \cdots & \cdots & d_{10} & d_{11} \end{array} \\
\begin{array}{c} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_{10} \\ d_{11} \end{array} &
\begin{pmatrix}
0 & 2 & \cdots & \cdots & 17 & 19 \\
2 & 0 & \cdots & \cdots & 15 & 17 \\
\vdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
\vdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
17 & 15 & \cdots & \cdots & 0 & 2 \\
19 & 17 & \cdots & \cdots & 2 & 0
\end{pmatrix}
\end{array}
$$

Figure 2.8: Distance matrix

2.3 Trip Planning Queries

In indoor space, there is no existing work that are proposed to handle trip planning queries, hence, we reviewed the techniques that are used to solve trip planning queries in both Euclidean space and spatial networks. Table 2.2 gives the details of the techniques that are reviewed in this section.

Techniques	Euclidean space	Spatial net-works	Optimal	Heuristic	TPQ	OSR	Route Search
[59]	✓	✓	✓	✓	✓		
[49]	✓	✓		✓			✓
[57]		✓		✓			✓
LORD & R-LORD [77]	✓		✓			✓	
PNE [77]		✓	✓			✓	

Table 2.2: Comparisons of existing techniques for TPQ queries

Li et. al. [59] propose a new type of query called Trip Planing Query (TPQ) in spatial databases. A set of points P=$\{p_1, p_2, ..., p_n\}$ are given, and each of the point belongs to a certain category. Given a starting point p_s, a destination point p_t and a few categories that the user wants to visit, the trip planning queries find the shortest path that starts from p_s, passes through one point in each required category and reaches to p_t. The brute-force method is to consider all the possible routes. However, it is proved to be a NP-Hard problem, hence, it is not efficient. Based on a triangular inequality property of the metric space, two fast approximation algorithms were studied in [59]. For one greedy algorithm called nearest neighbor algorithm, it incrementally adds the nearest neighbor of the last vertex added to the route from every category that has not been visited yet. However, it give a $(2^{(m+1)} - 1)$-approximation. Minimum distance algorithm is another greedy algorithm that choose the best point p in each category such that the distance of the route $p_s \longrightarrow p \longrightarrow p_t$ is the shortest among all points in the category. This method achieves better approximation m-approximation compared to the previous algorithm. An Integer Linear Programming approach [59] is proposed to further improve the approximation ration compared to the previous greedy algorithms. A linear approximation bound is achieved.

A variant of TPQ called the Optimal Sequenced Route(OSR) is discussed in [77]. In an OSR query, a category sequence to be visited is given, hence, OSR problem is not NP-Hard any more. For the current visiting category, there are at most n candidate routes where n is the number of points in the current visiting category. A Dijkstra based algorithm is proposed [77] to solve OSR queries, which is basically an expansion method considering all the possible candidate route.

After that, the authors propose two algorithms Light Optimal Route Discoverer (*LORD*) and R-tree based LORD ($R - LORD$) to solve OSR in a Euclidean space. LORD has the same flavor as Dijkstra's algorithm, but it is a threshold based algorithm. In terms of memory usage, it is proved to require less workspace compared to the Dijkstra based algorithm. A reverse way is utilized to build the partial sequenced route starting from the destination points. Only the point that leads to a route with shorter distance compared to the given threshold is added to the partial route until a full path is retrieved. To further improve the efficiency of LORD, a R-tree friendly version R-LORD is introduced. In addition to the point selection rules in LORD, a single range query based on R-tree is performed to further pruning the possible points to be added to the partial routes.

Apart from Euclidean space, a Progressive Neighbor Exploration (PNE) is discussed to deal with OSR in spatial networks. The main idea for PNE is that for the last visited point in a candidate route, the current nearest neighbor among the points belonging to next required category is added. This is under the intuition that the adding the closer point will lead to a shorter route. To efficiently handle nearest neighbor search, any existing techniques that solves nearest neighbor problems can be applied.

Constraint route search is another variant of TPQ. In [49], the route search algorithm is proposed over probabilistic geospatial data that contains spatial and textual information. For a search query, the user specifies the start and end locations, as well as some textual constraints like restaurant, park and river. They proposed several heuristic solutions under the bounded-length and bounded-probability semantics. After that [57] studied another type of route search that involves users' interaction. When the next point is returned to the user, the user has to provide the feedback whether this point satisfies his requirement. The order of the categories can be specified or partly specified that makes the solution more flexible.

2.4 Keyword Queries

In this section, we give a brief description of keyword queries studied in Euclidean space (Section 2.4.1) and spatial networks (Section 2.4.2) since for indoor space, there is no existing studies for keyword queries. Given a set of spatial objects and each object contains a list of keywords, for a query point q, keyword queries find the objects that satisfy both spatial and keyword constraints. Table 2.3 gives the details of the algorithms discussed in this section.

Techniques	Euclidean space	Spatial networks	Spatial part	Keyword part
[23]	✓		grid	inverted file
[79]	✓		grid	inverted file
Inverted R-tree [95]	✓		R-tree	inverted file
IR^2-tree [33]	✓		R-tree	bitmaps
IR-tree [28, 83]	✓		R-tree	inverted file
WIR-tree [84]	✓		R-tree	inverted file
[69]	✓	✓	Overlay indexing	inverted file
[47]		✓	Labelling method	inverted file
G-tree [94]		✓	G-tree	inverted file
DESKS [60]		✓	Region structure	inverted file

Table 2.3: Comparisons of existing techniques for keyword queries

2.4.1 Keyword Queries in Euclidean Space

Keyword queries are usually discussed in Euclidean space because of the efficiency of distance computations. A large number of techniques [60, 71, 84] are proposed to solve the problem. We provide some details of the techniques which are highly related to our problem.

Objects	**Keywords**
o_1	t_1, t_2
o_2	t_1, t_4
o_3	t_2, t_4
o_4	t_3, t_4, t_5
o_5	t_3, t_6
o_6	t_4, t_6
o_7	t_4, t_6, t_7
o_8	t_5, t_6, t_8
o_9	t_6, t_8

Table 2.4: Keywords information

Inverted index [24, 28, 36, 39, 42, 52, 61, 79, 95] is widely used in information retrieval applications. For each keyword term t, a list is created to store the objects that contains keyword t. In practice, with limited number of keywords input by the user, it is efficient to perform keyword search since a few lists have to be involved.

Given a set of objects $\{o_1, o_2, ..., o_9\}$ in Table 2.4 with 8 keywords $\{t_1, t_2, ..., t_8\}$. The locations of the objects are shown in Fig. 2.1. Fig. 2.9 shows the inverted files for part of the R-tree in Fig. 2.2. Take node R_3 as an example, two objects o_1 and o_2 are in R_3, based on the keywords of o_1 and o_2, the inverted file of R_3 contains 3 keywords t_1, t_2 and t_4. For t_1, both objects contain the keyword, hence, the object list for t_1 is o_1 and o_2.

However, due to the large amount of objects in the list, spatial indexes are utilized to organize the objects to speed up the processing. The inverted grid index is proposed in [23, 24, 79] to index the objects in the list. The problem is that grid-based indexes cannot scale well with the increasing

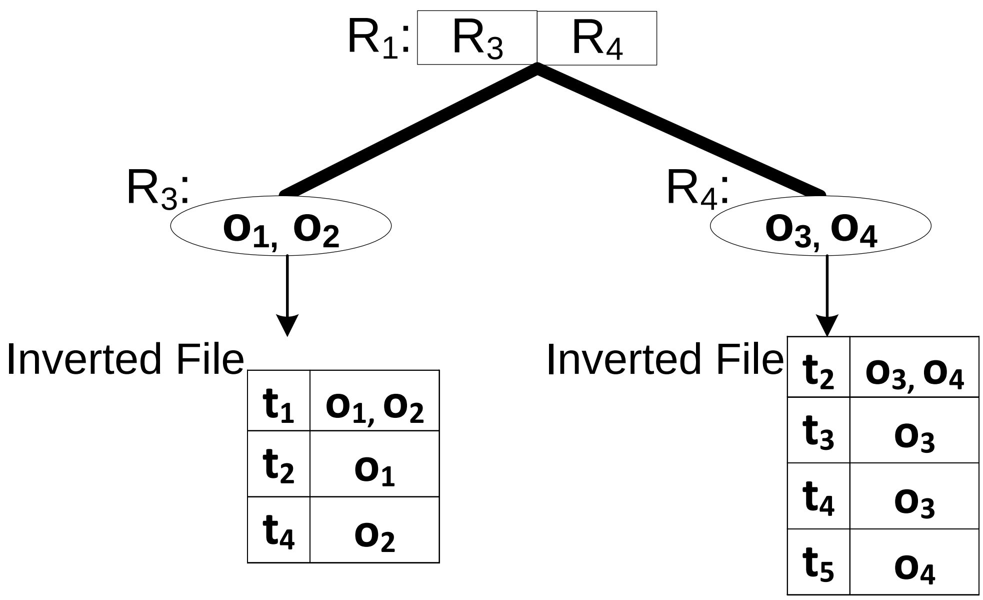


Figure 2.9: Inverted file

number of objects. Fig. 2.10 illustrates how the grid index looks like. For keyword t_4, there are 5 objects $\{o_2, o_3, o_4, o_6, o_7\}$ containing this keyword. The two dimensional space is then partitioned into 4 grid cells and the objects are allocated to the grid cell according to their locations.

Inverted R-tree [95] is proposed to solve this problem. For each keyword t, a R-tree is built based on the objects that contains the keyword t. Inverted R-tree is efficient when the number of query keywords is small because it needs to process only a few R-trees. For keyword t_4, the inverted R-tree is shown in Fig. 2.11. Since 5 objects containing t_4, a R-tree is built for these objects. For the query keyword t_4, only the inverted R-tree for t_4 needs to be processed, meanwhile, part of the objects that contain t_4 is processed instead of the large number of objects in total.

However, with the increasing number of query keywords, the performance of Inverted R-tree degrees significantly. Therefore, information retrieval R-tree (IR^2-tree) [33] is proposed that solves the problem by utilizing *signature* technique. All the objects are indexed by an R-tree according to their spatial lcoations. For each node of the R-tree, a *signature* is assigned that summarizes the keywords contained in its descendent data entries (objects). During query processing, once the *signature* of a node confirms that no objects in the node match the query keywords, no further processing is needed.

The information R-tree (IR-tree) [28, 83] is proposed as an improved technique compared to IR^2-tree since they have similar tree structures. Instead of *signatures*, IR-tree utilized inverted files for each node that maintains the keywords information in the node. Fig. 2.12 shows the IR-tree according to the R-tree in Fig. 2.2. For leaf node, the inverted files store the object list for

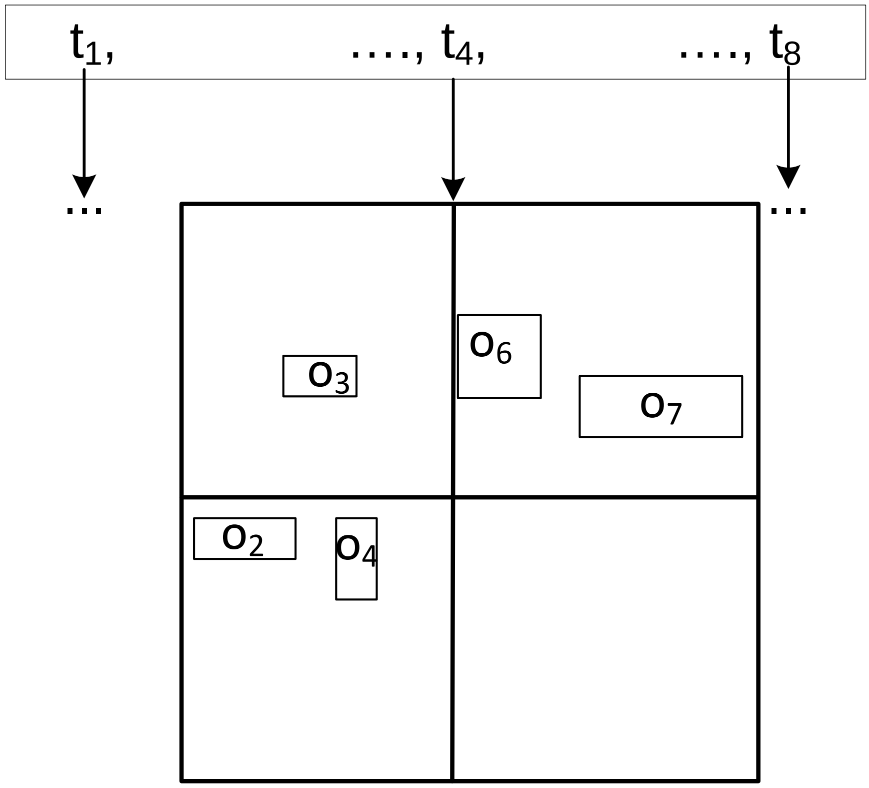


Figure 2.10: Grid index under keyword t_4

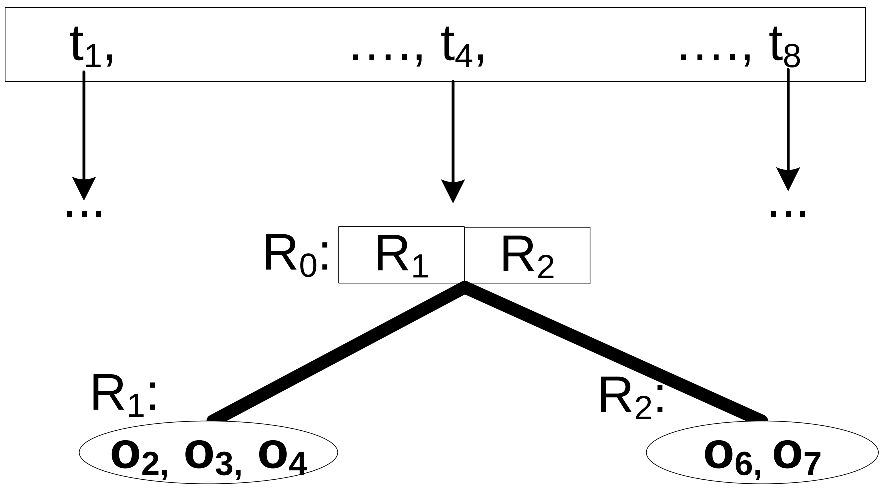


Figure 2.11: Inverted R-tree under keyword t_4

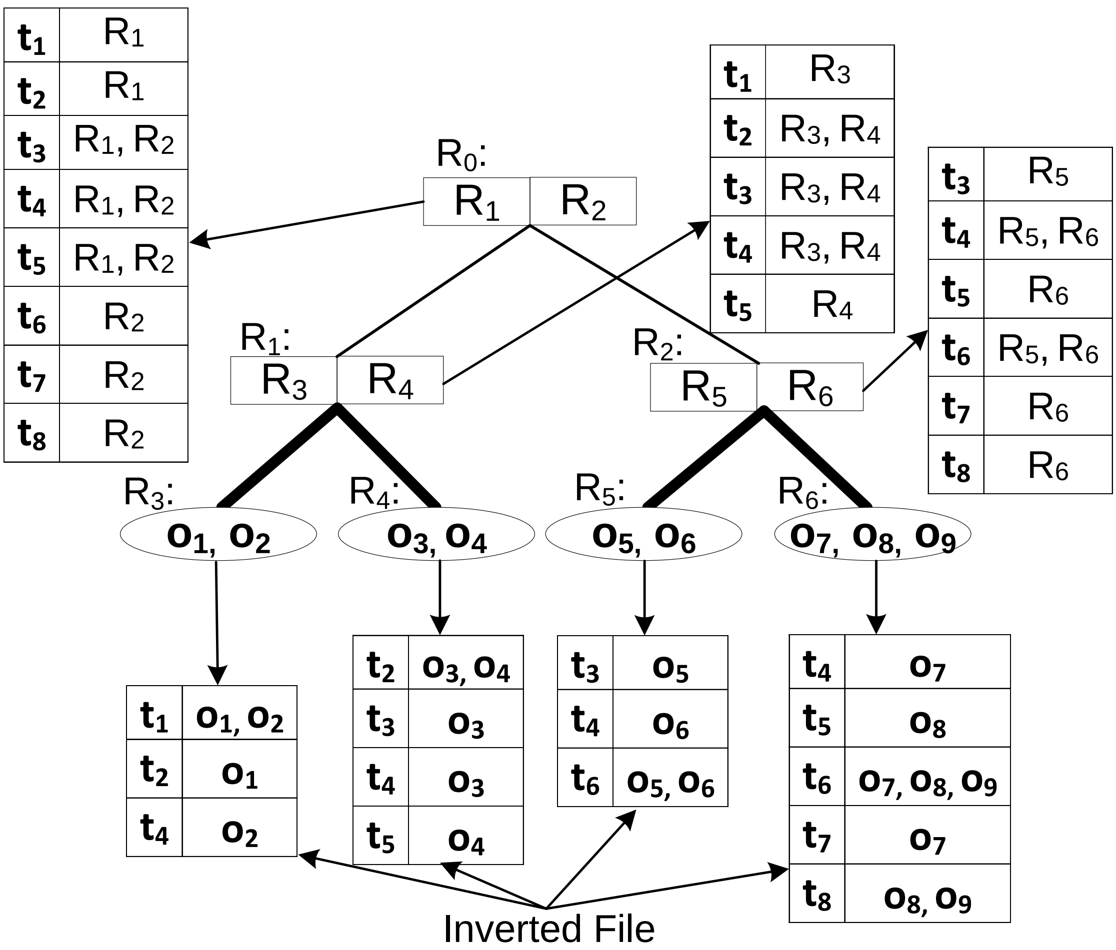
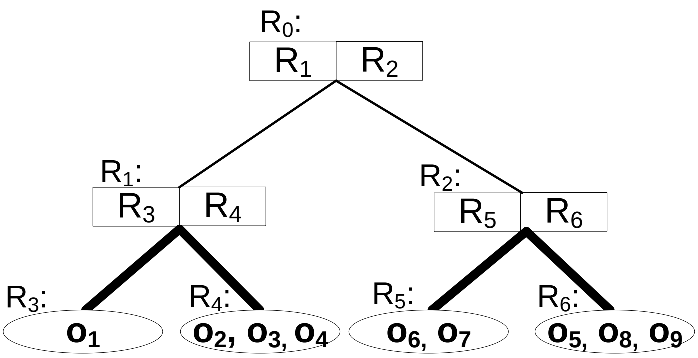


Figure 2.12: IR-tree

each keyword. For non-leaf node, the keywords in the node is the union of the keywords in the child nodes. The inverted file for non-leaf node is created as follows. For each keyword t, the list of child nodes containing t is stored. For example, the root node contains all keywords. For keywords t_3, t_4 and t_5, they are in both R_1 and R_2.

Some optimizations are applied to further improve the query processing. WIR-tree [84] utilizes the similar tree structure as IR-tree, but it partitions the objects according to keyword frequencies instead of spatial locations. Use the same object set, a WIR-tree is built in Fig. 2.13. First, the keywords are sorted in an ascending order according to the frequencies of each keyword. The sorted keyword list is $\{t_4, t_6, t_3, t_1, t_2, t_5, t_8, t_7\}$. Therefore, the objects are divided into two sets according to keyword t_4, $\{o_2, o_3, o_4, o_6, o_7\}$ and $\{o_1, o_5, o_8, o_9\}$. A parameter α is set to be the maximum number of objects in a leaf node, and we assume α is 3 here. Hence, there two objects sets have to be further partitioned according to keyword t_6. Consequently, 4 leaf nodes are created shown in Fig. 2.13. Once the leaf nodes are generated, each leaf node is considered as an object and they are partitioned according to the keywords sorted by the frequency computed by the leaf nodes. At last, WIR-tree is built. Compare Fig. 2.11 with Fig. 2.13, both indexes have the similar structure, but the objects contained in the leaf nodes are different due to the different grouping methods.



Node	Keywords
R_0	$t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8$
R_1	t_1, t_2, t_3, t_4, t_5
R_2	$t_3, t_4, t_5, t_6, t_7, t_8$
R_3	t_1, t_2
R_4	t_1, t_2, t_3, t_4, t_5
R_5	t_4, t_6, t_7
R_6	t_3, t_5, t_6, t_8

Figure 2.13: WIR-tree

2.4.2 Keyword Queries in Spatial Networks

A few techniques [47, 60, 69, 94] are proposed for keyword queries on spatial road networks. [69] is the first work to address the challenge of keyword queries in spatial networks. A basic approach is proposed for top-k keyword queries based on the existing state-of-the-art methods such as IR-tree [28, 83]. Furthermore, an indexing method is introduced by indexing the objects on each road segments with a detailed algorithm. To improve the efficiency of the query processing algorithm, an overlay network is proposed to prune the regions that does not contains any objects satisfying the keyword constraints. However, with the increasing size of spatial networks, the query performance downgrades significantly. To handle large spatial networks, [47] designs two specific methods: A forward search method is proposed when the query keywords are not frequent, while a forward backward search is performed to handle the query keywords that are frequent. Both of these two methods are based on the labels built by the 2-hop labelling method.

G-tree [94] is another technique that handles keyword queries in spatial networks. It is a graph partition technique that partitions the road network graph into sub-graphs and efficiently solves kNN queries as we discussed before. To solve keyword queries, the keywords information is embedded for the nodes. A textual function is utilized together with the existing distance function to efficiently pruning intermediate nodes during query processing.

After that, DESKS [60] considers the direction constraints for keyword queries in spatial networks. A novel direction-aware index is proposed that groups the objects according to their distances and directions. Similar to other spatial indexes, this direction-aware index efficiently prunes a large number of unnecessary objects during query processing. The cached results of previous keyword queries are utilized to answer new keyword queries as well.

2.5 Conclusion

In this Chapter, we presented a detailed studied for the existing techniques that are utilized to handle spatial queries in different settings. At first, the indexing and querying processing techniques for spatial-only queries like shortest distance/path, k nearest neighbors and range queries are discussed. Most of the techniques are focusing in Euclidean space and spatial networks. A few techniques are proposed to handle spatial queries in indoor space. Apart from the spatial-only queries, techniques for trip planning queries are discussed. All the techniques are in Euclidean space and spatial networks since no work has been done in indoor space. At last, spatial keyword queries are presented. Most of the existing works are in Euclidean space due to the efficiency of distance computation. There do exist a few techniques for spatial networks, but no techniques have been proposed in indoor space.

Chapter 3

VIP-Tree: An Effective Index for Indoor Spatial Queries

3.1 Overview

Research shows that human beings spend more than 85% of their daily lives in indoor spaces [43] such as office buildings, shopping centers, libraries, and transportation facilities (e.g., metro stations and airports). Due to this important fact, the recent breakthroughs in indoor positioning technologies (see [58], and its references), and the widespread use of smart phones, indoor location-based services (LBSs) are expected to boom in the coming years [3, 4, 82] and some reports suggest that indoor LBSs would have an even bigger impact than their outdoor counterparts [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.

Driven by recent advances in indoor location technology and popularity of indoor LBSs, there is a huge demand for efficient and scalable spatial query-processing systems for indoor location data. Unfortunately, as we explain next, the outdoor techniques provide below par performance for indoor spaces and the existing indoor techniques fail to fully utilize the unique properties of indoor venues resulting in poor performance[1].

This chapter is organized as follows. In Section 3.2, the limitations of existing techniques in both outdoor and indoor spaces are discussed. The main contributions are presented in Section 3.3. The detailed indexing method is proposed in Section 3.4 followed by the query processing algorithms in Section 3.5. Section 3.6 covers the comprehensive experimental evaluations. A conclusion is given in Section 3.7.

3.2 Background Information

3.2.1 Limitations of Existing Outdoor Techniques

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

One possible approach for indoor data management is to first model the indoor space to a graph using existing indoor data modelling techniques [62, 12] 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 [56] and G-tree [93] 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

[1]Published in 43^{rd} International Conference on Very Large Data Bases (VLDB) 2017

to California and Nevada states consists of around 4.6 million edges and 1.9 million vertices [13]. Thus, specialized techniques are required that carefully exploit the properties of indoor space to provide efficient results.

3.2.2 Limitations of Existing Indoor Techniques

Adopting the idea of mapping the indoor space to a graph and applying graph algorithms, existing techniques use door-to-door graph [89] and/or accessibility base graph [62] to process various indoor spatial queries.

Door-to-door (D2D) graph [89]. 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. 4.1 shows an example of an indoor space that contains 17 indoor partitions (P_1 to P_{17}) and 20 doors (d_1 to d_{20}). The corresponding D2D graph is shown in Fig. 3.2(a) where edge weights are not displayed for simplicity. The doors from 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 [62]. 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. 3.2(b) shows the AB graph for the indoor space shown in Fig. 4.1. Since partitions P_1 and P_2 are connected by door d_4, an edge labeled as d_4 is created between P_1 and P_2 in the AB graph. Partitions P_1 and P_3 are connected by two doors d_2 and d_3, and thus two labeled edges are created between P_1 and P_3. Although an AB graph captures the connectivity information, it does not support indoor distances.

Distance matrix (DM) [62]. 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 and pre-computed door-to-door distances.

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 [62] computes the shortest distance between a source point s and a target point t (shown as stars in

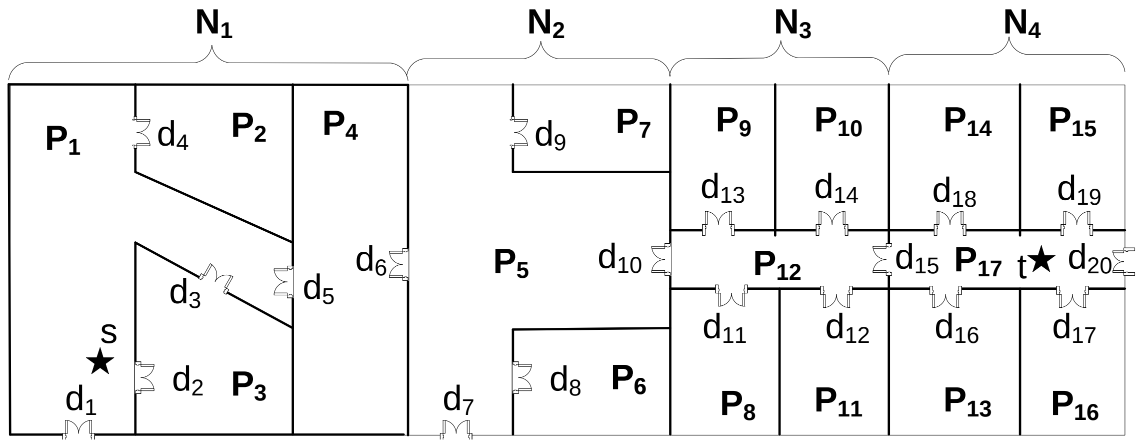


Figure 3.1: An indoor venue containing 17 partitions and 20 doors

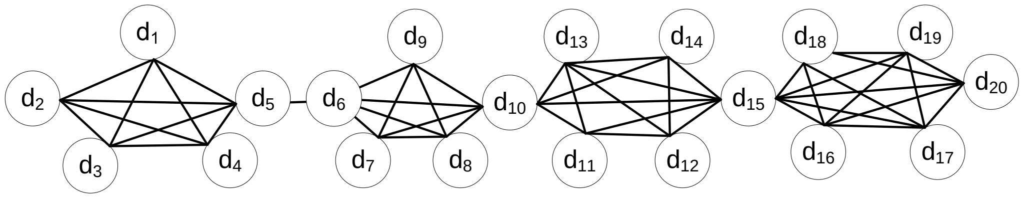
Fig. 3.1) using Dijkstra's like expansion on a D2D graph or AB-graph. Although several optimizations are employed in [62], these techniques essentially rely on a Dijsktra's like expansion over the entire graph which is computationally quite expensive. Consequently, the state-of-the-art indoor query processing takes more than 100 seconds to answer a single shortest path query on the Clayton campus data set used in our experiments.

3.3 Contributions

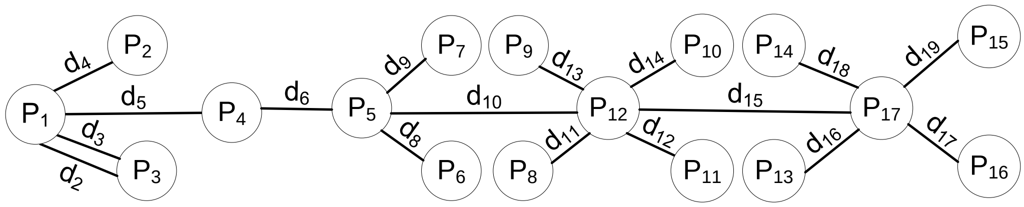
In this chapter, we propose two novel indoor indexes called Indoor Partitioning tree (IP-Tree) and Vivid IP-Tree (VIP-Tree) that optimize the indexing by exploiting the properties of indoor spaces. The basic observation is that the shortest path from a point in one indoor region to a point in another 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 [62] and road networks [93, 56] by several orders of magnitude. In comparison with the distance matrix, that allows constant time retrieval of distance between any two doors at the cost of expensive pre-computing and quadratic storage, our VIP-Tree also achieves comparable, near-optimal performance for shortest distance and path queries.

Low indexing cost. VIP-Tree and IP-Tree have small construction cost and low storage requirement. For example, for the largest data set used in our experiments that consists of around $83,000$ rooms (around 13.4 million edges), VIP-Tree and IP-Tree consume around 600 MB and can be



(a) Door-to-Door Graph



(b) Accessibility Base Graph

Figure 3.2: Indexing Indoor Space

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 not only provide practical efficiency but also have low storage and computational complexities. Table 3.1 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.

Table 3.1: *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 not covered in this chapter. For example, the proposed indexes can be used to answer spatial keyword queries in indoor space by integrating the inverted lists with the nodes of the tree, e.g., in a way similar to how R-tree is extended to IR-tree [22] to support spatial keyword queries in outdoor space.

3.4 Indexing Indoor Space

First, we define some terminology and the data model used in this thesis. 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. 4.1) 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 chapter, we choose $\gamma = 4$). In Fig. 4.1, 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 [89] 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 [11] data objects can be embedded in each partition. The results generated by our spatial query processing algorithms can be passed to other applications (e.g., [30, 40]) to provide visual/landmark-based navigation to the users. Next, we present the details of our indexes.

3.4.1 Indoor Partitioning Tree (IP-Tree)

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. 3.3 shows an IP-Tree of the indoor venue shown in Fig. 4.1 where the indoor space is first converted into four leaf nodes (N_1 to N_4). Each leaf node consists of several indoor partitions. Specifically, $N_1 = \{P_1, \cdots, P_4\}$, $N_2 = \{P_5, \cdots, P_7\}$, $N_3 = \{P_8, \cdots, P_{12}\}$, and $N_4 = \{P_{13}, \cdots, P_{17}\}$. The leaf nodes are iteratively merged until root node is formed, e.g., N_1 and N_2 are merged to form N_5 whereas N_3 and N_4 are merged to form N_6.

Definition 3.4.1. 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. 4.1, 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. 3.3 shows the access doors of each node in the boxes below the nodes, e.g., $AD(N_1) = \{d_1, d_6\}$ and $AD(N_5) = \{d_1, d_7, d_{10}\}$. Note that the shortest path to/from a point s in N_1 to/from a point t outside of N_1 must pass through one of its access doors d_1 and d_6.

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

Distance Matrix for N_5

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

Distance Matrix for N_7

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

Distance Matrix for N_1

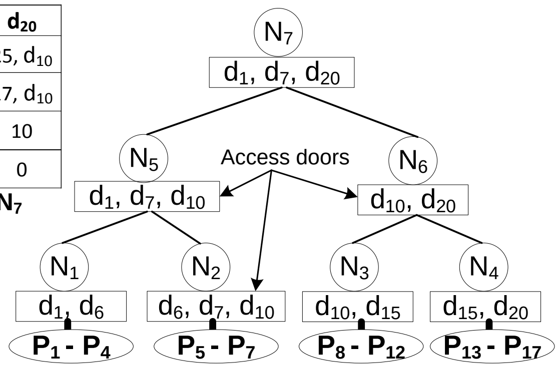


Figure 3.3: Indoor Partitioning Tree

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

Distance matrices for leaf nodes. For each leaf node N, the distance matrix stores distances between every door $d_i \in N$ to every access door $d_j \in AD(N)$. Fig. 3.3 shows an example of the distance matrix for the node N_1 where the distances between every door $d_i \in N_1$ (i.e., d_1 to d_6) and every access door $d_j \in AD(N_1)$ (i.e., d_1 and d_6) are stored.

To support the shortest *path* queries, the distance matrix also stores some additional information. Specifically, for a leaf node N, in addition to the shortest distance between $d_i \in N$ and $d_j \in AD(N)$, the distance matrix also stores a door d_k on the shortest path from d_i to d_j. d_k is called the next-hop door for the entry corresponding to d_i and d_j. Specifically, if the shortest path from d_i to d_j lies entirely inside the node N then d_k corresponds to the first door on the shortest path from d_i to d_j. In Fig. 4.1, 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. 3.3), 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. 4.1 does not have an example of it), this is critical to efficiently retrieve the shortest path between two points. We give a detailed example and reasoning of this later in Section 3.5.2. Finally, if the shortest path between d_i and d_j does not involve any other door (e.g.,

d_5 to d_6), the next-hop door is set as `NULL`. For better readability, the matrices in Fig. 3.3 show only non-null values.

Distance matrices for non-leaf nodes. Consider a non-leaf node N that has f children $N_1, N_2, \cdots, N_f$. The distance matrix of N stores distances between every access door of its children, i.e., it stores distances between all doors in $\cup_{i=1}^{f} AD(N_i)$. For example, in Fig. 3.3, 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. 3.3, 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.

Constructing IP-Tree

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

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

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

ii. Merging of a partition with a leaf node is not allowed if the merging will result in 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 1 : In Fig. 4.1, the partitions P_2 and P_3 are combined with the hallway partition P_1. The partition P_4 could be combined with either P_5 or P_1 because both P_1 and P_5 have exactly 1 common door with P_4 and are on the same floor. We assume that it is combined with P_1. Thus, P_1 to P_4 are combined to form the leaf node N_1. Note that the hallway P_5 cannot be included in the leaf node N_1 because doing so would violate the rule ii. The partitions P_6 and P_7 are combined with P_5 to form a leaf node N_2. Similarly, P_8 to P_{12} are combined to form the node N_3 and P_{13} to P_{17} are combined to construct the leaf node N_4. The algorithm stops because no further merging is possible without violating rule ii. ■

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

We define degree of a node N_i at level $l+1$ to be the number of level l nodes contained in N_i. A min-heap H is initialized by inserting all nodes in $\mathcal{N}_l$ and the key for each node is set to its degree initialized to one because no level l nodes are merged yet (line 1). If two nodes have the same degree, the heap prefers the node which has smaller number of adjacent nodes. This is because some nodes can only be merged with exactly one other node and such nodes should be given preferences in merging, e.g., in Fig. 4.1 and Fig. 3.3, N_1 is merged with N_2 and N_4 is merged with N_3 because both N_1 and N_4 can only be merged with exactly one other node.

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

Input : $\mathcal{N}_l$: nodes at the current level l, t: minimum degree
Output : $\mathcal{N}_{l+1}$: nodes at the next level $l+1$

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

The nodes are iteratively de-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 of at least t (line 2). This implies that every node in the heap contains at least t level l nodes, i.e., at least t children.

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

EXAMPLE 2 : To create the distance matrix of leaf node N_1 that contains doors d_1 to d_6, we first issue a Dijkstra's search starting at d_1 on the graph shown in Fig. 3.2(a) and expand the search until all doors d_1 to d_6 are reached. The distances and next-hop doors are populated in the distance matrix row corresponding to the door d_1. The same process is repeated for the other access door d_6. ■

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

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

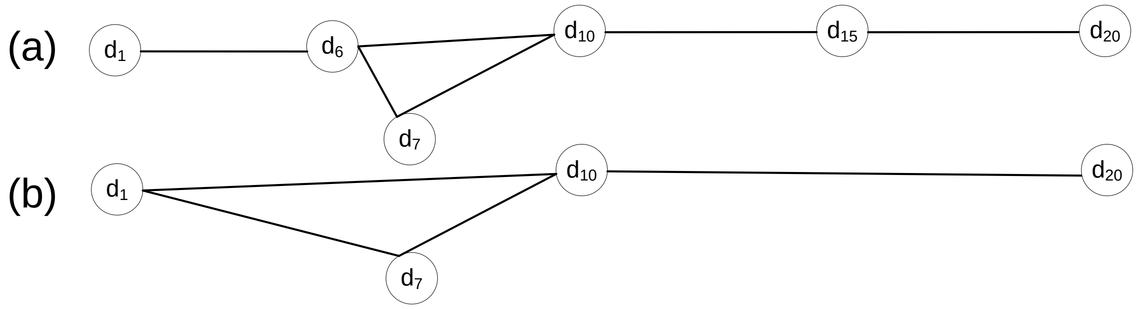


Figure 3.4: (a) $\mathcal{G}_2$: level-2 graph; (b) $\mathcal{G}_3$: level-3 graph

Fig. 3.4 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. 3.3, we use the graph in Fig. 3.4(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. 3.4(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. 3.3). Similarly, to construct the distance matrices of level 3 nodes, we use the graph shown in Fig. 3.4(b) where the vertices of the graph are the access doors of level 2 nodes.

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

EXAMPLE 3 : To construct the distance matrix of node N_5, the graph shown in Fig. 3.4(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. 3.4(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. ■

Storage Complexity

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

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

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

3.4.2 Vivid IP-Tree (VIP-Tree)

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

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

3.4.3 Discussions

Index update The conditions of the doors may change at different time periods. For example, some entrances of shopping malls will be closed after trading hours, or the doors will be closed

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

during the renovation. In order to handle these circumstances, our proposed indexes have to be updated accordingly. Two updates are required which are insertion and deletion. When a new door d_i is available, we have to insert this door to the proposed index. First, d_i is located to the leaf node. After that, we have to check whether d_i is an access door. If d_i is not an access door, changes only need to be made for the leaf node by adding d_i to the distance matrix and computing the distances. On the other hand, if d_i is an access door, the nodes that consider d_i as an access door have to be updated accordingly since d_i will affect the pre-computed distances. Meanwhile, the D2D graph of the highest level node affected by d_i has to be examined since adding d_i may affect the distances in the D2D graph. If it does affect the pre-computed distances, then updating the higher level nodes are required until the D2D graph on current level is not affected. Because the distance computations are very efficient for IP-Tree and VIP-Tree, tree update is very fast according to the experimental results. For deletion, similar process can be applied.

Directed doors In some indoor venues, doors are considered as directed doors. For example, in the airport, passengers can enter the door in security checkpoint, but they cannot go back through the same door. In order to handle directed doors, we revise the distance matrix in our proposed indexes. Assume d_i and d_j are two doors in the same partition, users can only enter through d_i and leave the partition through d_j. For pre-compute distance from d_i to d_j, it is the actual indoor distances between two doors, however, for pre-computed distances from d_j to d_i, it is set to be the distance of the path passing through the space outside the partition. If no path exists, the distance is set to be infinite.

3.5 Indoor Query Processing

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

3.5.1 Shortest Distance Queries

Shortest Distance Using IP-Tree

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

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

Shortest distance between s and an access door $d \in$ AD(`Leaf(s)`). In this chapter, an access door d of `Leaf(s)` that is also a door of `Partition(s)` is called a local access door of `Partition(s)`. If the access door d is not a door of `Partition(s)`, it is called a global access door for `Partition(s)`. Fig. 3.5(a) shows the leaf node N_1 which has two access doors d_1 and d_6. d_1 is a local access door of P_1 and d_6 is a global access door of P_1.

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

$$dist(s, d) = min_{\forall d_i \in \texttt{Partition(s)}} dist(s, d_i) + dist(d_i, d) \tag{3.1}$$

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

Definition 3.5.1. Superior door: Let P be a partition and `Leaf(P)` be the leaf node containing the partition P. A door $d_i \in P$ is called a superior door of P if either i) d_i is a local access door of P or ii) there exists a global access door d_j such that the shortest path from d_i to d_j does not pass through any other door of the partition P.

The doors that are not superior are called inferior doors. Consider the example of Fig. 3.5(a) that shows a leaf node containing partitions P_1 to P_4. The access doors of the node are d_1 and d_6 where d_1 is the local access door of P_1 and d_6 is its global access door. The superior doors of the partition P_1 are d_1 and d_5. d_1 is the superior door because it is a local access door of the partition. d_5 is a superior door because the shortest path from d_5 to the global access door d_6 does not pass through any other door. The doors d_2, d_3 and d_4 are the inferior doors for partition P_1.

For example, the door d_2 is an inferior door because the shortest path from d_2 to the global access door d_6 passes through at least one other door of the partition P_1.

Intuitively, the shortest path from any point $s \in P$ to any global access door d_j must pass through one of the superior doors of P. Therefore, we only consider the superior doors in Eq. 3.1. In the example of Fig. 3.5(a), the shortest path from $s \in P$ to d_6 must pass through one of its superior doors (d_1 or d_5). Hence, $dist(s, d_6) = min(dist(s, d_1) + dist(d_1, d_6), dist(s, d_5) + dist(d_5, d_6))$.

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

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

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

Lemma 3.5.1. *Let N_{parent} be the current node being processed and N_{child} be its child node. Let d be an access door of N_{parent}. The shortest path for a point $s \in N_{child}$ to d must pass through at least one access door of N_{child}.*

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

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

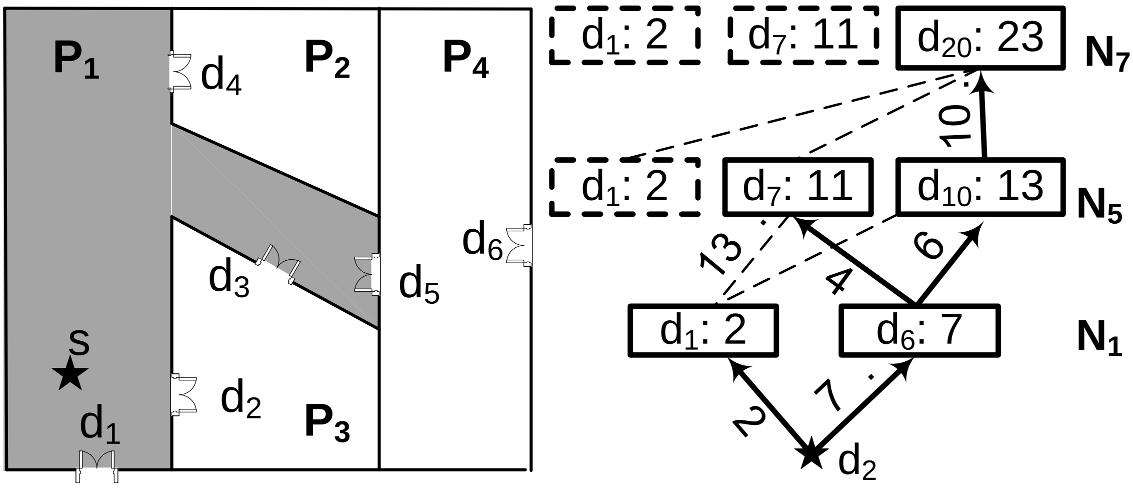
Algorithm 2: getDistances (s, N)

Input : s: source, N: an ancestor node of $Leaf(s)$
Output : $Distances$: shortest distance between s and every $d \in AD(N)$

Initialize N_{parent} to be the parent node of $Leaf(s)$;
Initialize N_{child} to $Leaf(s)$;
while N_{child} is not the same as N **do**
  for each unmarked $d \in AD(N_{parent})$ **do**
    $dist(s,d) = min_{\forall d_i \in AD(N_{child})} dist(s,d_i) + dist(d_i,d)$;
    mark d and then insert $dist(s,d)$ in $Distances$ if $d \in AD(N)$;
  $N_{child} \leftarrow N_{parent}$;
  $N_{parent} \leftarrow$ parent node of N_{parent};

$$dist(s,d) = min_{\forall d_i \in AD(N_{child})} dist(s,d_i) + dist(d_i,d) \tag{3.2}$$

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



(a) Superior doors

(b) Illustration of Algorithm 2

Figure 3.5: Shortest distance computation

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

EXAMPLE 4 : Consider the example of Fig. 4.1 and Fig. 3.3 and assume that we want to compute the shortest distances between d_2 and every access door of the root node N_7 (i.e., d_1, d_7 and d_{20}). `Leaf(`d_2`)` is the node N_1. The algorithm assumes that $dist(s,d)$ for every access door d of `Leaf(s)` has been computed as described above. For example, $dist(d_2,d_1) = 2$ and $dist(d_2,d_6) = 7$ have been computed (see Fig. 3.5(b)).

N_{parent} is initialized to be the parent node of N_1 (i.e., N_{parent} is N_5). The shortest distance to each access door in N_5 (e.g., d_1, d_7 and d_{10}) is then computed based on the distances from d_2 to the access doors in N_1. For instance, $dist(d_2,d_7) = min(dist(d_2,d_1) + dist(d_1,d_7), dist(d_2,d_6) + dist(d_6,d_7)) = min(2+13, 7+4) = 11$. Fig. 3.5(b) illustrates the processing of the algorithm where the incoming edges (thick arrows and broken lines) to a door demonstrate a possible path to the door and the thick arrows show the path that lead to minimum distance, e.g., d_7 has two incoming edges: one from d_1 and the other from d_6. The shortest distance is $dist(d_2, d_6) + dist(d_6, d_7) = 7 + 4 = 11$ and the edge between d_6 and d_7 is shown using a solid arrow. Similarly, $dist(d_2, d_{10}) = min(dist(d_2, d_1) + dist(d_1, d_{10}), dist(d_2, d_6) + dist(d_6, d_{10})) = 13$.

After $dist(s, d)$ is computed for every access door d of N_{parent}, the algorithm iteratively retrieves the parent node of N_{parent} to compute distances from s to its access doors (see lines 7 and 8). In Fig. 3.5(b), N_7 becomes N_{parent} and N_5 becomes N_{child} and the distances to the access doors of N_7 are computed using the previously computed distances to the access doors of N_5. For example, $dist(d_2, d_{20})$ is the the minimum of $dist(d_2, d_1) + dist(d_1, d_{20})$, $dist(d_2, d_7) + dist(d_7, d_{20})$ and $dist(d_2, d_{10}) + dist(d_{10}, d_{20})$. The thick arrows show the shortest path from d_2 to each access door.

If $dist(s, d)$ for a door d in N_{parent} is already known because d is also an access door for N_{child}, its distance is not needed to be recomputed. Fig. 3.5(b) shows such doors in a rectangle drawn in broken lines, e.g., $dist(d_2, d_1)$ is computed at node N_1 and it does not need to be recomputed when nodes N_5 and N_7 are accessed. In Algorithm 2, we mark each door d for which $dist(s, d)$ has been computed (line 6) and only compute the distances from s to the doors that are not marked (line 4). ■

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

Lemma 3.5.2. *Let $LCA(s, t)$ be the lowest common ancestor node of* `Leaf(s)` *and* `Leaf(t)`. *Let N_s (resp. N_t) be the child of $LCA(s, t)$ which is an ancestor of* `Leaf(s)` *(resp.* `Leaf(t)`*). The shortest path from s to t must path through at least one access door of N_s and at least one access door of N_t.*

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

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

Consider the example of Fig. 4.1 and Fig. 3.3 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. 4.1 passes through d_{10} which is an access door for both N_5 and N_6.

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

$$dist(s,t) = min_{\forall d_i \in AD(N_s), \forall d_j \in AD(N_t)} dist(s,d_i) + dist(d_i,d_j) + dist(d_j,t) \tag{3.3}$$

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

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

```
N_s ← ancestor of Leaf(s) and a child of LCA(Leaf(s), Leaf(t));
N_t ← ancestor of Leaf(t) and a child of LCA(Leaf(s), Leaf(t));
getDistances(s, N_s);                    /* Algorithm 2 */;
getDistances(t, N_t);                    /* Algorithm 2 */;
return min_{∀d_i∈N_s,∀d_j∈N_t} dist(s, d_i) + dist(d_i, d_j) + dist(d_j, t)
```

Complexity Analysis. First, we evaluate the cost of Algorithm 2. Let ρ be the average number of access doors in a node. To compute the distance from s to a door d in a node N_{parent}, the algorithm considers paths through all access door in the child node N_{child} (see Eq. 3.2). Hence, the cost to compute the distance of one door at node N_{parent} is $O(\rho)$ assuming that distances to every access door in N_{child} are known. Hence, the total cost to compute distances from s to all doors in a node N_{parent} is $O(\rho^2)$. Let h be the number of nodes between `Leaf(s)` and the node N. The total cost for computing distances from s to every $d \in AD(N)$ is $O(h\rho^2)$.

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

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

Shortest Distance Using VIP-Tree

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

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

3.5.2 Shortest Path Queries

Shortest Path Using IP-Tree

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

During the shortest distance computation (Algorithm 3), we maintain the intermediate doors on the path accessed by the algorithm. This gives a partial shortest path. For example, in the example of Fig. 3.5(b), the partial shortest path from d_2 to d_{20} is $d_2 \rightarrow d_6 \rightarrow d_{10} \rightarrow d_{20}$ (see thick arrows). Next, we describe how to decompose these edges to recover the complete shortest path.

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

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

1 **if** d_i and d_j both are non-access doors **then**
2     $d_i \rightarrow d_j$ is a final edge; `/* Lemmas 3.5.4 and 3.5.6 */;`
3 **else**
4   **if** d_i and d_j both are access doors **then**
5         $N \leftarrow$ the lowest common ancestor of d_i and d_j;
6   **else** `// only one of` d_i `and` d_j `is access door`
7         $N \leftarrow$ leaf node containing d_i & d_j; `/* Lemmas 3.5.4 and 3.5.7 */;`
8   Let d_k be the next-hop door of d_i and d_j in the distance matrix of N;
9   **if** d_k is NULL **then**
10         $d_i \rightarrow d_j$ is a final edge; `/* Lemma 3.5.3 */;`
11   **else**
12   Return $d_i \rightarrow d_k \rightarrow d_j$;

If both d_i and d_j are non-access doors then it can be proved that $d_i \rightarrow d_j$ is a final edge (Lemmas 3.5.4 and 3.5.6 in Section 3.5.2). Note that $d_i \rightarrow d_j$ is either an edge returned by Algorithm 3 or an edge resulting from decomposition of another edge by Algorithm 4. The proof is non-trivial and is given in the next section.

If both d_i and d_j are the access doors (line 4 of Algorithm 4), we will use the distance matrix of the lowest common ancestor node N of `Leaf(`d_i`)` and `Leaf(`d_j`)`. Otherwise, if only one of d_i and d_j is an access door, we will use the distance matrix of the leaf node N that contains both d_i and d_j. Lemmas 3.5.4 and 3.5.7 in the next section prove that, for each such edge $d_i \rightarrow d_j$ considered by Algorithm 4, we can always find both d_i and d_j in the same leaf node N.

Let N be the node as described above. We look up the distance matrix of N and retrieve the next-hop door d_k for the entry corresponding to d_i and d_j. The shortest path $d_i \rightarrow d_j$ is then decomposed to $d_i \rightarrow d_k \rightarrow d_j$. If d_k is `NULL` then $d_i \rightarrow d_j$ is a final edge and does not need to be decomposed (as we prove later in Lemma 3.5.3).

EXAMPLE 5 : Suppose we want to decompose $d_{10} \rightarrow d_{20}$. The lowest common ancestor of d_{10} and d_{20} is N_6 (see Fig. 3.3). The next-hop door for d_{10} and d_{20} in the distance matrix of N_6 is d_{15}. Therefore, $d_{10} \rightarrow d_{20}$ is decomposed into $d_{10} \rightarrow d_{15} \rightarrow d_{20}$. The algorithm then tries to decompose $d_{10} \rightarrow d_{15}$ using the lowest common ancestor N_3 of d_{10} and d_{15}. The next-hop door of d_{10} and d_{15} in the distance matrix of N_3 is `NULL`. Therefore, $d_{10} \rightarrow d_{15}$ is a final edge. Similarly, $d_{15} \rightarrow d_{20}$ is also a final edge.

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

A key property of Algorithm 4 is that if only one of d_i and d_j is an access door (see line 6) then there always exists a leaf node N that contains both d_i and d_j. We prove this later in Section 3.5.2. This property is made possible due to the special way we store next-hop door d_k for leaf nodes. Specifically, recall that if the shortest path from d_i to d_j passes outside of the leaf node N then next-hop door d_k is not any ordinary first door on the shortest path from d_i to d_j but d_k is the first access door on the shortest path from d_i to d_j. As shown in the next example, the above property cannot be ensured if d_k is not selected this way.

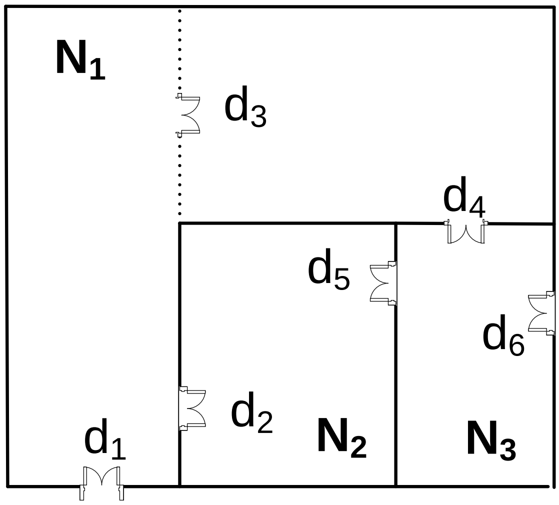


Figure 3.6: Choosing next-hop door for leaf nodes

EXAMPLE 6 : Consider the example of Fig. 3.6 that shows three leaf nodes N_1, N_2 and N_3. Suppose that we are creating the distance matrix of leaf node N_2 that contains two access doors d_2 and d_5. Assume that the shortest path from d_2 to d_5 is $d_2 \rightarrow d_3 \rightarrow d_4 \rightarrow d_5$ due to some obstacles inside N_2. Note that d_3 is the first door on the shortest path. If we choose d_3 as the next-hop door, the edge $d_2 \rightarrow d_5$ will be decomposed into $d_2 \rightarrow d_3 \rightarrow d_5$. Now, if we try to decompose $d_3 \rightarrow d_5$, there does not exist any leaf node that contains both d_3 and d_5 (d_3 is a non-access door and d_5 is an access door). Hence, Algorithm 4 will fail to decompose it. To address this, we choose d_4 as the next-hop door which is the first access door on the shortest path. Hence, $d_2 \rightarrow d_5$ is decomposed to $d_2 \rightarrow d_4 \rightarrow d_5$. Note that d_2, d_4 and d_5 all are access doors and each edge can be further decomposed using the distance matrix of the least common ancestor node (at line 4). ■

Proof of correctness

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

Lemma 3.5.3. *The next-hop door d_k for d_i and d_j in the distance matrix of N can only be* `NULL` *if $d_i \rightarrow d_j$ is a final edge.*

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

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

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

Lemma 3.5.4. *Let $d_i \rightarrow d_j$ be an edge returned by Algorithm 3. (1) $d_i \rightarrow d_j$ is a final edge if both d_i and d_j are non-access doors; (2) d_i and d_j can both be found in the same leaf node N if only one of the d_i and d_j is an access door.*

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

Note that the shortest path from s to d_j is $s \to d_i \to d_j$ where d_i is a door in `Partition(s)` (see Eq. 3.1). The edge $d_i \to d_j$ contains one access door (d_j) and it is easy to see that both d_i and d_j are in the same leaf node `Leaf(s)` - this proves (2). Now, we prove (1) by showing that every edge on the shortest path from s to d_i is a final edge. Recall that we compute the shortest path between two points in the same leaf node using a Dijkstra's like expansion on the original D2D graph. Hence, every edge on the shortest path from $s \to d_i$ is a final edge. □

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

Lemma 3.5.5. *Assume we decompose $d_i \to d_j$ into $d_i \to d_k \to d_j$ as described in Algorithm 4. If N is a non-leaf node then d_i, d_k and d_j all are access doors.*

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

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

Lemma 3.5.6. *Assume we decompose $d_i \to d_j$ into $d_i \to d_k \to d_j$ as described in Algorithm 4. Each edge in $d_i \to d_k \to d_j$ satisfies the following: if both doors in the edge are non-access doors then the edge is a final edge.*

Proof. As stated in Lemma 3.5.5, if N is a non-leaf node then d_i, d_k and d_j all are access doors and this lemma is not applicable. Therefore, this lemma only applies when N is a leaf node.

Since at least one of d_i and d_j is an access door for each *partial* edge $d_i \rightarrow d_j$ considered by Algorithm 4 (Lemma 3.5.4), this lemma is only applicable to either $d_i \rightarrow d_k$ (assuming d_i is a non-access door) or $d_k \rightarrow d_j$ (assuming d_j is a non-access door). Without loss of generality, assume that d_i is a non-access door. The lemma is not applicable to $d_k \rightarrow d_j$ because d_j is an access door. We prove the lemma for $d_i \rightarrow d_k$.

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

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

Lemma 3.5.7. *Assume we decompose $d_i \rightarrow d_j$ into $d_i \rightarrow d_k \rightarrow d_j$ as described in Algorithm 4. Each edge in $d_i \rightarrow d_k \rightarrow d_j$ satisfies the following: if only one of the doors is an access door then both doors can be found in the same leaf node.*

Proof. As stated in Lemma 3.5.5, if N is a non-leaf node then d_i, d_k and d_j all are access doors and this lemma is not applicable. Therefore, this lemma only applies when N is a leaf node.

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

Complexity Analysis

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

3.5.3 Shortest Path Using VIP-Tree

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

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

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

3.5.4 Querying Indoor Objects

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

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

The algorithm requires computing $mindist(q, N)$ for different nodes in the tree. $mindist(q, N)$ is the minimum distance from the query q to any point in the node N. $mindist(q, N)$ is zero if q is in a partition contained in the sub-tree of the node N. If N does not contain q, then

Algorithm 5: k Nearest Neighbors

Input : q: query point, k
Output : kNNs

$d^k = \infty$; /* d^k is distance to current k^{th}NN */;
getDistances(q,root); /* Algorithm 2 */;
Initialize a heap H with root of the tree;
while H is not empty **do**
  de-heap a node N from heap;
  if $mindist(q, e) > d^k$ **then**
    return kNN;
  if N is a non-leaf node **then**
    for each child N' of N **do**
      if N' contains objects **then**
        insert N' in heap with $mindist(q, N')$;
  else
    Use objects in N to update kNN and d^k;

$mindist(q, N)$ is the minimum distance from q to an access door of the node N, i.e., $mindist(q, N) = min_{\forall d \in AD(N)} dist(q, d)$. A straightforward way to compute $mindist(q, N)$ is to use Algorithm 3. Next, we show that we could optimize $mindist(q, N)$ for branch and bound algorithms because these algorithms access the nodes in a particular order.

Lemma 3.5.8. *Let N_1 and N_2 be the two sibling nodes. If N_1 contains q then $dist(q, d_i)$ for any access door $d_i \in AD(N_2)$ is $min_{\forall d_j \in AD(N_1)} dist(q, d_j) + dist(d_i, d_j)$.*

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

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

Lemma 3.5.9. *If N_1 does not contain q and N_2 is a child of N_1 then $dist(q, d_i)$ for any access door $d_i \in AD(N_2)$ is $min_{\forall d_j \in AD(N_1)} dist(q, d_j) + dist(d_i, d_j)$.*

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

Note that if $dist(q, d_j)$ for every access door d_j of N_1 is already known, then $mindist(q, N_2)$ can be computed in $O(\rho^2)$ using Lemma 3.5.8 or Lemma 3.5.9. Below are the details.

At line 2 of Algorithm 5, we compute distance from q to each access door of the root node by calling Algorithm 2. Note that Algorithm 2 computes distances from q to all access doors of each ancestor node of `Leaf(q)` in the process. We maintain these distances for each ancestor node of `Leaf(q)`. Now, when a child node N' of a node N is to be inserted in the heap at line 11, $mindist(q, N')$ can be computed using either Lemma 3.5.8 or Lemma 3.5.9.

Specifically, if N contains q then this implies that at least one sibling N_{sib} of N' contains q. Since we already know distances from q to every access door of N_{sib} (because it is an ancestor of `Leaf(q)`), Lemma 3.5.8 can be applied to compute $mindist(q, N')$. On the other hand, if N does not contain q then Lemma 3.5.9 is applied. Hence, $mindist(q, N')$ can be easily computed in $O(\rho^2)$ for each node accessed by the algorithm.

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

3.6 Experiments

3.6.1 Experimental Settings

Indoor Space. We use three real data sets: Melbourne Central [6], Menzies building [9] and Clayton Campus [7]. Melbourne Central is a major shopping centre in Melbourne and consists of 297 rooms spread over 7 levels (including ground and lower ground levels). Menzies building is the tallest building at Clayton campus of Monash University consisting of 14 levels (including basement and ground floor) and 1306 rooms. The Clayton data set corresponds to 71 buildings (including multilevel car parks) in Clayton campus of Monash University. We obtained the floor plans of all buildings and manually converted them to machine readable indoor venues. Coordinates of the buildings are obtained by using OpenStreetMap and the sizes of indoor partitions (e.g., rooms, hallways) are determined. A three dimensional coordinate system is used where the first two represent x and y coordinates of indoor entities (e.g., rooms, doors) and the third represents the floor number. For Clayton data set, the D2D graph also contains edges between the entry/exit doors of different buildings where the weight corresponds to the outdoor distance between the doors.

To evaluate the algorithms on even larger data sets, we extend Melbourne Central (denoted as MC), Menzies building (denoted as Men) and Clayton (denoted as CL) by replication. Table

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	1,280	56,009
Men-2	2 times Men	2722	2,560	112,062
CL	Clayton Campus	41,392	41,100	6,700,272
CL-2	2 times CL	83,138	82,540	13,400,884

Table 3.2: Indoor venues used in experiments

4.2 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 4.2 corresponds to the total number of edges in the D2D graph for each indoor space. The distance matrix used by the state-of-the-art indoor technique cannot be built on the venues larger than Men-2.

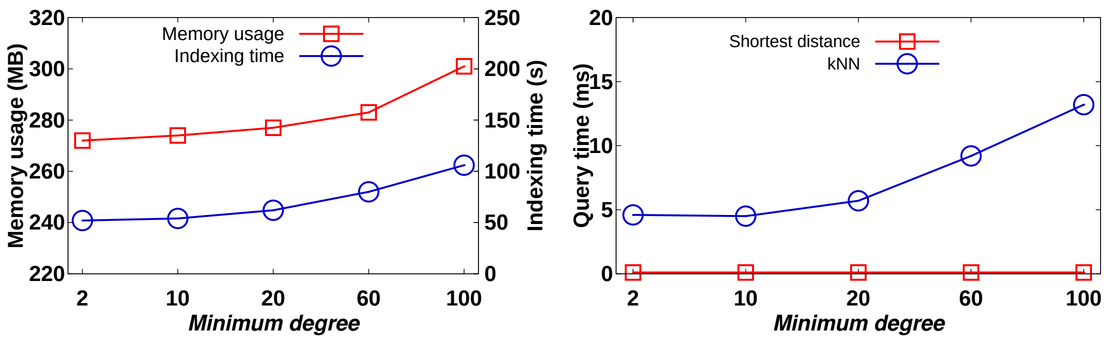
Competitors. All algorithms are implemented in C++ on a PC with 8GB RAM and Intel Core I5 CPU running 64-bit Ubuntu. We compare our proposed indexes (IP-Tree and VIP-Tree) with the following competitors.

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

Distance-aware model (DistAw) [62]. We also compare our algorithm with the state-of-the-art indoor query processing index called distance-aware model (shown as DistAw). For shortest distance/path queries, DistAw uses only an extended graph based on the accessibility base graph. For *k*NN and range queries, DistAw model also proposes to use DistMx to speed up the query processing. In the experiments, we use DistAw++ to denote the algorithm that exploits DistMx (requiring an additional $O(D^2)$ space). We use DistAw to denote the algorithm that does not required DistMx.

ROAD [56] and G-tree [93]. We also compare our algorithm with the state-of-the-art indexes for spatial query processing on road networks (G-tree and ROAD). These indices are constructed by passing the D2D graph as input and the query processing algorithms are adapted to suit indoor query processing. For each indoor venue, we experimentally choose the best value for the parameter τ inG-tree.

Queries and Objects. To evaluate the performance for shortest distance/path queries, $10,000$ pairs of source and target points are randomly generated in the indoor space. To evaluate *k*NN and range queries, 10,000 query points are randomly generated in the indoor space. We use washrooms in the buildings as the objects (e.g., the query is to find the nearest washroom). The number of washrooms in Men-2 is 50. We also generate synthetic object sets consisting of 10, 50, 100 and 500 objects - 50 is the default value. We choose a small set of objects because the kNN queries are more challenging for smaller object sets (as also reported in existing work on road networks [13]). This is because a larger area is to be explored to compute the kNNs when the number of objects is small. Furthermore, we believe that the real world scenarios for kNN queries contain a small number of objects, e.g., ATM machines, washrooms, charging-kiosks etc. k is varied from 1 to 10 and the default value of k is 5. The range is varied from 50 to 1000 meters and the default value is 100 meters. The figures report average query processing cost for each algorithm.

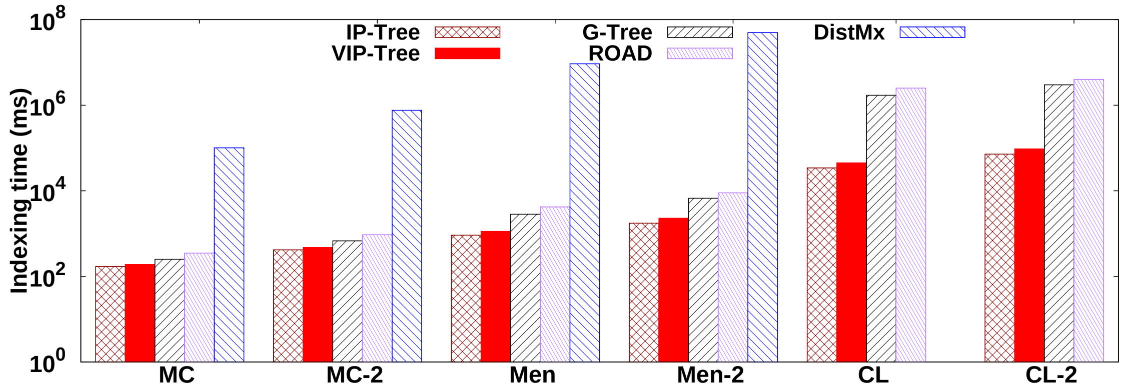


(a) Construction cost (b) Querying cost

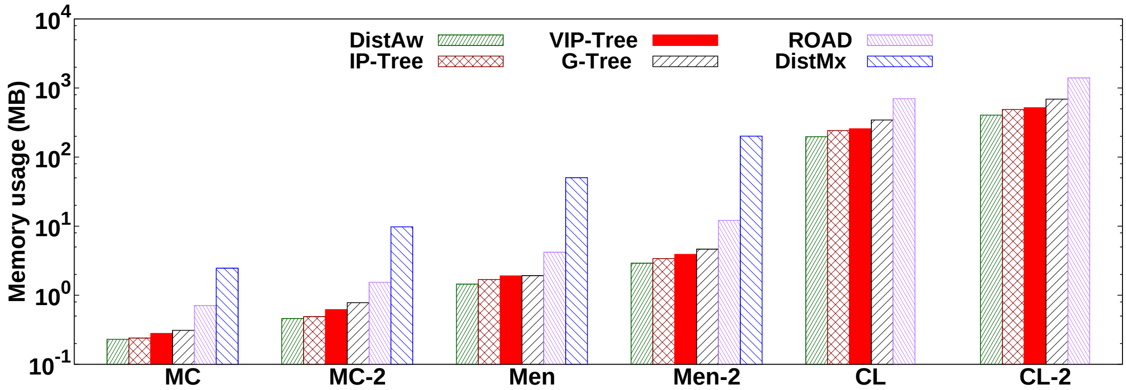
Figure 3.7: Effect of minimum degree t on VIP-Tree

Choosing t for IP-Tree and VIP-Tree. We evaluated the effect of the minimum degree t (see Algorithm 1) on our indexes and found that the best performance is achieved for $t = 2$. Fig. 3.7 shows the index construction cost and query time of VIP-tree on Clayton data set. The construction time and construction cost increases as t increases mainly because the size of distance matrices increases which requires more storage and more computation time to materialize the distances. The size of t does not affect the query time for shortest distance queries mainly because the cost is independent of the height of the tree – recall that VIP-tree computes shortest distance in $O(\rho^2)$ and ρ is not affected by t. The cost of kNN query increases with t mainly because fewer nodes can be pruned when t is large which requires the algorithm to access a larger number of nodes. The trend for IP-tree are similar. In the rest of the experiments, we use $t = 2$ for our indices. We also found that the average number of access doors and superior doors is less than 4 for all data sets and the maximum number is around 8. This provides an insight on why our indices perform exceptionally well for indoor spaces.

3.6.2 Indexing Cost



(a) Construction time



(b) Index Size

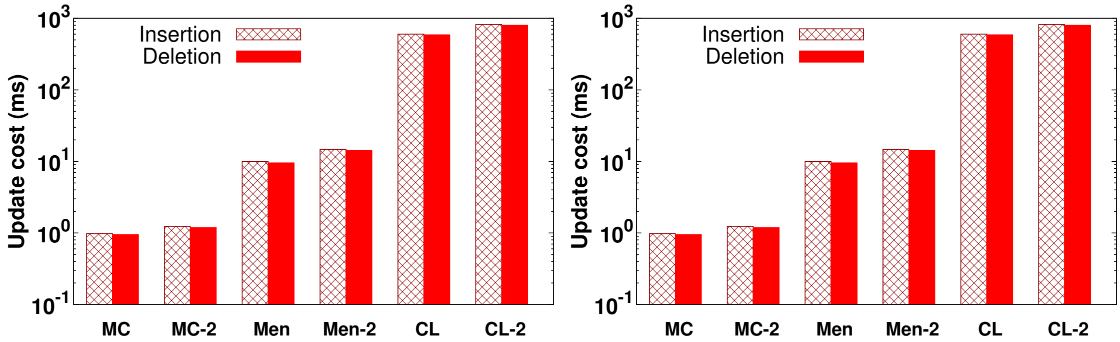
Figure 3.8: Indexing Cost

Construction time. Fig. 3.8(a) compares the time to construct each index using the accessibility base graph and D2D graph. Since DistAw only uses the extended graph based on the accessibility base graph, its index construction is not shown. Note that DistAw++ does use DistMx and its construction cost is the same as DistMx. To construct DistMx, for each door, we use a Dijkstra's like expansion until all other doors in the graph have been marked. This requires $O(D)$ expansions on D2D graph which is quite expensive. Consequently, DistMx has a high construction cost and it took almost 14 hours to construct DistMx for Men-2 with $2,738$ doors requiring computing almost 7.5 million shortest distances/paths.

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

Index size. Fig. 3.8(b) compares the size of different indexes. As expected, DistMx is the largest index. DistAw has the smallest index size because it only needs the extended graph based on the accessibility base graph. IP-Tree, VIP-Tree and G-tree have sizes comparable to DistAw index. The storage cost of VIP-Tree is slightly higher than IP-Tree, which demonstrates that materializing the distances to the access doors of all ancestors nodes does not increase the storage cost dramatically but significantly improves the query processing cost as we show later. G-tree and ROAD consume more space than IP-Tree and VIP-Tree mainly because these are designed for road networks having a small average outdegree (2 to 4) as compared to the D2D graph which has a much higher out-degree (up to 400). This results in a larger number of *border* nodes and hence consuming more space.

Update cost. We evaluate the update cost of our proposed indexes IP-Tree and VIP-Tree according to the door insertion and deletion. Fig. 3.9(a) and 3.9(b) shows the results for the update cost for IP-Tree and VIP-Tree respectively. For the largest dataset CL-2, the update cost for both operations can be done within 1 second, which is very efficient. Meanwhile, for VIP-Tree, the update cost is very similar to IP-Tree that proves the efficiency of distance materialization.



(a) IP-Tree

(b) VIP-Tree

Figure 3.9: Indexing Cost

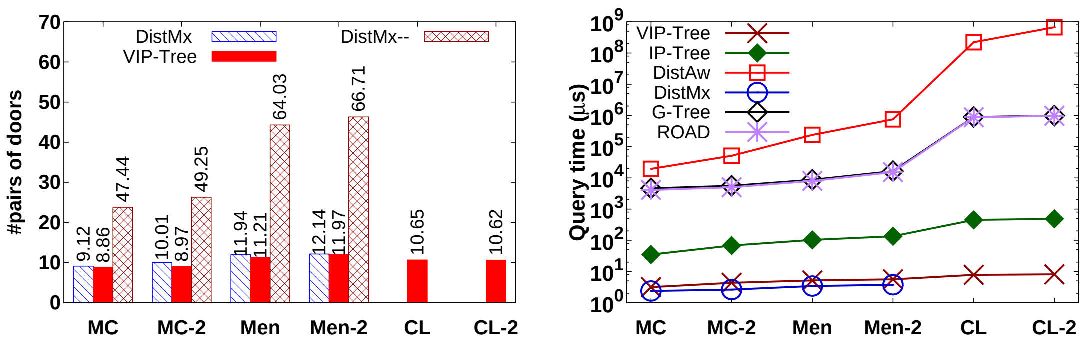
3.6.3 Query Performance

Shortest distance queries

In Fig. 3.10 we evaluate the algorithms for shortest distance queries on different indoor data sets. First, we present a simple optimization to improve the performance of DistMx. A straightforward approach to compute the distance from s to t is to use DistMx to calculate distances between every door d_i in `Partition(s)` and every door d_j in `Partition(t)` and picking the pair d_i and d_j that minimizes $dist(s, d_i) + dist(d_i, d_j) + dist(d_j, t)$. Let D_s and D_t be the number of doors in `Partition(s)` and `Partition(t)`, respectively. This requires checking $D_s \times D_t$

pairs of doors to retrieve the shortest distance and the cost may be high if $D_s \times D_t$ is large. A simple optimization is to ignore the doors in `Partition(s)` and `Partition(t)` that lead to no-through partitions.

The above optimization significantly reduces the pairs of doors that need to be considered. Fig. 3.10(a) shows the effect of this optimization where DistMx uses this optimization and DistMx-- does not use this optimization. The numbers on top of bars correspond to the number of pairs needed to be considered by each algorithm. As can be seen, this simple optimization significantly reduces the number of pairs and improves the performance of DistMx by up to several times. In the rest of the experiments, we use this optimization for DistMx. The numbers for VIP-Tree correspond to the pair of superior doors to be considered. This number is slightly smaller than the number of pairs considered by DistMx but the cost is slightly higher because VIP-Tree needs to first compute distances from s and t to the access doors of the children of lowest common ancestor which requires more computation.



(a) Optimizing DistMx (b) Comparing all algorithms

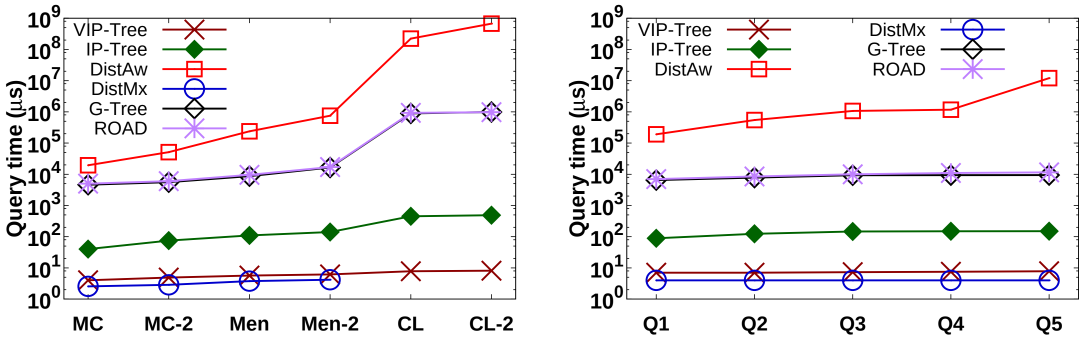
Figure 3.10: Shortest Distance Queries

Fig. 3.10(b) compares the performance of all techniques for shortest distance queries. Since DistMx returns distance between any two doors in the graph in $O(1)$, it gives the best performance. However, VIP-Tree provides a comparable performance. Note that DistMx has quadratic storage cost and huge construction cost. Recall that we were not able to construct DistMx for indoor venues larger than Men-2. VIP-Tree significantly outperforms IP-Tree at the expense of a slightly higher storage cost. Both VIP-Tree and IP-Tree outperform the other three techniques by several order of magnitude, e.g., for CL-2 data set, VIP-Tree processes a shortest distance query in around 10 microseconds as compared to ROAD and G-tree that take almost one second to answer a single shortest path query.

Shortest path queries

Fig. 3.11 compares the techniques for shortest path queries. We note that the overhead of recovering shortest paths is negligible, i.e., for each algorithm, the cost of shortest distance queries is similar to the cost of shortest path queries (compare Fig. 3.10(b) and Fig. 3.11(a)).

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

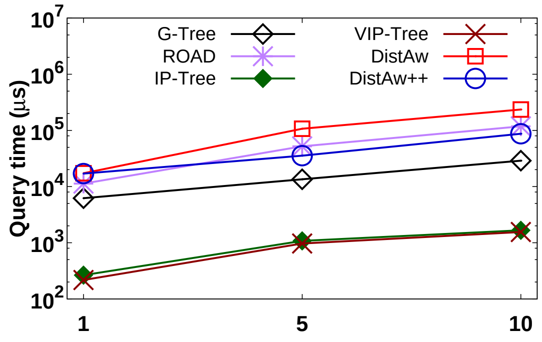


(a) Different buildings

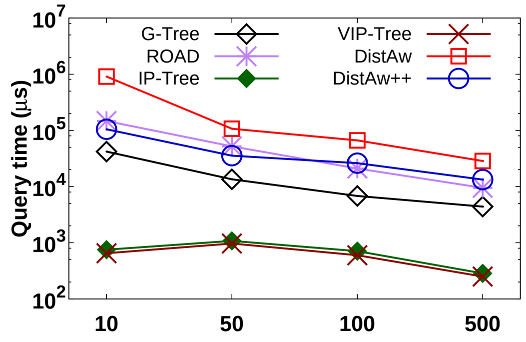
(b) Effect of distance b/w s and t

Figure 3.11: Shortest Path Queries

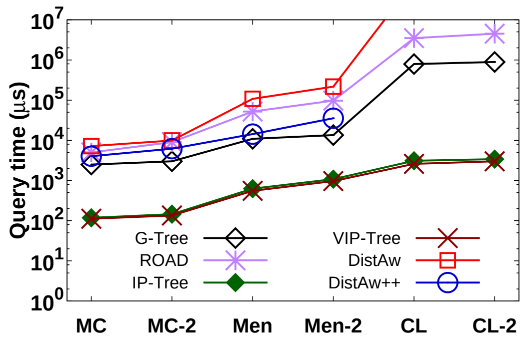
Fig. 3.11(b) shows the effect of distances on the performance of different algorithms. The cost of DistAw increases by almost two orders of magnitude as the distance increases. The cost for IP-Tree slightly increases from $Q1$ to $Q3$ because the lowest common ancestor is at a higher level when source and target are further from each other. This requires visiting more levels of the tree resulting in an increased cost. However, the cost does not increase further for $Q4$ and $Q5$ because, in most of the cases for $Q3$, the lowest common ancestor is already the root node. A similar behavior can be observed for G-tree and ROAD. The effect of distance is negligible on DistMx and VIP-Tree because these algorithms require retrieving relevant entries from the distance matrices which is independent on the distances between the source and target points. A similar trend was observed for shortest distance queries.



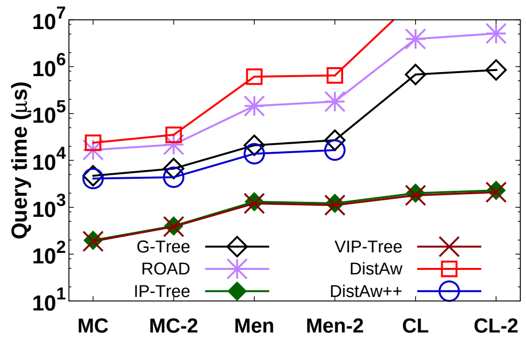
(a) Effect of k (kNN query)



(b) # of objects (kNN query)



(c) Indoor venues (kNN query)



(d) Indoor venues (range query)

Figure 3.12: kNN and Range Queries

Querying Indoor Objects

kNN Queries. Fig. 3.12(a), Fig. 3.12(b) and Fig. 3.12(c) evaluate different algorithms by varying k, the number of objects, and the indoor buildings, respectively. VIP-Tree and IP-Tree perform equally well. This is because IP-tree computes $mindist(q, N)$ for a node N with the same complexity as that of VIP-Tree due to the optimizations presented in Section 3.5.4. Both VIP-Tree and IP-Tree outperform the other algorithms by several orders of magnitude. Note that DistAw++ is the existing method that utilizes DistMx to speed up the query processing. Nevertheless, it is outperformed by our proposed techniques.

Fig. 3.12(b) shows that the cost of all algorithms decreases as the number of objects increases. This is because kNNs can be found closer to the query point as the number of objects increases. Hence, the algorithms require exploring a smaller area. On the other hand, the query processing cost increases for all algorithms as the value of k or the data set size increases.

Range Queries. Fig. 3.12(d) evaluates the performance of different techniques for range queries. The cost of all algorithms increases with for larger venues mainly because the sizes of the indexes increase. VIP-Tree and IP-Tree both perform equally well and outperform the other competitors by several orders of magnitude.

3.7 Conclusion

In this chapter, we propose two novel indexes, IP-Tree and VIP-Tree, for efficiently processing indoor spatial queries. We also present efficient algorithms to answer shortest path queries, shortest distance queries, k nearest neighbors queries and range queries. IP-Tree and VIP-Tree have low storage requirement, small 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.

Chapter 4

Indoor Trip Planning Queries

4.1 Overview

Location-based services (LBS) are applications that allow mobile users to search for nearby points of interest and are valuable in many domains, such as building emergency services, recommendation systems, and navigation systems. To enable LBSs, GPS technology is used to detect the user locations in outdoor, but it cannot be applied in indoor as an indoor space contains different levels. There have been breakthroughs in indoor positioning technologies [58] that can locate the user locations in indoor spaces. Consequently, indoor LBSs are expected to be booming in the coming years [82, 3, 4].

One of the well known LBSs is Trip Planning Queries (TPQ) that enables users to visit their desired places with a minimum travel distance. For its counterpart in indoor spaces, indoor TPQ ($iTPQ$) is valuable as well since the recent research shows that humans spend more than 85% of their time in indoor spaces, such as houses/apartments, office buildings and shopping centres [44]. Fig. 4.1 shows an sample indoor floor plan containing 17 indoor partitions (P_1 to P_{17}). In these partitions, nine indoor points belonging to three categories (e.g. ATMs) are located in C_1=$\{p_1, p_2, p_3\}$, C_2=$\{p_4, p_5, p_6\}$, and C_3=$\{p_7, p_8, p_9\}$. Let the two stars p_s and p_t be the start and end points of a trip; an $iTPQ$ is to find the shortest route starting from p_s, passing through only one point in each category, and reaching to p_t. Thus, a possible route can be $\{p_s \rightarrow p_1 \rightarrow p_4 \rightarrow p_7 \rightarrow p_t\}$.

Take people's daily shopping as an example. A user wants to buy *milk*, *flowers* and *oranges* after he finishes his work. He drives to the car parking in a shopping centre. Starting from his current location, he wants to buy all the items and come back to his car using a shortest walking

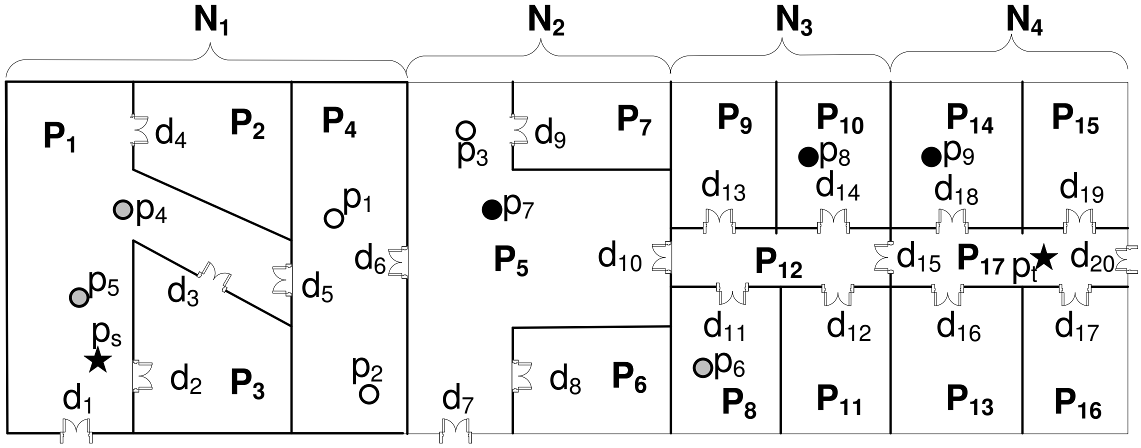


Figure 4.1: An indoor venue containing 17 partitions, 20 doors and 9 points in 3 categories

distance. Hence, an *i*TPQ helps him to plan a shortest route. Another example could be borrowing books from a library. A student wants to borrow a *math* book, a *science fiction* and a *Japanese cartoon*. An *i*TPQ provides the shortest route for him to get all these three books and return to the reception.

*i*TPQ will be a critical part in indoor LBSs. The problem is that existing technologies focus on outdoors, and they do not offer efficient processing time for *i*TPQ. In indoor spaces, no solution has been proposed to deal with *i*TPQ.

TPQ is one of the classical problems in spatial road networks, which already has a number of solutions [59, 20]. Unfortunately, no existing technique is available for indoor spaces. If we apply any existing TPQ techniques from spatial road networks to indoor spaces, we must adopt a spatial road network kind of graphs for an indoor setting. In our previous paper [75], we have shown that if this were to be done, the performance to compute shortest paths in indoor spaces would degrade significantly. Consequently, to address the problem, we adopt VIP-Tree [75] in this chapter.

In outdoor spaces, most of the research aims to find a heuristic solution for TPQ. One method focusing on the exact solution is the Progressive Neighbor Exploration (PNE) method [77]. It was designed to solve the Optimal Sequenced Route (OSR) queries (Note that OSR is a variant of TPQ which defines the visiting order of categories). However, for OSR queries, PNE is not efficient in terms of processing time, while TPQ is more complex compared with OSR. Thus, the revised PNE to handle *i*TPQ is not efficient that is proved in our experimental section[1].

This chapter is organized as follows. Section 4.2 gives the formal definition of indoor trip planning queries followed by the contributions presented in Section 4.3. A brute-force method is discussed in Section 4.4. We present our main method in Section 4.5. In Sections 4.6 and 4.7, the

[1]Published in The Computer Journal

pruning techniques in both pre-processing and query processing are proposed. The techniques are evaluated in Section 4.8 and a conclusion is given in Section 4.9.

4.2 Background Information

4.2.1 Problem Definition

Given a set of n indoor points $V = \{p_1, p_2, ..., p_n\}$ and a set of m categories $\mathcal{C} = \{C_1, C_2, ..., C_m\}$, a mapping function π: $p_i \longrightarrow C_j$ maps each indoor point $p_i \in V$ to a category $C_j \in \mathcal{C}$.

Definition 4.2.1. ***i*TPQ.** Given a set $\mathcal{R} \subseteq \mathcal{C}$ ($\mathcal{R} = \{R_1, R_2, ..., R_k\}$), a starting indoor point p_s and an ending indoor point p_t, a route $\tau = \{p_s, p_1, p_2, ..., p_k, p_t\}$ from p_s to p_t that visits at least one indoor point of each category in $\mathcal{R}$ ($\cup_{i=1}^{k} \pi(p_i)$) and has the minimum route distance $\mathcal{C}(\tau)$, is an Indoor Trip Planning Query (*i*TPQ).

4.2.2 Limitation of Exisiting Techniques

PNE [77]. The main idea of PNE is to explore nearest neighbors one-by-one and to keep adding the first nearest neighbor to the last point of the candidate routes. Once a candidate route is processed, the current nearest neighbor of second last point is replaced by the next nearest neighbor. Hence, the efficiency of PNE relies on the method used to find nearest neighbors. PNE is designed for spatial road networks, and does not consider any properties that only exist in indoor spaces. Thus, the pruning power for PNE is much lower than our proposed VIP-Tree Neighbor Expansion (VNE) algorithm. Our experiments show that VNE is much more efficient in all settings compared to PNE, no matter G-tree or VIP-Tree is utilized for nearest neighbor computations.

PNE [77]+G-Tree [93]. We also compare our method with PNE+G-Tree. G-tree is the state-of-the-art algorithm for query processing in spatial road networks, such as shortest distance/path, kNN and range queries. As in [77], PNE can use any existing algorithms in spatial road networks to find kth nearest neighbor such as IER [67] and VN^3 [53]. Logically, we adopt G-Tree with PNE to process kNN, because G-Tree is the most efficient methods for kNN in spatial road networks. However, our performance evaluation shows that even PNE combined with the best spatial road network indexing, G-Tree, cannot beat our proposed method VNE. One reason is that no efficient pruning techniques are introduced in PNE. Hence, PNE processes most parts of the candidate routes. Although G-tree is efficient for kNN in spatial road networks, it is not efficient in indoor spaces.

PNE [77]+VIP-Tree [75]. The last competitor is PNE combined with our specialised indoor space indexing, VIP-Tree. As noted in our previous paper [75], VIP-Tree is proven to be much more efficient than G-Tree, in all accounts in indoor query processing. Hence, using VIP-Tree with PNE is a sensible solution. However, though VIP-Tree is efficient for indoor *k*NN queries, PNE+VIP-Tree is not efficient for *i*TPQ due to the lack effective pruning techniques in PNE. In the original paper [77], PNE takes more than twenty seconds to process a OSR with six categories and twelve points in each category. The computational complexity for OSR is $O(\rho^m)$ (where ρ is the average number of points in each category, m is the number of categories), which is much less than that of TPQ ($O(\rho^m m!)$). In the same settings, processing time for TPQ is much longer than that for OSR. On the contrary, our proposed VNE algorithm performs $iTPQ$ much more efficient than PNE+VIP-Tree. This demonstrates the need for a specialised *i*TPQ algorithm for indoor spaces, which is the aim of this chapter.

4.3 Contributions

In this chapter, we propose a VIP-Tree Neighbor Expansion algorithm to deal with $iTPQ$. This includes new pruning techniques applied during the pre-processing phase and the query processing phase.

Effective pruning techniques. We propose pruning techniques during the pre-processing phase and the query processing phase. With these pruning methods, VNE avoids to process a large number of unnecessary candidate routes. In addition, we set a benchmark to show the effectiveness of the proposed pruning techniques, together with the combination of all techniques.

High efficiency. Our experimental results show that our proposed method VNE outperforms the other three methods (DBE to be discussed in Section 4.4, PNE+G-tree [93], PNE+VIP-Tree) by several orders of magnitude. No matter whether G-tree or VIP-Tree is utilized in PNE to find the nearest neighbors, our VNE algorithm is more efficient and is able to handle all the settings well.

Low indexing cost. Computing shortest distance/path or *k*NN is a crucial part in TPQ. G-tree is the state-of-the-art method in spatial road networks, while VIP-Tree is the counterpart in indoor spaces. In VNE, we add the information of indoor points in VIP-Tree and we are able apply the pruning techniques efficiently, although we need a small extra indexing cost. However, our experimental results show that, VNE requires a similar indexing cost to the others, but delivers a much better query processing performance.

4.4 A Dijkstra-based Expansion (DBE)

A naive approach to process *i*TPQ is to consider all possible routes. The result is the route with the minimum distance. However, the performance will be very low, since the computational cost is $O(\rho^m m!)$. Therefore, we propose an improved version of the naive algorithm that prunes a number of candidate routes. Algorithm 6 shows the details of this Dijkstra-based approach (DBE). Note that all distances between any two indoor points is still computed by the point-to-point distance algorithm, but uses the VIP-Tree for efficiency.

Algorithm 6: DBE Algorithm

Input : p_s: a starting point, p_t: a ending point, G
Output : τ

1 Initialize a Minheap $H = \emptyset$;
2 **for** $p_i \in V$ **do**
3   add $\tau_i = \{p_s, p_i\}$ into H;
4 **while** H is not empty **do**
5   de-heap $(\tau_c, \mathbf{c}(\tau_c))$;
6   **if** $|\tau_c| = m+2$ **then**
7   return τ_c;
8   **else**
9   **if** $|\tau_c| = m+1$ **then**
10   add $\tau_c = \tau_c + \{p_t\}$ into H;
11   **else**
12   **for** $p_j \in V$ $(\pi(p_j) \notin \cup_{i=1}^{|\tau|-1} \pi(p_i), \tau = \{p_s, p_1, p_2, ..., p_{|\pi|-1}\})$ **do**
13   add $\tau = \tau + \{p_j\}$ into H;

At the first stage, a minimum heap H is initialized. Starting from p_s, every point $p_i \in V$ is added to the candidate routes and is inserted into H together with its route distance. After that, in each step, a candidate route τ_c is de-heaped from H. According to the length of τ_c, there are three cases. If the length of τ_c equals to m+2 (lines 6-7), it means that the shortest route in the candidate set is a complete route that has visited all categories and reached the destination p_t. Note that only the candidate route with the shortest distance is de-heaped from H, thus, this route is the optimal route and we return this as the query result. If the length of τ_c equals to m+1 (lines 9-10), p_t is added to this route as all categories have been visited but this route has not reached p_t. After that, the new route is inserted into H. In the last case (lines 12-14), since τ visits parts of the categories only, hence, for point p_j that does not belong to any visited categories is added to τ and τ is inserted into H.

4.5 Our Approach

In this section, we present our proposed algorithm for processing *i*TPQ. Firstly, the VIP-Tree is added with categories information of all points. The query processing algorithm contains two main functions, particularly first nearest neighbor and next nearest neighbor queries. The algorithm also features several pruning techniques in both pre-processing phase and query processing phase. These pruning techniques demonstrate the efficiency of the proposed algorithm as shown in the performance evaluation section.

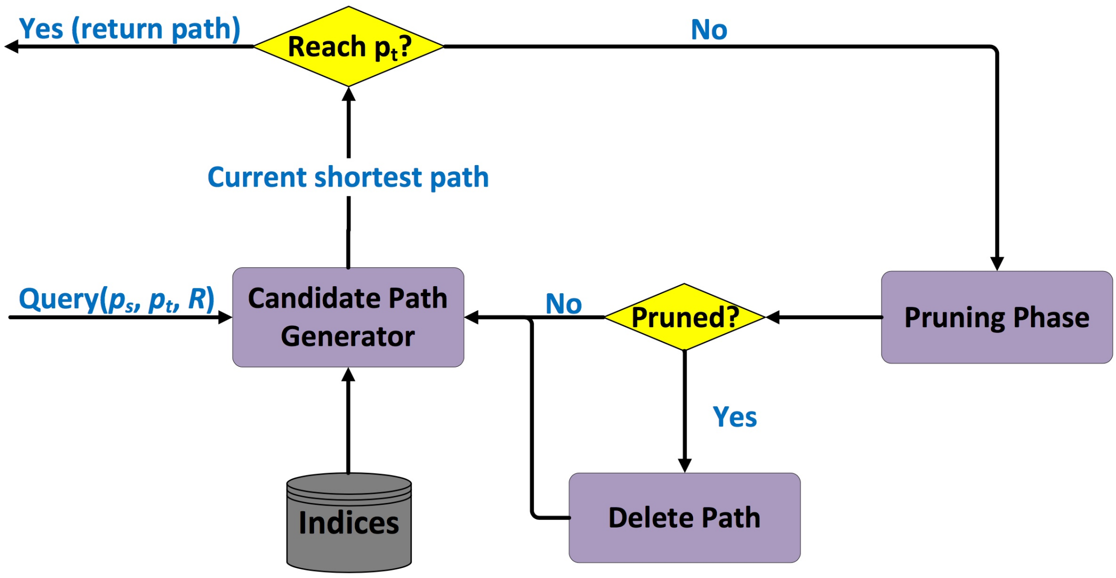


Figure 4.2: The process to solve an *i*TPQ

Fig. 4.2 shows the detailed framework of how to process an *i*TPQ. Before processing an *i*TPQ, the indices are pre-computed for the available indoor venues. An *i*TPQ represented by $Query(p_s, p_t, R)$ is invoked by a user, where p_s and p_t are the starting and ending points respectively, while R is the categories that the user wants to visit. At the next stage, candidate path generator keeps generating the candidate path by adding one more point (this point does not belong to any visited category of the current path) to the existing path. All distance calculations are solved by the proposed algorithms based on the indices. After that, we have to check for the current shortest path, if this path reaches p_t, return this path as the result. Otherwise, the pruning phase are utilized to determine if this path can be pruned. If current path has to be pruned, then return to candidate path generator to generate the consequential paths. Otherwise, a new shortest path is retrieved.

4.5.1 VIP-Tree with Categories

Each indoor point $p_i \in V$ belongs to a category C_i. The original VIP-Tree does not have any information about categories. Hence, the first step is to update VIP-Tree with the category information of each point.

The update process is divided into two steps. Firstly, for each leaf node, the distance matrix is updated by adding the distances between every access doors and every point inside the indoor partitions of the leaf node. The distances can be computed efficiently using the shortest distance algorithm in [75]. Meanwhile, the D2D graph of this leaf node is updated by adding the edges between p_i and every door inside the indoor partition containing p_i. This D2D graph will be used to compute the distance between two points in the same leaf node.

Secondly, for each non-leaf node, the distances between every access door and every point inside the indoor partitions of the sub-tree are computed and stored in the distance matrix.

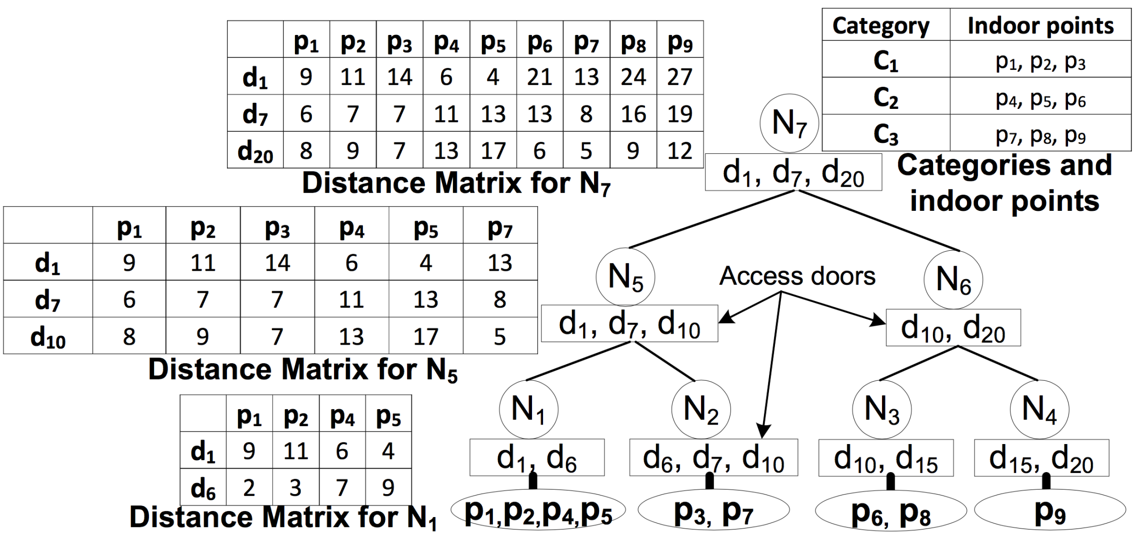


Figure 4.3: VIP-Tree with the Category information

Fig. 4.3 shows the VIP-Tree with the distance matrices for nodes N_1, N_5 and N_7. The indoor venue shown in Fig. 4.1 contains 9 indoors points that belong to 3 categories: specifically, V_{C_1}=$\{p_1, p_2, p_3\}$, V_{C_2}=$\{p_4, p_5, p_6\}$, V_{C_3}=$\{p_7, p_8, p_9\}$. In the node, we omit the doors that are not access doors. The distance matrices shown in Fig. 4.3 display the distances between every access door and every point inside the partitions contained in the node. Take N_7 as an example, $AD(N_7)$=$\{d_1, d_7, d_{20}\}$. N_7 contains V=$\{p_1, p_2,...,p_9\}$. Therefore, the distances matrix of N_7 stores the distances between every $d_i \in AD(N_7)$ and every point $p_i \in V$.

4.5.2 Query Processing

In this section, we describe our proposed algorithm to solve *i*TPQ. The algorithm progressively builds candidate routes by adding the next nearest neighbor of the last point in the candidate route. The rationale is that comparing to the points that are far away from the last point, nearby points are more likely to generate a route in a shorter distance. Thus, an expansion algorithm is used as shown in Algorithm 7.

Algorithm 7: Query Processing Algorithm

Input : VIP-Tree, p_s: a start point, p_t: an end point, V: an indoor point set
Output : τ

1 Initialize a Minheap H=$\emptyset$;
2 **for** i from 1 to m **do**
3   p=$NN(p_s, C_i)$;
4   add ($\tau = \{p_s, p\}$,$c(\tau)$) into H;
5 **while** H is not empty **do**
6   de-heap (τ_c,$c(\tau_c)$) from H;
7   **if** $|\tau_c|$=m+2 **then**
8   return τ_c;
9   **if** $|\tau_c|$=m+1 **then**
10   add (τ_c + $\{p_t\}$,$c(\tau)$) into H;
11         $p_l \longleftarrow$ last point of τ_c;
12         $p_j \longleftarrow$ second last point of τ_c;
13   p=$NextNN(p_j, \pi(p_l))$;
14   add (τ_c-$\{p_l\}$+$\{p\}$,$c(\tau)$) into H;
15   **else**
16         $p_l \longleftarrow$ last point of τ_c;
17         $p_j \longleftarrow$ second last point of τ_c;
18   **for** i from 1 to m **do**
19   **if** C_i is not visited in τ_c **then**
20   p=$NN(p_l, C_i)$;
21   add (τ=τ_c+$\{p\}$,$c(\tau)$) into H;
22   **else**
23   p=$NextNN(p_j, C_i)$;
24   add (τ_c-$\{p_l\}$+$\{p\}$,$c(\tau)$) into H;

Algorithm 7 shows the details on how to process an *i*TPQ. In the initialization phase (lines 1-3), it finds the nearest neighbor of the start point p_s within all categories, and forms m candidate routes. A minimum heap H is initialized to store the candidate routes along with their route distances. After that, one candidate route τ_c is de-heaped from H. If the length of τ_c equals to m+2 (lines 7-8), it means that this is the query result, since the distances of any routes in the candidate set are longer than τ_c. Hence, τ_c is returned as the query result. On the other hand, if the length of τ_c is less than m+2, this route has not reached the destinations yet. Therefore, more points have to be added according to the following two cases:

1. $|\tau_c|$=m+1 (lines 9-14). This means that τ_c has visited all categories, so the next point is the destination point p_t. Thus, we add p_t to τ_c. On the other hand, in order to cover all possible routes, we find the next nearest neighbor of p_j computed by NextNN(p_i, C_i) (Function NextNN(p_i, C_i) is utilized to find the next nearest neighbor of p_i in a specified category C_i) to replaces p_l in τ_c. For the new τ_c, insert it into H with $c(\tau_c)$.

2. $|\tau_c|$ <m+1 (lines 15-24).

   (a) We expand τ_c to the next category, in which we find the nearest neighbor of the last point of τ_c in all unvisited categories. Once the nearest neighbor is found by NN(p_i, C_i) (Function NN(p_i, C_i) is utilized to find the nearest neighbor of p_i in a specified category C_i), it is added to τ_c, and $(\tau_c, c(\tau_j))$ is insert into H.

   (b) We then find the next nearest neighbor of the second last point in τ_c. A candidate route is updated by replacing the last point with the next nearest neighbor of the second last point in τ_c, and this is then inserted into H.

EXAMPLE 7 : We discuss Algorithm 7 in more details using the example of Fig. 4.1 and 4.3. Let p_s and p_t be the start and end points, and there are 3 categories (V_{C_1}=$\{p_1, p_2, p_3\}$, V_{C_2}=$\{p_4, p_5, p_6\}$, V_{C_3}=$\{p_7, p_8, p_9\}$) that will be visited. Fig. 4.1 depicts the contents of the minimum heap H in each step. In step 1, the first nearest neighbor of p_s in 3 categories is computed as p_1, p_5 and p_7. 3 candidate routes are inserted into H and the distance of the shortest route in H is shown in the first place of the heap. In step 2, the shortest candidate route τ_c=(p_s, p_5 : 2) is de-heaped from H. Since τ_c has visited one category only, two operations are performed. For the unvisited categories, NN(p_5, C_1) and NN(p_5, C_3) are used to find the first nearest neighbor for p_5 in categories C_1 and C_3. p_1 and p_7 is return as the query results. Hence, two candidate paths (p_s, p_5, p_1: 14), (p_s, p_5, p_7: 15) are inserted into H. For category C_2 that has been visited, we use NextNN(p_s, C_2) to find the second nearest neighbor of p_s in C_2, which in this case is p_4 (the first nearest neighbor p_5 has been found, hence, the next nearest neighbor is the second nearest neighbor). By replacing p_5 with p_4, (p_s, p_4: 4) is inserted into H. Similarly, this process is repeated until the shortest candidate route in H has visited 3 categories and the last point is p_t. The algorithm returns this path as the query result. The only requirement for Algorithm 7 is to efficiently perform NN(p_i, C_i) and NextNN(p_i, C_i). Hence, we are going to discuss these two functions in more detail.

■

Step	Heap Contents
1	**(p_s, p_5: 2)**, (p_s, p_1: 8), (p_s, p_7: 12)
2	**(p_s, p_4: 4)**, (p_s, p_1: 8), (p_s, p_7: 12), (p_s, p_5, p_1: 14), (p_s, p_5, p_7: 15)
3	**(p_s, p_7: 8)**, (p_s, p_4, p_1: 11), (p_s, p_7: 12), (p_s, p_5, p_1: 14), (p_s, p_5, p_7: 15), (p_s, p_4, p_7: 16), (p_s, p_6: 20)
4	**(p_s, p_4, p_1: 11)**, (p_s, p_7, p_3: 11), (p_s, p_7: 12), (p_s, p_5, p_1: 14), (p_s, p_5, p_7: 15), (p_s, p_4, p_7: 16), (p_s, p_7, p_4: 19), (p_s, p_6: 20), (p_s, p_8: 24)
5	**(p_s, p_7, p_3: 11)**, (p_s, p_7: 12), (p_s, p_5, p_1: 14), (p_s, p_4, p_2: 14), (p_s, p_5, p_7: 15), (p_s, p_4, p_7: 16), (p_s, p_4, p_1, p_7: 18), (p_s, p_7, p_4: 19), (p_s, p_6: 20), (p_s, p_8: 24)
...	...
Final	**(p_s, p_4, p_1, p_7, p_t: 32)**, ...

Table 4.1: Query processing for the example in Fig. 4.1

Function NN(p_i, C_i). Given an indoor point p_i and a category C_i, $NN(p_i, C_i)$ computes the first nearest neighbor of p_i among the points in category C_i. We use a best-first search algorithm, widely used on various branch and bound structures, such as R-tree, Quad-tree etc. Different from that in [75], the first nearest neighbor search algorithm is a simplified one. Note that in VIP-Tree with categories, for any point $p_i \in V$, the distances between p_i and the access doors $AD(N_i)$ (N_i is the ancestor node of `Leaf(`p_i`)`) are pre-computed and stored in the distance matrices in N_i. However, in [75], only the distances between every object and access doors of the leaf node are pre-computed. Therefore, NN(p_i, C_i) is more efficient. Before we discuss the detailed algorithm of NN(p_i, C_i), we propose an algorithm to compute the distance between any point p_i ($p_i \in V$) and any non-ancestor node N_i of p_i because the distance between p_i and any ancestor node is zero.

Algorithm 8: getDistance(p_i, N_i)

Input : p_i: an indoor point, N_i: a tree node
Output : $dist(p_i, N_i)$: shortest distance between p_i and N_i

1 Initialize N_{LCA} to be the lowest common ancestor node between `Leaf(`p_i`)` and N_i;
2 $N_l \longleftarrow$ the child node of N_{LCA} and the ancestor node of `Leaf(`p_i`)`;
3 **if** N_{LCA} is the parent node of N_i **then**
4     $N_r \longleftarrow N_i$;
5 **else**
6     $N_r \longleftarrow$ the child node of N_{LCA} and the ancestor node of N_i;
7 $dist(p_i, N_i) = min_{d_i \in AD(N_l), d_j \in AD(N_r), d_k \in AD(N_i)} dist(p_i, d_i) + dist(d_i, d_j) + dist(d_j, j_k)$;
8 return $dist(p_i, N_i)$;

Algorithm 8 illustrates the distance computation between any point $p_i \in V$ and any non-root node N_i. Firstly, it locates the lowest common ancestor node N_{LCA} between `Leaf(`p_i`)` and N_i. In lines 2 and 3, we find the child nodes of N_{LCA}. Meanwhile, for these two child nodes, they have to be the ancestor nodes for `Leaf(`p_i`)` and N_i, respectively. The shortest distance computation contains 3 parts: distances between p_i and $AD(N_l)$, distances between $AD(N_l)$ and $AD(N_r)$, and distances between $AD(N_r)$ and N_i.

EXAMPLE 8 : Considering the example of Fig. 4.1 and Fig. 4.3 and assuming that we want to compute the distance between p_1 and N_4. In Fig. 4.4, the arrows depict the actual distances that are pre-computed and stored in the distance matrices, while the dashed lines indicate that two doors are the same door. The lowest common ancestor node is N_7. According to Algorithm 8, the next step is to find the child nodes of N_7 that contains `Leaf(`p_i`)` and N_i. Consequently, N_l and N_r are N_5 and N_6 respectively. It clearly shows that in Fig. 4.4, $dist(p_i, d_i)$ is the distance between p_1 and every access doors of N_5 which are d_1, d_7 and d_{10}. $dist(d_i, d_j)$ is the distance between every access door in N_5 and every access door in N_6. The last part $dist(d_j, d_k)$ is the distance between the access doors in N_5 and N_4.

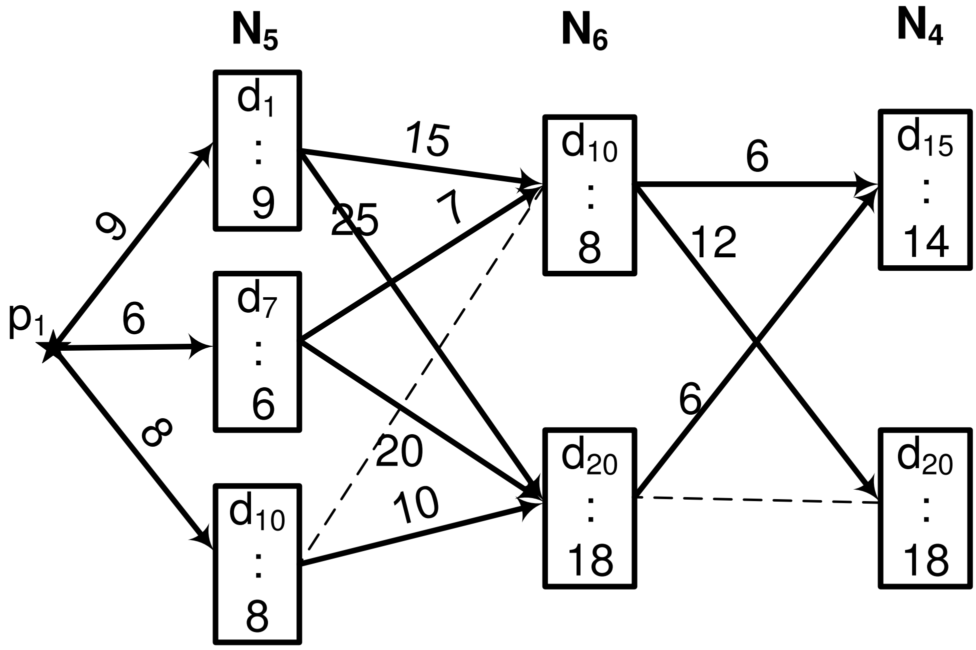


Figure 4.4: getDistance(p_i, N_i) computation

■

Algorithm 9 describes how to find the first nearest neighbor given an indoor point p_i and a category C_i. Firstly, a minimum heap is initialized with the root node of the tree and the distance from p_i to root is zero. In each iteration, one node is de-heaped from H. If the distance between p_i and N is larger than d_k (the current distance from p_i to the first nearest neighbor), p_j is returned as the result (lines 5-6). Otherwise, if N is a non-leaf node (lines 7-12), according to their distance to p_i, we perform two operations. If the distance equals to 0, it means that this node contains p_i.

Algorithm 9: NN(p_i,C_i)

Input : p_i: an indoor point, C_i: a category
Output : p_j: nearest neighbor of p_i, $p_j \in C_i$
1 d_k=∞; /* d_k is the distance to current first nearest neighbor */;
2 Initialize a Minheap H with the root of the node;
3 **while** H is not empty **do**
4   de-heap N from H;
5   **if** $dist(p_i, N) \geq d_k$ **then**
6   return p_j;
7   **if** N is a non-leaf node **then**
8   **if** $getDistance(p_i, N) = 0$ **then**
9   **for** each child node N' of N **do**
10   insert N' into H;
11   **else**
12   use $p_k \in C_i$ in N to update p_j and d_k;
13   **else**
14   use $p_k \in C_i$ in N to update p_j and d_k;

For its child nodes, we insert them into H with their distances to p_i. For the non-leaf node with a non-zero distance, we do not need to expand to its child nodes anymore. For p_k in the sub-tree of N, the distances between p_k and every access door of N is pre-computed, hence, we can easily update the distances between p_i and p_k. If N is a leaf node, then update p_j and d_k (lines 13-14). Since NextNN(p_j, C_i) uses the status of NN(p_j, C_i), we need to store H and the nearest neighbors found so far.

Function NextNN(p_i, C_i) Given an indoor point p_i and a category C_i, NextNN(p_i, C_i) computes the next nearest neighbor in category C_i. Based on the saved status, we can perform a next nearest neighbor search.

Algorithm 10: NextNN(p_i,C_i)

Input : p_i: an indoor point, C_i: a category, H: a heap
Output : p_j: next nearest neighbor of p_i, $p_j \in C_i$
1 d_k=∞; /* d_k is the distance to current nearest neighbor */;
2 **while** H is not empty **do**
3   de-heap N from H;
4   **if** $dist(p_i, N) \geq d_k$ **then**
5   return p_j;
6   **if** N is a non-leaf node **then**
7   **if** $getDistance(p_i, N) = 0$ **then**
8   **for** each child node N' of N **do**
9   insert N' into H;
10   **else**
11   use $p_k \in C_i$ in N to update p_j and d_k;
12   **else**
13   use $p_k \in C_i$ in N to update p_j and d_k;

Algorithm 10 illustrates the details. Note that, different from OSR queries that have a category sequence, no sequence is given in *i*TPQ. In an OSR query, assuming that fifth nearest neighbor of p_i is found, when p_i comes up again, it must be finding the sixth nearest neighbor because there is only one route that ends up with p_i and its fifth nearest neighbor. However, for *i*TPQ, there are several possible candidate routes that can end with p_i and its ith ($1 < i < 5$) nearest neighbor as no category sequence is given. As a result, for the saved status that finds the fourth nearest neighbor, one candidate route tries to find the third nearest neighbor. Therefore, we have to save the current i nearest neighbors having been found so far. If the next nearest neighbor is computed, the NextNN(p_i, N_i) search does not need to be performed.

4.5.3 Proof of Correctness

In this section, we prove that Algorithm 7 correctly answers *i*TPQ. This means that Algorithm 7 processes all the candidate routes. Let C=$\{C_1, C_2, ..., C_m\}$ be the category set that an *i*TPQ is going to visit. The start and end points are denoted as p_s and p_t. Let C_{sub}=$\{C_1, C_2, ..., C_k\}$ ($k < m$) be a subset of C. We use τ_c to refer to the candidate route that satisfy C_{sub}, τ is an optimal route that satisfies C. Hence, we get $c(\tau_c) \leq c(\tau)$.

Algorithm 7 generates and examines all the possible candidate routes that have a shorter distance compared to the optimal route τ. Note that H is a minimum heap, therefore, in each iteration, the candidate route with the shortest distance is de-heaped. If the length of the de-heaped candidate route equals to m+2, Algorithm 7 stops as the other candidates in H will not generate a shorter route. Therefore, the candidate routes that are left in H after the algorithm stops will not be used. We prove that the candidate routes de-heaped from H before τ contain all the possible routes that have a shorter distance than τ. Lemma 4.5.1 shows the details.

Lemma 4.5.1. *For a given iTPQ, Algorithm 7 examines all the candidate routes that have a shorter distance than the optimal route* τ.

Proof. The proof is done by induction on k, the size of the candidate route τ_c. Firstly, for k=1, we show that it examines all the candidate routes that start from p_s and visit only one category. During the initialization phase (lines 2-4), the first nearest neighbor of p_s in each category is computed. Take C_1 as an example, the first nearest neighbor is p_1. There are two cases for τ_c=$\{p_s, p_1\}$. For the first one, if $c(\tau_c) \geq c(\tau)$, this means that the candidate routes generated by τ_c will never have a shorter distance than $c(\tau)$, because of the following two reasons. If p_1 is replaced by the next nearest neighbor of p_s (lines 22-24), the route distance is larger than τ_c. On the other hand, if τ_c appends any point belonging to C_i ($C_i \neq C_1$, $C_i \in C$). The length of the new routes exceeds the size which equals to 1 in this case. For the second case, if $c(\tau_c) < c(\tau)$. Similarly, there are two ways to generate candidate paths according to τ_c. For appending points in different categories, it exceeds the maximum size, hence, we are not going to discuss. When replacing the last point in τ_c with the next nearest neighbor of p_s (lines 22-24), if the generated route has a shorter distance than $c(\tau)$, we keep expanding to the nearest neighbor until the distance of generated route is longer than $c(\tau)$ or all points in C_1 have been expanded. Once we found the generate candidate has a longer distance than $c(\tau)$, for the rest of the next nearest neighbors, we do not need to consider since our condition is only applicable for candidates routes that are shorter than $c(\tau)$.

Now, we are going to examine the candidate routes that satisfy $C_{sub}=\{C_1, C_2, ..., C_k\}$ ($k < m$). Let $C_{sub1}=\{C_1, C_2, ..., C_k, C_{k+1}\}$ ($k+1 < m$) be a subset of C. The first k categories in both C_{sub} and C_{sub1} are the same. We are going to prove that Algorithm 7 examines all the candidate routes satisfying C_{sub1} generated by the candidate routes satisfying C_{sub}. $\tau_c=\{p_1, p_2, ..., p_k\}$ is the current candidate route de-heaped from H, for every unvisited category, we use NN(p_i, C_i) to find the first nearest neighbor of p_k (lines 19-21). After appending the first nearest neighbor to τ_c, if the current distance is shorter than τ, it means that this candidate route will be de-heaped before τ from H. Meanwhile, NextNN(p_i, C_i) is utilized to find the nearest nearest neighbor of p_k in the same category until the generate candidate path is longer than τ. This ensures that for each candidate routes satisfying C_{sub}, we are able to generate all the possible candidate routes satisfying C_{sub1} and having a shorter distance than τ. □

4.6 Pruning in The Pre-processing Phase

Our algorithm features two levels of pruning: (*i*) at the pre-processing phase, and (*ii*) at the query processing phase. These pruning techniques are effective to prune candidate routes. In this section, we are going to discuss three pruning methods in the pre-processing phase.

4.6.1 Partition-based Pruning

Indoor partitions are quite common in an indoor space and indoor points are located within partitions. As a result, one indoor partition may contain several indoor points belonging to one or more categories. For the indoor points in the same category and located in the same partition, Lemma

4.6.1 is used to prune part of the edges between these indoor points and the indoor points outside the partition.

Lemma 4.6.1. *Let* $V_1 = \{p_1, p_2, ..., p_i\}$ *be a set of indoor points located in the same indoor partition* P_i*,* $\forall p_i \in V_1$*,* $\pi(p_i)$ *is the same.* $D = \{d_1, d_2, ..., d_i\}$ *is a set of doors in* P_i*. For any indoor point* p_j *located outside* P_i*, for a candidate route* $\tau = \{p_s, ...p_j, p_k, p_n, ..., p_t\}$ *(*p_j *and* p_n *are outside* P_i*), if* $\exists p_m \in V_1 - \{p_k\}$*,* $\forall d_j \in D$*,* $dist(p_m, d_j) < dist(p_k, d_j)$*,* τ *is pruned.*

Proof. Consider a route $\tau = \{p_s, ...p_j, p_k, p_n, ..., p_t\}$ and an alternative route $\tau_a = \{p_s, ...p_j, p_m, p_n, ..., p_t\}$, where $p_m \in V_1 - p_k$. $c(\tau)$-$c(\tau_a)$=$dist(p_j, p_k) + dist(p_k, p_n) - dist(p_j, p_m) - dist(p_m, p_n)$. To prove $c(\tau)$-$c(\tau_a)$>0, we need to prove that $dist(p_j, p_k) + dist(p_k, p_n) > dist(p_j, p_m) + dist(p_m, p_n)$. We compute two parts of the equation separately (the left and right parts are represented by Δ_1 and Δ_2 respectively):

$\Delta_1 = [dist(p_j, d_{jk}) + dist(d_{jk}, p_k)] + [dist(p_k, d_{kn}) + dist(d_{kn}, p_n)]$

$\Delta_2 = [dist(p_j, d_{jm}) + dist(d_{jm}, p_m)] + [dist(p_m, d_{mn}) + dist(d_{mn}, p_n)]$

Let d_{ij} be the door in P_i in which the shortest path between p_i and p_j has to pass through. Based on the condition that $\exists p_m \in V_1 - \{p_k\}$, $\forall d_j \in D$, $dist(p_m, d_j) < dist(p_k, d_j)$, we know that $dist(p_j, d_{jk}) + dist(d_{jk}, p_k) > dist(p_j, d_{jk}) + dist(d_{jk}, p_m)$. As the shortest path between p_j and p_m passes through d_{jm}, $dist(p_j, d_{jk}) + dist(d_{jk}, p_m) > dist(p_j, d_{jm}) + dist(d_{jm}, p_m)$. According to the above two equations, we can conclude that $dist(p_j, d_{jk}) + dist(d_{jk}, p_k) > dist(p_j, d_{mn}) + dist(d_{mn}, p_m)$. For the same reason, $dist(p_k, d_{kn}) + dist(d_{kn}, p_n) > dist(p_m, d_{mn}) + dist(d_{mn}, p_n)$. Thus, we are able to prove that $\Delta_1 > \Delta_2$. This means that for p_k to exist in a route τ, $c(\tau)$ can be reduced by replacing p_m with p_k. Hence, τ is pruned, since it will never be the shortest route. □

EXAMPLE 9 : Consider the example in Fig. 4.5, where the indoor partition P_4 has two doors, d_5 and d_6. For d_5, the nearest indoor points inside P_4 is p_1, while for d_6, p_1 is the nearest one as well. Both p_1 and p_2 belong to category C_1. In partition P_1, there are two points p_4 and p_5 belonging to C_2, while in partition P_5, one point p_7 is located belonging to C_3. According to Lemma 4.6.1, for a given route $\tau = \{p_s, ..., p_4, p_2, p_7, ...p_t\}$, it is pruned because there exists p_1 that is located in the same partition as p_2 and the distances from p_1 to every door in P_4 is shorter than that for p_2.

■

Lemma 4.6.2. *Let* $V_1 = \{p_1, p_2, ..., p_i\}$ *be a set of indoor points located in partition* P_i*,* $\forall p_i \in V_1$*, and* $\pi(p_i)$ *is the same. For* $p_i \in V_1$ *and* $p_j \in V_1$*, a perpendicular bisection is drawn that divided* P_i

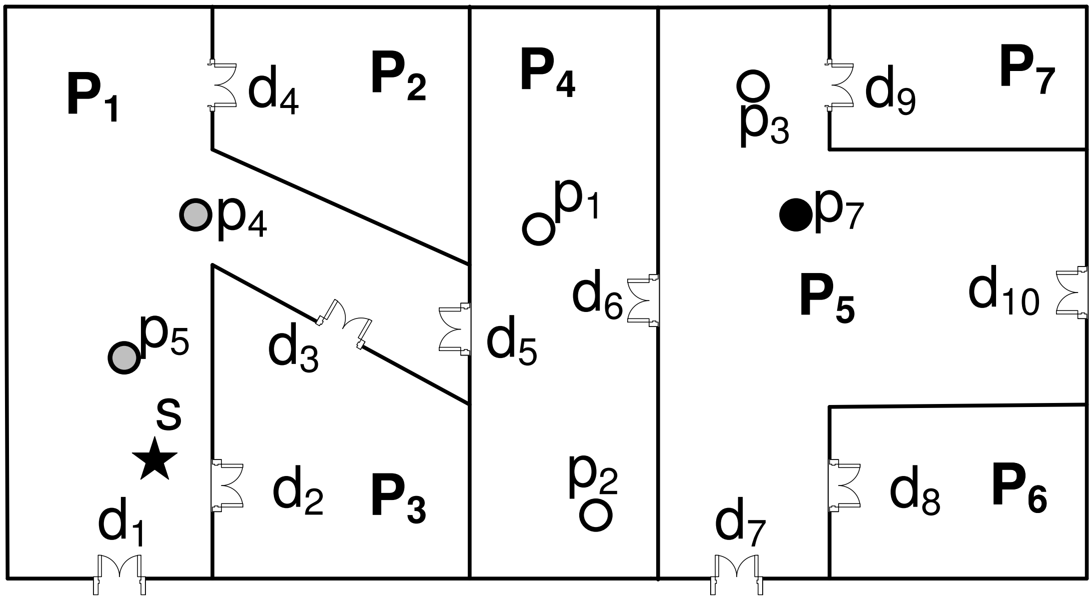


Figure 4.5: $\pi(p_1)$=$\pi(p_2)$; $\pi(p_1)\neq\pi(p_4) \neq \pi(p_7)$, $\pi(p_4)$=$\pi(p_5)$

into two areas. For a candidate route τ=$\{p_s,...p_m,p_i,p_n,...,p_t\}$ *(*p_m *and* p_n *are inside* P_i*), if both* p_m *and* p_n *are in the same half space with* p_j*,* τ *is pruned* [2]*.*

Proof. Consider a route τ=$\{p_s,...p_m,p_i,p_n,...,p_t\}$ and an alternative route τ_a=$\{p_s,...p_m,p_j,p_n,...,p_t\}$, where $\pi(p_i) = \pi(p_j)$. A perpendicular bisector is drawn for p_i and p_j. According to the properties of a perpendicular bisector, for any point p_a located in the same area with p_i, distance between p_i and p_a is shorter than that between p_j and p_a, and vice versa. Hence, $\forall p_k$ in P_i which share the same area with p_i, they are closer to p_i than p_j. As we want to prove that $c(\pi)$-$c(\pi_a) < 0$, we compute $[dist(p_m, p_i) - dist(p_m, p_j)] + [dist(p_i, p_n) - dist(p_j, p_n)] < 0$. Note that p_m/p_n is in partition P_i and both p_m and p_n are in the same half space with p_j. According to the property of perpendicular lines, $dist(p_i, p_m/p_n) < dist(p_j, p_m/p_n)$. Thus, $c(\pi)$-$c(\pi_a) < 0$ is proven. This means τ will never be the shortest route, hence, τ is pruned. □

EXAMPLE 10 : Consider the example in Fig. 4.6, p_1 and p_2 are two indoor points in the same category located in partition P_4. A perpendicular bisector is drawn for these two points denoted as the dashed line. Two indoor points p_{10} and p_{11} are located in P_4 as well and $\pi(p_1)\neq\pi(p_{10})\neq\pi(p_{11})$. Let τ=$\{p_s,...p_{10},p_2,p_{11},...,p_t\}$ and τ_c=$\{p_s,...p_{10},p_1,p_{11},...,p_t\}$. Note that p_{10} and p_{11} are in the same half space with p_1, hence, $c(\tau) > c(\tau_c)$. This means τ cannot be the shortest route anymore, therefore, τ is pruned. ■

[2]Perpendicular bisector is just for an illustration purpose. In practice, we will compute the actual distances to see whether p_m/p_n is closer to p_i or p_j

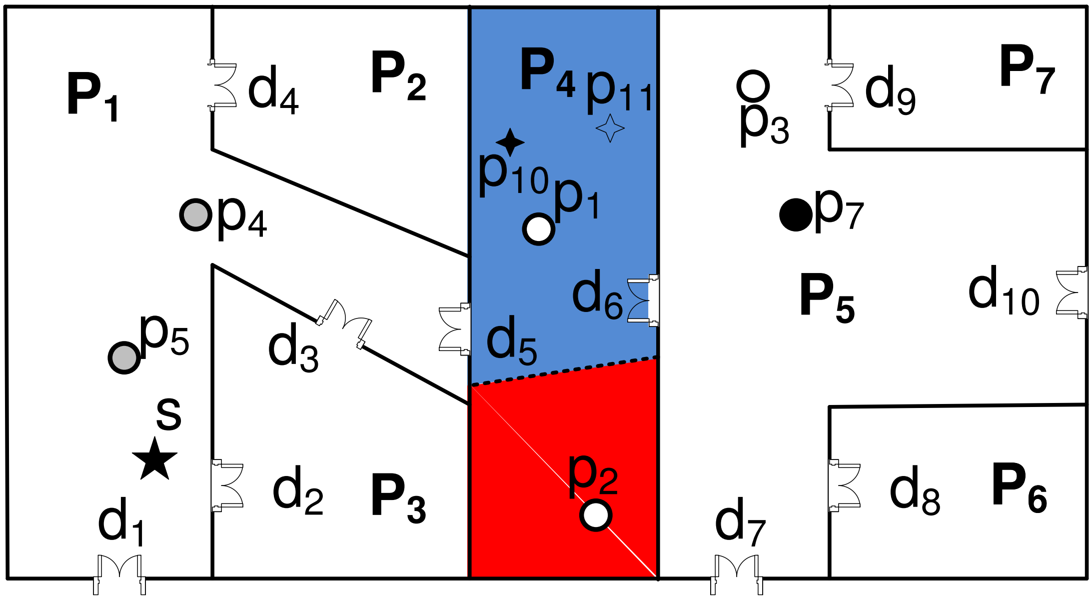


Figure 4.6: $\pi(p_1)$=$\pi(p_2)$≠$\pi(p_{10})$≠$\pi(p_{11})$

4.6.2 Node-based Pruning

VIP-Tree progressively merges indoor partitions [75]. As mentioned before, a node in VIP-Tree represents a partition that combines several indoor partitions and considers access doors as the doors of the partition, hence, for each node in the VIP-Tree, it is easy to adapt the pruning techniques that discussed in Section 4.6.1.

Lemma 4.6.3. *Given a tree node N_i in the VIP-Tree and $AD(N_i)$ denotes the access doors of N_i. Let NP_i be the partition representing the combined partitions in N_i and V is a set of indoor points inside NP_i, $\forall p_i \in V$, $\pi(p_i)$ is the same. For a candidate route τ=$\{p_s,...p_j,p_i,p_n,...,p_t\}$ (p_j and p_n are outside N_i), if $\exists p_j \in V_1 - \{p_i\}$, $\forall d_j \in AD(N_i)$, $dist(p_m, d_j) < dist(p_k, d_j)$, τ is pruned.*

Proof. Since the proof for Lemma 4.6.3 is similar to that for Lemma 4.6.1, we omit the details. □

4.6.3 Applying Pruning Techniques in VIP-Tree

Algorithm 11 shows how to apply Lemmas 4.6.1, 4.6.2 and 4.6.3 on VIP-Tree. Use Lemma 4.6.1 as an example. If one indoor point p_j can be found in the distance matrix of `Leaf(`p_j`)` between p_i and $AD($`Leaf(`p_j`)`$)$. This means that the distances between any door in the partition and p_i is longer than that for p_j. Hence, according to Lemma 4.6.1, once a path contains p_i, it will be pruned. Note that for updating VIP-Tree based on Lemma 4.6.2, although it uses multiple loops, it is still efficient since the number of indoor points located in the same partition will not be very large (In our experiments, we have shown that in reality, the number of indoor points in the same

Algorithm 11: Apply Pruning Techniques in VIP-Tree

for every indoor partition P_i **do**
  for every indoor point p_i inside P_i **do**
    if $\forall d_i \in P_i, \exists p_j \in P_i$ $dist(p_j, d_i) < dist(p_i, d_i)$ **then**
      add p_j between p_i and $d_j(d_j \in AD($`Leaf(`p_i`)`$))$ to distance matrix of `Leaf(`p_i`)`; `/* Lemma 4.6.1 */;`
    for every indoor point p_j inside P_i $(p_i \neq p_k, \pi(p_i) = \pi(p_j))$ **do**
      if $\exists p_m, p_n, \pi(p_i) \neq \pi(p_m) \neq \pi(p_n)$, $dist(p_i, p_m/p_n) < dist(p_j, p_m/p_n)$ **then**
        add p_i between p_m/p_n and $AD(N_i)$ to distance matrix of `Leaf(`p_i`)`; `/* Lemma 4.6.2 */;`

for every node N_i in Combined VIP-Tree **do**
  for every indoor point p_i inside N_i **do**
    if $\forall d_i \in AD(N_i), \exists p_j \in N_i$ $dist(p_j, d_i) < dist(p_i, d_i)$ **then**
      add p_j between p_i and $d_j(d_j \in AD(N_i))$ to distance matrix of N_i; `/* Lemma 4.6.3 */;`

room are very small, therefore, computational cost here is very small). To differentiate the stored indoor points in the distance matrix, we use different three different tags for three lemmas.

4.7 Pruning in The Query Processing Phase

In this section, we propose three techniques to efficiently prune the candidate routes during the query processing phase.

4.7.1 3-Candidate Pruning

Given a route $\tau = \{p_s, ..., p_{m-1}, p_m\}(|\tau| \geq 3)$, any three sequenced points are chosen to form a sub-route $\tau_a = \{p_i, p_j, p_k\}$ where $\pi(p_i) \neq \pi(p_j) \neq \pi(p_k)$. If we can find a point p_m $(\pi(p_m) = \pi(p_j))$ to form a route $\tau_b = \{p_i, p_m, p_k\}$ such that $c(\tau_a) \geq c(\tau_b)$, hence, τ_a will never be shorter than τ_b and is pruned.

Lemma 4.7.1. *Given part of a candidate route $\tau = \{p_i, p_j, p_k\}$, If $\exists d_i \in N$(N is a node of VIP-Tree), $\exists p_m(\pi(p_m) = \pi(p_j)$, p_m is in N), $dist(p_i, d_i) + dist(p_k, d_i) + 2dist(p_m, d_i) \leq c(\tau)$, τ is not an optimal route.*

Proof. Let $\tau_a = \{p_i, p_m, p_k\}$, to prove τ is not an optimal route, we have to prove $c(\tau) \geq c(\tau_a)$. $c(\tau_a)$ is computed as $dist(p_i, p_m) + dist(p_m, p_k)$. As we know $dist(p_i, p_m) \leq dist(p_i, d_i) + dist(d_i, p_m)$ and $dist(p_m, p_k) \leq dist(p_m, d_i) + dist(d_i, p_k)$, therefore, it is easy to conclude $dist(p_i, p_m) + dist(p_m, p_k) \leq dist(p_i, d_i) + dist(p_k, d_i) + 2dist(p_m, d_i)$. Thus, $c(\tau_a) \leq dist(p_i, p_m) + dist(p_m, p_k) \leq c(\tau)$. We can conclude that τ is not an optimal route.

□

Choosing N and p_m. In Lemma 4.7.1, N/p_m can be any node/point in the VIP-Tree with categories. Note that our main goal is to find N and p_m that minimize $dist(p_i, d_i) + dist(p_k, d_i) + 2dist(p, d_i)$. Naturally, if p_m is close enough to the shortest path between p_i and p_k, the cost to add p_m between p_i and p_k may be minimized. Thus, in the updated VIP-Tree, N is chosen to be the nodes on the shortest path between p_i and p_k. The distance between N_i and N_j are computed as $min_{\forall d_i \in AD(N_i), \forall d_j \in AD(N_j)} dist(d_i, d_j)$. Note that in the updated VIP-Tree, we store the distances between every point p_i and AD(N_i)(N_i is an ancestor node of `Leaf(`p_i`)`). Hence, $dist(p, d_i)$ can be retrieved in $O(1)$ time. If N is chosen to be not the lowest common ancestor node between p_i and p_k but a node on the shortest path between p_i and p_j, to compute $dist(p_i, d_i)/dist(p_k, d_i)$, a door-to-point shortest distance query has to be performed (this is similar to a point-to-point shortest distance query) while another distance can be retrieved in $O(1)$. On the other hand, for the nodes on the shortest path, the lowest common ancestor node contains the most number of points, it is likely to find a point p_m that satisfies Lemma 4.7.1. Therefore, N is chosen to be the lowest common ancestor node.

Since we aim to minimize the distance, after choosing the node N, d_i is chosen to the one that minimize $dist(p_i, d_i) + dist(d_i, p_k)$ Note that in the VIP-Tree with categories, for any access door d_i in a node N_i, any point p_i in N_i is sorted according to $dist(d_i, p_i)$. Therefore, p_m is chosen to be the nearest point to d_i. Algorithm 12 shows all the details. Note that, if the current p_j is the optimal one, Algorithm 12 returns p_j.

Algorithm 12: 3-candidate pruning

Input : $\tau = \{p_i, p_j, p_k\}$, N_i: a node
Output : p_m: a point better than p_j

Initialize a heap $H = \emptyset$;
$d = c(\tau)$;
for every $d_i \in AD(N_i)$ **do**
  insert $dist(p_i, d_i) + dist(d_i, p_k)$ into H;
while H is not empty **do**
  de-heap $dist(p_i, d_i) + dist(d_i, p_k)$;
  $p_i \longleftarrow$ nearest point to d_i in N_{LCA};
  if $dist(p_i, d_i) + dist(d_i, p_k) \geq d$ **then**
    return p_m;
  update d and p_m;

In query processing phase, if for any 3-candidate sub-route τ=$\{p_i, p_j, p_k\}$, we can quickly retrieve the optimal point p_m ($\pi(p_m) = \pi(p_j)$). For any route $\{p_i, p_j, p_k\}$ that starts from p_i and ends at p_k, it is pruned if $p_j \neq p_m$. Let ρ be the number of points in category $\pi(p_j)$, for ρ routes that starts from p_i and ends at p_k, only one route is a candidate route. Considering all τ_c ($|\tau_c| = m + 2$) that contains τ (τ does not contain p_m), they are pruned. Hence, it prunes a large number of possible route. However, find the optimal points p_m for any two points p_i and p_k is time consuming. Thus, during the query processing phase, we store the current optimal point for two points in different categories and keep updating it. According to this, any 3-candidate routes that have a longer distance is pruned.

4.7.2 End Point Pruning

Let three candidate routes τ_a=$\{p_s, p_1, p_2, p_3, p_e\}$, τ_b=$\{p_s, p_2, p_1, p_3, p_4, p_e\}$, τ_c=$\{p_s, p_3, p_1, p_2, p_4, p_5, p_e\}$, these candidate routes are ending with the same point p_e. During the query processing, it is quite common to have a large number of candidate routes. Note that the number of points is $m \cdot \rho$, and it is relatively small compared to the number of candidate routes.

Lemma 4.7.2. *Let two candidate routes τ_a= $\{p_s, p_1, p_2, ..., p_j, p_e\}$ and τ_b=$\{p_s, p_1, p_2, ..., p_k, p_e\}$, $|\tau_a| \leq |\tau_b|$. $R_a \subseteq R_b$ ($R_a = \cup_{i=1}^{j}\pi(p_i)$, $R_b = \cup_{i=1}^{k}\pi(p_i)$)). If $c(\tau_a) \geq c(\tau_b)$, τ_a is pruned.*

Proof. To prune τ_a, we have to prove that for any candidate route τ_c ($|\tau_c|$=$m + 2$), a candidate τ_d ($|\tau_d|$) can be found to be shorter than τ_c. Let $\tau_c = \tau_a + \{p_{e+1}, p_{e+2}, ..., p_m, p_t\}$ and $R_c = C - R_a$. Accordingly, $\tau_d = \tau_b + \{p_{e+1}, p_{e+2}, ..., p_m, p_t\}$ and $R_c = C - R_b$. Note that $R_a \subseteq R_b$, we can get $R_d \subseteq R_c$. Since τ_d add the same points in the same order with τ_c and $c(\tau_a) \geq c(\tau_b)$, hence, $c(\tau_c) \geq c(\tau_d)$. On the other hand, $|\tau_d| > m$, we delete the points in $\{p_{e+1}, p_{e+2}, ..., p_m, p_t\}$ that has the same category with the points in τ_b and get a candidate route $\tau_{d'}$ ($|\tau_{d'}| = m + 2$). We can get $c(\tau_d) \geq c(\tau_{d'})$ since deleting a point in the route result is a shorter route. Finally, we can conclude that $c(\tau_c) \geq c(\tau_{d'})$. Hence, for any τ_c, there exists $\tau_{d'}$ that has a shorter distance.

□

EXAMPLE 11 : Take τ_a and τ_b as an example. τ_b visits one more category than τ_a which is p_4. Assuming that $c(\tau_a) \geq c(\tau_b)$ and m=6, let τ_c=$\{p_s, p_1, p_2, p_3, p_e, p_4, p_5, p_t\}$ and τ_d=$\{p_s, p_2, p_1, p_3, p_4, p_e, p_4, p_5, p_t\}$. We can get $c(\tau_c) \geq c(\tau_d)$. After deleting p_4 in τ_d, $\tau_{d'}$=$\{p_s, p_2, p_1, p_3, p_4, p_e, p_5, p_t\}$. Therefore, we can get $c(\tau_c) \geq c(\tau_{d'})$ that means τ_a is pruned. ■

4.7.3 Lower Bound for Candidate Route

For any candidate route τ_c that has not visited all categories, if we can predict the route distance from the last point of τ_c (p_l) to p_t by visiting the un-visited categories, it is very effective to prune a candidate route in an early stage. In this section, we use $c(\tau_c)$+$dist(p_l, p_t)$ as the lower bound of

τ_c. Note that for one point p_i, it may appear a lot of times at the end of τ_c, even though computing a single $dist(p_l, p_t)$ is efficient, we pre-compute $dist(p_i, p_t)$ ($p_i \in V$).

4.7.4 Upper Bound for Optimal Route

In this section, we discuss two fast approximation methods to quickly compute the candidate routes with length $m + 2$.

Minimum Distance Algorithm [59]. This is a fast approximation algorithm proposed in [59]. For each category C_i, it computes the point p_i ($p_i \in C_i$) that minimises the route distance of $\{p_s, p_i, p_t\}$. After retrieving one point from each category, the route starting from p_s visits the retrieved points in a nearest neighbor order. Note that the distances between p_i ($p_i \in V$) and p_t are pre-computed and we do the same as for p_s. Therefore, retrieving a candidate route using Minimum Distance Algorithm can be computed efficiently.

Minimum Route Expansion. In Section 4.7.1, we mentioned that if one point p_i is closer enough to the shortest path from p_s to p_t, the cost to add this point to the shortest path is relatively small. According to this intuition, we proposed a Minimum Route Expansion algorithm to quickly retrieve a candidate path with length $m + 2$. Let sp=$\{p_s, d_1, d_2, ..., d_i, p_t\}$ (how to compute the shortest path can be found in [75]) be the shortest path between p_s and p_t. For sp, only access doors on the shortest path is stored because the distances between p_i and ADN (N is any ancestor node of `Leaf(`p_i`)`) can be retrieved in $O(1)$ time by looking up the distance matrix of N.

Algorithm 13: Minimum Route Expansion

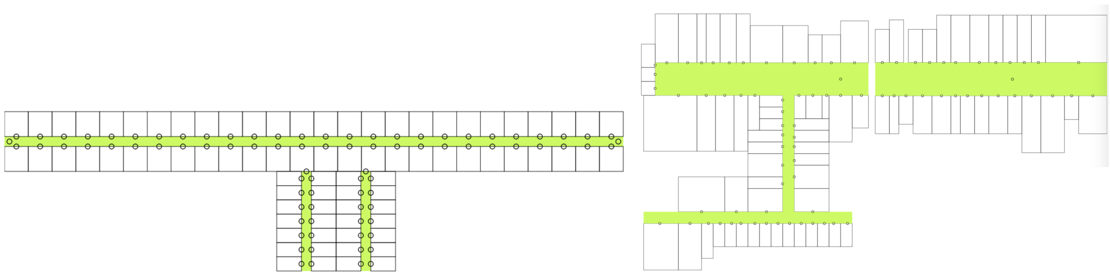
Input : sp: shortest path from p_s to p_t, VIP-Tree
Output : $distance$: upper bound for optimal route
1 Initialize $distance$ to $dist(p_s, p_t)$;
2 **for** each category C_i **do**
3   Initialize d_c=∞;
4   **for** each access door d_i in sp **do**
5         $N \longleftarrow$ node contains d_i;
6         $d = min_{p_j \in N, \pi(p_j)=C_i} dist(d_i, p_j)$;
7   **if** $d < d_c$ **then**
8             $d_c = d$;
9     $distance = distance + 2d_c$;
10 return $distance$;

Algorithm 13 shows the details on how to get the upper bound. Note that in line 5, N is not always the leaf node, it is the actual node that the shortest path passes through. After comparing the upper bounds computed by Minimum Distance Algorithm and Minimum Route Expansion, the smaller upper bound is used to prune the candidate routes. Meanwhile, during the query processing phase, once a candidate route with m categories has been found, we will use the distance of this route to update the upper bound.

4.8 Performance Evaluation

4.8.1 Experimental Settings

Indoor Space. We created three real datasets: Melbourne Central [6], Menzies building [9] and Clayton Campus [7]. 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 1280 rooms. Fig. 4.7(a) and 4.7(b) show the floor plans of the first floor in these two buildings. We use the application in [76] to visualize these two floor plans. The Clayton dataset corresponds to 71 buildings (including multilevel car parks) in Clayton campus of Monash University. We obtained the floor plans of all buildings and manually converted them into machine readable indoor venues. Coordinates of the buildings are obtained by using OpenStreetMap and the sizes of indoor partitions (e.g. rooms, hallways) are then determined. A three dimensional coordinate system is used where the first two represent x and y coordinates of indoor entities (e.g. rooms, doors) and the third represents the floor number. For the Clayton dataset, a D2D graph also contains edges between the entry/exit doors of different buildings, whereas the weight corresponds to the outdoor distance between the doors.



(a) Menzies building (b) Melbourne Central

Figure 4.7: Floor plans for two buildings

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 4.2 gives the 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.

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	1,280	56,009
Men-2	2 times Men	2722	2,560	112,062
CL	Clayton Campus	41,392	41,100	6,700,272
CL-2	2 times CL	83,138	82,540	13,400,884

Table 4.2: Indoor venues used in experiments

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

Dijkstra-based Expansion (DBE). A improved naive algorithm proposed in Section 4.4.

PNE [77]+G-tree [93]. PNE is used to solve OSR queries in spatial road networks. We revised it to process *i*TPQ as well. For the nearest neighbor search, the state-of-the-art method G-tree in spatial road network is employed.

PNE [77]+VIP-Tree [75]. Different from the previous one, PNE is implemented with the state-of-the-art method VIP-Tree in indoor space.

Queries and Indoor Points. To evaluate the performance of *i*TPQ, we randomly generated 50 pairs of start and end points. For the indoor points, we use both synthetic and real data. For the synthetic ones, we set the number of categories to 2, **3**, 4, 5, 6, while for the number of indoor points in each category is set to 10, 20, **30**, 40, 50, 60. The bold numbers are the default values. As discussed before, in an indoor space, the number of indoor points in a category is not large, thus, in our synthetic setting, maximum number of indoor points is set to be 60. On the other hand, when a user issues a *i*TPQ, the number of categories will not be a large value either, hence, we set the maximum number of categories to 6 – enough for indoor settings. To make the synthetic indoor points closer to real world scenarios, we vary the number of indoor points in each category. In each category, the numbers of indoor points are 5, 100, 55, 30, 75 and 45 respectively. For the real indoor points, we investigated the Monash University Clayton Campus and recorded the indoor points of 4 categories: printing rooms, restaurants, free study spaces and vending machines. The numbers of indoor points in these 4 categories are 65, 21, 28, 16.

4.8.2 Indexing Cost

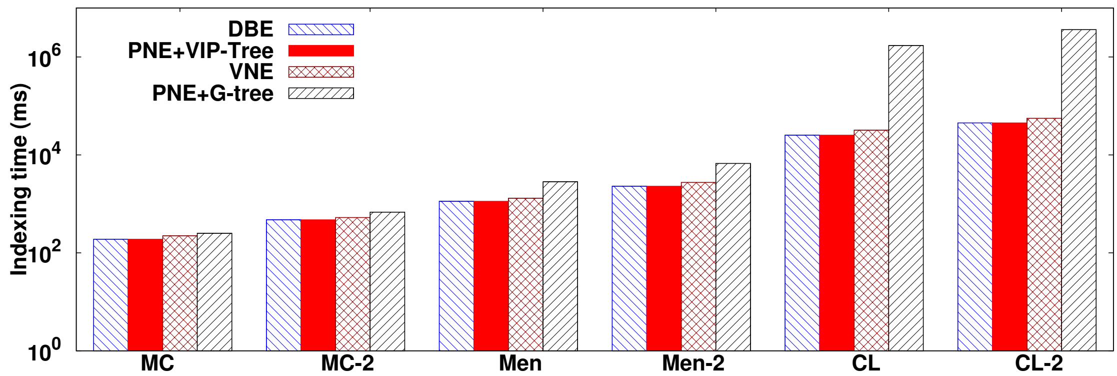
Construction time. Fig. 4.8(a) shows the construction cost for each method to build the indices according to the D2D graph and other information, such as rooms and doors. For DBE, as it is an expansion method, the distances between any two indoor points are computed by VIP-Tree since VIP-Tree is most efficient to perform indoor point-to-point shortest distance queries. Meanwhile, PNE+VIP-Tree utilizes VIP-Tree to compute the *k*NN queries. Therefore, both of these two methods have to construct VIP-Tree before the actual query processing. G-tree is constructed based on the D2D graph before it can be used to perform *k*NN queries used in PNE; Hence, the index construction time of PNE+G-tree is the time to build G-tree index. Our own method VNE has to build the VIP-Tree at first. After that, based on the indoor points and the above three lemmas (Lemma 4.6.1, 4.6.2 and 4.6.3), VIP-Tree will be updated to include the category information. Therefore, it takes more time compared to the time to build the original VIP-Tree. However, adding the category information to VIP-Tree after the VIP-Tree is constructed does not increase the time significantly. Thus, our proposed method takes similar construction times to those of the other methods. We can see that even for the largest datasets (CL-2) that consists of more than 83,000 doors and more than 13 million edges in the D2D graph, the construction time is less than 90 seconds, which is considered very efficient.

Indexing size. As discussed in previous section, both PNE+VIP-Tree and DBE utilize VIP-Tree, therefore, the indexing sizes of these two method are actually the same, which are the indexing size of the VIP-Tree. For PNE+G-Tree, it's indexing size is the same as G-tree. The indexing size of our own method is slightly larger than that for DBE and PNE+VIP-Tree as we add some extra information in the VIP-Tree. However, the extra information does not take a lot of storage cost and it is still better than G-tree. Fig. 4.8(b) shows the indexing sizes.

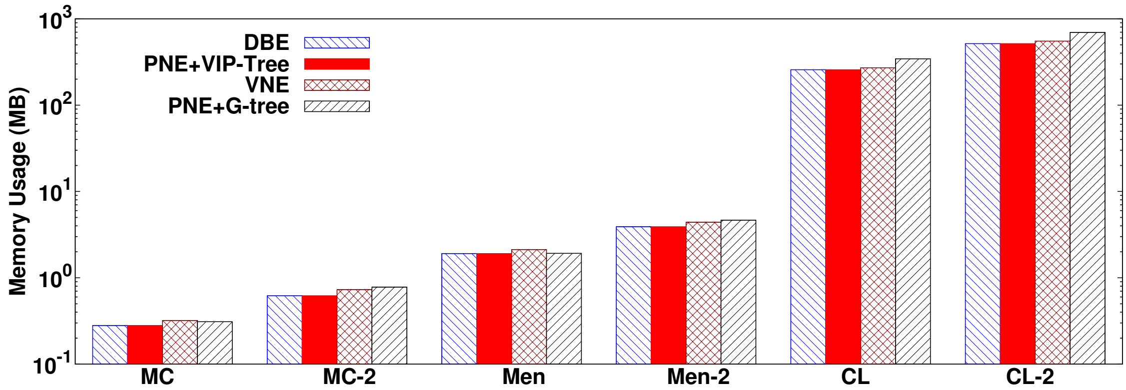
4.8.3 Query Performance

In this section, we compare the query processing time among the four methods by varying the number of indoor points, number of categories and datasets. We also tested these four methods using the real indoor points we collected at Monash University Clayton Campus.

Varying number of indoor points. The number of categories is set to be 3, and the dataset Men-2 is used. Fig. 4.9(a) shows the query processing time when the number of indoor points in each category changes from 10 to 60. DBE performs very badly as expected. When the number of indoor points in each category increases to 30, it takes more than 16 minutes to finish, which

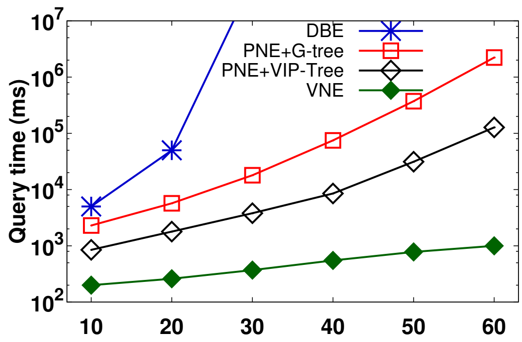


(a) Construction time

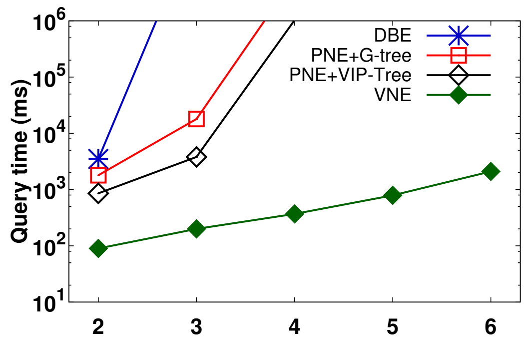


(b) Indexing size

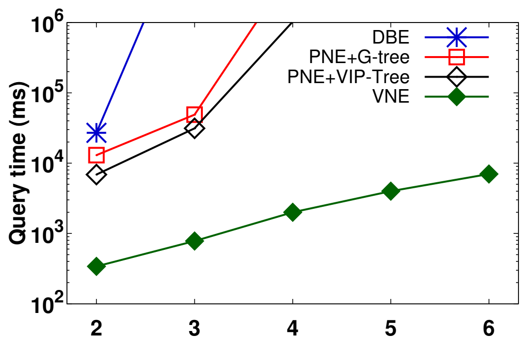
Figure 4.8: Indexing costs



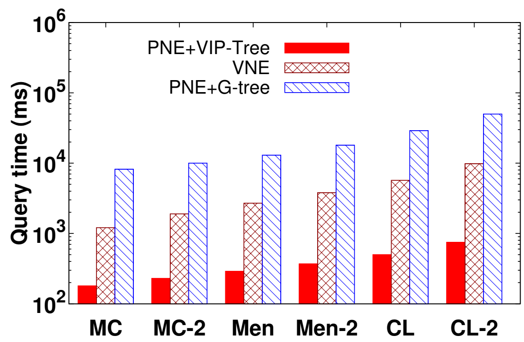
(a) # of indoor points



(b) # of categories



(c) # of categories



(d) Indoor venues

Figure 4.9: Indoor Trip Planning Queries

is far from acceptable. PNE+VIP-Tree has the closest query time compared to our method VNE. It is able to finish the *i*TPQ is slightly more than 100 seconds when the number of indoor points increases to 60. PNE+G-tree is slower than PNE+VIP-Tree which is expected because we have proven that VIP-Tree is much more efficient than G-tree, in terms of indoor *k*NN query processing as discussed in our previous paper [75]. Our own method, VNE, performs very well and it takes approximate 1 second to complete the *i*ITPQ, even though the number of indoors points increases to 60. Meanwhile, the processing time is quite stable with the increasing number of indoor points in each category. This is because no matter how many indoor points each category has, the 3-candidate pruning performs very well to find the optimal 3-candidate sub-routes. For the end point pruning, it ensures that the candidate routes are pruned, when the end points are the same and the visited categories are the subsets of the category set representing the longest route.

Varying number of categories. We set the number of indoor points in each category to 30, and Men-2 is used as the dataset. In Fig. 4.9(b), number of categories are ranging from 2 to 6. When there are only 2 categories, the three methods except our method finish in several seconds, while it only takes around 0.2 second for our method. This proves that the 3-candidate pruning and the end point pruning are effective. With the increasing number of categories, the other three methods become very slow because 1 category increased results in $(m+1)\rho$ (m is the number of categories before increasing by 1) times the previous number of possible candidate routes. However, for our method, since we employ several steps of pruning processes, the increasing number of possible candidate routes does not increases significantly. Therefore, for 6 categories, the processing time for our method performs far better than the other competitors.

Varying number of indoor points and categories. The number of indoor points are 5, 100, 55, 30, 75 and 45 respectively. For the *i*th category, the number of indoor points in this category is the *i*th number of indoor points. For example, 4th category has 30 points. The result is shown in Fig. 4.9(c). Our proposed algorithm is quite stable even though the numbers of indoor points in each category have a big difference. The processing time is longer than that in Fig. 4.9(b) since the average number of indoor points in Fig. 4.9(c) is around 50 while it is 30 in Fig. 4.9(b).

Varying datasets. We evaluate the four methods using the six datasets, while setting the number of indoor points to 30 and number of categories in 3. We exclude DBE in Fig. 4.9(d) since it always performs very badly. It clearly shows that changing of the datasets does not affect the query processing time dramatically. This is because the processing time for point-to-point shortest

	MC								CL							
# of points	PT_1	PT_2	PT_3	PT_4	PT_5	PT_6	PT_7	ALL	PT_1	PT_2	PT_3	PT_4	PT_5	PT_6	PT_7	ALL
10	0.5	0.3	21.2	49.8	63.1	20.2	9.1	86.3	0	0	23.1	49.4	64.3	19.3	8.7	87.6
20	1	0.3	22.9	49.1	64.3	19.1	8.2	87.1	0	0	24.9	49.7	64.8	19.8	8.2	89.1
30	1.5	0.3	26.1	50.8	63.5	18.8	7.7	88.3	0	0	27.2	50.1	65.2	19.8	8.0	90.6
40	1.8	0.3	28.3	50.1	65.7	19.2	7.2	89.8	0.1	0	29.5	50.3	65.7	19.2	7.5	91.9
50	2.6	0.3	33.7	51.2	65.1	20.3	6.5	92.5	0.1	0	34.4	51.4	66.3	20.5	7.1	93.6
60	3.3	0.4	37.2	51.7	66.4	21.1	6.4	94.8	0.1	0	37.9	52.3	67.4	21.3	6.8	95.8

Table 4.3: Pruning percentage for MC and CL varying # of points

distance and kNN queries using G-tree/VIP-Tree is quite stable for different datasets. However, our method is around 1 order of magnitude compared with the other two PNE based methods.

Real indoor points. We use CL dataset in this experiment. The number of indoor points in the four categories is mentioned in Section 4.8.1. For 2 categories, printing rooms and restaurant. Free study space is added to form 3 categories. All the 4 categories are used at the end. Fig. 4.10 shows the query processing time. We get the similar result here while using the real indoor points. This proves that in both synthetic and real settings, our algorithm performs very well and is more than 1 order of magnitude compared with the other three methods.

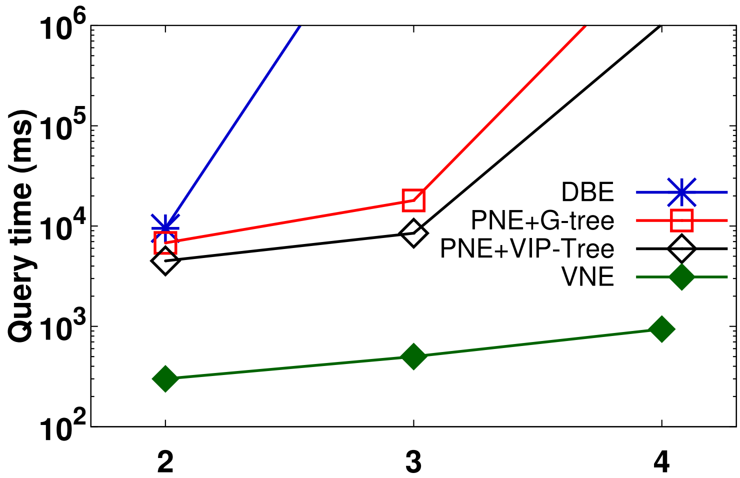


Figure 4.10: Real indoor points

4.8.4 Pruning Efficiency

In the query performance section (section 4.8.3), we have shown that our proposed algorithm VNE is much more efficient than the competitors. One of the main reason is that we have introduced the use of pruning techniques to eliminate a large number of candidates. Hence, in this section, we would like to evaluate the effectiveness of our pruning techniques.

	MC								CL							
# of categories	PT_1	PT_2	PT_3	PT_4	PT_5	PT_6	PT_7	ALL	PT_1	PT_2	PT_3	PT_4	PT_5	PT_6	PT_7	ALL
2	0.5	0.3	25.3	50.5	63.2	18.3	8.9	87.7	0	0	25.3	50.4	66.3	19.2	8.4	89.6
3	1.5	0.3	26.1	50.8	63.5	18.8	7.7	88.3	0	0	27.2	50.1	65.2	19.8	8.0	90.6
4	2.1	0.3	27.4	49.8	65.1	19.2	7.3	89.8	0	0	30.2	49.3	65.8	20.4	7.7	91.6
5	2.5	0.3	28.9	48.4	64.6	19.4	7.9	90.8	0	0	33.9	50.8	66.1	19.8	7.2	93.1
6	3.6	0.4	33.7	49.3	66.5	19.9	8.1	93.5	0.1	0	37.3	52.3	67.1	21.3	6.9	95.4

Table 4.4: Pruning percentage for MC and CL varying # of categories

# of categories	PT_1	PT_2	PT_3	PT_4	PT_5	PT_6	PT_7	ALL
2	0	0	24.5	50.2	65.6	20.2	9	88.5
3	0	0	26.7	49.7	68.7	19.5	8.2	90.3
4	0	0	29.2	49.5	69.3	19.8	7.7	91.9

Table 4.5: Pruning percentage for real points in CL varying # of categories

In the previous sections, we have explained seven pruning techniques: three pruning techniques used in the pre-processing phase, and the others used in the query processing phase. The seven pruning techniques are labeled as follows: (*i*) partition-based pruning Lemma 4.6.1 (PT_1), (*ii*) partition-based pruning Lemma 4.6.2 (PT_2), (*iii*) node-based pruning (PT_3), (*iv*) 3-candidate pruning (PT_4), (*v*) end point pruning (PT_5), (*vi*) lower bound pruning (PT_6), and (*vii*) upper bound pruning (PT_7). We have one more label for the combined seven pruning methods, denoted as ALL.

In the evaluation, we would like to see how much each pruning method can do the pruning job (PT_1 to PT_7), and how much the combined pruning method (ALL) can prune candidate routes. For evaluating the effectiveness, we use the percentage $\frac{number\ of\ pruned\ routes}{number\ of\ total\ possible\ routes}$ to represent the pruned percentage. The higher the percentage is, the more routes are pruned. To evaluate the effectiveness for the pruning techniques separately, we run the algorithm with only one pruning technique each time to get the pruned routes. Meanwhile, we randomly generate five synthetic indoor points each time. For example, five synthetic indoor points are generated when the number of categories is 3, the number of indoor points is 10. The MC and CL dataset is used in the experiment. After running the five sets of indoor points, the average percentage is reported.

Table 4.3 shows the prune percentage for MC and CL by varying the number of points, while Table 4.4 is the results by varying number of categories. For PT_1 and PT_2, they can only prune a very small percentage of the total routes. This is because the condition of these two techniques is that there are at least two points within the same category in the same room. Since in our datasets, this situation rarely happens, hence, PT_1 and PT_2 do not prune that much.

For the MC dataset, it consists of 297 rooms. 30 indoor points is relatively small. When the number of indoor points increases to 180, it is more likely that two indoor points within the same category are in the same room. Therefore, we can see that with the increasing number of indoor points and categories, pruning percentage for $PT1$ increases. However, for PT_2, since it must have 3 points in the same room, the percentage does not increase a lot. Their pruning percentages are not more than 4%, which is relatively very small. In Table 4.3, although the number of indoor points increases, the percentage for both PT_1 and PT_2 is almost 0 as the CL dataset contains more than 40, 000 rooms that makes it less possible to have two points in the same room.

PT_3 works much better than the previous two techniques as it does not rely on single partitions. In terms of the indexing tree nodes, it contains more points than an indoor partition, and it is more likely to have two points satisfying Lemma 4.6.3.

For PT_4 and PT_5, they are the two most effective ones which are expected. For any sub-route consisting of 3 points, there is only one optimal middle point. This means if there exist n points in the same category as the category of middle point, $n-1$ sub-routes with the same starting/end points are pruned. For PT_5, the condition is not strict. For example, for different candidate routes ending with the same point and visiting the same categories, only the shortest route is not pruned. Our experimental results show that these two pruning techniques can prune up to around 50% to 65% which is very effective.

For the rest of the two techniques (PT_6 and PT_7), they perform quite stable no matter what the number of categories and the number of indoor points are. On the other hand, when the same experiments are conducted using the CL dataset, we get a similar result although CL is much larger than MC. It means that our pruning techniques are scalable. For evaluating PT_6, we get the first route until one of the candidate route τ_c has reached to the destination. Therefore, for any candidate route de-heaped later, if the lower bound of the current candidate route is larger than $C(\tau_c)$, the route is pruned. While for PT_7, we compute the upper bound that is the shorter distance between the routes computed by two heuristic algorithms at the first place. When a candidate route is de-heaped, it is pruned only if its distance is larger than the upper bound. Since the length of de-heaped candidate route is shorter than $m+2$, it is less likely to be pruned. This is why using PT_7 only does not achieve a good pruning percentage compared with PT_6.

The percentage for *ALL* is around 90% in all the experimental settings, this proves that our proposed algorithm is quite efficient. Combining all pruning techniques together does not mean the pruning percentage is the percentage sum of all seven pruning techniques. This is because the candidate routes pruned by different pruning techniques have some intersections. On the other hand, different pruning techniques are able to help each other. For example, combing PT_6 and PT_7 achieve much better result because the upper bound computed by PT_7 can be utilized at the start of PT_6 such that PT_6 takes effects at the beginning, not waiting for a candidate route with length $m + 2$ is de-heaped.

For the real indoor points, we evaluate the effectiveness of the pruning techniques as well. Note that the number of points cannot be changed here, subsequently, we evaluate the real indoor points based on the different number of categories. The dataset used here is the CL dataset. Similar results are achieved compared to the synthetic indoor points and the overall pruned routes are around 90% shown in Table 4.5. As no indoor points are located in the same indoor partition, nothing is pruned by the partition-based pruning which are PT_1 and PT_2. For the other pruning techniques, similar results are achieved because the number of categories and points are in similar scales compared to synthetic indoor points.

4.9 Conclusion

In this chapter, we studied an new type of indoor query called the Indoor Trip Planning Query (*i*TPQ). Our proposed algorithm, called the VIP-Tree Neighbor Expansion (VNE) algorithm, exploits the features of indoor spaces, such as rooms and hallways. The pruning techniques used in VNE avoid processing a large number of unnecessary candidate routes. In the experimental section, we benchmark all the pruning techniques and show the effectiveness of our proposed pruning techniques. As a result, VNE performs much faster (by several orders of magnitude) than any competitor algorithms in both synthetic and real datasets in terms of processing time with low indexing cost. Additionally, VNE produces exact routes as the query results, instead of only approximation like any other TPQ algorithms for spatial road networks.

Chapter 5

KP-Tree: An Effective Index for Keyword Queries

5.1 Overview

In this chapter, we study a new type of indoor query, indoor boolean keyword query. Addition to the spatial-only queries such as shortest distance/path, *k* nearest neighbors and range queries, textual information is added to the indoor objects. For example, a pack of biscuits considered as an indoor objects can be tagged as 1"biscuit, pizza, flavour, creamy". For an indoor keyword query, the results are the *k* closest objects that contain the query keyword as well.

Consider an example in people's daily life, a person would like to buy a bottle of coke with raspberry flavour. "coke" and "raspberry" are considered as the query keywords. In a large shopping mall, several stores such as Woolworths and Coles sell raspberry coke, but these objects are only a small portion compared to the large number of products in the stores. A possible way to solve this problem is that the techniques handling *k* nearest neighbors query are utilized. The keywords of the objects are checked when the object is accessed. However, due to the small ration of objects containing keywords "coke" and "raspberry", it is not efficient because in the worst case, all objects have to be accessed. Another solution is that the objects are indexed based on the keywords information. The query algorithm only checks the objects that contains the objects. The problem is that for some frequent keywords, a large number of objects have to be accessed as well. Furthermore, the objects is not sorted such that all of them have to be accessed to retrieve the query results.

A few techniques have been proposed to solve the keyword queries in both Euclidean space and spatial networks. For Euclidean-based methods, they are not applicable to indoor space as the distance metric in indoor space is the minimum working distance. While for outdoor techniques that utilize the D2D graph, the efficiency becomes a big issue since we have proved that the state-of-the-art indexes ROAD and G-tree performs very bad for indoor queries. Driven by this, we extend our proposed VIP-Tree to solve indoor keyword queries, furthermore, a partition-specific index KP-Tree is discussed in this chapter to efficiently answer indoor keyword queries.

This chapter is organized as follows. In Section 5.2, we formally define the indoor boolean keyword queries and a few possible solutions are briefly described based on the existing techniques. In Section 5.3, the proposed index KP-Tree is discussed and the detailed algorithms is introduced. The detailed experimental evaluations are provided in Section 5.4 followed by the conclusion in Section 5.5.

5.2 Background Information

5.2.1 Problem Definition

In this chapter, 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. An indoor boolean kNN spatial keyword query (iBkNN-SK) is defined below.

Definition 5.2.1. iBkNN-SK Query. Given a set O of *spatio-textual* objects, a query object q where $q.loc$ is the spatial location and $q.\mathcal{T}$ is a set of query keywords, an iBkNN-SK is to find k closest objects to $q.loc$ that contains every keyword in $q.\mathcal{T}$.

Hereafter, whenever there is no ambiguity, we use o to refer to $o.loc$.

EXAMPLE 12 : Take Fig. 5.1 as an example. A set of *spatio-textual* objects $O = \{o_1, o_2, ..., o_{12}\}$ are located in the indoor venue. For simplicity, we ignore the labels of doors and partitions. 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. ■

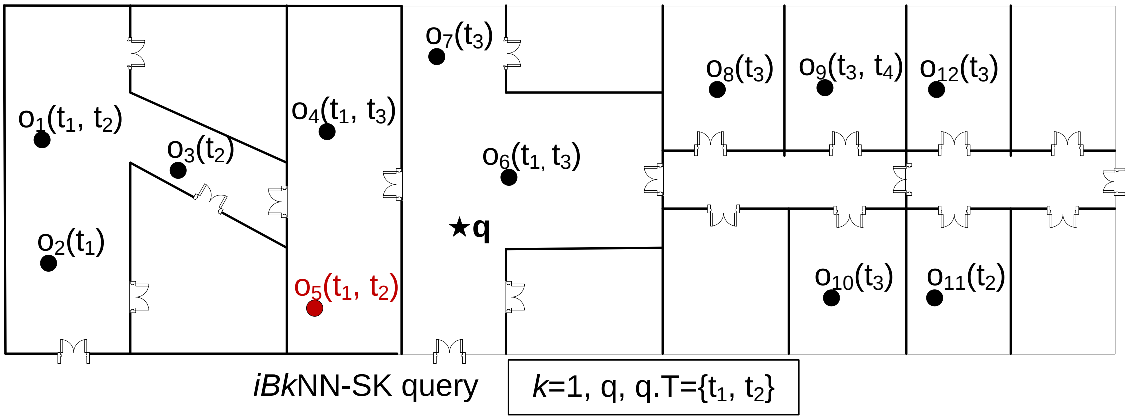


Figure 5.1: Example of *i*Boolean-*k*NN Query

5.2.2 Some possible solutions

To the best of our knowledge, we are the first to study the spatial keyword queries in indoor venues. In this section, we briefly discuss how to extend existing indoor/outdoor techniques and the VIP-tree to answer spatial keyword queries in indoor environment.

Extending Distance-aware Model (DistAw) [62]. The distance aware model is the state-of-the-art algorithm for indoor query processing. To solve *i*B*k*NN 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.

Extending DistAw++. In DistAw, indoor distances are computed during the expansion process. [62] utilized distance matrix to materialize indoor distances between any two doors with an extra $O(D^2)$ storage. This results in significant improvements in query processing time. Hence, we use DistAw++ to indicate a version of DistAw that uses a distance matrix to accelerate the query processing.

Extending G-tree [94]. 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 un-necessary 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 [28, 83], 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. 5.6(b) in Section 5.4.3) show that IVIP-tree is up to an order of magnitude better than all other approaches. This shows the effectiveness and adaptability of VIP-tree for different settings.

In next section, we propose another novel data structure called 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.

5.2.3 Contributions

In this chapter, we extended the previous VIP-Tree to Inverted VIP-Tree that solves indoor keyword queries. To further improve the efficiency, we designed keyword partitioning tree (KP-Tree) that indexes the objects inside the indoor partition.

Low indexing cost. For IVIP-Tree, the construction and storage cost is very small since it is extended from VIP-Tree. The partition-specific index KP-Tree requires low construction and storage cost. For example, for the largest dataset used in our experiments that consists of around 30,000 objects, KP-Tree requires only about 3 MBs and can be constructed within 4 seconds.

High efficiency. Among the existing techniques that build the index based on indoor venue only (objects are indexed based on inverted list), IVIP-Tree, the simple extension of VIP-Tree performs much better. For the indoor partition specific indexes that build the index for the objects in each partition and utilized IVIP-Tree for indexing the indoor venue, KP-Tree achieves the best performance. For the largest dataset that consists of about 140,000 with 60,014 unique keywords, it takes around 0.1 second for KP-Tree[1].

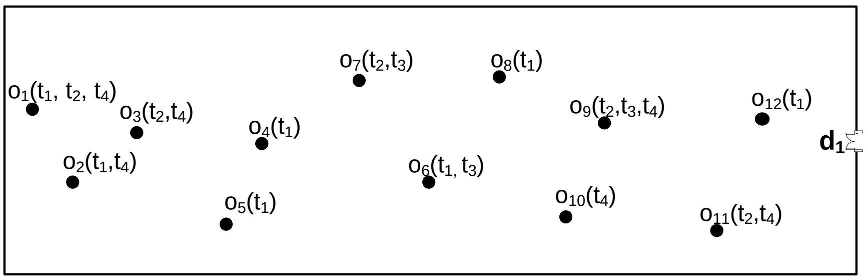
5.3 Keyword Partitioning Tree

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

[1]Under review for The VLDB Journal

library or medicines in a pharmacy etc. For example, in the real world data set that we use in 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 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. 5.2 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			

Inverted List for d_1

Figure 5.2: Inverted List

Another possible approach is to use spatial keyword indexes like IR-Tree [28] 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. 5.2, assume that a node groups the objects o_1, o_2 and o_3. 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 remark that there exist some indexing techniques such as WIR-Tree [84] 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. 5.2 contains all the keywords t_1, t_2, t_3 and t_4. Fig. 5.3(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.

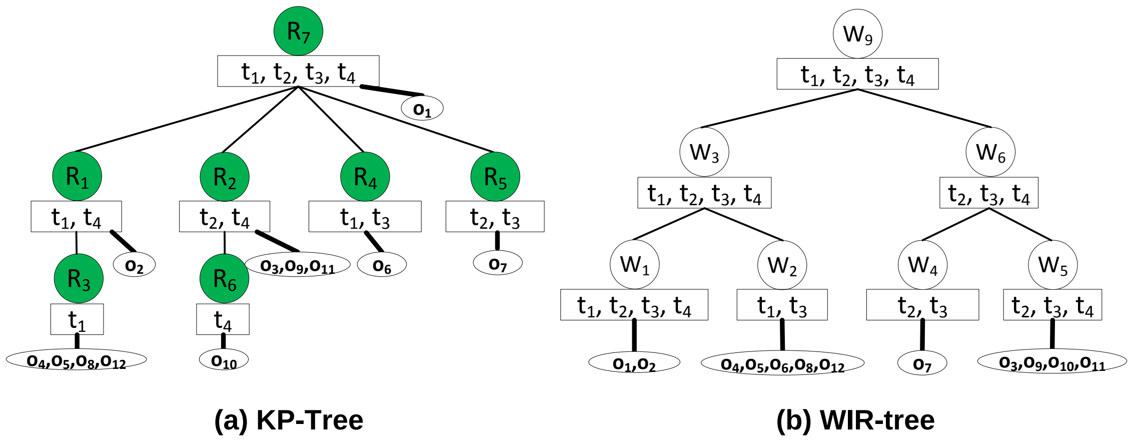


Figure 5.3: KP-Tree and WIR-tree for the objects in Fig. 5.2 except that we assume o_1 contains all four keywords

In this chapter, 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. 5.3(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 chapter, we use the objects in Fig. 5.2 as represented and do not assume that o_1 contains all keywords.

5.3.1 Overview of KP-Tree.

First, we give a brief overview of KP-Tree and some of its properties before formally describing its construction in next section. Fig. 5.4 is used as an example to illustrate KP-Tree for the objects in Fig. 5.2. Each node R in KP-Tree consists of a list of keywords represented as $R.\mathcal{T}$. The root node contains all unique keywords associated with objects in the partition. In Fig. 5.4, $R_9.\mathcal{T} = \{t_1, t_2, t_3, t_4\}$. For every node R, $R.\mathcal{T}$ is the union of the keywords contained in its children. For example, $R_9.\mathcal{T} = R_7.\mathcal{T} \cup R_8.\mathcal{T}$. 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. Specifically, KP-Tree has two kinds of nodes: *fruitful nodes* (shaded nodes) and *fruitless nodes* (white nodes). A fruitful node is a node that has some objects attached to it. On the other hand, a frutiless node does not have any object attached to it. In Fig. 5.4, 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. 5.4. 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 $e.\mathcal{T} \not\subseteq R.\mathcal{T}$. For each node retrieved from the 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

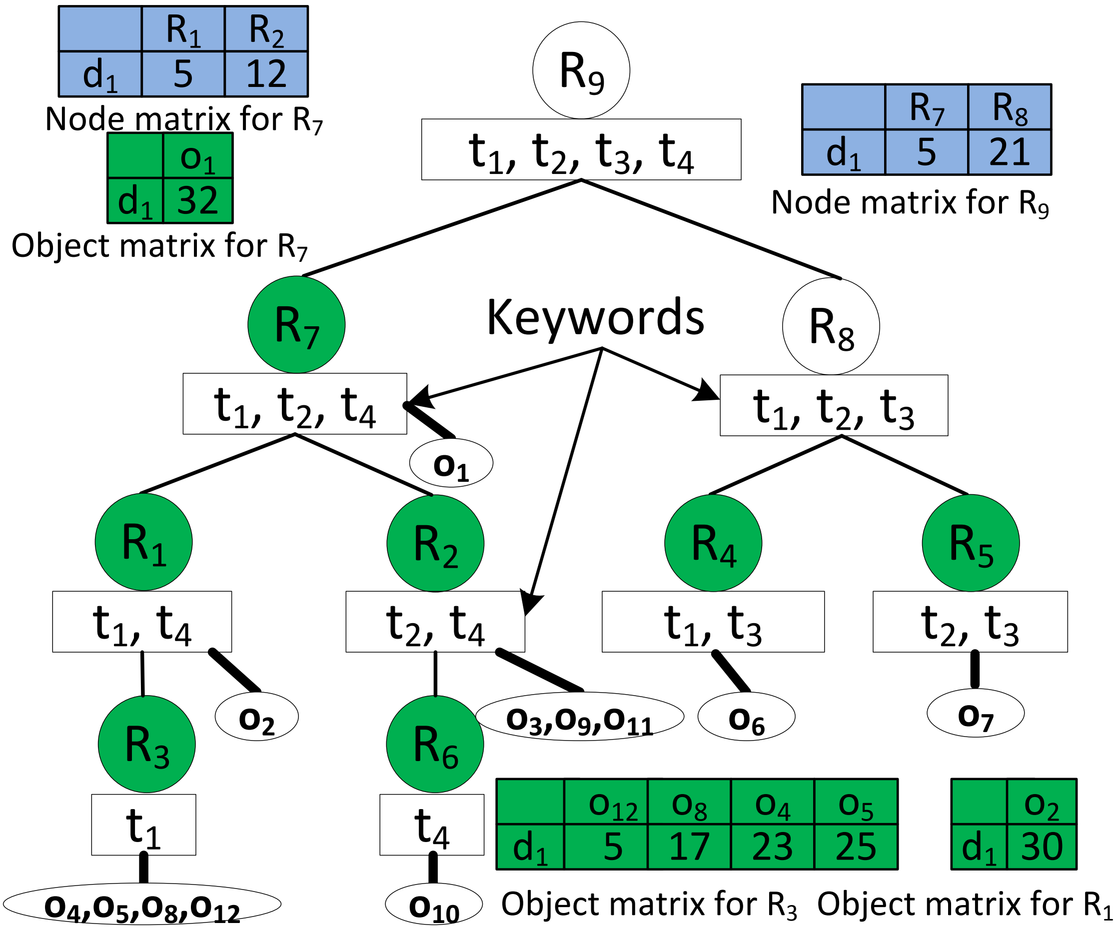


Figure 5.4: KP-Tree for the objects in Fig. 5.2

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 5.3.3, we present the details of how the VIP-Tree and KP-Tree are utilized to process queries in an indoor venue containing many paritions.

5.3.2 Constructing KP-Tree

The KP-Tree is constructed in 4 steps: 1) fruitful nodes are created by grouping the objects having exactly the same set of keywords; 2) *fruitful subtrees* are constructed using the fruitful nodes 3) the KP-Tree is constructed using a keyword graph and the fruitful subtrees constructed in the previous step; 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. 5.4, 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. 5.4, 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 14: Constructing sub-trees

Input : $\mathcal{R}$: a set of fruitful nodes

for each node $R \in \mathcal{R}$ in ascending order of # of keywords **do**
  choose a parent node R_p;
  if R_p is NULL **then**
    Set R as the root of its subtree;
  else
    set R_p as the parent of R in its subtree;

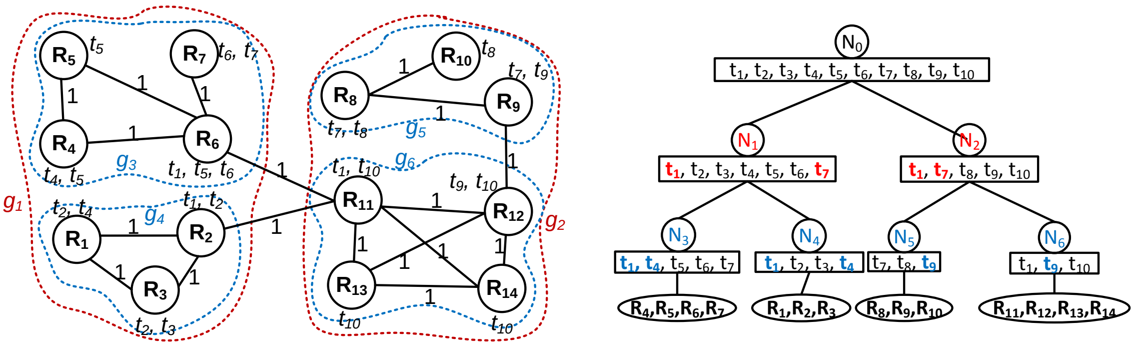
Algorithm 14 shows the details of constructing fruitful subtrees using a set of fruitful nodes $\mathcal{R}$. The nodes are accessed in ascending order of their number of keywords, i.e., the subtrees are constructed in a bottom-up approach. If there is no potential parent for a node R, it indicates that this node is the root node for a fruitful subtree. For example, in Fig. 5.4, 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).

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. We aim to minimize the overlap of keywords among children to ensure effectiveness of the KP-tree for querying. We use a *keyword graph* to guide the KP-Tree construction algorithm. 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.

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. 5.4 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 [50] for graph partitioning.

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. 5.5(a) shows a sample keyword graph. To avoid mixup between the nodes in KP-Tree and vertices in the keyword graph, in this example, 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 $(t_1, \cdots, t_{10})$ of the keyword graph. Assuming $f = 2$, the graph is partitioned into two graphs g_1 and g_2 as shown in Fig. 5.5(b) minimizing the total weight of the edges connected 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 $t_1, \cdots, t_7$) 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 keywords in N_1 and N_2 are shown in bold). Next, the children of N_1 are computed by 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.



(a) Graph partition (b) KP-Tree

Figure 5.5: Keyword graph ($R_1 = \{t_2, t_4\}$, $R_2 = \{t_1, t_2\}$, $R_3 = \{t_2, t_3\}$, $R_4 = \{t_4, t_5\}$, $R_5 = \{t_5\}$, $R_6 = \{t_1, t_5, t_6\}$, $R_7 = \{t_6, t_7\}$, $R_8 = \{t_7, t_8\}$, $R_9 = \{t_7, t_9\}$, $R_{10} = \{t_8\}$, $R_{11} = \{t_1, t_{10}\}$, $R_{12} = \{t_9, t_{10}\}$, $R_{13} = \{t_{10}\}$, $R_{14} = \{t_{10}\}$)

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. 5.2 which only has one door d_1. For node R_3 in Fig. 5.4, 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. 5.4, 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 distance matrices of the children.

Tree update. In real world, due to construction or relocation, items in a shop may change the location. This requires the proposed index to be updated accordingly. However, the change is not happened frequently. In the experiment part, we have shown that the tree updating time for

VIP-Tree is less than one second. Meanwhile, the construction time for KP-Tree is in seconds which achieves enough efficiency to re-build the index.

5.3.3 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 indoor boolean kNN (iBkNN) query is to use an algorithm very similar to our kNN algorithm (Algorithm 5) 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 15 shows our proposed algorithm to answer indoor boolean kNN (iBkNN) queries. Similar to our kNN algorithm, 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

Algorithm 15: indoor boolean kNN query

```
Input  : q: query point, k
Output : kNNs
d^k = ∞;                          /* d^k is dist. to current k^th NN */;
getDistances(q,root);             /* Algorithm 2 */;
Initialize a heap H with root of the IVIP-tree;
while H is not empty do
    de-heap an entry N from heap;
    if mindist(q, N) ≥ d^k then
        return kNN;
    if N is a non-leaf node of IVIP-tree then
        for each child N' of N do
            if q.T ⊆ N'.T then
                insert N' in heap with mindist(q, N');
    if N is a leaf node of IVIP-tree then
        for each partition P_i in N do
            R_i ← root node of KP-Tree of P_i;
            if q.T ⊆ R_i.T then
                insert R_i in heap with mindist(q, R_i);
    if N is a node of KP-Tree then
        for each child N' of N do
            if c.T ⊆ N'.T then
                insert N' in heap with mindist(N', c);
        if N is a fruitful node then
            for each object o associated with it do
                if q.T ⊆ o.T then
                    insert o in heap with dist(q, o);
    if N is an object then
        add the object to kNN and update d^k;
```

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.

5.4 Experiments

In this section, we provide the detailed experimental evaluations between our proposed methods and other competitors.

5.4.1 Experimental Settings

Indoor Venue and Keyword Datasets. We use Chadstone Shopping Centre [8] 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. Coordinates of the buildings are obtained by using OpenStreetMap and the sizes of indoor partitions (e.g., rooms, hallways) are determined. A three dimensional coordinate system is used where the first two represent x and y coordinates of indoor entities (e.g., rooms, doors) and the third represents the floor number. 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 5.1.

Category	**Store Name**	**# unique products**	**# unique keywords**
Technology	EBGames (EB)	12,848	8,432
	JB Hifi (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

Table 5.1: Details of Stores

We use these stores to obtain several real world object data sets. Table 5.2 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 Hifi) 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.

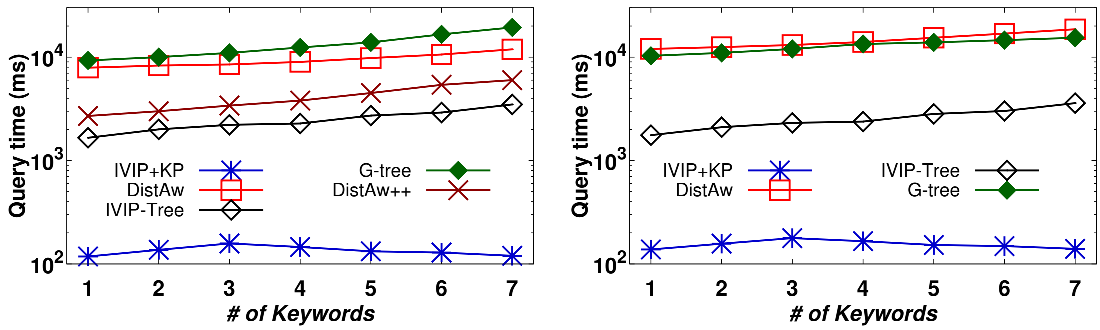
Dataset	Vocabulary size	# products
TS	35,803	72,539
TSH	50,056	105,394
TSHLP	60,014	139,899

Table 5.2: Details of keyword datasets

To evaluate our algorithms on larger indoor venues, we use Monash University Clayton Campus as the indoor venue and, for each of the object datasets in Table 5.2, 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 [21]. 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 *i*B*k*NN-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 5.4.2, 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 5.4.3, 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).



(a) Chadstone (b) Clayton

Figure 5.6: Effect of # keywords

5.4.2 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 [62]. As described before, DistAw utilizes the AB graph of the indoor venue and keywords information is embedded for each partition.

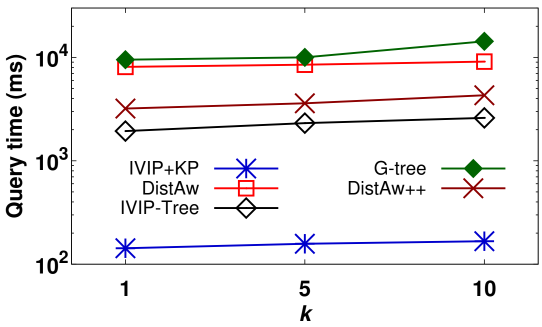
DistAw++. To accelerate distance computations, distance matrix is used to compute the distances between any two doors in the indoor venue. However, it need $O(D^2)$ storage cost for the distance matrix where D is the number of doors in the indoor venue.

G-tree [94]. 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.

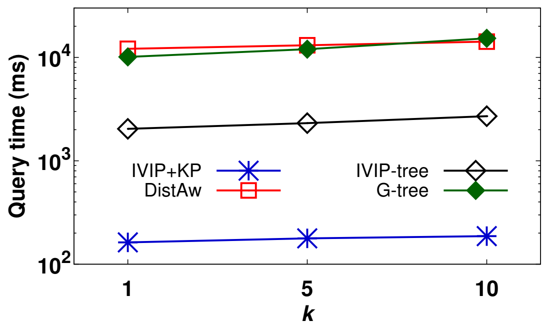
IVIP-Tree and IVIP+KP: IVIP-Tree is our venue-level index which, like other competitors in this section, uses inverted lists for each indoor partition. We also show the performance of IVIP-Tree when it uses KP-Tree to index the objects in each partition. This is shown as IVIP+KP in the figures.

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 only queries in the previous section.

Results. Fig. 5.6, 5.7 and 5.8 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

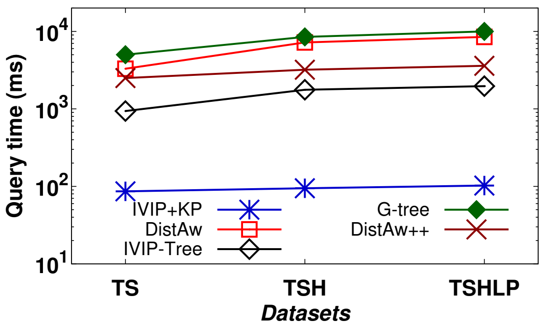


(a) Chadstone

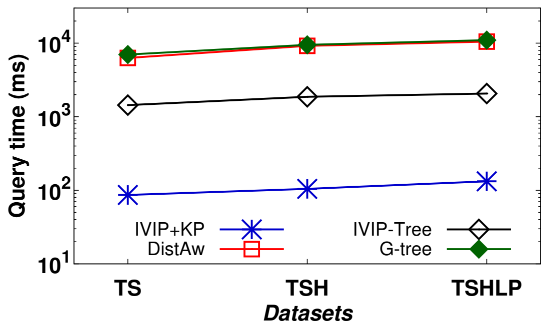


(b) Clayton

Figure 5.7: Effect of k



(a) Chadstone



(b) Clayton

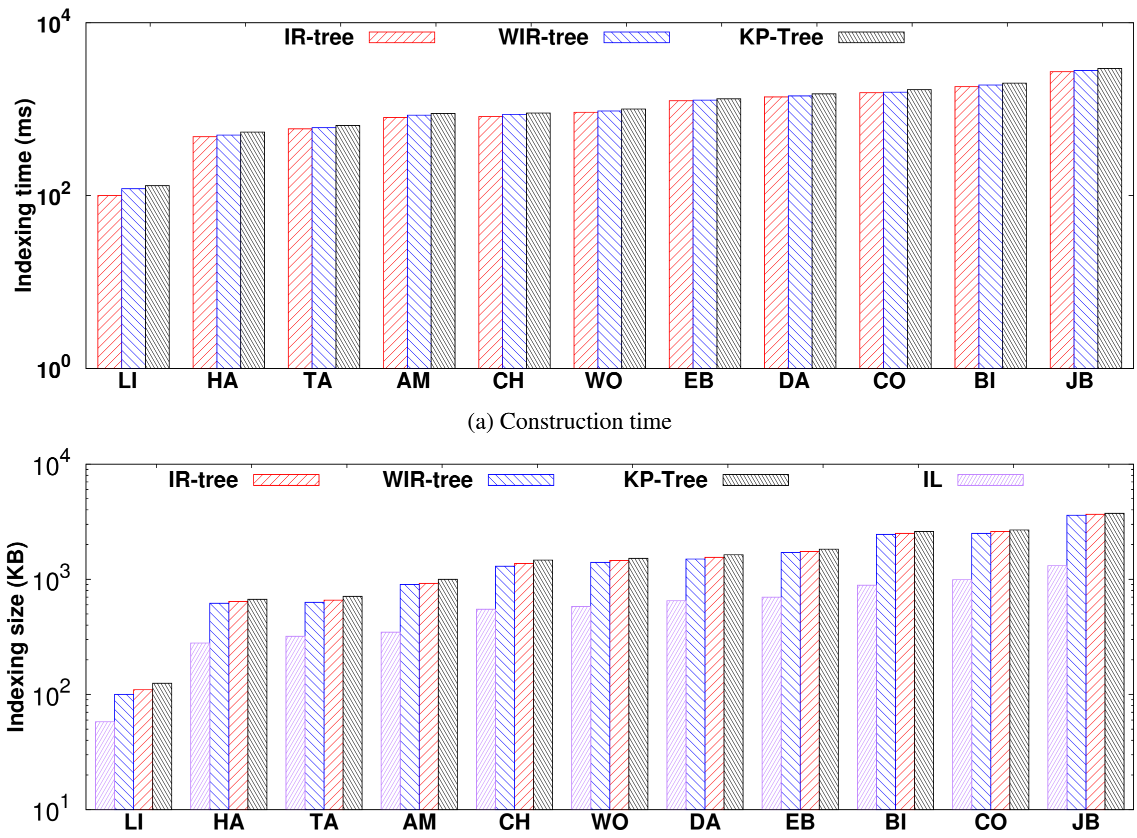
Figure 5.8: Effect of object data sets

shows the effectiveness of our venue-level index IVIP-Tree as well as our partition-specific index KP-Tree. As mentioned earlier, DistAw++ is only available for smaller indoor venues due to the $O(D^2)$ construction time and storage requirement. Therefore, results for DistAw++ are not shown for the Clayton data set.

Fig. 5.6(a) 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 [21] for some spatial keyword query processing techniques in Euclidean space.

5.4.3 Evaluating Partition-Specific Indexes

Partition-specific indexes. To evaluate partition-specific indexes, we use IVIP-Tree to index the indoor venue and use the following techniques to index objects in each partition.



(a) Construction time

(b) Index size

Figure 5.9: Indexing Cost

Inverted Lists (IL). IVIP-Tree is used to index indoor venue and, for each partition, inverted lists are created to index objects in the partition.

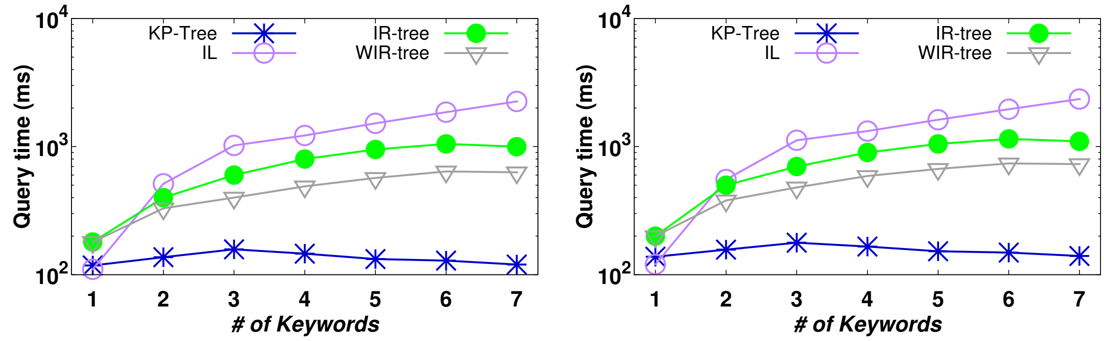
IR-tree [28]. Indoor venue is indexed by IVIP-Tree and, for each indoor partition, objects are indexed by IR-tree.

WIR-tree [84]. IVIP-Tree is utilized for indexing the indoor venue and, for each indoor partition, objects in it are indexed by WIR-tree.

KP-Tree. IVIP-Tree indexes the indoor venue and, for each partition, KP-Tree is used to index indoor objects.

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 the maximum number of fruitful nodes in the leaf node is set to 32.

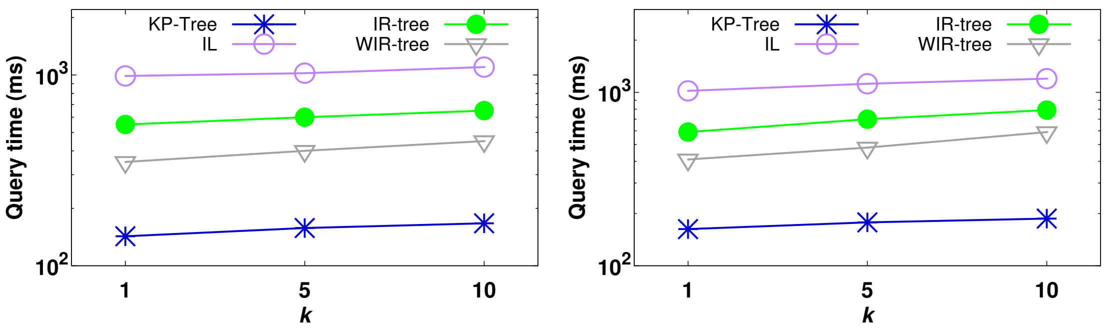
Indexing cost. Fig. 5.9 compares the construction time and index size for each indexing technique for different stores in our data sets (see Table 5.1 for the details of each abbreviation). The stores on x-axis are listed in increasing order of the total number of unique products in each store. As expected, inverted lists can be constructed significantly more efficiently as compared to other approaches because the construction cost mainly consists of sorting each list based on distances



(a) Chadstone

(b) Clayton

Figure 5.10: Effect of # keywords



(a) Chadstone

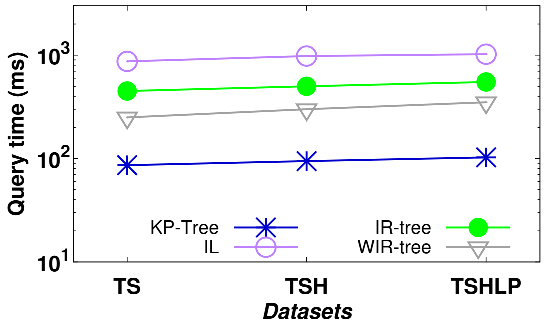
(b) Clayton

Figure 5.11: Effect of k

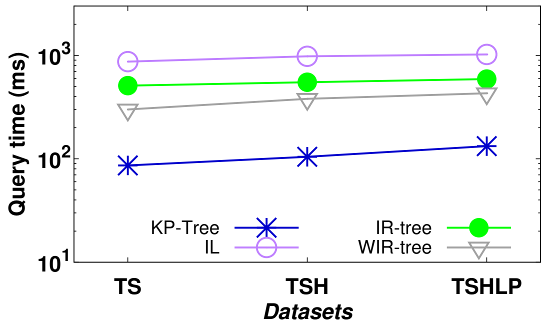
of the objects from each door in the partition The index size of inverted list is also the smallest. The construction time and index size of KP-tree is comparable with other approaches although a little higher. For the biggest store (JB HiFi) containing around $30,000$ unique products, KP-tree is constructed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

Querying cost. Fig. 5.10, 5.11 and 5.12 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. 5.10 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.



(a) Chadstone



(b) Clayton

Figure 5.12: Effect of object sets

5.5 Conclusion

In this chapter, we extend the previous VIP-Tree to IVIP-Tree by embedding keyword information on the nodes and inverted list is utilized to index the objects in the indoor partitions. After that, KP-Tree is proposed to build the index for the objects inside one indoor partition. According to the proposed KP-Tree, a detailed algorithm is developed to solve indoor keyword queries. The experimental evaluation shows that IVIP-Tree performs better compared to the existing techniques that do not have specific indexes for objects in the partition. For KP-Tree, it is much efficient compared to the other algorithms, both indoor indexes and partition-specific indexes.

Chapter 6

Final Remarks

6.1 Overview

In this thesis, we present two efficient indexes for indoor space that support efficient indoor query processing. Chapter 3 present our research on the indexes for indoor space and efficient algorithms are proposed to solve the spatial queries like shortest distance/path, k nearest neighbors and range queries. In Chapters 4 and 5, two more advanced indoor spatial queries are studied: indoor trip planning queries and indoor boolean keyword queries. This chapter is organized as follows. We gives the detailed contributions in Section 6.2. Section 6.3 presents two possible directions for future works.

6.2 Contributions

In Chapter 3, we carefully exploit the unique properties of indoor space and propose the IP-Tree that indexes the indoor space using a tree structure. To further improve the query processing efficiency, VIP-Tree is proposed by materilizing the distances. The two proposed indexes achieve low theoretical complexities. Based on the indexes, efficient algorithms have been studied to solve shortest distance/path, k nearest neighbors and range queries. During the experimental evaluation, our proposed indexes outperform the other state-if-the-art techniques in both indoor and outdoor space by several orders of magnitude.

In Chapter 4, indoor trip planning query is studied. As trip planning query is proved to be NP-hard, the exact algorithms in spatial networks achieve very low efficiency. In indoor space, due to the limited number of objects in each category, we proposed a expansion-based algorithms with efficient pruning techniques. A large number of distances computations are required to solve trip planning queries, VIP-Tree is utilized to handle the distance computation. Hence, the storage cost is low since the index of VIP-Tree is used. The experimental evaluation shows that our proposed algorithms outperform the existing techniques. Meanwhile, the proposed pruning techniques are proved to be effective due to the large pruned ration of candidate routes.

In Chapter 5, another type of query, indoor boolean keyword query is discussed. Spatial keyword queries are discussed mostly in Euclidean distances since the efficiency of distance computations. No previous work has been done in indoor space. We perform a simple extension for VIP-Tree to solve indoor boolean keyword query. Furthermore, KP-Tree is proposed to index the objects in the indoor partition. An efficient algorithm is developed based on IVIP-Tree and KP-Tree. In the experimental evaluation, IVIP-Tree, the simple extension of VIP-Tree, outperforms the techniques according to indoor indexes such as G-tree. Meanwhile, for KP-Tree, it achieves comparable construction time storage cost compared with the existing indexes such IR-tree and WIR-tree, but KP-Tree outperforms the other techniques much better.

6.3 Directions of Future Work

In this section, we propose several possible directions for future works.

6.3.1 Query Processing on Both Indoor and Outdoor Space

In this thesis, we have studied a few indoor spatial queries and efficient algorithm have been proposed. In spatial networks, spatial queries are much well studied. However, in real applications, indoor and outdoor spaces are not seperate. For example, a student park his car in the car parking area in Monash University Clayton Campus. He wants to go to the lecture theatre in Menzies building. Assume the student does not have any knowledge about the spatial network in the campus and the indoor map of Menzies building. If only outdoor technique is utilized, he will lost after he enters into Menzies building. On the other hand, while only indoor techniques are utilized, he has no idea about how to reach to Menzies building. Hence, an efficient algorithm that handles both indoor and outdoor space is required. There is an existing research that combines the spatial network with the D2D graphs transferred from indoor venues [46]. The problem is that the combined graph is very large, the Dijkstra based algorithm proposed in the paper is not efficient. No other works have been done on this problem, hence, an efficient algorithm is required.

6.3.2 Continuous Queries

Continuous query is another important spatial query. In indoor space, it is valuable for vendors. For example, in a large shopping centre, vendors want to send promotions to the potential customers. Meanwhile, vendors do not want to send the promotions to the customers who are far away since they are less likely to come to the stores. According to vendor locations, we need to continuously get the customers' locations in order to send them the promotion information. There is no existing researches on continuous queries in indoor space yet. Hence, in near future, more techniques will be proposed on indoor continuous queries.

References

[1] `http://www.ourjiangsu.com/a/20171006/1507251980426.shtml`.

[2] `https://maps.google.com.au/`.

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

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

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

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

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

[8] `https://www.chadstone.com.au/directions/centre-map`.

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

[10] `https://developers.google.com/maps/documentation/geocoding/intro`.

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

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

[13] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. k-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *PVLDB*, 2016.

[14] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Experimental Algorithms* -

10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings, pages 230–241, 2011.

[15] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Hierarchical hub labelings for shortest paths. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 24–35, 2012.

[16] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013.

[17] Asif Iqbal Baba, Hua Lu, Torben Bach Pedersen, and Xike Xie. Handling false negatives in indoor RFID data. In *MDM*, pages 117–126, 2014.

[18] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 322–331, 1990.

[19] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 1(1):7–28, 2004.

[20] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. The multi-rule partial sequenced route query. In *16th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2008, November 5-7, 2008, Irvine, California, USA, Proceedings*, page 10. ACM Press, New York, NY, USA, 2008.

[21] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*.

[22] Lisi Chen, Gao Cong, Christian S Jensen, and Dingming Wu. Spatial keyword query processing: an experimental evaluation. In *PVLDB*, 2013.

[23] Yen-Yu Chen, Torsten Suel, and Alexander Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.

[24] Maria Christoforaki, Jinru He, Constantinos Dimopoulos, Alexander Markowetz, and Torsten Suel. Text vs. space: efficient geo-search query processing. In *Proceedings of the*

20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011, pages 423–432, 2011.

[25] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 937–946, 2002.

[26] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[27] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[28] Gao Cong, Christian S. Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1), 2009.

[29] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[30] Zhixiang Fang, Qingquan Li, Xing Zhang, and Shih-Lung Shaw. A GIS data model for landmark-based pedestrian navigation. *International Journal of Geographical Information Science*, 26(5):817–838, 2012.

[31] Jay Farrell and Matthew Barth. *The global positioning system and inertial navigation*, volume 61. Mcgraw-hill New York, NY, USA:, 1999.

[32] Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Cleaning trajectory data of rfid-monitored objects through conditioning under integrity constraints. In *EDBT*, pages 379–390, 2014.

[33] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[34] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[35] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms,*

7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings, pages 319–333, 2008.

[36] Richard Göbel, Andreas Henrich, Raik Niemann, and Daniel Blank. A hybrid index structure for geo-textual searches. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 1625–1628, 2009.

[37] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

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

[39] Ramaswamy Hariharan, Bijit Hore, Chen Li, and Sharad Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *19th International Conference on Scientific and Statistical Database Management, SSDBM 2007, 9-11 July 2007, Banff, Canada, Proceedings*, page 16, 2007.

[40] Harlan Hile, Radek Grzeszczuk, Alan L. Liu, Ramakrishna Vedantham, Jana Kosecka, and Gaetano Borriello. Landmark-based pedestrian navigation with enhanced spatial reasoning. In *Pervasive Computing*, pages 59–76, 2009.

[41] Bernhard Hofmann-Wellenhof, Herbert Lichtenegger, and James Collins. *Global positioning system: theory and practice*. Springer Science & Business Media, 2012.

[42] J. B. Rocha-Junior *et al.* Efficient processing of top-k spatial keyword queries. In *SSTD*, 2011.

[43] Peggy L Jenkins, Thomas J Phillips, Elliot J Mulberg, and Steve P Hui. Activity patterns of Californians: Use of and proximity to indoor pollutant sources. *Atmospheric Environment. Part A. General Topics*, 1992.

[44] Peggy L Jenkins, Thomas J Phillips, Elliot J Mulberg, and Steve P Hui. Activity patterns of californians: use of and proximity to indoor pollutant sources. *Atmospheric Environment. Part A. General Topics*, 26(12):2141–2148, 1992.

[45] Christian S. Jensen, Hua Lu, and Bin Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD 2009, Aalborg, Denmark, July 8-10, 2009, Proceedings*, pages 208–227. Springer, Berlin Heidelberg, 2009.

[46] Søren Kejser Jensen, Jens Thomas Vejlby Nielsen, Hua Lu, and Muhammad Aamir Cheema. Outdoor-indoor space: unified modeling and shortest path search. In *Proceedings of the Eighth ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness, ISA@SIGSPATIAL 2016, Burlingame, California, USA, October 31, 2016*, pages 35–42, 2016.

[47] Minhao Jiang, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. Exact top-k nearest keyword search in large networks. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 393–404, 2015.

[48] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 500–509, 1994.

[49] Yaron Kanza, Eliyahu Safra, and Yehoshua Sagiv. Route search over probabilistic geospatial data. In *International Symposium on Spatial and Temporal Databases*, pages 153–170. Springer, 2009.

[50] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, page 29, 1995.

[51] Norio Katayama and Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 369–380, 1997.

[52] Ali Khodaei, Cyrus Shahabi, and Chen Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *Database and Expert Systems Applications, 21st International Conference, DEXA 2010, Bilbao, Spain, August 30 - September 3, 2010, Proceedings, Part I*, pages 450–466, 2010.

[53] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 840–851. VLDB Endowment, 2004.

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

[55] Ken C. K. Lee, Wang-Chien Lee, and Baihua Zheng. Fast object search on road networks. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 1018–1029, 2009.

[56] Ken C. K. Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. ROAD: A new spatial object search framework for road networks. *IEEE Trans. Knowl. Data Eng.*, 24(3):547–560, 2012.

[57] Roy Levin, Yaron Kanza, Eliyahu Safra, and Yehoshua Sagiv. Interactive route search in the presence of order constraints. *PVLDB*, 3(1):117–128, 2010.

[58] Fan Li, Chunshui Zhao, Guanzhong Ding, Jian Gong, Chenxing Liu, and Feng Zhao. A reliable and accurate indoor localization method using phone inertial sensors. In *The 2012 ACM Conference on Ubiquitous Computing, Ubicomp '12, Pittsburgh, PA, USA, September 5-8, 2012*, pages 421–430. ACM, New York, NY, USA, 2012.

[59] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *Advances in Spatial and Temporal Databases*, pages 273–290. Springer, 2005.

[60] Guoliang Li, Jianhua Feng, and Jing Xu. Desks: Direction-aware spatial keyword search. In *ICDE*, 2012.

[61] Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lun Lee, and Xufa Wang. Ir-tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[62] Hua Lu, Xin Cao, and Christian S. Jensen. A foundation for efficient indoor distance-aware query processing. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 438–449. IEEE Computer Society Press, Washington, DC, USA, 2012.

[63] Hua Lu, Xin Cao, and Christian S Jensen. A foundation for efficient indoor distance-aware query processing. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 438–449. IEEE, 2012.

[64] Hua Lu, Bin Yang, and Christian S. Jensen. Spatio-temporal joins on symbolic indoor tracking data. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 816–827. IEEE Computer Society Press, Washington, DC, USA, 2011.

[65] Pratap Misra and Per Enge. Global positioning system: Signals, measurements and performance second edition. *Massachusetts: Ganga-Jamuna Press*, 2006.

[66] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 802–813. VLDB Endowment, 2003.

[67] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 802–813. Morgan Kaufmann, San Francisco, CA, USA, 2003.

[68] Ira Pohl. *Bi-directional and heuristic search in path problems*. PhD thesis, Dept. of Computer Science, Stanford University., 1969.

[69] João B. Rocha-Junior and Kjetil 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.

[70] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM, 1995.

[71] S. B. Roy *et al.* Location-aware type ahead search on spatial databases: semantics and efficiency. In *SIGMOD*, 2011.

[72] Maytham Safar. K nearest neighbor search in navigation systems. *Mobile Information Systems*, 1(3):207–224, 2005.

[73] Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. Efficient query processing on spatial networks. In *Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 200–209. ACM, 2005.

[74] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518, 1987.

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

[76] Zhou Shao, Joon Bum Lee, David Taniar, and Yang Bo. A real time system for indoor shortest path query with indexed indoor datasets. In *Databases Theory and Applications - 27th Australasian Database Conference, ADC 2016, Sydney, NSW, September 28-29, 2016, Proceedings*, pages 440–443. Springer, Berlin Heidelberg, 2016.

[77] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. The optimal sequenced route query. *The VLDB journal*, 17(4):765–787, 2008.

[78] Nathan R. Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1392–1397, 2005.

[79] Subodh Vaid, Christopher B. Jones, Hideo Joho, and Mark Sanderson. Spatio-textual indexing for geographical search on the web. In *Advances in Spatial and Temporal Databases, 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005, Proceedings*, pages 218–235, 2005.

[80] Wei Wang, Jiong Yang, and Richard R. Muntz. Pk-tree: A spatial index structure for high dimensional point data. In *The 5th International Conference of Foundations of Data Organization (FODO'98), Kobe, Japan, November 12-13, 1998*, pages 27–36, 1998.

[81] Xiaolin Wang, Yingwei Luo, Lishan Yu, and Zhuoqun Xu. PK+ tree: An improved spatial index structure of PK tree. In *Computational Science - ICCS 2005, 5th International Conference, Atlanta, GA, USA, May 22-25, 2005, Proceedings, Part III*, pages 511–514, 2005.

[82] Martin Werner. *Indoor Location-Based Services: Prerequisites and Foundations*. Springer, Berlin Heidelberg, 2014.

[83] Dingming Wu, Gao Cong, and Christian S. Jensen. A framework for efficient spatial web object retrieval. *VLDB J.*, 21(6):797–822, 2012.

[84] Dingming Wu, Man Lung Yiu, Gao Cong, and Christian S. Jensen. Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.*, 24(10):1889–1903, 2012.

[85] Xike Xie, Hua Lu, and Torben Bach Pedersen. Efficient distance-aware query evaluation on indoor moving objects. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 434–445. IEEE Computer Society Press, Washington, DC, USA, 2013.

[86] Xike Xie, Hua Lu, and Torben Bach Pedersen. Distance-aware join for indoor moving objects. *IEEE Trans. Knowl. Data Eng.*, 27(2):428–442, 2015.

[87] Kefeng Xuan, Geng Zhao, David Taniar, Wenny Rahayu, Maytham Safar, and Bala Srinivasan. Voronoi-based range and continuous range query processing in mobile databases. *Journal of Computer and System Sciences*, 77(4):637–651, 2011.

[88] Bin Yang, Hua Lu, and Christian S. Jensen. Scalable continuous range monitoring of moving objects in symbolic indoor space. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 671–680. ACM Press, New York, NY, USA, 2009.

[89] Bin Yang, Hua Lu, and Christian S. Jensen. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. In *EDBT*. ACM Press, New York, NY, USA, 2010.

[90] Wenjie Yuan and Markus Schneider. Supporting continuous range queries in indoor space. In *Eleventh International Conference on Mobile Data Management, MDM 2010, Kanas City, Missouri, USA, 23-26 May 2010*, pages 209–214. IEEE Computer Society Press, Washington, DC, USA, 2010.

[91] Geng Zhao, Kefeng Xuan, J. Wenny Rahayu, David Taniar, Maytham Safar, Maytham L. Gavrilova, and Bala Srinivasan. Voronoi-based continuous k nearest neighbor search in mobile navigation. *IEEE Trans. Industrial Electronics*, 58(6):2247–2257, 2011.

[92] Geng Zhao, Kefeng Xuan, David Taniar, Maytham Safar, Marina L. Gavrilova, and Bala Srinivasan. Multiple object types KNN search using network voronoi diagram. In *Computational Science and Its Applications - ICCSA 2009, International Conference, Seoul, Korea, June 29-July 2, 2009, Proceedings, Part II*, pages 819–834, 2009.

[93] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. G-tree: an efficient index for knn search on road networks. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 39–48. ACM, 2013.

[94] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.*, 27(8):2175–2189, 2015.

[95] Yinghua Zhou, Xing Xie, Chuang Wang, Yuchang Gong, and Wei-Ying Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.