ADJACENCY-BASED INDEXING FOR MOVING OBJECTS IN SPATIAL DATABASES

Sultan Mofareh Alamri

MONASH University

Adjacency-Based Indexing for Moving Objects in Spatial Databases

by

Sultan Mofareh Alamri

Supervisor: A/Professor David Taniar

Associate Supervisor: A/Professor Vincent CS Lee

Thesis submitted in total fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Clayton School of Information Technology

May 2014

Declaration of Authorship

I, Sultan Alamri, declare that this thesis titled, 'Adjacency-Based Indexing for Moving Objects in Spatial Databases' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Sultan Alamri

Date: 1/9/2014

"Keep your face always toward the sunshine and shadows will fall behind you."

Walt Whitman

Copyright Notices

Abstract

With the currently available positioning devices such as Global Positioning System (GPS), RFID, Bluetooth and WI-FI, the locations of moving objects constitute an important foundation for a variety of applications. However, with the recent developments new challenges are presented to traditional database technology. In traditional database systems, the data structures do not support the high-update objects since these structures have not been designed to accommodate dynamic updates of moving objects. Therefore, much research has gone into the outdoor moving objects (e.g. indexing and querying). However, moving objects in indoor spaces no less important than in outdoor. Indoor space refers to cellular spaces where objects are located based on their cells/rooms. Similarly, in some outdoor topographical spaces, moving objects applications focus only on the region/cell where mobile/moving objects are located (not the exact coordinate location). Therefore, this thesis addresses three important basic issues in moving objects databases: (i) understating the variety of the moving objects' features and queries, (ii) adjacency/cellular indexing for moving objects in indoor space, and (iii) adjacency indexing for moving objects in outdoor space.

This thesis starts by presenting a new *taxonomy* for moving object queries. This taxonomy provides a better understanding of the moving objects' features and their variety of queries. The taxonomy for moving object queries is based on five perspectives. First is the *Location* perspective, which includes common spatial queries such as K nearest neighbours (KNN), range queries and others. Second is the *Motion* perspective, which covers direction, velocity, distance and displacement queries. Third is the *Object* perspective, which includes the type status queries and the form status queries. Fourth is the *Temporal* perspective which includes many queries such as the trajectory, timestamped, and period queries. Last is the *Patterns* perspective, which includes many patterns such as spatial movement patterns and temporal movement patterns. Each perspective is explained with illustrated examples.

In addition, in **indoor spaces**, we propose a new cells *adjacency-based index* structure for moving objects. The new index structure focuses on the moving objects based on the notion of cellular space. Therefore, an indoor filling space algorithm is proposed to represent overlapping between the cells. Then, the index structure for indoor space uses the indoor filling space algorithm for

efficient adjacency grouping and serving the indoor spatial queries. The temporal side has been presented using three different techniques: Trajectories Indoor-tree (*TI-tree*), Moving Objects Timestamping-tree in indoor cellular space (*MOT-tree*) and Indoor Trajectories Deltas index based on the connectivity of cellular space (*ITD-tree*). Moreover, in order to improve the performance of the index structure in indoor space, the *density* of the moving objects needs to be considered. Thus, this thesis presents a new index for moving objects (*Indoor^d-tree*) that distinguishes between the high and low density cells. Moreover, it presents a new index structure for moving objects in *multi-floor indoor environments* (*Graph-based Multidimensional Indoor-Tree* or *GMI-tree*) which uses the indoor connectivity graph to group the moving objects multidimensionally and to support wings/sections positioning queries.

Moreover, in **outdoor spaces**, we extended the *regional/adjacency indexing* in indoor spaces to be applicable in topographical outdoor spaces. Thus, we propose an efficient data structure index (*Topographical Outdoor-tree* or *TO-tree*) for the moving objects based on their cellular location. The proposed Adjacent Level Algorithm and Connection Cells Algorithm are used to determine the connectivity level of each cell in the topographical outdoor space. Therefore, the TO-tree will group the moving objects based on their adjacency, thereby reducing the update costs and efficiently supporting the spatial queries and adjacency queries. Furthermore, based on our taxonomy, a very limited number of data structures concentrates on the motion vectors in the construction of the moving objects' data. Therefore, we propose a new index structure (*DV-TPR*-tree*) that supports *velocity and direction* queries, beside the common spatial queries.

To sum up, this thesis introduces a new level of indexing called **adjacency indexing** which proves to be efficient and robust in indoor and outdoor spaces.

Acknowledgements

My foremost thanks goes to my supervisor Associate Professor David Taniar. Without him, this thesis would not have been possible. I appreciate his vast knowledge in many areas, and his insights, comments and guidance, and his incisive and rapid feedback.

I would like to thank my co-supervisor Associate Professor Vincent CS Lee for his patience and valuable advice during my PhD study. I thank all my friends students in our spatial database group (Dr Geng Zhao, Dr Kefeng Xuan, Dr Thao P. Nghiem and Haidar Al-Khalidi and Kiki Maulana) whose presence and lightheartedness made the otherwise grueling experience tolerable. Also professor Maytham Safar and Dr Kinh Nguyen for their assistance and cooperation. Also, I thank Dr Abdullah Alamri, Dr Adil Fahad and Mohammed Aladalah for their assistance and their valuable suggestions. Furthermore, my heartfelt thanks is due to the Ministry of Higher Education of Saudi Arabia for sponsoring me in my PhD study.

Last but not least, I thank my family for always being there when I needed them most, and for supporting me throughout all these years.

Monash University, May 5th, 2014

Sultan Alamri

Contents

List of Figures

List of Tables

Abbreviations

MBR	Minimum Bounding Rectangle
GIS	Geographic Information System
ChildPTR	The pointer to the child node
PTR	The pointer
[x, y]	x and y coordinates of a point
P	Set of moving objects
RC	Range of expand points (cells)
LE	Largest expand point

For my Family

List of Publications

Journals:

- Sultan Alamri, David Taniar, and Maytham Safar. "A Taxonomy for Moving Object Queries in Spatial Databases". *Future Generation Computer Systems*, 2014. doi: 10.1016/j.future.2014.02.007.

- Sultan Alamri, David Taniar, Maytham Safar, and Haidar Al-Khalidi. "Tracking Moving Objects Using Topographical Indexing". *Concurrency and Computation: Practice and Experience* 2013. doi: 10.1016/j. 2013.03.035.

- Sultan Alamri, David Taniar, Maytham Safar, Haidar Al-Khalidi: "Spatiotemporal Indexing for Moving Objects in an Indoor Cellular Space". *Neurocomputing* 122: 70-78 (2013).

- Sultan Alamri, David Taniar, Maytham Safar, and Haidar Al-Khalidi. "A Connectivity Index for Moving Objects in an Indoor Cellular Space". *Personal and Ubiquitous Computing*, pages 1-15, 2013. ISSN 1617-4909. doi: 10.1007/s00779-013-0645-3.

- Sultan Alamri, David Taniar, Maytham Safar: "Indexing Moving Objects for Directions and Velocities Queries". *Information Systems Frontiers* 15(2): 235-248 (2013).

- Sultan Alamri, David Taniar, Haidar Al-Khalidi and Kinh Nguyen: "Indexing Mobile Objects in Multi-Floor Indoor Environments". *ACM Transactions on Spatial Algorithms and Systems* 2014.(under review).

- Sultan Alamri, David Taniar, Haidar Al-Khalidi and Kinh Nguyen: "Density-Based and Adjacency Indexing for Moving Objects in Multi-Floor Indoor Spaces". *IEEE Transactions on Knowledge and Data Engineering* 2014.(under review).

Conferences:

- Sultan Alamri, David Taniar, Maytham Safar: "Indexing of Spatiotemporal Objects in Indoor Environments". *AINA* 2013: 453-460.

- Sultan Alamri: "Indexing and Querying Moving Objects in Indoor Spaces". *ICDE Workshops* 2013: 318-321.

- Sultan Alamri, David Taniar, Maytham Safar: "Indexing Moving Objects in Indoor Cellular Space". *NBiS* 2012: 38-44.

CHAPTER 1

Introduction

Chapter Plan:

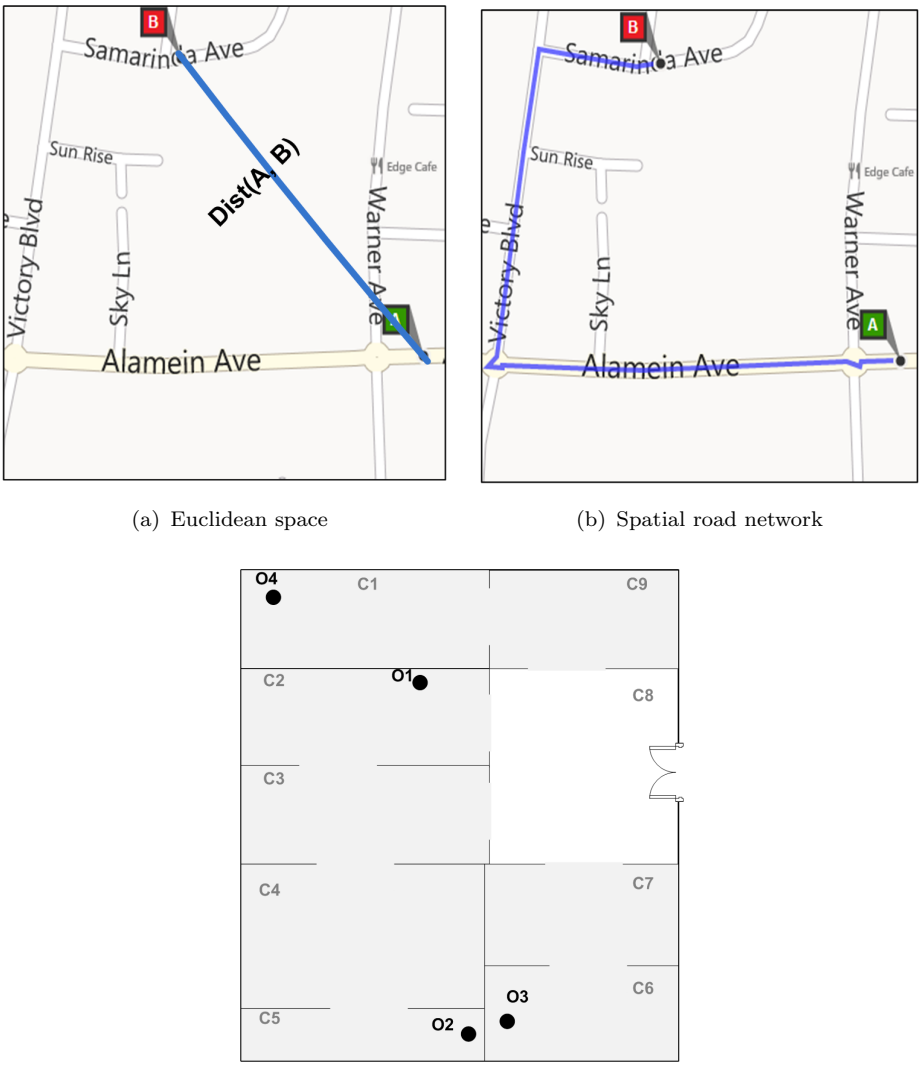
1.1 Overview

A Spatial Database is a database that is designed for storing and querying data related to objects in a location, such as point, line and regions. A spatial index structure is necessary for the spatial databases in order to query the spatial data. Moreover, the spatial databases have been increasingly developed since the fast development of the positioning systems such as the Global Positioning System (GPS) and WI-FI [2–4]. Therefore, many studies have focused extensively on spatial databases over the last two decades, resulting in large number of query processing, index structures and conceptual model techniques.

However, in these traditional spatial databases, the data is assumed to be static, which prevents the direct adoption of these index structures techniques to

the moving objects database [5]. There is an important difference between indexing moving objects and indexing static objects, that is, objects that are stationary [6, 7]. With static objects, spatial indexes basically assume that the objects of interest are constant unless conspicuously updated. Whereas, spatial indexes basically will require frequent update of the locations of continuously moving objects. Multidimensional databases indexing, such as the R-tree and its variants have been designed to support and enable efficient query processing for applications that have relatively fewer updates [8]. The main challenge facing the indexing of moving objects derives from the need to accommodate frequent updates along with obtaining an efficient query processing [9–11]. **Therefore, this thesis presents a new indexing technique named *(adjacency/cellular-indexing)* for moving objects in indoor and outdoor spaces for more efficient update and optimal query processing.**

Several new index structures have been proposed for indexing moving-objects; however, these types of moving objects indexing methods were unable to embrace new query types and new environments. Existing moving objects databases indexes can be divided into three categories: coordinate-based index structures which concentrate on current/anticipated future locations of moving objects [6, 12, 13]; trajectory-based which concentrate on the trajectories of the moving objects; and historical-based which focus on the temporal/historical aspects of the moving objects [14–17]. However, the majority of these works focuses on the Euclidean space and Spatial Road Network underline structures (space-based) of the moving objects. Moreover, the indexing of moving objects must take into consideration the varieties of queries that relate to the moving objects' features in order to establish a solid data structure for the moving objects.



(a) Euclidean space (b) Spatial road network

(c) Cellular space

FIGURE 1.1: Space structures of moving objects

In terms of the *space-based* indexing of moving objects, most of the current studies are based on Euclidean distance which measures the distance between moving objects by the straight line between them (see Figure 1.1 (a)) [18, 19]. Also, some works focus on the road network, which is based on measuring the length of the road between moving objects (the actual network distance) (see Figure 1.1 (b)) [20, 21]. However, in terms of the *space-based*, there is one more which is a representation of location by sets of cells that include moving objects (named **cellular-space** or **adjacency-based**) [22]. This environment is clear in indoor spaces where it is a promising research area. Here, a query in Euclidean

space or a spatial road network is given with coordinates such as $(x_i;\ y_i)$ and $(x_j;\ y_j)$, whereas, a query in cellular space is based on cellular notations such as "What are the moving objects in cell/room C_1?". In this example, the location descriptions are represented by sets and a located object in that case is considered as a member of these sets. Indoor spaces usually are treated as *cellular space* where the object locations are not given with coordinates. For example, (based on Figure 1.1 (c)), object O_4 location is C_1. Here the cell C_1 is considered to be the container/location of the moving object O_4. Also, the distance is not metric, but is based on the number of hops (adjacency/connectivity). For example, if the O_3 in Figure 1.1 (c) wants to return the nearest object to its location, the measurement will be based on the number of hops between the cells, where O_1 is the nearest to O_3. Here, although O_2 is the closest by distance, it is based on the adjacency/connectivity concept, so O_2 is the furthest from O_3.

Therefore, this thesis undertakes the following: **(i)** It provides a moving objects queries **taxonomy** which explains the variety of the moving objects queries from various perspectives. **(ii)** It presents a new indexing technique which focuses on the moving objects in **indoor space**, using the cellular space structure. The index structure introduces a new indexing technique called **adjacency-based index structure** for moving objects. Also, it provides a spatial-temporal index structure which assist in supporting the temporal queries in indoor spaces. **(iii)** It presents a new **density-based** and adjacency-based index structure for the moving objects which improves the performance and supports the density queries. **(iv)** An index structure for moving objects in **multi-floor** indoor environments is also introduced, based on the adjacency in multi-floor indoor areas. **(v)** For the outdoors, it extends the regional/adjacency indexing in indoor spaces so that it can be applied to outdoor cellular spaces (**outdoor adjacency indexing**). **(vi)** This thesis proposes a new index structure that indexes moving objects based on the

spatial, **direction** and **velocity** domains in order to support queries pertaining to velocity and direction.

1.2 Major Challenges

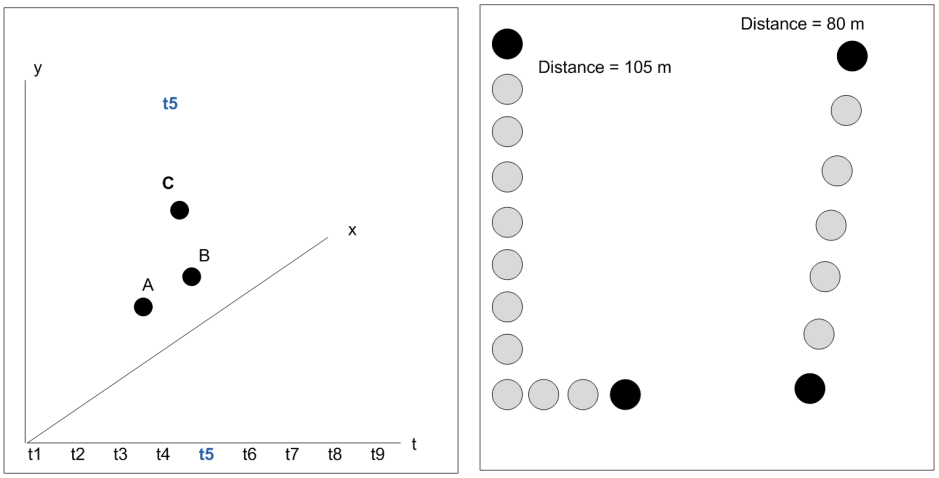
This section describes the challenges and research problems related to moving objects databases.

1.2.1 Queries in Moving Objects Databases

Most studies on moving objects have concentrated on the common spatial queries which are generally used in most spatial databases [6, 23, 24]. These queries are coordinate-based queries or location-aware; that is, a query based on the current user's location. The most common types of queries are point queries, range queries and k-nearest neighbour queries:

- Point queries: "Find the location of an object O_i at a specific time t" For example, where is car number 123 at time 12:00 p.m.? The answer will return the location of car number 123.
- Range queries: "Return all objects whose locations intersect with a specific range R from time t_1 to t_2". For example, return the car that is located at Monash University between 02:00 p.m. and 03:00 p.m.?
- K-nearest neighbour queries: "Find the k nearest objects of a given location L at a specific time t". For instance, find the 3 taxis nearest to Monash University.

The proposed index techniques that support the above queries can be found in [6, 10, 12, 25]. However, these works did not realize that the moving objects are associated with different dimensions. These new dimensions produce new queries such as velocity and direction queries. Figure 1.2 shows an example of a time-based query and a motion (distance) query. Therefore, this thesis presents a taxonomy study which explains the possible variety of queries that can be raised in moving objects databases.



(a) B is the nearest object to A at t_5 (Time-based query)

(b) Distance query

FIGURE 1.2: Moving objects' time-based query and motion (distance) query

1.2.2 Indexing Moving Objects in Indoor Cellular Space

The common spatial data structures are developed mainly in order to speed up retrievals where the objects are assumed to be static. Hence, in order to capture the dynamic moving objects, the common spatial data structures such as R*-tree will produce a massive updating of the locations of moving objects [10, 16, 26]. Therefore, these data structures are not suitable for indexing the moving objects [13, 25, 27, 28]. Many works focus on indexing the moving objects; however, they are not suitable for indoor spaces for the following reasons. First, these data

structures mainly use the GPS as a basic positioning system, whereas positioning devices such as WI-FI, RFID reader or Bluetooth are the most suitable positioning systems for indoors [29]. Second, the measurements in most of these data structures are based on Euclidean space or a spatial network, whereas, indoor space is related to the symbol/notion of cellular space [30]. Indoors moving objects are usually located based on their cells/rooms (e.g. Object O_1 is located in room/cell 12) where the exact location is not needed. Figure 1.3 shows the importance of constructing moving objects in the indoor spaces based on connectivity between the rooms/cells. Therefore, this thesis proposed data structures that are based on the adjacency of the cells and locate the moving objects based on their cells with no consideration given to the exact locations.

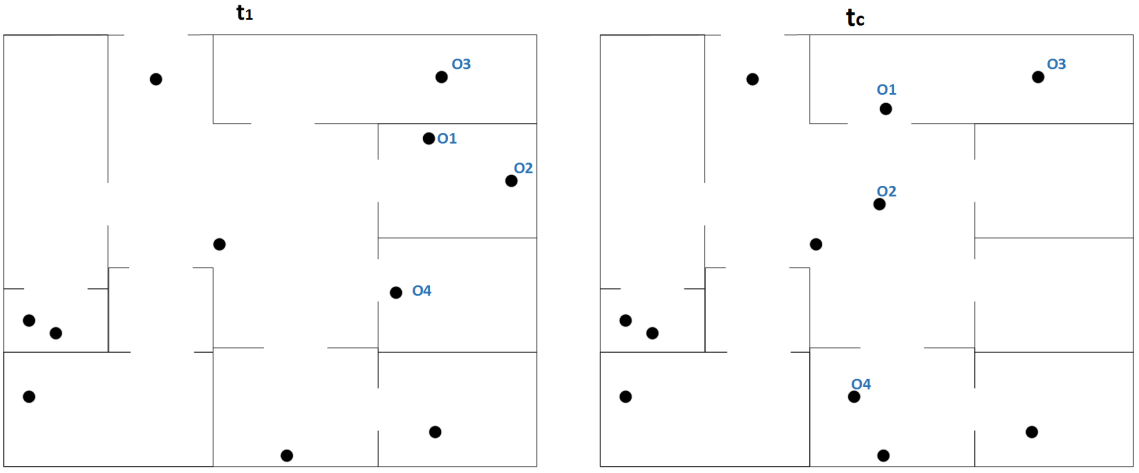


FIGURE 1.3: Moving objects in indoor environments in t_1 and t_c

1.2.3 Data Density in Indoor Cellular Space

In indoor space, the rooms or cells are the container of the moving object. However, the influence that high density cells have on the performance of the data structure is not well understood [31, 32]. How will the data structure perform if the cells are not treated equally when indexing the moving objects? Obviously, in

indoor spaces, some cells usually contain a high density of moving objects such as classrooms, whereas teachers' offices usually contain a low density of moving objects. Hence, we present an index structure of the moving objects in indoor spaces which distinguish between the cells that have a high density and those with a low density of moving objects. This density-based index for moving objects in indoor spaces ensures that the index structure will perform better when the density of the cells is considered. Figure 1.4 shows that the density of moving objects in indoor spaces needs to be considered in order to improve the data structure performance.

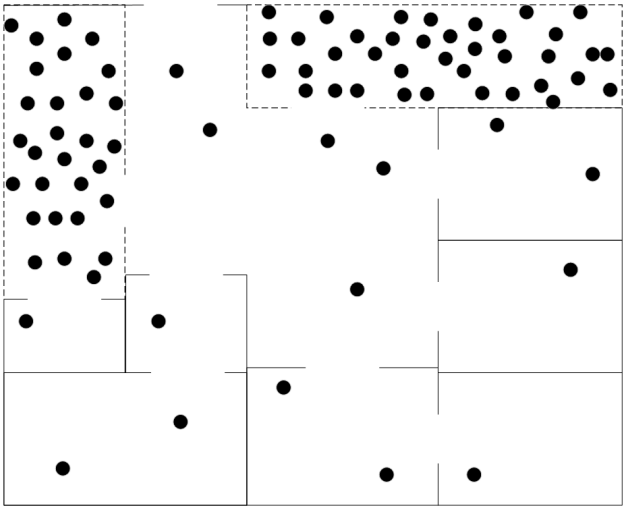


FIGURE 1.4: Data density in indoor environments

1.2.4 Moving Objects in Multi-Floor Indoor Space

Most of the current data structures that focus on moving objects in indoor spaces have treated the moving objects in each floor separately. However, some queries would consider the moving objects in the whole building section without any attention given to their exact floor location. If a query asks about a building section/wing in which a moving object is located, the index structure (assuming that the index treats each floor individually) will need to engage in several verification

steps to reach the targeted answer. However, the importance of the location accuracy of the mobile objects usually varies depending on the query. For instance, "Where is object O_1?" The exact location here is not important since the query can be satisfied with the answer "O_1 is in the north section/wing". Obviously, some buildings (such as shopping centers) which contain wings, affects the success of the positioning queries. Figure 1.5 shows an example of multi-floor indoor environments. Hence, this thesis proposes a new index structure that considers grouping the moving objects multidimensionally in each wing of the multi-floor indoor space.

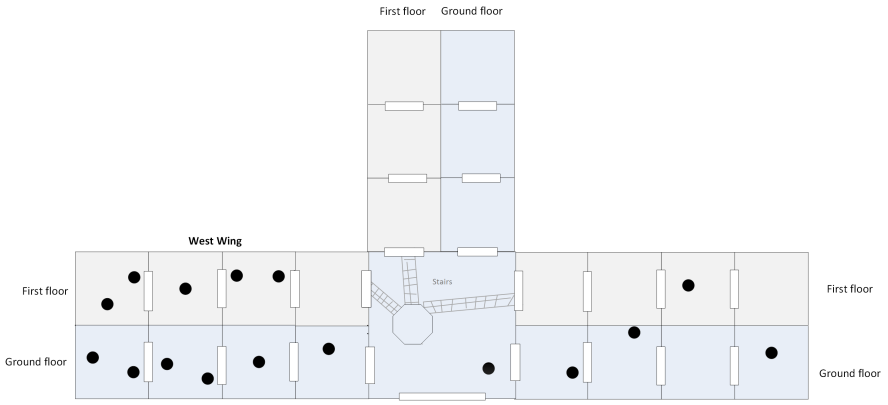


FIGURE 1.5: Moving objects in multi-floor indoor environments

1.2.5 Moving Objects in Outdoor Cellular Space

Many applications involving moving objects concentrate on the area or the cells that contain the moving objects, where exact locations are not important. The locations of moving objects are frequently and continuously changing thereby, producing a high volume of updates [18]. However, with adjacency-based applications, the updates are performed only when the moving object leaves its territory cell and checks into a new cell. An example of an adjacency-base application is the taxi systems in many cities around the world that partition city areas into cells already fixed by a wide wireless network system [33, 34]. When a client contacts

the taxi centre to request a vehicle, most of these taxi centres check the location of the client and then check the available taxis in that cell and send a request to the first one in the queue. If there is no taxi available in that cell, the taxi centre will look at the adjacent cell and send a request to the first available taxi. As is clear, the exact location of the taxi is not important in this application, since the cells are the key points. Figure 1.6 shows an example of a topographical outdoor space. Therefore, this thesis extends the regional/cellular indexing in indoor spaces to be applicable in topographical outdoor spaces.

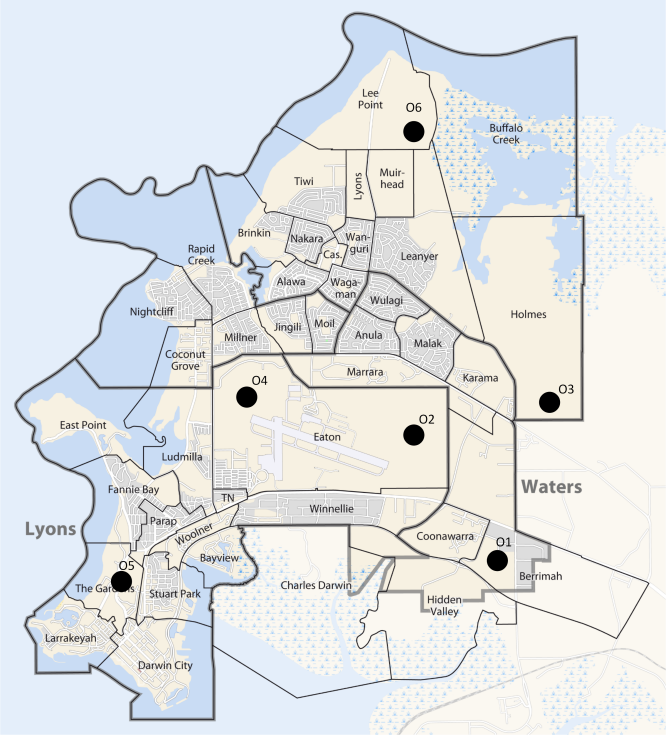
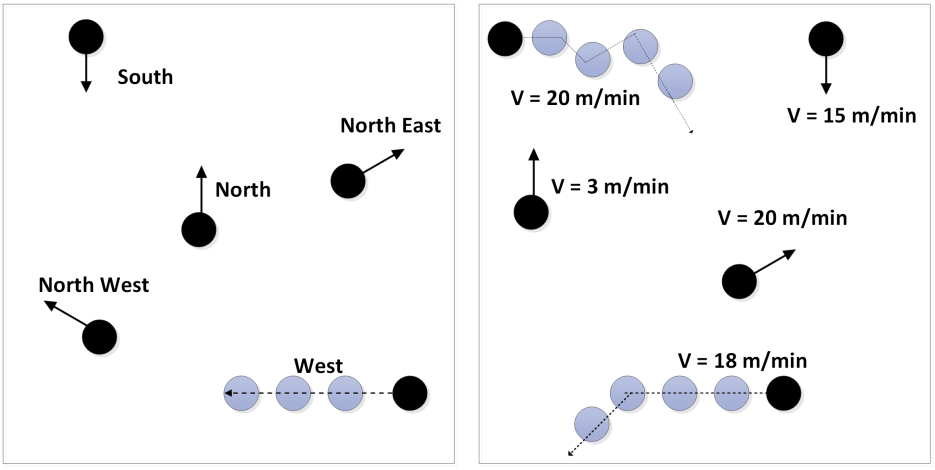


FIGURE 1.6: Moving objects in a topographical outdoor space which is represented as cellular space

1.2.6 Supporting Velocity and Direction Queries

Most of the current moving objects data structures focus only on the common spatial queries which are generally used in the static objects data structure; however,

moving objects often include other dimensions such as velocity/direction queries and movements queries [24, 35]. The data structure in the TPR-tree and its successors are based on the space domain, without considering any distribution of the moving objects' direction and velocity. Three types of queries are supported by the TPR-Tree and its successors to retrieve points with positions within specified regions (the timeslice query, window query and moving query) [6]. The three query types are distinguished according to the regions that are specified by the query. If the TPR-tree and its successors want to process velocity and direction queries, they will need to engage in several verification steps to reach the targeted answer which incur a high cost for access. Therefore, in this thesis, we propose a novel index structure to include efficiently the direction queries and the velocity queries for moving objects. Figure 1.7 shows an example of the velocity and direction of moving objects.



(a) (b)

FIGURE 1.7: Velocity and direction of moving objects

1.3 Objectives and Contributions of This Thesis

This thesis has six major aims: (i) to develop a taxonomy to show all possible queries that can be raised in moving objects databases; (ii) to design a new spatial-temporal index structure (adjacency-based) for moving objects in indoor spaces which serves the location and the temporal queries more efficiently; (iii) to develop a density and adjacency-based index structure to improve the data structure performance and to support the density queries; (iv) to introduce a new index structure for multi-floor indoor environments; (v) to introduce a new adjacency index structure for moving objects in outdoor cellular spaces; (vi) to design a novel index structure for moving objects in outdoor spaces to support motion queries such as those pertaining to velocity and direction.

1.3.1 Contributions to Taxonomy for Moving Object Queries

This thesis presents a taxonomy for moving object queries. The taxonomy explains the moving object queries from various perspectives. *First* is the Location perspective, which includes common spatial queries such as K nearest neighbours (KNN), range queries and others. *Second* is the Motion perspective, which covers direction, velocity, distance and displacement queries. *Third* is the Object perspective, which includes the type status queries and the form status queries. *Fourth* is the Temporal perspective which includes the trajectory, timestamped, inside, disjoint, meet, equal, contain, overlap and period queries. *Last*, it explains the patterns perspective which include many patterns such as spatial movement patterns and temporal movement patterns.

This research [24] was published in the *Future Generation Computer Systems* journal in 2014.

1.3.2 Contributions to Adjacency Index Structures in Indoor Spaces

This thesis proposes a cells connectivity-based index structure for moving objects. The novel index structure focuses on the moving objects based on the notion of cellular space, in contrast to the outdoor space structures which are based on the space domain, Euclidean or spatial network. Moreover, the key idea behind our moving objects indexing is to take advantage of the entities such as doors and hallways that enable and constrain movement in an indoor environment. Therefore, the contributions here can be summarized as follows: **(i)** An indoor filling space algorithm is proposed to represent overlapping between the cells. Because the indoor space cannot precisely be transformed to a straight line (such as Hilbert space); therefore, we propose an expansion idea that provides an optimal representation of the filling indoor space. **(ii)** A unique index structure for indoor space (indoor-tree) is introduced which can manage memory wisely via a neighbours' distance lookup table, and use the indoor filling space algorithm for efficient adjacency grouping and serving the indoor spatial queries.

This research [30] was published in the *Personal and Ubiquitous Computing* journal in 2013. Also, it has been published in *The 15th International Conference on Network-Based Information Systems* (NBiS 2012) [36].

In addition, since the tracking of moving objects and updates in indoor spaces are computationally expensive, the aim is to reduce the tracking and the

updating by means of efficient temporal techniques. Thus, we extended our index to support the temporal side which has been presented using three different techniques. The *first technique* is Trajectories Indoor-tree (TI-tree) which is based on the non-leaf nodes timestamping and the FindPredecessor algorithm that helps to connect the objects with their last location. The *second technique* is the Moving Objects Timestamping-tree in indoor cellular space (MOT-tree), which is based on the cells' adjacency for the purpose of monitoring and dealing with objects that updated their cells. The *third technique* is the Indoor Trajectories Deltas index based on the connectivity of cellular space (ITD-tree), which assists in reducing the update cost by using the deltas which is a temporary recorder of the objects' modifications in the floor space.

This research [37] was published in the *Neurocomputing* journal 2013, and in the *IEEE 27th International Conference on Advanced Information Networking and Applications* (AINA-2013) [38].

1.3.3 Contributions to Density Data Structure in Indoor Spaces

This thesis proposes a new index structure, density and adjacency-based, for moving objects in indoor spaces. Here, the contributions can be summarized as follows: **(i)** We study the impact of data density in Indoor spaces. Since the moving object density in indoor spaces greatly affects the efficiency of the index structure, we distinguish between cells of different densities in our index structure. **(ii)** We propose an adjacency clustering (Low-density tree) which groups the moving objects that are located in low density cells based on their connectivity. **(iii)** The

objects in dense cells are grouped separately in an indoor high-density tree. **(iv)** A combination of the high-density tree with the low-density tree is performed in connectivity basics.

The density-based work [39] was submitted to *IEEE Transactions on Knowledge and Data Engineering* journal (TKDE 2014), and is currently under review.

1.3.4 Contributions to Multi-Floor Data Structure in Indoor Spaces

This thesis proposes a new index structure that groups mobile objects in a multi-floor indoor space. Indoor space is a connectivity environment where room/cells are connected to each other through doors. The contributions here can be summarized as follows: **(i)** In this work, we propose an Indoor Multidimensional Connectivity Graph which introduces two edges of the indoor connected graph. **(ii)** This graph will be used to group the moving objects multidimensionally based on their building section/wing. The Graph-based Multidimensional Index structure (*GMI-tree*) groups the objects based on their adjacency on the same floor and the same section (based on Indoor Multidimensional Connectivity) and extends that to group the objects in each section as a multidimensional grouping for the multi-level.

The graph multidimensional grouping for the multi-floor was submitted in the *ACM Transactions on Spatial Algorithms and Systems* journal (2014) [40], and is currently under review.

1.3.5 Contributions to Adjacency Index Structure in Outdoor Cellular Spaces

This thesis proposes an adjacency-based index structure for moving objects in outdoor environments. The key idea of our moving objects indexing is to take advantage of the adjacency and the connections within topographical outdoor environments. The contributions here can be summarized as follows: **(i)** An optimal representation of the outdoor cells has been proposed in order to deal with the outdoor cells environment based on adjacency and connections. **(ii)** The proposed Adjacent Level Algorithm and Connection Cells Algorithm are used to determine the connectivity level of each cell in the topographical outdoor space. **(iii)** The Topographical Outdoor tree (TO-tree) will group the moving objects based on their adjacency which can enable us to answer spatial queries, topological queries and adjacency queries more efficiently.

This research [41] was published in the *Concurrency and Computation Practice and Experience* journal, 2013.

1.3.6 Contributions to Velocity and Direction Index Structure in Outdoor Spaces

This thesis proposes a novel index structure that will include the direction queries and the velocity queries for moving objects. The data structure in TPR-tree and its successors is based on the space domain, and does not take into account any distribution of the moving objects' direction and velocity. In this work, **(i)** we focus on the construction of the moving objects based on their direction and

velocity domains. For example, the moving objects that are heading in the same direction (e.g. north) will be grouped together in the same MBR. **(ii)** The key purpose of this moving objects index (DV-TPR*-tree) is to cover new types of queries, namely directions and velocity queries (DV queries).

This research [5] was published in the *Information Systems Frontiers* journal, 2013.

1.4 Thesis Outline

An overview study of this thesis study is illustrated in Figure 1.8. The rest of the thesis is organized as follows:

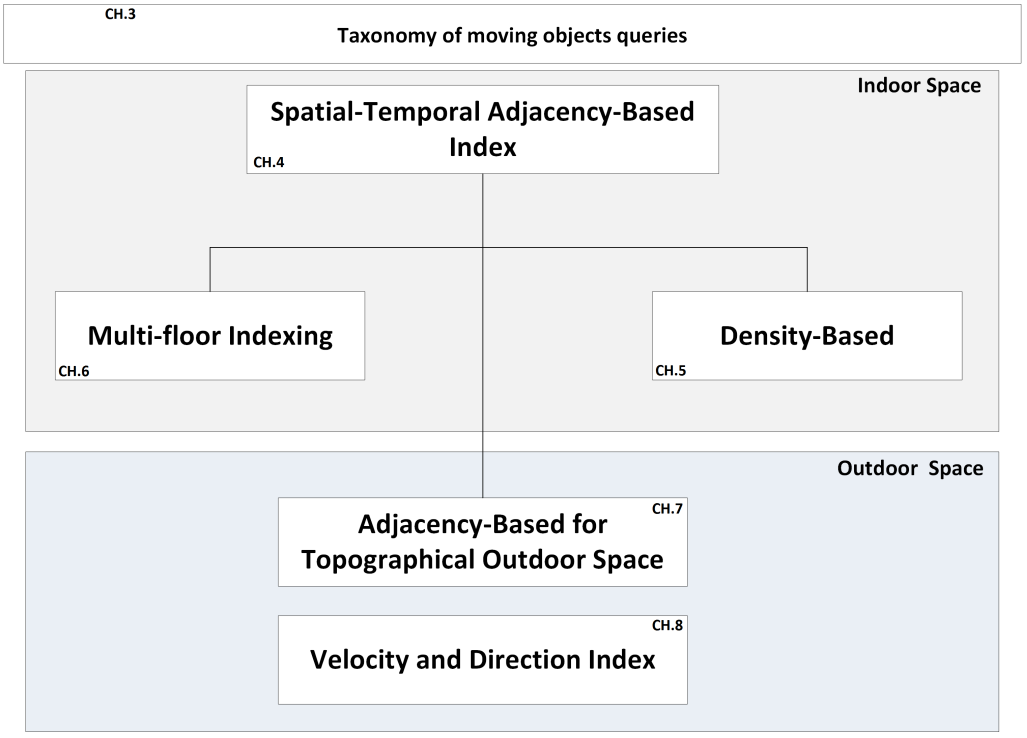


FIGURE 1.8: An overview of this Thesis Study

- Chapter 2 reviews the data structures of the static spatial databases. Moreover, it reviews the data structures of the moving object databases, and provides an analysis and discussion.
- Chapter 3 presents our proposed taxonomy for moving object queries in spatial databases.
- Chapter 4 presents a novel connectivity-based index structure for moving objects which focuses on moving object and is based on the notion of indoor spaces. Also, it presents the temporal part of the adjacency index structure that has been presented using three different techniques.
- Chapter 5 presents a new density and adjacency-based index structure for moving objects in indoor spaces.
- Chapter 6 presents an index structure that groups the mobile objects in the multi-floor indoor space using the proposed connectivity graph of the indoor building.
- Chapter 7 presents a novel adjacency-based index structure for moving objects in topographical outdoor environments.
- Chapter 8 presents an index structure that includes direction queries and the velocity queries for moving objects in outdoor spaces.
- Chapter 9 concludes this thesis and suggests directions for future work.

CHAPTER 2

Literature Review

Chapter Plan:

This chapter starts with the background of the positioning systems in indoor environments (indoor cellular space) of moving objects. Then it briefly reviews the spatial queries in spatial databases, alongside the traditional indexes in spatial databases which are the basis of many of the moving object index structures. Then it investigates the existing index structures for moving objects which is classified into four sections as follows: the trajectories moving objects indexes, historical moving objects indexes, future and current moving objects indexes and indoor moving objects indexes. Finally, it evaluates the extent to which these structures are applicable to indoor environments, alongside with some thoughts about how indoor environments should be indexed.

2.1 Positioning Systems Background

As mentioned previously, GPS technologies are not suitable for indoor spaces [42, 43]. Hence, many positioning technologies need to be adjusted in order to track objects effectively in indoor spaces. One of the common indoor positioning systems is Wi-Fi positioning which has been developed extensively over the past few years [44]. For example, the Wi-Fi Cell-ID is one of the common indoor positioning systems that already developed in the consumer device which are developed to fill the gap. Wi-Fi positioning is rapidly earning acceptance as a complement to and supplement for the global navigation satellite system or GNSS positioning for indoor spaces. Since GNSS has started to struggle and has many issues with the new smart devices, Wi-Fi nodes are popular in the many areas which are already provided with a Wi-Fi system which can support the positioning applications [43–45]. Many service providers can offer the essential databases which basically can enable and facilitate the overall positioning capability.

Moreover, indoor positioning techniques based around the pattern of observations include multiple Wi-Fi hotspots (nodes), and fingerprinting where observations are compared to the previous mapped locations, and "trilateration" which is used to indicate of distance from the transmitter and to locate the device using the geometric calculation against known transmitter locations. Note that the Wi-Fi-based positioning technologies take advantage of the fast enhancements of wireless access points (nodes) in indoor environments and urban areas. Therefore, the proposed indoor index structures in this thesis assume that the Wi-Fi-based positioning system is the basic positioning system in these structures.

With the development of the new smart phones, many systems have been developed to enhance the Wi-Fi indoor positioning. Cell of Origin or (Cell-ID) is one of the common mechanisms used to estimate the approximate location in any system based on "referenced cells" and is the concept of Cell-ID (which uses access points in Wi-Fi 802.11 systems) [46, 47]. Basically, this technique does not explicitly attempt to return the location of the mobile device beyond indicating the cell with which the mobile devices are (or have been) registered. Here, the complicated algorithms of the Cell-ID mechanism do not require an implementation which makes positioning system performance fast. Therefore, the ease of implementation is the main advantage of this technique. Example of systems that use this mechanism in indoor spaces is Navizon and PATS systems [46, 48]. Navizon was an early developed system which makes it possible to identify the geographic location of a mobile device using the reference-id of the location of a cell using the Wi-Fi-based access points instead of GPS. Also, Navizon was improved to be compatible and efficient for tracking the mobile devices indoors and in multi-floor indoor spaces.

The basic idea is to use the indoor Wi-Fi positioning triangulation, whereby multiple access points (nodes) will be fixed on the multi-floor [49]. The nodes are fixed within a short distance from each other which allows a three-dimensional space to be covered, which make it efficient in a multi-floor building. The overlapping between the nodes coverage is already fixed, where the aggregate data from multiple nodes can provide an efficient positioning of each moving object. Figure 2.1 illustrates the Wi-Fi-based positioning system in an indoor space.

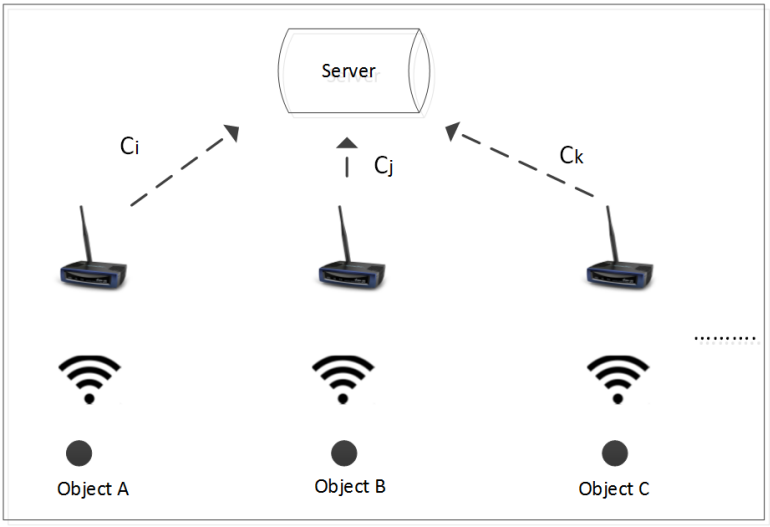


FIGURE 2.1: Indoor Wi-Fi-based positioning system

In addition, people within indoor environments are usually located by abstract symbols such as room number and floor number "in which room is object O located?". Moreover, the moving objects in indoor environments settle for long periods inside the rooms/venues, so that they can be considered as being not fully dynamic moving objects. Therefore, based on these facts, the tracking of moving objects indoors and inside the rooms/cells is not needed. The idea is to track the moving objects only when they change room/cell. As long as a moving object is still in the same cell, there is no need to update its location to the server. The cell's boundaries are determined by the Wi-Fi fingerprinting [43, 44] (see Figure 2.2). Thus, we argue that the use of this tracking technique will not incur a large

update overhead.

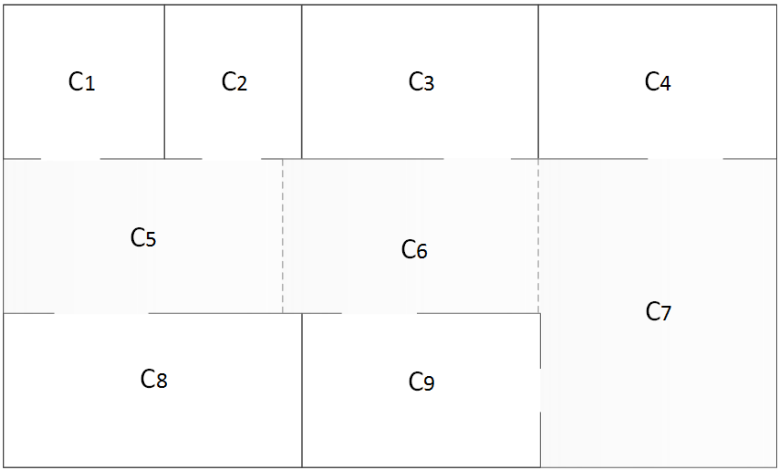
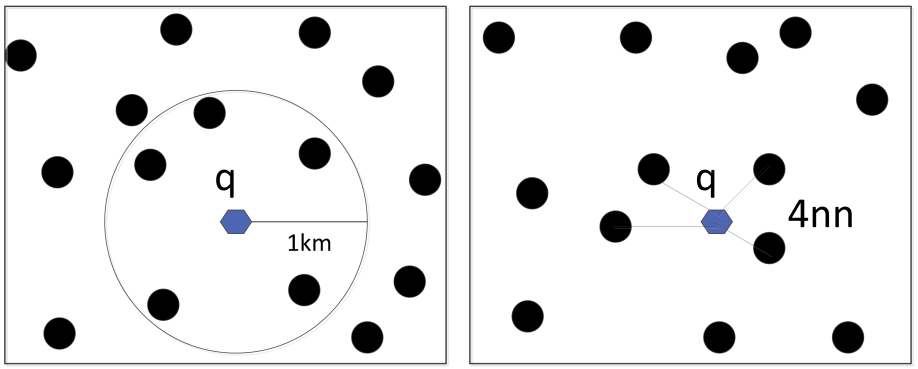


FIGURE 2.2: An example of indoor cells

2.2 Traditional Indexes in Spatial Databases

In spatial applications, querying the objects of interest has become vital with the rapid growth of mobile systems and positioning systems such as GPS. Spatial queries are queries about the location of points of interests such as restaurants, gas stations and shopping centers [19, 50, 51]. Many types of location/spatial queries have become essential in spatial databases such as point queries, Knn queries, and range queries [2, 21, 52]. Figure 2.3 illustrates the Knn queries and range queries.

These types of query have produced the need for spatial index structures to process the spatial queries more efficiently. In order to know the nearest object to a certain location, the location database needs to be indexed based on distance which allows an easy and efficient query processing of that query. Many data structures have been proposed in order to allow an efficient query processing for



(a) Range query (b) 4NN query

FIGURE 2.3: Range and KNN queries

the spatial queries [3, 50, 53]. Note that these traditional indexes assume that the objects of interest are static. This section briefly explains these indexes in order to give a better understanding of this issue. The R-tree [8] (see Figure 2.4) is a multi-dimensional version of B+-tree [53]. Since B+-tree is designed to fulfil non-spatial data , the purpose of R-tree is to be a spatial data retrieval as B+-tree works with non-spatial data [53]. In R-tree objects are represented by minimum bounding rectangles (MBRs).

As shown in Figure 2.4, MBRs are recursively grouped into larger MBRs to form a tree, whereby the larger MBRs cover all spatial objects to include the smaller MBRs. Every leaf node of the R-tree points to the MBRs that contain the spatial objects whereas; the non-leaf node points to the child of the node or MBRs that contain another MBRs [8, 54, 55]. Clearly then, a possible overlapping of the MBRs might lead the search to descend all subtrees that intersect with each other. To insert an object in the R-tree, the insertion algorithm will start scanning for a suitable leaf node which will enlarge the least to include the new objects. If the insertion causes any overlapping, the MBR will be spilt by one of the splitting methods: Exhaustive splitting method, Quadratic splitting method or Linear splitting method. Deletion might cause underflow, where the number of

objects within a node is less than the required minimum. In this case, all entries in this node will be deleted and then reinserted.

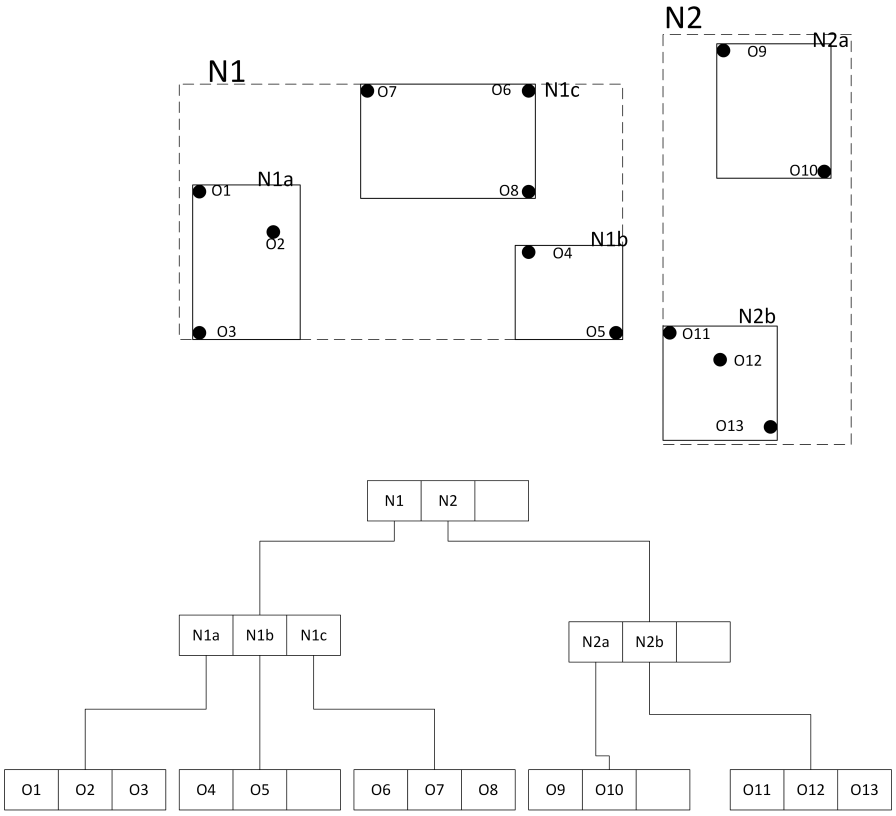


FIGURE 2.4: An example of the R-tree structure

Many weakness of the R-tree have led to the development of the R*-tree [56]. The R*-tree introduces a new policy named 'forced reinsert'. If a node overflows, it is not split immediately. Rather, part of the entries of the node are deleted from the node and then reinserted into the tree. The R*-tree has been wisely reorganized the insertion critical for good search performance. Moreover, since its performance is up to 50% better than that of the R-tree, it is considered to be the most successful variant of the R-tree [55, 57].

Furthermore, since the major issue of the R-tree structure and its variants is the overlap of the MBRs in the directory, which leads to increases with growing dimension, the X-tree has been proposed to resolve this issue [57]. The X-tree is

designed for high dimensional space where the main idea is to avoid overlap as possible without affecting the tree. If that is not possible, then the X-tree uses extended nodes, named supernodes. The X-tree contains three different types of nodes: data nodes, normal directory nodes, and supernodes. The unique feature of the X-tree is the supernodes which are large directory nodes of variable size that in charge to avoid any splitting in the directory that would result in an inefficient directory structure. Note that the X-tree is different from the R-tree since it has larger nodes when necessary. Therefore, the X-tree might become a heterogeneous tree as shown in Figure 2.5.

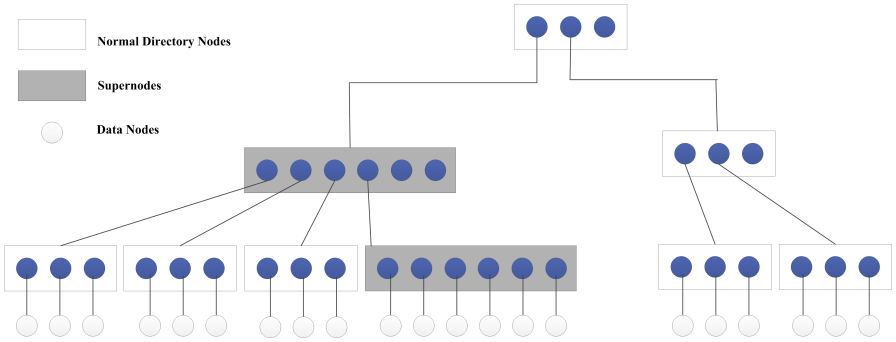


FIGURE 2.5: An example of the X-tree structure

Quadtree [58] is another commonly used index structure which recursively decomposes the space into square cells of different sizes in order to obtain the same value in each cell [59]. Adopting the quadtree data structure directly in the moving object database will cause similar problems to those in the R-tree, which are an expensive numbers of updates. Figure 2.6 shows a Quadtree index structure.

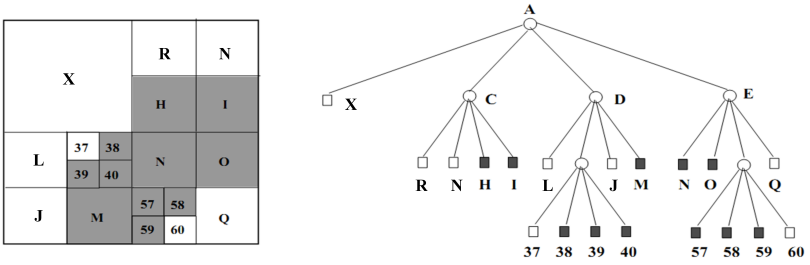


FIGURE 2.6: An example of Quadtree structure

Another common data structure that has been used in spatial data is Voronoi Diagram. Voronoi Diagram is a special type of a metric space that is determined by distances to a specified distinct set of objects in the space. Voronoi Diagram data structure can be Euclidean distance based or Network Voronoi Diagram (NVD) (see Figure 2.7). A Network Voronoi Diagram is a Voronoi diagram based on spatial road network, whereby the distance between two points of interest (or locations) is based on network distance, (not Euclidean distance) [19, 50, 54].

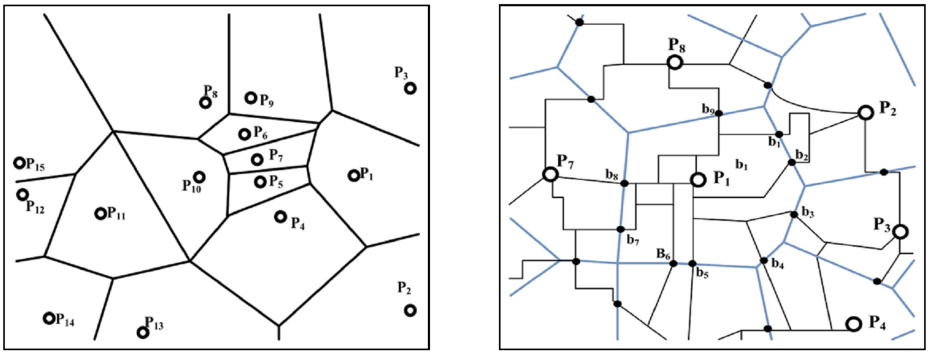


FIGURE 2.7: An example of Voronoi Diagram Euclidean based and network distance based

2.3 Moving Objects Indexes

Multidimensional databases indexing, such as the R-tree and its variants was designed to support and enable efficient query processing and efficient updates [2, 24, 41, 60]. This works well in applications where queries are relatively much more frequent than updates (static objects). On the other hand, applications that involve moving objects have different aspects which include large loads of updates and varieties of queries. Therefore, many index structures have been proposed to index moving objects to and to minimize the large update cost [22]. This section explains the existing moving objects data structures which are classified as:

trajectories moving objects indexes, historical moving objects indexes, future and current moving objects indexes, and indoor moving objects indexes.

2.3.1 Trajectories Moving Objects Indexes

Many works concentrated on the *trajectories* of the moving objects [14, 15]. The Trajectory Bundle tree (TB-tree), which is based on the R-tree [8, 54], indexes trajectories by allowing a leaf node to contain line segments only from the same trajectory, which assists in retrieving the trajectory of an individual object, but negatively influences the spatiotemporal range queries [14, 26]. Note here that the TB-tree preserves trajectories such that leaf nodes only contain segments belonging to the same trajectory. The drawback of the TB-tree is when the overlap increases, the traditional range query cost increases due to the decreases of the space discrimination. Figure 2.8 shows the TB-tree structure.

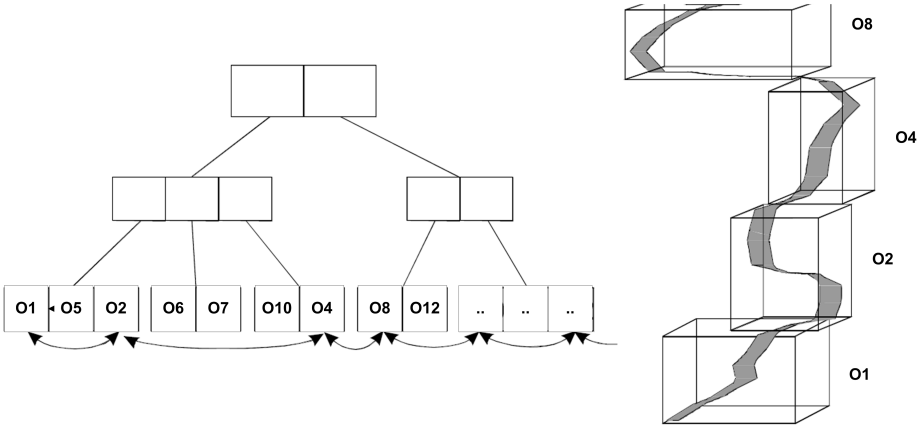


FIGURE 2.8: An example of TB-tree structure

Another trajectory index is named the STR-tree (Spatio-Temporal R-tree) [4, 14, 15]. STR is based not only on spatial closeness, but also on trajectory

preservation. STR was introduced in order to balance spatial locality with trajectory preservation. The main idea of the STR-tree is to keep line segments within the same trajectory [3].

The SETI (Scalable and Efficient Trajectory Index) [15] basically partitions space-dimensions into non-overlapping cells. The goal is to ensure that trajectory line segments in the same index partition belong to the same trajectory. Then, inside each partition, the line segments are indexed by a separate spatial index (R-tree) [15]. Another work that focuses on the trajectory of the data, but for a road network, is the Fixed Network R-tree (FNR-tree) proposed by Frentzos [11]. The idea basically is to take into account the constraints in road networks. The FNR-tree contains a 2-dimensional (2D) R-tree and a 1-dimensional (1D) R-tree. The 2D R-tree is intended to index the spatial data of the network; whereas, the 1D R-tree is intended to index the time interval of each moving object. Note that in each leaf node entry of the 2D R-tree ($Line_{Id}$, MBR, $Orientation$), and the structure of each non-leaf of the 2D R-tree is (Ptr, MBR). Moreover, leaf nodes of the 2D R-tree contain a pointer to the root of an 1D R-tree (see Figure 2.9).

In the FNR-tree each edge corresponds to a line segment in the network which leads to the main drawback of the FNR-tree index: the FNR-tree will greatly increase the number of objects, leading to high updates in the data structure, since distinct objects are needed for every line segment that the object traverses. Hence, the Moving Objects in Networks Tree (MON-tree) is proposed to efficiently store and retrieve moving objects in road networks [61]. The data structure stores entire trajectories of the objects and is able to process past queries of the database. Moreover, the index structure is contain by a top R-tree (2D R-tree) which responsible for indexing polyline bounding boxes, and bottom R-trees (2D R-trees)

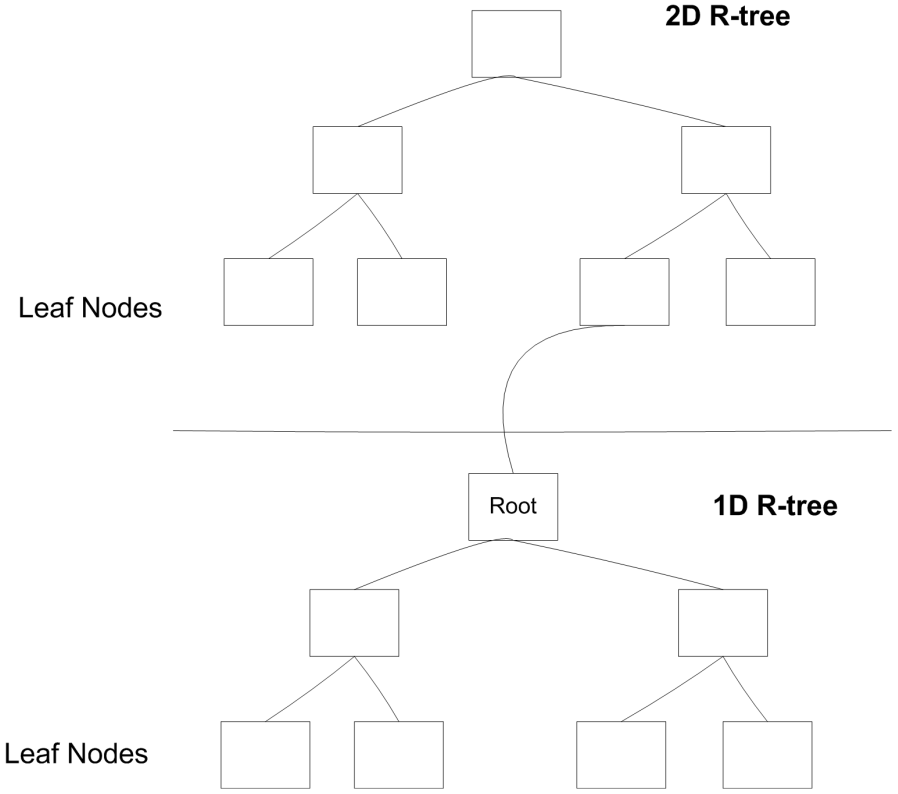


FIGURE 2.9: An example of FNR-tree structure

to index moving objects along the polylines. The main disadvantage here is the great amount of dead space in polyline MBBs [61].

2.3.2 Historical Moving Objects Indexes

Works that concentrate on *historical* moving objects [62–64]: Many applications such as road planning applications and security applications use the historical data of moving objects. The Historical R-tree (HR-tree) is one of the earliest data structures to concentrate on historical data [17, 64]. The main idea is to use the timestamp history to construct the R-tree. R-trees can make use of common paths if objects do not change their positions, and new branches are created only

for objects that have moved. It is clear that HR-trees are efficient in cases of timestamp queries, as search is reduced to a static query for which R-trees are very efficient. However, the massive duplication of objects can lead to large space consumption which affects the performance [62–64]. Figure 2.10 shows an example of the HR-tree.

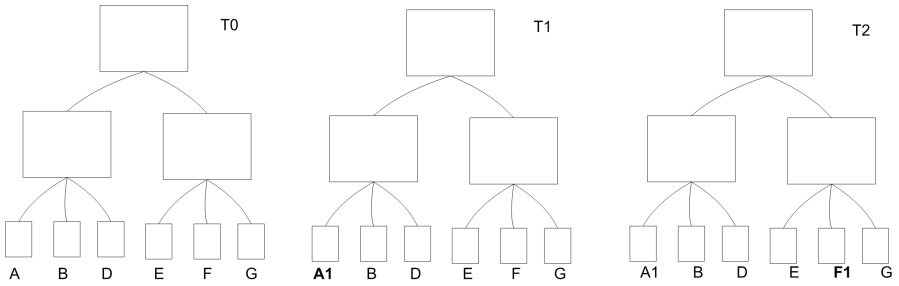


FIGURE 2.10: An example of HR-tree structure

Another work that focuses on the historical data is the Multi-version 3D R-tree (MV3R-tree) [62] which basically uses Multi-version B-trees [26, 54] and combines them with 3D R-trees [63]. The MV3R-tree includes significant improvements which produce huge space savings without influencing the performance of the timestamp queries compared to the HR-tree. Since the HR-tree and 3D R-trees require a huge space, the MV3R-tree overcomes this problem by combining two structures: a multi-version R-tree (Known as MVR-tree) and 3D-tree. Note that the MVR-tree includes multi R-trees, each one containing (MBR, T_{start}, T_{end} and ptr) [62]. Moreover, the 3D R-trees are built on the leaf nodes of the MVR-tree (see Figure 2.11).

In addition, the Multiple TSB-tree (MTSB-tree) supports the range close-pair queries and historical for moving objects [16]. A range close-pair query is used to find two moving objects that are close to each other (closer than the given threshold distance) at certain time intervals. The space in the MTSB-tree is divided into several cells, where each cell has its own index structure for the moving objects inside it. The basic idea here is that the MTSB-tree adopts the a Time-Split B-tree

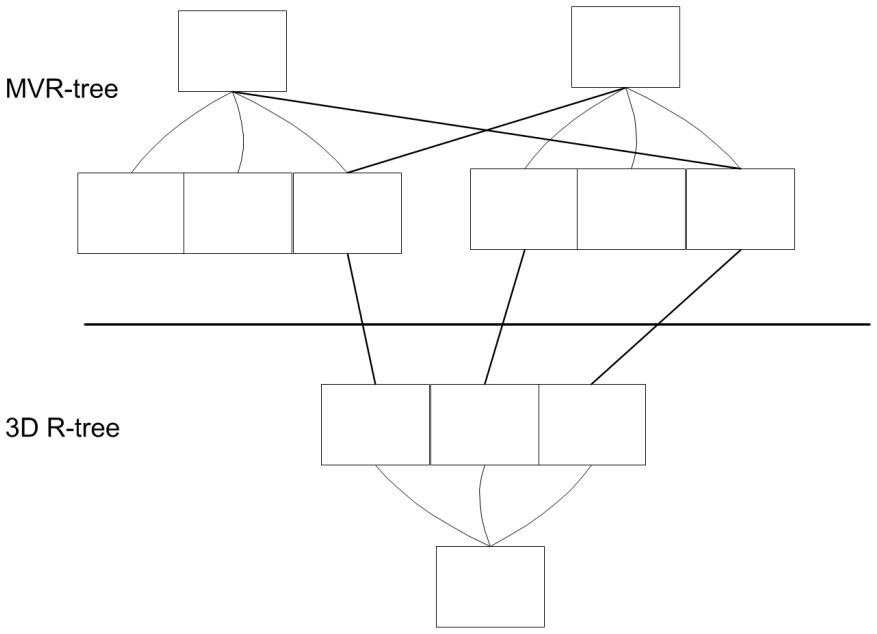


FIGURE 2.11: An example of MV3R-tree structure

(TSB-tree) [65] in each cell. The MTBS-tree has a plane-sweep algorithm which sweeps the alongside the space in order to produce close-pair objects. Therefore, in the temporal and spatial range query, the trajectory segments will be produced with the increase of the time which is ordered by combining results from the TSB-trees of all cells.

2.3.3 Future and Current Moving Objects Indexes

In this section, we review the future and current indexes in spatio-temporal databases. Saltenis et al. [6] introduced the TPR-tree, (Time Parameterized R) which is based on the R*-tree in order to construct and manage moving objects. The main idea of the TPR-Tree is that the index stores the velocities of objects along with their positions in nodes. Moreover, the intermediate nodes' entries will store MBRs (minimum bounding rectangles), in addition to its velocity vector which they call VBR. As shown in Figure 2.12, the lower boundary of a MBR is moving with

the minimum velocity of the enclosed points, while the upper boundary is moving with the maximum velocity of the enclosed points. This ensures that the MBRs consider all times of the moving objects. The insertion and the deletion of the TPR-tree is similar to the R*-tree. The insertion algorithm considers the MBR area along with the following integral:

$$\int_{t_1}^{t_1+H} A(t)dt \tag{2.1}$$

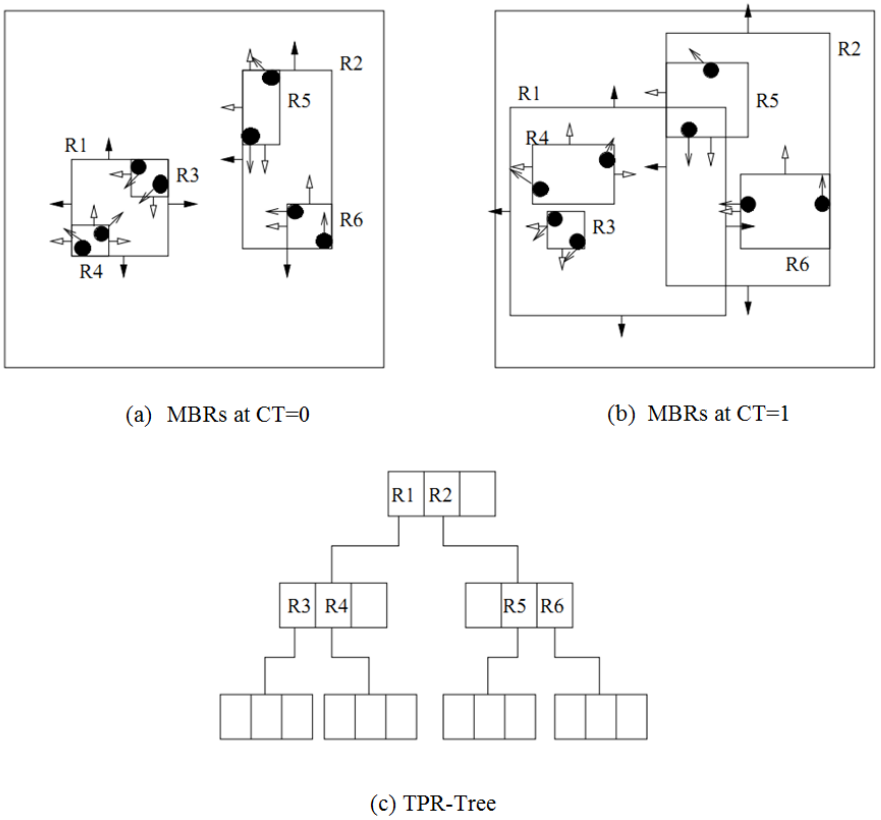


FIGURE 2.12: An example of TPR-Tree.

Where t_1 is the current time and H is the length of the time interval, and $A(t)$ is the area of the MBR. The deletions in TPR-tree will exactly hold the methodology in R*-tree. For example, node $R3$ becomes under full, therefore its objects will be deleted, which will force the reinsertion [12, 66].

As an extension of the TPR-tree, Yufei Tao proposed the TPR*-tree [10], which develops the insertion/deletion algorithms in order to improve the performance of TPR-tree. The main idea in TPR*-tree is to use the swept region ASR instead of the integral mentioned in (Formula 2.1) (see Figure 2.13). Formula (2.2) represents the average number of node accesses for answering query q which is used in the insertion of TPR*-tree. Note that o is the moving rectangle and o' is the transformed rectangle of o, qT is the query interval $qT = \{qT_-, qT_+\}$.

$$cost\,(q) = \sum_{Every\ node\ o} ASR(o', qT) \tag{2.2}$$

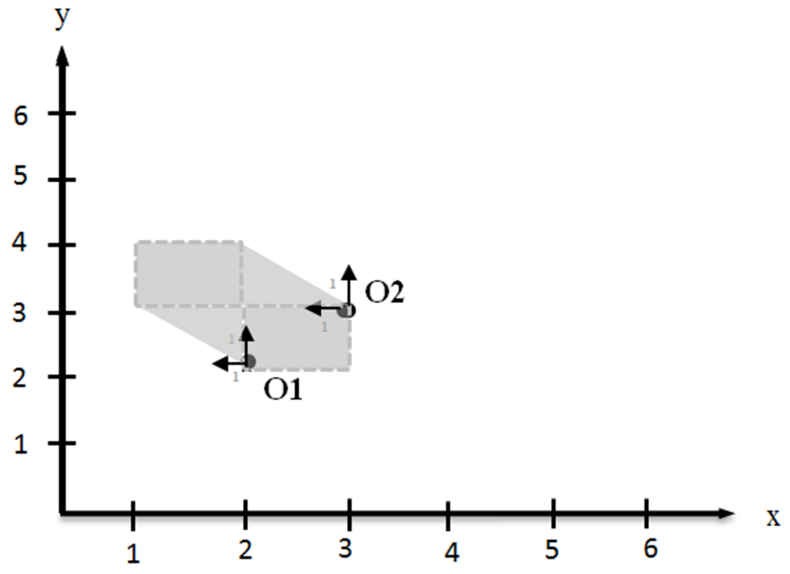


FIGURE 2.13: MBR R and its sweeping region ASR shown in gray

Following the TPR-tree concept, several methods have been proposed to improve its various aspects. Wei Liao et al. [67] proposed the VTPR-tree in order to enhance a good dynamic update performance and concurrency. The VTPR-tree takes into account both the space and velocity distribution of moving objects.

They kept the TPR-tree basics intact, apart from the new structure of a linear velocity bucket queue. Items in each velocity bucket queue will hold a summary about the amount of space and the velocity range of moving objects. In addition, because the TPR-tree is still intact, each bucket item will point to a TPR-tree structure of the moving objects in this bucket. Note that the velocity bucket queue will be visited when an update or query is performed; therefore, the organizing of this new structure (into main memory) will reduce frequent disk I/Os per update. Another TPR-tree successor is the Horizon TPR-tree or HTPR-tree [68]. In HTPR-tree, the focus was on developing a new motion function of time to alleviate the speed of growing sizes of MBR. This will lead to smaller MBRs which help to reduce overlap issues and increase the efficiency of the queries.

Moreover, some works such as [25] focus on indexing the moving objects in landmarks, where the tracking of moving objects in a topographical area is needed. In [25] (RP-tree) moving objects in the landmark are indexed based on the TPR-tree concept (see Figure 2.14). In the RP-tree, the space is partitioned into several subspaces based on the reference points (RP), and then each partition is constructed based on the concept of the TPR-tree, as an auxiliary data structure. Also, [25] developed a kNN search and update algorithms for the RP-tree data structure with concurrency control considerations.

In addition, in the Q+R-tree, Quadtree it combined with the R-tree for better query performance and efficient updating [13]. The R-tree is usually better for query performance, although, the Quadtree is efficient in updating. This work distinguishes between the slow-moving objects and fast-moving objects. Therefore, the data structure here is divided into three levels. First, an R-tree is built for the slow-moving objects. Here the R-tree will index the objects topographically, where

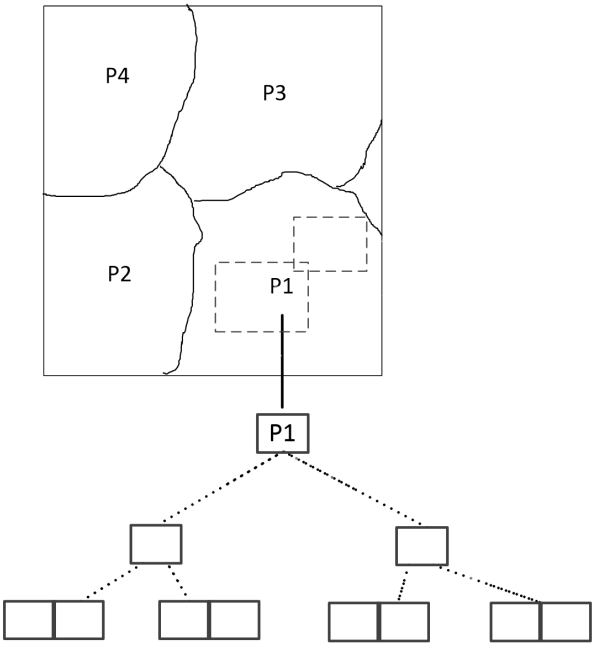


FIGURE 2.14: An Example of the RP-tree Structure

there is no restriction and splitting technique for the node (no Maximum/Minimum policy for splitting the node). The second level is the Quadtree for the fast moving objects. Here it applies the traditional operation for the Quadtree. At the third level, the R-tree and the Quadtree are combined.

2.3.4 Indoor Moving Objects Indexes

As mentioned previously, the indexing and querying of moving objects in indoor spaces is no less important than indexing and querying moving objects outdoors. Therefore, few works have concentrated on building an index structure for moving objects indoors [1, 29, 69].

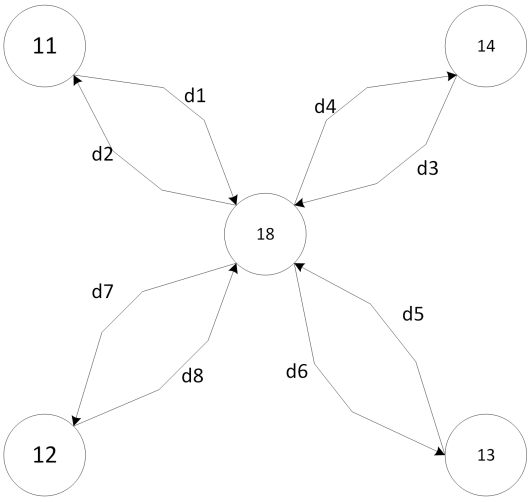
The RTR-tree and TP2R-tree [69] both collect the data via RFID readers that are fixed within the indoor space. The RTR-tree uses the same basic node organization as in the R-tree. The trajectory is organized as a set of line segments

in the indoor plane. To answer a query (For example range query), the Euclidean range space is changed to a set of RFID readers and the search is conducted similarly to that for the R-tree. Note that the Leaf nodes in the RTR-tree is formatted as (MBR, R_{ID}), where MBR is the minimum boundary of a record which is indicated by R_{ID}. Moreover, the R_{ID} indicates a reader on the Reader axis and t_s and t_e. Also the Non-leaf nodes, the RTR-tree have (MBR, c_p), where c_p is a pointer to a child leaf node in the RTR-tree and MBR is the higher minimum bounding rectangle which contains the lower MBRs of the child leaf node's entities.

The TP2R-tree is different from the RTR-tree in that it transforms the trajectory of data into a set of points in the indoor plane. Here in the TP2R-tree, the leaf nodes contain index entities of the form (MBR, $\lhd t$, R_{ID}), where MBR is the minimum boundary of a record which indicated by R_{ID}, and $\lhd t$ is a time parameter which indicates the period of the reading by a reader. Note that each $\lhd t$ will be equals to t_e - t_s. Also, the non-leaf nodes in the TP2R-tree contain index entities of the form (MBR, $\lhd t$, cp), where MBR is the higher box that contains all MBRs in the child node's entities, cp is a child pointer, and $\lhd t$ is a time parameter. The TP2R-tree handles the case of overflowing nodes by considering the amount of time needed to produce fewer node accesses. Therefore, the TP2R-tree succeeded in achieving a better node organization [69].

In [1], Hua Lu et al. proposed a distance-aware indoor space structure which integrates indoor space distances. This structure provides an algorithm to calculate the indoor space distances. Indoor space rooms are connected through doors; therefore, a door connects two adjacent rooms (partitions). This distance-aware model proposed a topology information mappings which basically maps the indoor floor partitions and doors. Here there are two concepts: first, D2P(di) which

can be unidirectional and bidirectional, if $|D2P(di)| = 1$ is considered as unidirectional. Whereas, $|D2P(di)| = 2$ is considered as bidirectional. For example, D2P($d5$) = ($v6, v7$), where $d5$ is door number 5, and $v6, v7$ are partitions number 6 and 7. The second concept is two derived mappings that can be reversed to capture relevant topology information in the opposite direction. In particular, the mapping P2D$\sqsupset$ maps a partition v to all the indoor doors through those that can enter v. On the otherwise, P2D$\sqsubset$ maps a partition v from the other side. For example, P2D $\sqsupset$ ($v5$) = d1, d10, d12 and P2D $\sqsubset$ ($v5$) = d10, d12. Figure 2.15 shows the accessibility Base Graph and Distance Matrix in this model.



	d1	d2	d3	d4	d5	d6	d7	d8
d1	0.0	2.0	3.2	2.1	3.6	1.3	2.2	4.9
d2	2.0	0.0	3.6	2.0	3.0	1.3	2.3	3.9
d3	3.2	3.6	0.0	3.2	2.6	1.4	2.4	3.9
d4	2.1	2.0	3.2	0.0	1.4	1.5	1.9	4.5
d5	3.6	3.0	2.6	1.4	0.0	2.4	2.3	5.9
d6	1.3	1.3	1.4	1.5	2.4	0.0	1.3	2.9
d7	2.2	2.3	2.4	1.9	2.3	1.3	0.0	3.9
d8	4.9	3.9	3.9	4.5	5.9	2.9	3.9	0.0

FIGURE 2.15: An Example of the accessibility Base Graph and Distance Matrix in [1]

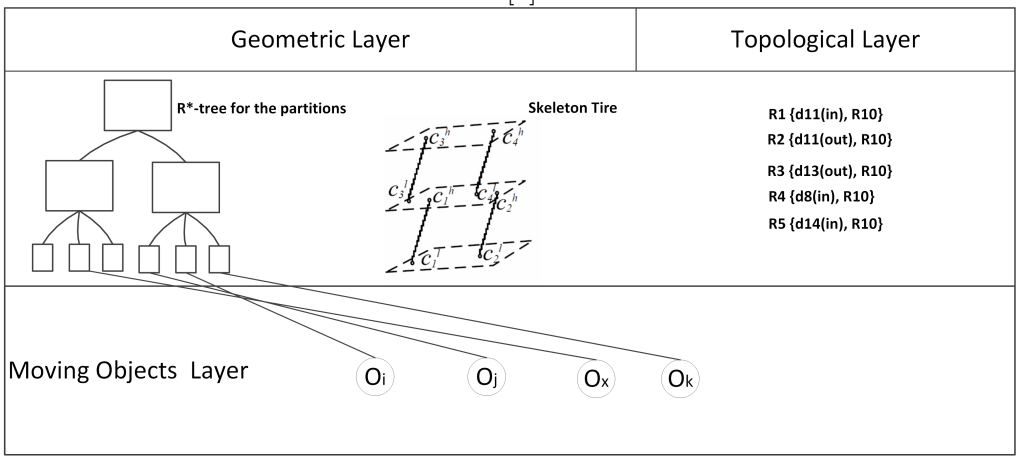


FIGURE 2.16: The composite index includes three layers

Xike Xieet al. [29] proposed indoor distance topological upper/lower bounds.

These topological upper/lower bounds are used in range and Knn query processing. It also focused on the distance of the indoor space between a query point q and an object O_1, the locations of which are obtained through an indoor positioning system. Therefore, the basic idea here is to divide the not accurate location of a moving object into disjoint subregions where each of them is falling inside one indoor partition. Then, based on the topological properties, it classifies the distances between the query and the subregions, and without any calculation of indoor distance, it can determine by the bounds the disqualified objects in query processing. It also designs a composite index for indoor environments as illustrated in Figure 2.16. The composite index includes three layers. First is the geometric layer which uses the R*-tree structure to index all indoor partitions. It also contains the skeleton tier that is responsible for maintaining a distance between staircases. Second is the topological layer which is responsible for connectivity information between indoor partitions. Third is the object layer which is responsible for storing all indoor objects through partitions at the leaf level.

2.4 Limitations

It is clear that the techniques appropriate for outdoor data structures are not suitable for indoor spaces for the following reasons: (i) The outdoor data structures use the GPS as a basic positioning system, whereas positioning devices such as WI-FI, RFID reader or Bluetooth are the optimal positioning systems in indoor environments. (ii) The measurements in these data structures are based on Euclidean space or a spatial network, whereas, indoor space is related to the symbol/notion of cellular space. (iii) Indoor environments contain obstacles or entities (e.g. rooms and walls) that control the movement of the objects, whereas outdoors

it is mostly free movement in Euclidean space. As a result, moving objects in indoor space should be indexed based on the symbol/notion of indoor cellular space, with no attention to the metric distance. Also, indoor spaces need to be pre-fixed with advanced indoor positioning technologies such as WI-FI and RFID [44].

Moreover, people spend the majority of their lives in indoor spaces such as offices, shopping centres, schools, transport stations, and many others. This becomes clear with the fast deployment of indoor positioning systems such as RFID, Bluetooth and Wi-Fi [43, 49, 70]. Hence, moving objects in indoor spaces is a promising research area which is just as important as research into outdoor spaces. Many applications include indoor objects navigation, moving objects positioning, security, way finding and indoor information (such as aggregation about how many objects visited a certain indoor space which is important for the planning of the retail stores). Therefore, the indexing of moving objects in indoor spaces needs to be efficient in order to obtain an efficient query processing for the variety of indoor queries [30, 35, 45].

TABLE 2.1: Differences in responding to indoor and outdoor queries

Queries examples	**Indoor respond**	**Outdoor respond**
What is the location of object O_1?	Room/Cell 212	(41.850033,-87.6500523) coordinates
What is the distance between objects O_1 and O_2?	two rooms/-cells away	845 meters

In addition, the current moving object indexes in indoor spaces have the following drawbacks. The location of moving objects in indoor space is usually indicated by a symbol/notion; whereas, the current indoor index structures consider the exact locations. For example, if a query is performed to locate object O_1, then in indoor space, the logical answer is the room that O_1 has checked into

(e.g. room/cell 212). Therefore, a object indoors should be located based on its region (room/cell) with no attention given to the exact location. Moreover, using distance calculation between the doors on an indoor floor is useless in most of the indoor applications, since the distance between two objects is based on the number of cells/rooms between the objects. Table 2.1 illustrates queries which clarify how indoor data are better described using a symbol/notion of cellular space based on their distance.

Moreover, the majority of the data structures used for moving objects have focused on the construction of the moving objects on each floor separately, with no consideration given to the multi-floor environments. Some buildings contain several levels where the moving objects on different floors can be the targets to some queries. For example, in some buildings, the positioning of the mobile objects can be based on their building section/wing with no consideration as to the exact floor. Furthermore, data density plays an essential role in the data structure performance. Some works focus on the density of the moving objects in outdoor environments. However, in indoors, there is an obvious lack of approaches that monitor the density of the moving objects. Therefore, the index structure of the moving objects in indoor spaces should consider the high density cells and low density cells, eliciting a faster performance from the spatial database.

2.5 Chapter Summary

This chapter initially explains the positioning systems of moving objects in indoor environments (indoor cellular space). Then, it briefly reviews the traditional indexes in spatial databases such as R-tree and Quadtree, with the spatial queries

in spatial database. Then it investigates the existing index structures for moving objects which is classified into four sections as follows: the trajectories moving objects indexes such as TB-tree, STR-tree and FNR-tree; the historical moving objects indexes such as MV3R-tree and MTSB-tree; the future and current moving objects indexes such as TPR-tree, TPR*-tree, Q+R-tree and RP-tree; and, it discusses the indoor moving objects indexes such as RTR-tree, TP2R-tree and distance-aware indoor space structure. Finally, this chapter discussed and evaluated the applicability of these structures for indoor environments, and indicated how indoor environments should be indexed.

CHAPTER 3

A Taxonomy for Moving Object Queries in Spatial Databases

Chapter Plan:

Publications and Submissions:

- Sultan Alamri, David Taniar, and Maytham Safar. "A taxonomy for moving object queries in spatial databases". *Future Generation Computer Systems*, 2014. doi: 10.1016/j.future.2014.02.007.

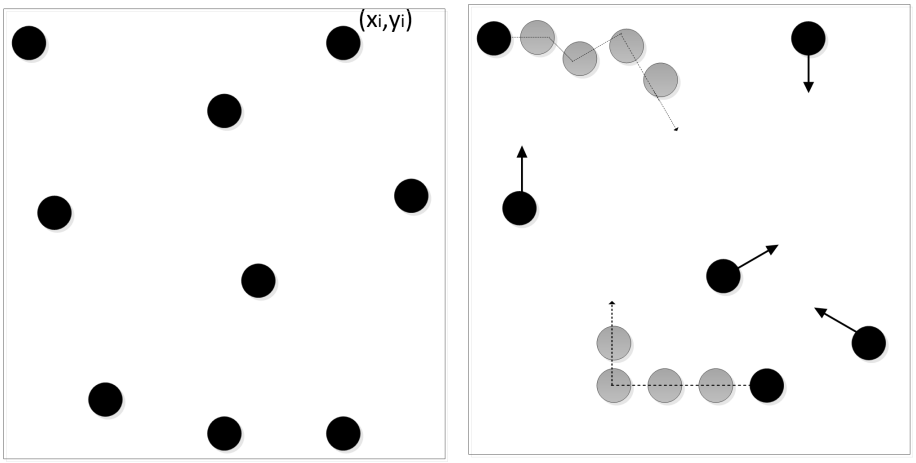
Before introducing the proposed index structures for the moving objects, we need to understand the features of the moving objects databases and the variety of possible queries of the moving objects. A large number of moving object applications have different perspectives regarding the querying of the moving objects [5, 71]. Besides the typical types of queries such as point queries, range queries and k-nearest neighbor queries which are widely used in static object applications, the querying of moving objects has many other dimensions. For example, moving objects change their locations with time; therefore, tracking moving objects at certain times (temporal queries) is essential in security applications. Moreover, tracking moving objects that enter or leave a certain area (topological queries), is another unique perspective of moving objects querying. These queries illustrate the different vectors of the moving objects which do not exist in applications for static objects.

In addition, different researches have been developed to analyse objects which change their spatial location over time in order to track and monitor cars,

trains, and ships [6, 70, 70, 72]. Most of these researches on moving objects have concentrated on the common spatial queries with a lack of attention being paid to the variety of queries about moving objects. This chapter concentrates on building a taxonomy that gives a better understanding of the moving object queries in order to build data structures for moving objects. The main goal is to explore the variety of possible queries about moving objects in spatio-temporal databases. Figure 3.1 illustrates the difference between moving objects and static objects. Figure 3.1 (a) shows that the locations of static objects are constant unless conspicuously altered (e.g. change address). On the other hand, Figure 3.1 (b) shows that the locations of moving objects are constantly updated. Moreover, Figure 3.1 (b) shows the features of moving objects such as direction, velocity and movement patterns, which static objects do not have.

Note that the space structure of the moving objects can be one of the following: *Euclidean space, spatial road network* and *cellular space*. In Euclidean space the distance between object O_i and O_j is the straight and direct distance between them [69, 73]. In this space, the queries will be considered from the perspective of Euclidean space measurement. For example, a range query performed in Euclidean space could return the moving objects within a 2 km radius. In this example, the Euclidean space's measurement will be used to measure the distance from the query point to the points of interest up to a 2 km radius. Figure 3.2 (a) illustrates the Euclidean space measurement query where q wants to retrieve the Euclidean distance between itself and a moving object O_4. The *spatial road network* is the other space that can be considered for moving objects. In geospatial database setting, moving objects can be located in road network where the distance between O_i and O_j is a network distance (or shortest road) between them [27, 36, 74]. In this space, the queries will be considered from the road network measurement perspective. For instance, a 2NN query performed in a spatial road network to

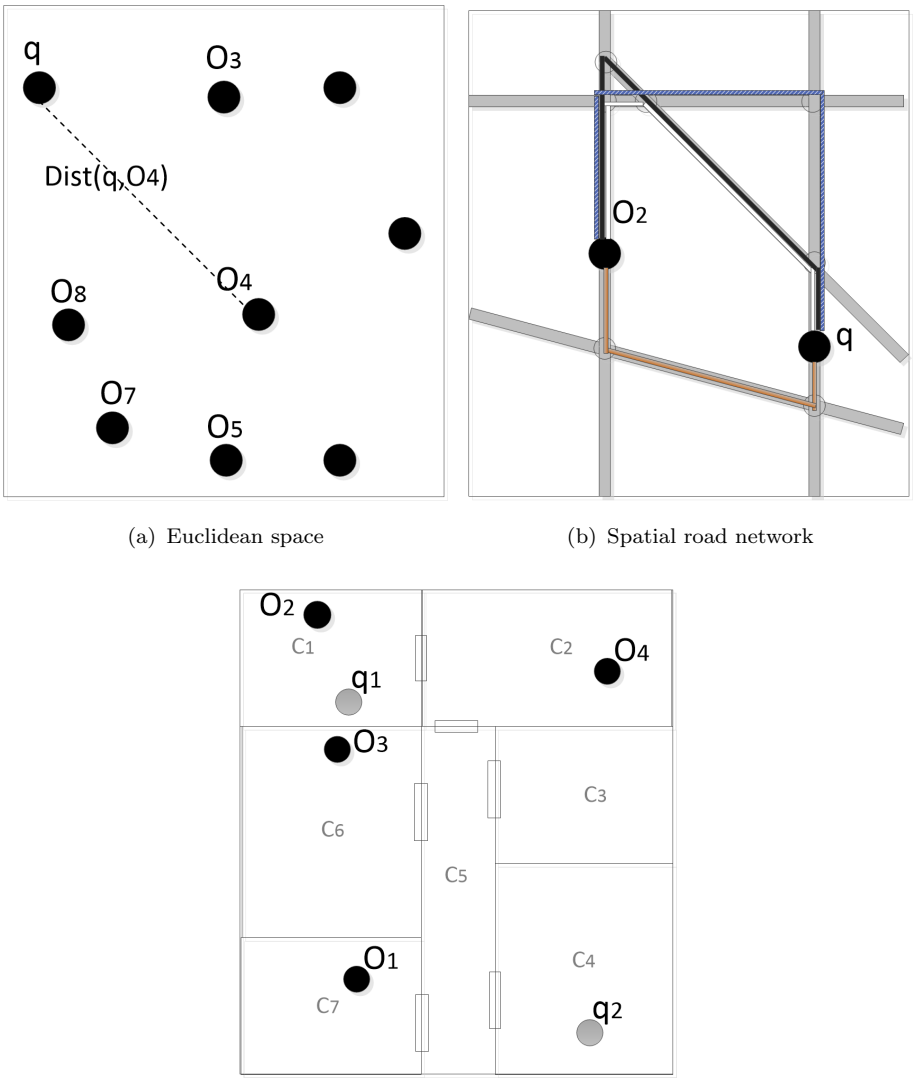
return the two moving objects nearest to q. In this example, the shortest path measurement is used to retrieve the results. Figure 3.2 (b), shows a spatial road network, where q wants to retrieve the shortest path between its position and the moving object O_2.



(a) Static Objects (b) Moving Objects

FIGURE 3.1: Moving objects and static objects

Cellular space is a representation of location by sets of cells that include moving objects. The basic difference between cellular space and Euclidean space or spatial road networks is the dependence on the geometric representation of spatial property [30, 73]. A query in Euclidean space or a spatial road network is given with coordinates such as $(x_i; y_i)$ and $(x_j; y_j)$. On the other hand, the queries in cellular space are usually based on cellular notations such as "What are the moving objects in cell 48?" In this example, the cell number represents a cell identifier which is the main difference from the outdoor coordinates in Euclidean space or spatial road networks [36, 37, 67, 69]. In cellular space, the location descriptions are represented by sets and a located object in that case is considered as a member of these sets. Indoor spaces usually are treated as *cellular space* where the object locations are not given with coordinates. The locations of the moving objects in indoor spaces are usually based on cellular notations. Moreover,



(a) Euclidean space

(b) Spatial road network

(c) Cellular space

FIGURE 3.2: Space structures of moving objects

many outdoor applications depend on the outdoor cellular space, where the exact locations of the moving objects are not useful. In this space, the queries will be considered from the cellular space's measurement point of view. The distance is not metric, but is based on the number of hops. For example, if the q_1 in Figure 3.2 (c) wants to retrieve the object nearest to its location, O_2 will be considered as the 1NN to q_1 since it is located in the same cell. Another example based on Figure 3.2 (c) where q_2 wants to return the nearest object to its location, the measurement will be based on the number of the hops between the cells, where O_1 is the nearest to q_2.

In this chapter, the taxonomy for moving object queries is based on five perspectives. First is the *Location* perspective, which includes common spatial queries such as K nearest neighbours (KNN), range queries and others. Second is the *Motion* perspective, which covers direction, velocity, distance and displacement queries. Third is the *Object* perspective, which includes the type status queries and the form status queries. Fourth is the *Temporal* perspective which includes the trajectory, timestamped, inside, disjoint, meet, equal, contain, overlap and period queries. Last, it explains the *Patterns* perspective, where the moving objects are using undefined movement or predefined movement patterns which include many patterns such as spatial movement patterns and temporal movement patterns. Each perspective is explained with examples.

3.1 Moving Object Queries Taxonomy

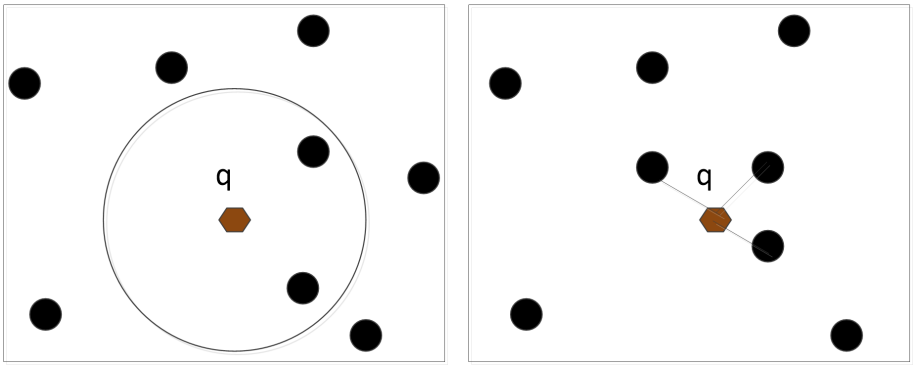
This section illustrates the five perspectives of the moving objects queries taxonomy. Each perspective is explained and examples of queries are provided.

3.1.1 Location perspective

The location perspective considers the location as the key element in the moving object queries. Therefore, the results will be retrieved based on the location requirements of the queries. For example, range queries use the location as the main element to return the points of interest in a certain location with a certain range distance to the query point. This perspective includes the common location queries which are widely used in spatial databases. Location perspective includes

many query types; however, this section limits the illustration to some examples such as *spatial, adjacency, navigational, topological, N-body constraints, density queries* and *aggregate* queries.

Spatial queries are common spatial queries which are widely-used in spatial databases such as join, point, K Nearest Neighbor (KNN), Reverse Nearest Neighbor (RNN), range, spatiotemporal range and within-distance queries [19, 75]. These queries usually depend on the location (the coordinates) of the query (see Figure 3.3). Examples of KNN queries and range queries are "return the three taxis nearest to location B and all taxis within 2 km radius range". Spatiotemporal range queries consider the moving objects within the spatial range and temporal interval, such as "retrieve moving objects inside a 2 km radius within 5 minutes from now". A within-distance query example is as follows: "return moving objects in the last 1 km distance to q".



(a) Range query (b) 3NN query

FIGURE 3.3: Range and KNN queries

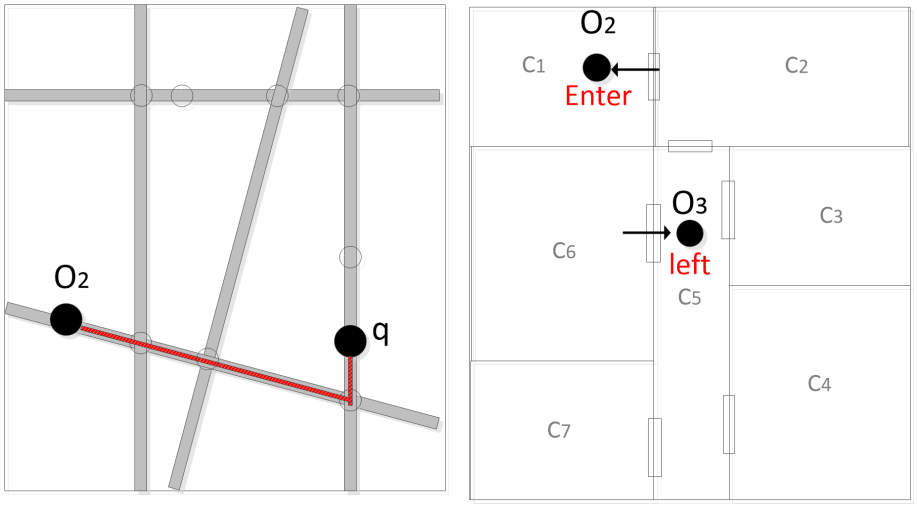
Adjacency queries focus on moving objects that are located in the neighbors regions. These types of queries are common in the *cellular space* (e.g. indoor spaces) where the moving objects are located inside cells (the exact location is not tracked) [37]. The adjacency cells are usually divided into several levels where

each cell has first level neighbor cells and second level until the furthest level of the adjacency cells . Therefore, the query will be concentrated on the moving objects in these levels. For example, a query may be about the moving objects that are located in the first level cells of C_1. Another example is: "return the moving objects that are located at the furthest level cells of C_2".

Navigational queries focus on data such as *heading, speed, travel distance, location, path* and many more (see Figure 3.4 (a)). Note that the overhead of computing these answers will be incurred because the information that is requested is not stored directly. An example of a navigational query is a traffic management center that wants to return the current *location* or the current speed of a particular vehicle, or another example, "return the *distance* between object O_2 and q".

Topological queries concentrate on the data of a trajectory object stored partially or fully [70, 72]. *Enters, leaves, bypasses* and *crosses* are common operations that are involved in topological queries (see Figure 3.4 (b)). Examples of topological based queries include: "Which vehicles entered Monash University most recently?" and "Which object crosses C_6?" Note that the complexity of processing this type of query includes the consideration of the moving object locations as cellular notations.

N-body constraint queries are the types of queries whereby the user can specify location constraints such as greater than or less than a specific distance. Therefore, the query returns a set of N objects which fulfil the alert location constraint. A query example to satisfy a 2-body constraint (based on Figure 3.5) is: "retrieve taxis within a circular range with r=3 from the center station". Figure 3.5 shows an example of N-body constraints queries.



(a) Path query (b) Topological query

FIGURE 3.4: Topological and navigational queries

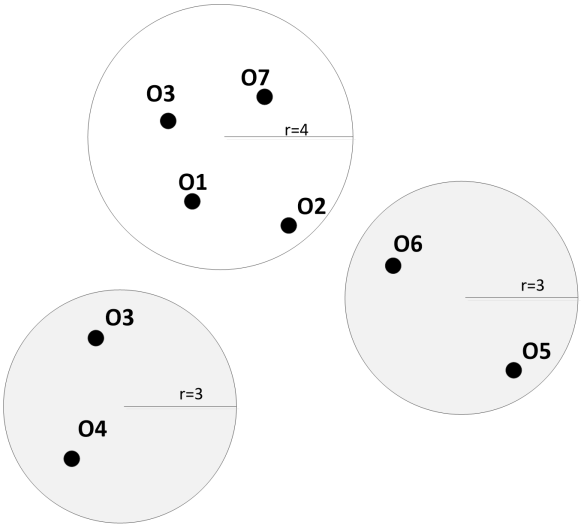


FIGURE 3.5: N-body constraints queries. <O_3, O_4> and the pair <O_5, O_6> satisfy a 2-body constraint with alerting distance r=3

Density queries focus on regions that have a high or low density of moving objects. These types of queries are common in *cellular space* (e.g. indoor spaces) since the moving objects are located inside cells which can raise the dense cells queries. For example, given a set of P moving objects in indoor space, if a thresholds $\partial 1$ is given, find cells $C = C_1,..., C_n$ at t_i, where Density(C_i, t_i) > $\partial 1$ $i \in$

[1, n]. Another example is: "return the dense cells at the second floor at time 1 p.m.".

Aggregate queries or window aggregate queries concentrate on aggregate data over regions that satisfy several spatiotemporal predicates. This type of query is processed somewhat differently from a traditional relational database [76, 77]. The difference is that in window aggregate queries, detecting the pre-defined hierarchies from ranges and locations of the spatiotemporal query window is quite difficult, which leads to some difficulties and complexities in using OLAP operations. For example: "return the number of moving objects that have been in A area between 2 p.m. and 3 p.m." Another query example is: "return the maximum number of the visitors to B area".

To sum up, the results from this perspective are retrieved based on the location requirements in the queries. We explain several examples such as spatial, adjacency, navigational, topological, N-body constraints, density and aggregation queries. One of the important complexities in this perspective is maintaining the frequent update of the queries; for example, maintaining the update of moving objects which are involved in a range query, or maintaining the update of moving objects' speeds or travel distances in navigational queries.

3.1.2 Motion Perspective

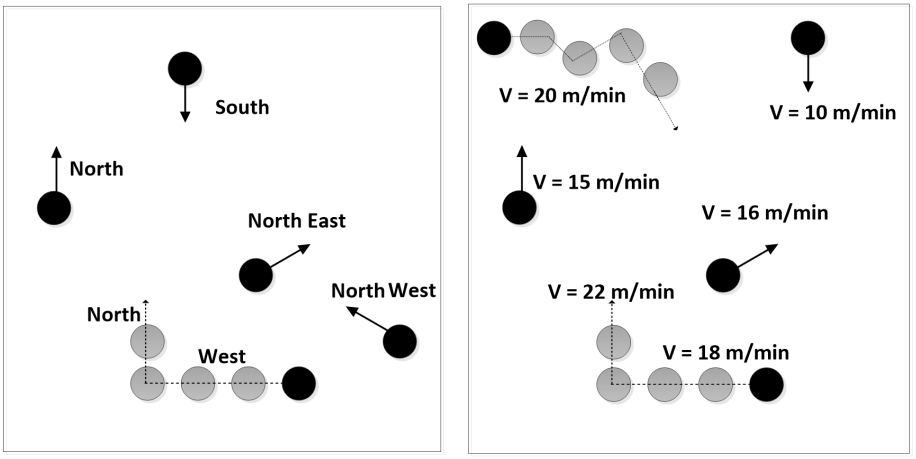
Motion can be observed by linking a reference to a moving object and measuring its location change relative to another reference frame [78]. The motion perspective is a unique dimension of moving objects which does not exist for static objects.

Therefore, motion queries can be raised for moving objects. There are many motion vectors of the moving objects which can be classified as motion vectors related to non geo-referenced moving objects (such as wind) and geo-referenced moving objects (such as vehicles). Since we focus on the geo-referenced moving objects which consume geographical space, the motion vectors are classified as follows: *velocity*, *direction*, *distance* and *displacement*.

Velocity queries are queries that illustrate the relevance of the different speeds of each moving object. There is no doubt that each moving object will have a different velocity range for its own movement. For example, a transport company has a number of moving points (such as taxis); hence, the velocity range between the moving objects will be different. Moreover, the velocity range of each single object trip will be different as well. Therefore, object velocity queries can be classified as *similar range velocities* or *different range velocities*. *Similar range velocities* indicate that the target's moving objects on the map are moving within a similar speed range, whereas *different range velocities* means that the query is interested in whether objects are moving at different speeds [5]. Note that speed similarity or difference can be applied to one single moving object, or to multiple moving objects. For example: "What are the objects that are moving within the range 60 kmph to 65 kmph in the northern area?" This illustrates a query that is related to each moving object's velocity.

Direction queries are those queries that depend explicitly on the moving objects' directions and whether or not they have a similar direction. For a *similar direction query*, an example is: "return the vehicles in the suburb of Clayton that changed their direction to First St North", While a *different direction query*, "the traffic management system wants to return all moving vehicles in the suburb of Clayton which are moving in opposite directions". Note that the moving object's

velocity and direction can make a difference because it involves different data structures and algorithms. The complexity of processing these types of queries include considering both the velocity and direction in the tree structure of the moving object. Figure 3.6 illustrates the direction and velocity of moving objects.



(a) Moving object's direction (b) Moving object's velocity

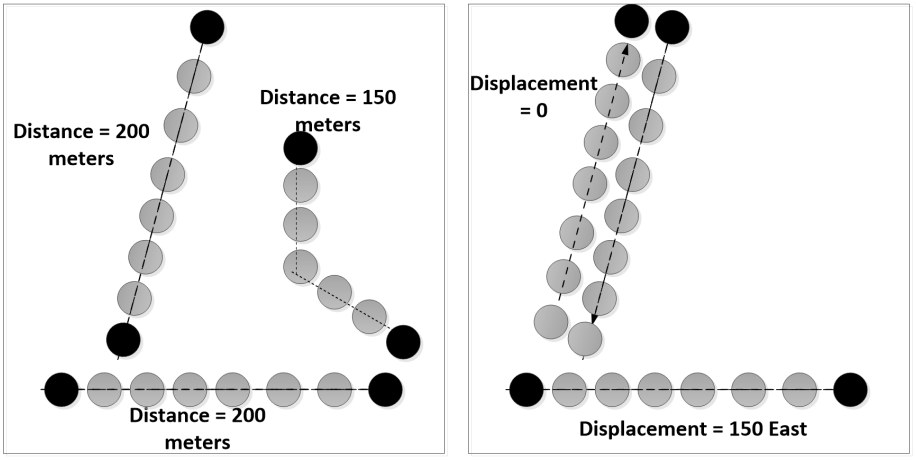
FIGURE 3.6: Direction and velocity of the moving objects

Distance queries are queries that illustrate the relevance of the different distances of each moving object. The distance vector is the path followed by the moving object during its motion. Here, this section focuses on the relevant distances between the moving objects [78]. Note that relevant distances can be applied to one single moving object, or to multiple moving objects; for example: "return the moving objects that moved 2000 metres distant". Another example which illustrates a query that is related to the moving object's distance is: "return the location of moving object A when its mileage (moved distance) reaches half the moving object B mileage".

Displacement queries are queries that illustrate the relevance of the different displacement of each moving object. The displacement vector is the shortest distance starting from the initial point to the last destination point in the motion of

a moving object [79]. Moreover, displacement has two vectors which are distance and direction [80]. Here, this section focuses on the relevance of the between displacement of the moving objects. For example: "return all moving objects that moved around the field with the displacement = 0". In this example, the results will include all moving objects that moved from an initial point and returned to the same initial point. Another example which illustrates queries that are related to moving object's displacement is: "return the location of moving object A when its displacement is equal to its *moved distance* / 2". Figure 3.7 illustrates the distance and displacement of moving objects.

To sum up, the motion perspective can be observed by linking a motion vector to a movement. Our focus on the geo-referenced moving objects includes velocity, direction, distance and displacement. The issues raised in this perspective include how to associate the motion vectors in the query processing, and how to involve the location perspective with the motion vectors.



(a) Moving objects' distance (b) Moving objects' displacement

FIGURE 3.7: Distance and displacement of the moving objects

3.1.3 Object Perspective

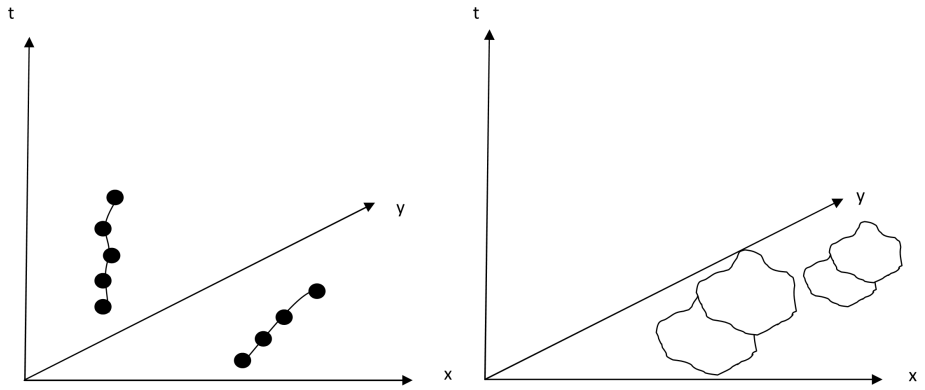
The object perspective considers the object characteristics as the key element of the query. Therefore, the results will be retrieved based on the object's characteristics (type or form) that are indicated in the query. There are many types of moving objects; therefore, the object queries make a difference because they involve different data structures and algorithms. Queries about a moving object can involve different types of moving objects, or can specify only one type. Consequently, we need to illustrate the queries that are related to the object perspective of moving objects. Object perspective can be classified as follows: *object-type queries* which include single-type and multi-types and *object-form queries* which include point, line and region objects.

Object-type queries are queries that depend explicitly on the moving object types and whether or not they are of a similar type. There is no doubt that the types of moving objects are different and are determined by the main sponsor of the moving object. For example, a transport company has different types of moving points such as taxis, buses etc. The *type query* will consider the object from a *single-type* or *multi-types* perspective. *Single type queries* means that the outcomes of the query will be only one type of moving object (object type = 1), which means the objects are similar in all factors of the queries. For example, *cohort* [81, 82], which means a group that has a factor in common that will be statistically relevant. For example, a similar gender (females): "return the locations of all females employees of the company". *Multi-type queries* means that the outcome of the query will be more than one type of moving object (object type > 1). For example: "return the locations of all moving objects in Second St". Here, the objects are *heterogeneous* because they are not the same type (such as same

vehicle). Therefore, the outcome of this query might include different types of moving objects such as different makes of car and different mobile users.

Object-form queries are related to the body/form of the moving objects. Defining the object form to a large degree influences the successful design of a spatial data structure and definitely influences the performance of spatial database systems [83]. The majority of the geo-referenced moving objects in spatial databases are considered as points such as individuals, mobile users, vehicles, and many more. However, the moving objects can also be considered as *lines* and *regions* [11, 84]. A group of moving objects which are moving in a line format can be considered as line-moving objects. For example, a procession of cars constitutes a moving line on the roads. A query example is: "return the length of the line that is moving towards Second Street". Another body status of moving objects is the moving region. This is clear in the movements of army troops and land extensions. The queries about moving regions can be about locations, size and other characteristics. Figure 3.8 shows a representation of moving points and moving regions.

To sum up, the object perspective focuses on the object characteristics as the key element of the queries. This perspective can include object-type queries and object-form queries which include points, lines and regions objects. The issues include the difference that can be made in the data structures and algorithms considering new types of moving objects, and the issues associated with indexing unstructured moving objects.



(a) Moving points (b) Moving regions

FIGURE 3.8: An example of moving points and moving regions

3.1.4 Temporal Perspective

The temporal perspective includes queries about the moving objects which concern the temporal aspects and characteristics of the moving objects. Therefore, the temporal aspects will be included in this perspective's queries in different ways as given or requested. The time aspect is a unique vector of moving objects compared with static objects. Moreover, historical (temporal) queries are based on the time of the moving object and show the time difference for objects moving from one place to another [64]. Each database state has a link with a *timestamp* [12]. In this state, the value of the dynamic object will be taken to be the value of the moving object at the time *(t)* [61, 83, 85]. The temporal perspective is classified as follows: *trajectory, timestamped, inside, disjoint, meet, equal, contain, overlap* and *period* queries.

Trajectory means that the sequence time of a moving object with the timestamps of each visit are stored and available for analysis. Trajectories can be used to describe the movement behavior of objects and therefore they can assist in determining groups of objects with the same trajectory. A query example is: "return

the trajectory of object O_l". In this example, the location path of O_l is retrieved with the timestamp of each location.

The database history is a sequence of database states, one for each time, depending on the fixed global clock. Hence, *timestamps* will be strictly applied on the database history. Moreover, the value of an object will be different in two respective database states; therefore, timestamps queries can be classified as *lower timestamp*, *current timestamp*, *future timestamp* and *interval (range) timestamps* [70, 76, 83, 86]. *A lower timestamp* shows the recorded times of each moving object over previously visited location (past). For example, a delivery company wants to know the recorded time of delivery car X at its last location (historical queries). A lower timestamp is based on the previously recorded information for a moving object in the database. *A current timestamp* shows the current time of the moving object's visited locations [9, 27]. For example, "where is Object O_i Now". On the other hand, *the estimation future time-stamp* is based on the estimated time for the moving object to reach its next location (predictive queries) (see Figure 3.9) [28]. Here, the query must cooperate with the distance of the next location and the current velocity of the moving object to determine the expected future time, for example, "return the nearest ambulance that will reach Monash University Clayton Campus in the next 2 hours". Interval (range) timestamps specify a certain period of time which has a start and an end, for example, "return the ambulances that entered Monash University Clayton Campus between 2 p.m. and 5 p.m.".

Inside temporal queries indicate moving objects inside a certain time determined by the query. Inside temporal queries are different from the range timestamp queries because the specified times are not timestamps, but can be a day or a week; for example, "return the moving objects that were active yesterday". In

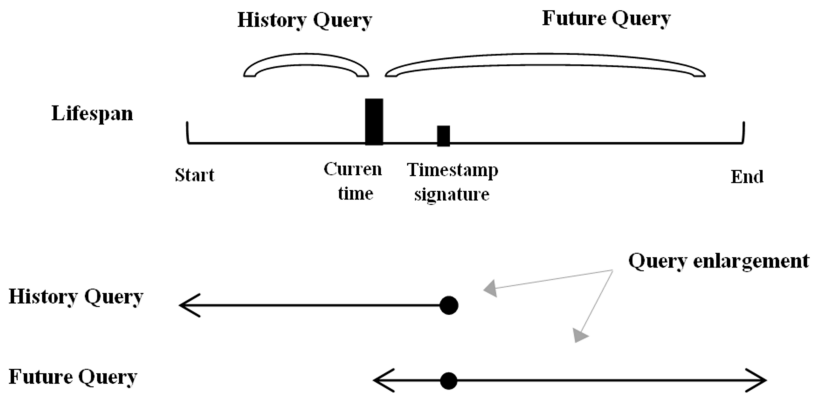


FIGURE 3.9: Timestamped queries (lower, current and future timestamps)

this example, the result should be the moving objects that were moving at any time during the previous day.

Disjoint temporal queries indicate moving objects that have nothing in common from the temporal perspective. In these queries, the times that are determined for each moving object are different. For example, "return two moving objects where one works on night shift and the other on day shift". In this example, the result should be two moving objects that have not worked together on the same shift.

Meet temporal queries are queries about moving objects whose time boundaries touch at a certain time. In other words, these are queries about moving objects that meet in their event times. For example, "return the moving objects that meet in any of their finishing or starting working times". In this example, the result would be the moving objects where the finish of their working time meets with the beginning of the other object's working time or otherwise (e.g. return O_1 and O_2, because O_1 working time finishes at 5 p.m. and O_2 working time starts 5 p.m.).

Equal temporal queries are queries about moving objects that are equal on the temporal side. The results of the equal temporal queries should have exactly

equal times. An example of an equal temporal query is: "return the moving objects that arrived at building A yesterday at 2 p.m. and left the building at 3 p.m.". In this example, the result should be only the moving objects that have met all the temporal aspects of the query, which are yesterday, 2 p.m. and 3 p.m.

Contain temporal queries are queries about moving objects that have several temporal similarities. Contain temporal query have some equal times between the moving objects. An example of a contain temporal query is "return the moving objects that arrived of building A at 3 p.m. or 5 p.m. or 6 p.m.". In this example, the result should be the moving objects that have any of the temporal aspects in the query, which are 3 p.m. or 5 p.m. or 6 p.m.

Overlap temporal queries are queries about moving objects that overlap at certain targeted times. An example of an overlap temporal query is: "return all moving objects where their working duties are overlapped with the day shift and the night shift". The result should be the moving objects that were in that place at both shift times. Figure 3.10 illustrates some of the temporal operations.

Period temporal queries are queries about moving objects that have a specified period of time. An example of a period temporal query is: "return all taxis that were at Monash University for 20 minutes". The result should be all taxis that were in the specified place (Monash University) for the specified period of time (20 minutes).

To sum up, the temporal perspective is associated with queries about moving objects which concentrate on the temporal aspects and characteristics of moving objects. Many issues and complexities can be considered in temporal perspective queries such as maintaining the ongoing updating of the time with the location

of the moving objects. Moreover, it is important to develop an index structure that facilitates temporal operation queries such as overlap and contains temporal queries.

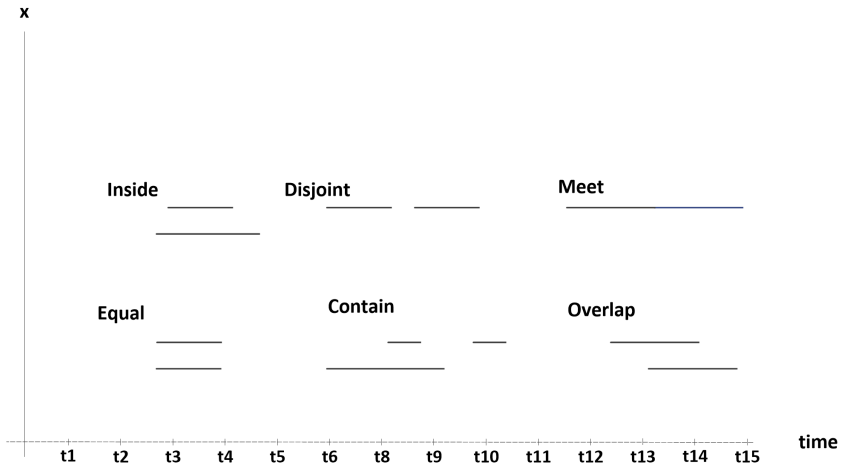


FIGURE 3.10: Temporal operations

3.1.5 Patterns Perspective

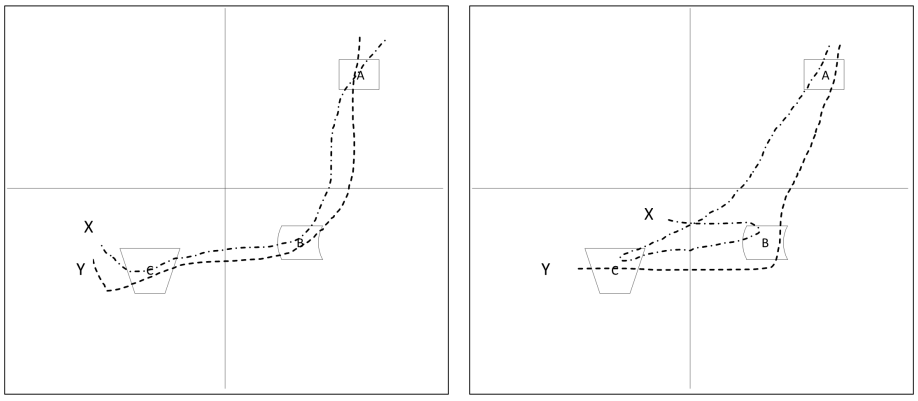
Here we argue that, according to this perspective, some queries will be raised only when their movement *patterns* or behaviors are known. Movement patterns are any interesting relationships in a set of moving objects or any recognizable regularity in spatio-temporal data. In this perspective, the query depends entirely on the objects' predefined movement patterns. Patterns perspective queries are classified as: *spatial patterns, spatio-temporal movement* and *temporal patterns*. This section explains these pattern categories with some movement pattern examples in terms of a number of characteristics and query examples.

3.1.5.1 Spatial patterns

Spatial patterns are patterns of movement that include the spatial concentration of the moving objects. In other words, spatial patterns are concerned with the spatial aspects of the moving objects. Therefore, these patterns address the spatial features of the moving objects and the impact on existing unique queries. Spatial patterns include many movement patterns and this section will be addressing some of them such as *co-location*, *concentration* and *Levy flights*, with several examples of queries.

Co-location occurs when the movements of the objects have similar locations in common [71, 87]. There are three types of co-location patterns: *ordered co-location* which exists when the common locations are reached in similar order; *unordered co-location* exists when the common locations are reached in different orders; and, *symmetrical co-location* exists if the common locations are reached in opposite orders [81]. For example, tourists are visiting four different areas in Victoria: Melbourne Zoo, Melbourne Museum, the Great Ocean Road and the Yarra Valley. If the tourists reach the locations in the same order, this will achieve the ordered co-location pattern; whereas, if the tourists reach the areas in different orders, an un-ordered co-location pattern will result. Moreover, if the areas are reached in reverse order, then symmetrical co-location is achieved. Figure 3.11 illustrates two types of co-location. The query example is: "return the tourist buses that are moving in the same order".

Concentration indicates the spatial concentration of moving objects at a certain instance of time; for example, congestion indicates an area that is affected by a crowd of vehicles in a transportation network. The query example is: "return the ambulances that are located in crowded areas of the city".



(a) Ordered co-location (b) Un-ordered co-location

FIGURE 3.11: Co-location patterns

The *Levy flights pattern* is named after Paul Pierre Levy, the French mathematician. It means a kind of random walk in which the increments are distributed based on a heavy-tailed probability distribution [88, 89]. In other words, after many steps, the distance from the original random walk will become a stable distribution [89, 90]. The Levy flight concept has been used in a large number of fields such as nature and biology [89, 90]. It has also been used in tracking stock market fluctuations, turbulent flow and human travel [88–90].

It is clear from Figure 3.12 that there is a range of small random flights, separated by a transformation to some extent jumping to another area. This can be illustrated with a simple example of a Levy flight random movement: a maintenance company receives a call from Monash University Clayton Campus for a maintenance callout, so the company's maintenance car jumps from the company location to the University. During their operations at the University, they receive an emergency call from the Chadstone Shopping Center for another maintenance callout. It is clear that the movement patterns here are random and not pre-planned, but the movement adopts the Levy flight aspect, where a stable distribution (in the University and Chadstone Shopping Center) has a jumping

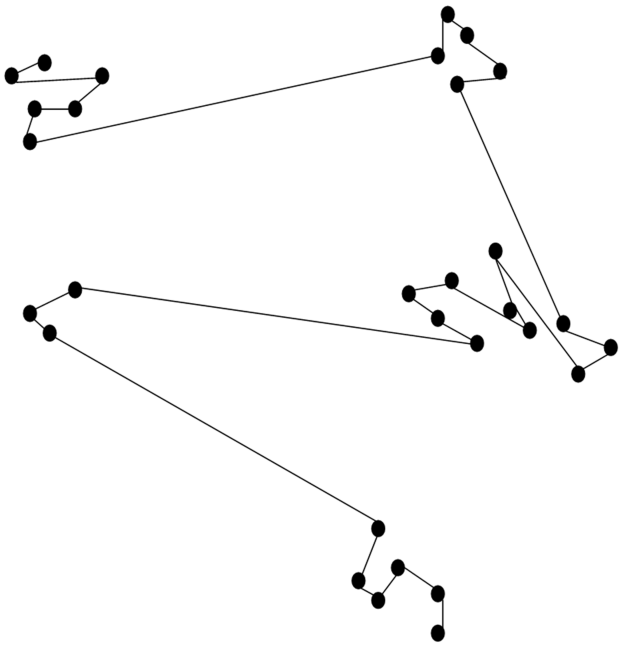


FIGURE 3.12: illustrates a jump in Levy flight

stage in between. Therefore, a query can be raised about this movement pattern such as: "return the total number of jumps which were made by maintenance car *A* today".

3.1.5.2 Spatio-temporal patterns

This pattern of movement concerns the spatio-temporal aspects of the movement. Therefore, these patterns address both the spatial and temporal features of the moving objects and the impact on existing unique queries. There is a variety of movement patterns in the spatio-temporal category; however, our aim is to not limit the movement patterns. Our aim is to illustrate that, with knowledge of the movement patterns, unique queries about the moving objects can be raised. To illustrate, this section explains some of the spatio-temporal patterns such as *incidents*, *constancy, sequence, periodicity, meet* and *moving cluster* with some query examples.

First, *incidents* movements among multiple objects can be divided into four pattern types: *concurrence, co-incidence, opposition* and *dispersion* [81]. *Concurrence* refers to an incident that contains a set of entities having similar values of movement attributes for a certain duration of time, whereas *co-incidence* is a specific type of incident that considers the similarity of the moving objects' positions. This type can be *full co-incidence* which means the same locations are reached at the same time, or *lagged co-incidence*, meaning that the same locations are reached after a time delay. *Opposition* indicates a multi-polar arrangement of movement parameter values, such as a sudden group splitting of moving objects into two opposite movement directions. *Dispersion* indicates a group of moving objects that perform a non-uniform motion (opposite of concurrence)[91]. A query example is: "retrieve the moving objects that reach location A at 9:00 p.m." (full co-incidence pattern).

Second, *constancy* is when the motion consists of parameters still with the same values or only slightly changed for a duration of time [81]. For example, a group of cars are moving on a straight road at a specific velocity, and direction and the derived parameters remain the same for a particular duration. The query example is: "track all the vehicles that are heading north and not accelerating at an unreasonable speed".

The third pattern is *Sequence*, which is a series of locations that have been visited as an ordered list. This type of pattern indicates a known start and end point in space and time. Sequence patterns are based on the locations and their timestamps. An example of a sequential pattern is a group of tourists visiting a set of places (Zoo, Museum and Gallery) in a particular sequence Zoo $\longrightarrow$ Museum $\longrightarrow$ Gallery within a specified duration of time [92].

Periodicity: this type indicates a cyclical pattern through a period of time (e.g. weekly or daily). This type shows a regular repetition of the movement (spatio-temporal periodicity) for a particular duration. For example, security patrolling cars regularly monitor specific locations through a regular repetition of movement in each period of time. The query example is: "return the locations that security car x regularly visits".

The *Meet pattern* consists of a set of moving objects that meet at a particular point or location. Note that the meet pattern can be a fixed meet or un-fixed meet depending on whether or not the moving objects that stay together for a certain duration are planned for the meeting region [81, 93]. For example, a group of friends move to meet in a certain cafe for a meeting (fixed), or a group of taxis move toward the airport to drop off passengers (un-fixed). Figure 3.13 illustrates the meet pattern. The query example is: "in a marathon running competition, a runner invokes a query to return the number of runners ahead of his/her location".

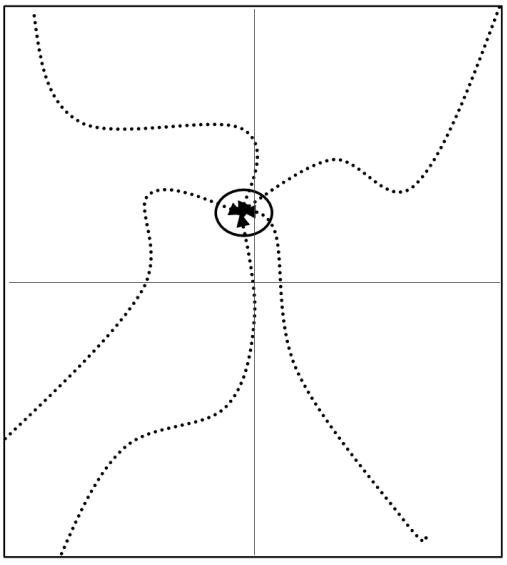


FIGURE 3.13: Meet pattern

A *moving cluster* is a set of moving objects that, during movements, stay close to each other while moving in a similar path for a specific duration. Note

that it is not necessary for the moving objects participating in the pattern to stay the same; an example is: a query about moving troops in parallel methods on a military battlefield.

3.1.5.3 Temporal patterns

The temporal patterns are the patterns of the movement that focus on temporal characteristics of moving objects. In other words, this pattern of movement is interested in the temporal aspects of the movement. There are a variety of movement patterns in the temporal category and for illustration we will explain some of the temporal patterns such as *temporal relations*, and *synchronization in time* with some query examples.

Temporal relations are patterns which are based on the time axis, such as a query about a group of moving objects stopping after a long trip. Figure 3.14 explains how the *temporal relations* pattern depends on the time axis.

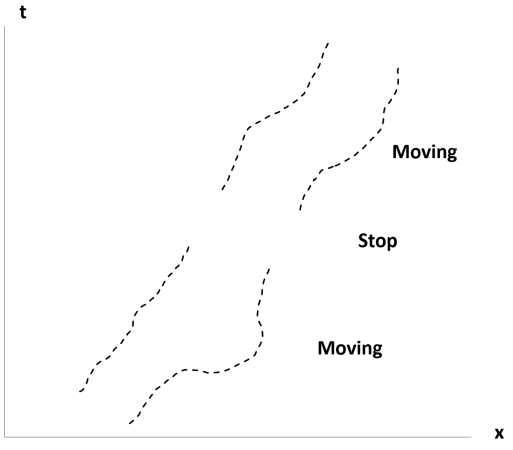


FIGURE 3.14: Temporal relations pattern

Synchronization in time is divided into two types. First, *full synchronization* indicates a pattern when changes of movement parameters occur simultaneously

with no time delay (e.g., velocities, direction). *Lagged synchronization* indicates a pattern when changes of movement parameters happen simultaneously but after a delay of time. For example, "return the police cars that change their direction on Monash Hwy 5 minutes from now".

Note that the aim is not to limit the number of movement patterns as there are a large number of other movement patterns. The main idea is to illustrate that with knowledge of movement patterns, new queries about moving objects can be raised. Moreover, these new queries can be integrated into any type of previous movement patterns and thus produce a set of different queries that depend on the combined movement patterns. Many issues and complexities can be considered using the patterns perspective, such as associating certain movement patterns in the index structure. Moreover, pre-defining movement patterns can influence the updating of the moving objects.

3.2 Discussion

Many works have been proposed to accommodate the intensive updating which is the main issue when indexing moving objects databases. As an example, suppose that we are tracking the positions of 2 million mobile phone users in Melbourne. Each user updates his/her position every 10 seconds, and a single location server keeps track of them. The location server continuously receives the location update stream as a sequence of location update records in a form of $(ObjectID, p(x, y))$, where $ObjectID$ is the moving object identifier and p is its location coordinates. The location server needs to efficiently handle 200,000 updates per second. Here, for each update operation, the location server will update the locations of the

moving objects which maintain the spatial index. This is necessary in order to answer the variety of spatial and location queries. Therefore, many works have been focusing on minimizing the above-mentioned update cost for each update. Moreover, index structures also aim to obtain a logarithmic search complexity.

This section summarizes our taxonomy which targets queries in most of the current moving object data structures. Note that the movement patterns in all of these index structures do not specify moving patterns for the moving objects. Many data structures have focused on the *location perspective* which includes the common queries in the spatial databases. For example, Saltenis et al. introduced the TPR-tree, (Time Parameterized R-tree) which is based on the R*-tree, in order to construct and manage moving objects. The TPR-tree and its successors [10, 67, 68] focused on the *spatial queries* such as range queries. Works such as [69] index moving objects in indoor spaces, which can then process the *navigational, topological* and *aggregation* queries. The RTR-tree and TP2R-tree [69] are two R-tree-based indexes for trajectories of objects moving in symbolic space.

In addition, moving objects have many types; therefore, the object queries can make a difference because they involve different data structures and algorithms. The majority of the current data structures focus on *moving points* but do not consider moving lines and moving regions. For example, the MV3R-tree, TPR-tree, FNR-tree, RP-tree and many others index the moving points as coordinates (x, y) [6, 10, 15, 25]. Moreover, works that concentrate on *temporal* moving objects include [17, 63, 64]. The Historical R-tree (HR-tree) is one of the earliest data structures to focus on historical data [64]. The HR-trees are efficient in cases of *timestamp queries*, as the search degenerates into a static query for which R-trees

are very efficient. Another work that focuses on *temporal queries* is the Multi-version 3D R-tree (MV3R-tree) [62] which basically uses multi-version B-trees and combines them with 3D R-trees [62, 63].

Works that concentrate on *trajectory queries* of moving objects are [14, 15], and include the trajectory bundle tree (TB-tree) which is based on the R-tree [8]. Another trajectory index is the STR tree (Spatio-Temporal R-tree) [4, 14, 15]. STR is based not only on spatial closeness, but also on trajectory preservation; therefore, it can easily support *trajectory queries*. Many other works such as [69, 85] focus on indexing the trajectory queries of moving object. Table 3.1 summarizes the taxonomy queries of moving objects in spatial databases, with some examples of the moving object data structures of the targeted queries.

TABLE 3.1: Taxonomy queries of moving objects

Perspective	Queries	Examples of Data Structures	Unique feature
Location perspective	Spatial queries Adjacency queries Navigational queries N-Body constraints queries Topological queries Density queries Aggregation queries	TPR-tree, TPR*-tree, RP-tree, Q+Rtree, RTR-tree, TP2R-tree and LUGrid	Location as main element
Motion perspective	Direction queries Velocity queries Displacement queries Distance queries		Motion vectors
Object perspective	Type queries Moving points Moving lines Moving regions	STR-tree, TPR-tree, TPRuv-tree and TPR-tree	Similarity of the objects' type or form
Temporal perspective	Trajectory queries Timestamped queries Inside temporal queries Disjoint temporal queries Equal temporal queries Contain temporal queries Overlap temporal queries Period temporal queries	HR-tree, MV3R-tree, TB-tree, SETI, STR-tree	Trajectories or timestamps
Patterns perspective	Spatial patterns Spatio-temporal patterns Temporal patterns		Movement patterns

As is clear, the majority of the moving object data structures focus on the location queries and use the metric distance as the indexing based since the underlying structures is Euclidean space or road network. Noticeably, there is a lack of support for location queries in **cellular space environments**. Moreover, the process of responding to location queries in cellular space will be different since the metric distance is not used for these; therefore, ***Chapter 4*** presents a new index structure that can serve all location queries. Here, the index structure is based on the adjacency between the cells which presents a unique data structure that can serve the *spatial queries* such as range, Knn queries.

In addition, since the indoor space is based on cellular space where the cell/rooms are connected with each other through doors, *adjacency queries* will be supported in the data structure presented in ***Chapter 4***. Note that no one of the previous data structures has considered this new query type. Moreover, the data structure considers the indoor characteristics such as doors, rooms and hallways and topology predicates (enter, leave, cross, etc.). Therefore, these queries involve location, path (navigational queries), and enter, leave (topological queries) and *aggregation queries* are easily supported by our proposal presented in ***Chapter 4***.

Furthermore, temporal queries are essential in indoor space. Many queries can be raised from the temporal prospective in indoor space such as security queries like: "where is Object O_1 at 9 a.m.?" Therefore, ***Chapter 4*** also presents a spatiotemporal adjacency-based index structures for moving objects. The temporal properties are supported in our proposed indexes which support the *timestamps* and *Interval queries*. Moreover, the temporal data is stored based on the idea of non-leaf node timestamping which also supports the *trajectories queries*.

The majority of the current data structures do not consider the *density queries*. Since the indoor space is based on cellular space where the objects positions are the cells/rooms, then the density cells queries is essential for indoors. Thus, ***Chapter 5*** indexes the moving objects by distinguishing between the high density cells and the low density cells which easily improve the performance and support the *density cells queries* in addition to the other location queries. Moreover, moving objects in the multi-floor indoor environments are indexed as multidirectional MBRs based on the proposed mutligraph which support the *wings/sections* positioning queries (***Chapter 6***). Also ***Chapter 7*** presents an extension of the adjacency indexing in indoor cellular space to the topographical outdoor space. The proposed index structure in ***Chapter 7*** supports the *location queries* such as adjacency, spatial, topological and aggregation queries.

Moreover, the characteristics of moving objects are different from those of static objects, such as *motion vectors*. A limited number of data structures concentrate on the motion vectors in the construction of the moving objects' data. Therefore, ***Chapter 8*** presents the DV-TPR*-tree which indexes moving objects based on the *spatial*, *direction* and *velocity* domains. Therefore, the DV-TPR*-tree is the only data structure that easily supports *velocity* and *direction queries*, apart from the common spatial queries.

3.3 Chapter Summary

This chapter presents a taxonomy for moving object queries to address the variety of queries that can be raised about moving objects of interest. The queries' taxonomy mainly uses five perspectives to retrieve moving objects and includes:

First, the *Location* perspective, which includes *spatial, adjacency, topological, navigational, N-body constraints, density* and *aggregation* queries; second, the *Motion* perspective, which covers the *distance, velocity, direction* and *displacement* queries; third, the *Object* perspective which covers type queries and form status queries (points, lines and regions); fourth, the *Temporal* perspective, in which the query can be a *trajectory, timestamped, inside, disjoint, meet, equal, contain, overlap or period* queries; and last, from a *Patterns* perspective, whereby the moving objects can be predefined movements which include many patterns (spatial movement patterns, spatio-temporal movement patterns and temporal movement patterns). Each perspective has been defined in a concise and expressive way according to a number of characteristics and by using examples. Moreover, this chapter summarizes the moving objects index structures and the query types that the supported in that index structures. It also provides a discussion of the query types that are considered in this thesis.

CHAPTER 4

Spatial and Temporal Connectivity Index for Moving Objects in Indoor Space

Chapter Plan:

Publications and Submissions:

- Sultan Alamri, David Taniar, Maytham Safar, and Haidar Al-Khalidi. "A connectivity index for moving objects in an indoor cellular space". *Personal and Ubiquitous Computing*, pages 1-15, 2013. ISSN 1617-4909. doi: 10.1007/s00779-013-0645-3.

- Sultan Alamri, David Taniar, Maytham Safar: "Indexing Moving Objects in Indoor Cellular Space". *NBiS* 2012: 38-44.

- Sultan Alamri, David Taniar, Maytham Safar, Haidar Al-Khalidi: "Spatiotemporal indexing for moving objects in an indoor cellular space". *Neurocomputing* 122: 70-78 (2013).

- Sultan Alamri, David Taniar, Maytham Safar: "Indexing of Spatiotemporal Objects in Indoor Environments". *AINA* 2013: 453-460.

In the past few years, much research has gone into the development of outdoor applications involving moving objects [12, 31, 94], and the indexing and querying of the trajectories of moving objects and their locations [4, 14, 74]. However, as mentioned previously, the outcomes of those researches are not suitable for indoor scenarios for the measurement differences and the suitable positioning devices. The measurement in outdoor space, Euclidean space or a spatial network, is typically used; whereas, indoor space is related to the notion of cellular space. Moreover, the positioning technologies differ for outdoor spaces and indoor spaces. In an outdoor space, a GPS is capable of continuously reporting the velocity and location of a moving object with varying accuracy. On the other hand, a proximity analysis is given by indoor positioning technologies which are not able to report the exact velocities or exact positions of objects [35, 70].

Therefore, this chapter presents a cells-connectivity-based index structure for moving objects. The index structure focuses on the moving objects based on the notion of cellular space, in contrast to the outdoor space structures which are based on the space domain, Euclidean or spatial network. Moreover, the key idea behind our moving objects indexing is to take advantage of the entities such as doors and hallways that enable and constrain movement in an indoor environment. Therefore, this chapter obtains an optimal representation of the indoor environment that is different from the outdoor environment. Moreover, if the indoor environment is seen in terms of connectivity between the adjacent cells, then the spatial queries can be answered more efficiently.

Furthermore, tracking the moving objects' updates in the indoor spaces is computationally expensive; therefore, we aim to reduce the tracking and the updating by means of efficient temporal techniques. Therefore, the temporal part of the index has been presented using three different techniques. The first uses

the non-leaf nodes timestamping where the FindPredecessor algorithm will be performed to connect the objects with its last location. The second monitors the objects that change their cells and delay the others, and the third uses the deltas for each cell to reduce the update cost. The contributions of this chapter can be summarized as follows:

- A moving objects index structure for indoor space (indoor-tree) which can manage memory wisely via a neighbours' distance lookup table, and efficiently serve the traditional spatial queries in addition to queries related to the connectivity in indoor environments.
- An indoor filling space algorithm to represent overlapping between the cells. The indoor space cannot precisely be transformed to a straight line (such as Hilbert space); therefore, we propose an expansion idea that provides an optimal representation of the filling indoor space.
- A spatial and temporal index that is intended to reduce the tracking and the updating of the moving objects. The temporal index is proposed with three techniques as follows:
  - Trajectories Indoor-tree (TI-tree), on the non-leaf nodes timestamping and the FindPredecessor algorithm that assists to connect the objects with their last location.
  - A Moving Objects Timestamping-tree in indoor cellular space (MOT-tree), which is based on the cells adjacency for the purpose of monitoring and dealing with objects that updated their cells.
  - Indoor Trajectories Deltas index based on the connectivity of cellular space (ITD-tree), which assists in reducing the update cost by using the deltas which is a temporary recorder of the objects' modifications in the floor space.

- We conduct experiments in order to evaluate the proposed indexes by studying the insert, search and the update performances. Furthermore, we evaluate the indexes based on the number of moving objects and the connections and complexities of the indoor space.

4.1 Preliminary

4.1.1 Motivation

To illustrate the problem, a database that records the position of a moving object in a coordinate base is considered. For each object, an initial $[x, y]$ is stored in the current time instant. Therefore, it is possible to determine the moving object's location based on its coordinates on the floor. Here we assume that the objects are indexed by a metric data structure such as (TPR-tree). Moreover, the system is dynamic; i.e., objects may be deleted or new objects may be inserted. Table 4.1 explains the notations used throughout this chapter.

Let $P(t_c) = [x_1, y_1]$ be the initial location of an object O_1 at the current time t_c. Then, the object changes its position which can be calculated as $P(t_c) = [x(t_c), y(t_c)] = [x_1 + v_x(t - t_c), y_1 + v_y(t - t_c)]$, note that $[v_x, v_y]$ is the velocity vector.

Now, we would like to answer a typical spatial query such as (kNN) of the form (based on Figure 4.1): "What is the 1NN objects to object O_1?" Assuming that the query q perform at t_c and the velocity vector is static at t_c. Given a location of

Table 4.1: Notations used throughout this chapter

Notation	**Definition**
C	Cell
S	Space
P	Set of moving objects
tc	The current time
F	Floor number
$NX(Oi)$	Next cell of object Oi
$Dist(q, O)$	The Euclidean distance between query q and object O
$ActDist(q, O)$	The actual distance between query q and object O (with consideration of the constraints entities)
$MINDIST(q, O)$	Minimum distance between query q and O
$CellDist(Oi, Oj)$	The number of hops of the cells between Oi and Oj
$R(P)$	P indexed on metric structures such as R-tree.
$CU(P)$	P based on cellular base
$Ci\ \chi\ Cj$	Ci is connected to Cj
$Ci \dashrightarrow Cj$	Ci continue the movement from the DC toward the last cell.
$Ci(N\chi)$	The number of connections in Ci
MIN	Minimum values
RC	Range of expand points (cells)
LE	Largest expand point
DC	Default Cell
N	Leaf node
$EX - P$	*Expand point*
MBR	Minimum Boundary Rectangle
$[x, y]$	x and y coordinates of a point
v	velocity vector
$ChildPTR$	The pointer to the child node
PTR	The pointer
O_n	The maximum capacity of a leaf node

$O_1 = [x_i, y_i]$ and its adjacent Objects $O_2 = [x_j, y_j], O_3 = [x_r, y_r]$ and $O_4 = [x_e, y_e]$. Based on the metric properties of the object's structure the result will be O_2. However, it can be noticed that O_2 is not the right answer, because of the entities that enable and constrain the movement in an indoor environment (doors and hallways). Based on the actual distance ($ActDist(O_1, O_2) > Dist(O_1, O_2)$), (see definition 4.1) O_2 is the farthest one from O_1, and O_3 is the right result as 1stNN. The indoor environment is not Euclidean space where objects can move freely

without restrictions. Everything that restricts the movement must be considered in order to obtain accurate results. Since the indoor space has these entities, and it needs to be fixed with multiple positioning devices, we argue that the connectivity between the cells is the optimal method of indexing the indoor space. Figure 4.2 illustrates the notion of cell connectivity (where the number of hops between the cells is used instead of Euclidean distance (see definitions 4.2, 4.3), (see Figure 4.3)) giving a more effective performance than do the metric structures. Moreover, indoor environments are not based on GPS, where the velocity and the location can be reported continuously. (Note that kNN queries illustrate the weakness of using a metric structure on indoor environments; *other spatial queries* can prove the same).

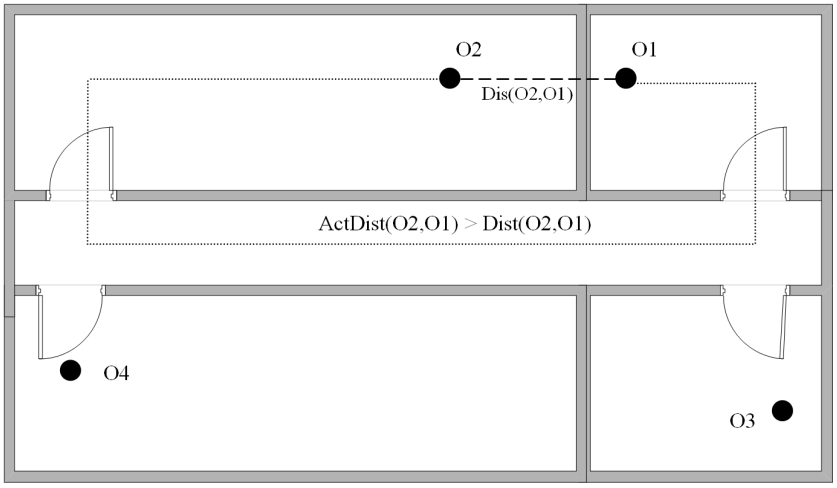


FIGURE 4.1: Using $R(P)$ in indoor space can return wrong results; in this example, O_2 is chosen as the 1stNN of O_1, where the right result is O_3 because of $ActDist(O_1, O_3) < ActDist(O_1, O_2)$.

Definition 4.1: let $P = \{O_1, O_2, ..., O_n\}$ a set of moving objects in indoor space, If $R(P)$, the (1stNN) of O_1 is O_2 if $ActDist(O_1, O_2) < ActDist(O_1, O_i))$ where $i \neq 2$.

Definition 4.2: Given a set of moving objects $P = \{O_1, O_2, ..., O_n\}$, If $CU(P)$, the $CellDist(O_x, O_y)$ is the number of hops of the cells, and is calculated as:

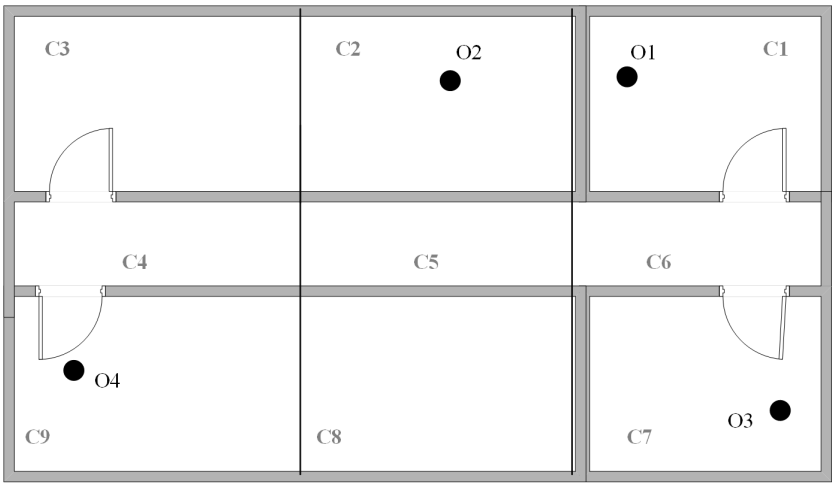


FIGURE 4.2: An example of the cells' coverage and distribution which is more likely to return an accurate result based on the neighbours and connections between the cells

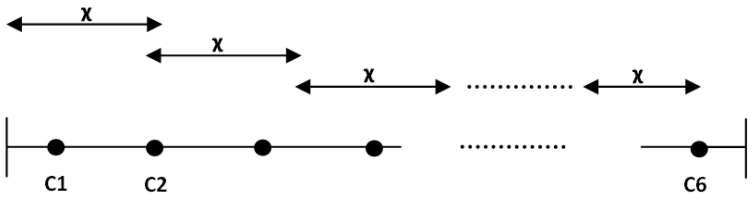


FIGURE 4.3: The number of cells hops from C_1 to C_6 $CellDist(O_1, O_6)$

$CellDist(O_x, O_y) = |\{C_1, C_2, ..., C_n\} - 1|$, where $O_x \overset{in}{\rightarrow} C_1$ and $O_y \overset{in}{\rightarrow} C_n$, $\forall$ $C_1 \chi C_2 \chi C_8 .....\chi C_n$.

Definition 4.3: Let $P = \{O_i, ..., O_n\}$ be a set of moving objects. In indoor cellular space, O_2 is considered as the 1stNN of O_1 if $CellDist(O_1, O_2) < CellDist(O_1, O_j))$ where $j \neq 2$.

Lemma 4.1: Let $P = \{O_i, ..., O_n\}$ be a set of moving objects in indoor space, O_j has minimum distance to O_i $MINDIST(O_j, O_i)$; however, O_x is considered as the 1stNN of O_i iff:

- $ActDist(O_i, O_j) > Dist(O_i, O_j)$ and

- $ActDist(O_i, O_x) < ActDist(O_i, O_j)$.
- $CellDist(O_i, O_x) < CellDist(O_i, O_j)$, $\forall$ j $\neq i$ and $j \neq x$.

Proof. Let $\{O_1, O_2, ..., O_n\}$ be the ordering of P moving objects at time t_c, sorted by position coordinates (x, y). Assume that we maintain a 1stNN query to O_1. Based on definition 4.1, since $ActDist(O_1, O_2) > Dist(O_1, O_2)$ and $ActDist(O_1, O_3) < ActDist(O_1, O_2)$. Moreover, based on definition 4.2, 4.3 where the $CellDist$ is the basis for the indoor spatial queries where $CellDist(O_1, O_3) < CellDist(O_1, O_j))$ where $j \neq 2$; therefore, O_3 is the 1stNN of O_1.

4.1.2 Cellular Space

Indoor space has restriction entities such as rooms, doors and walls; therefore, it is not possible to fix the whole indoor floor with one positioning device with high accuracy. Hence, the indoor floor will be fixed with many positioning devices and will be treated as cellular space. *The cellular space* is not based on any geometric representation of spatial property. Cellular space is a representation of the location according to sets of cells that include spatial objects (see definitions 4.4, 4.5). The indoor space data structure has an important requirement related to the notion of cellular space.

Definition 4.4: Given a set of cells $C = \{C_1, C_2, ..., C_n\}$, space S and a set of spatial objects $\{O_1, O_2, ..., O_n\}$, the space is called *Cellular Space* iff: $S = \bigcup C_i$, where C_i is the cell and $\exists\ C_j$ such that $O_i \overset{in}{\rightarrow} C_j$.

Definition 4.5: Given a set of cells $C = \{C_1, C_2, ..., C_n\}$, and a spatial object O_i, C_i is *an adjacent cell* to C_j $(C_i \chi C_j)$ iff $O_i \overset{in}{\rightarrow} C_j$ and $NX(O_i)$ is C_i.

The basic difference between cellular space and Euclidean space is the dependence of geometric representation of spatial property [35, 42, 69]. A query in outdoor space is given with coordinates such as (x_i, y_i) and (x_j, y_j). On the other hand, the queries in indoor space are usually based on cellular notations such as: "What are the moving objects in room 431?". In this example, the room number represents a cell identifier, which is the main difference from the outdoor coordinates in Euclidean space [43, 45, 70].

To illustrate further, take as an example a location within a train which is an indoor space. The location is identified by the wagon and seat numbers and not by its coordinates [49, 95]. For more explanation, in a geometric space, both the located objects and locations are shown as coordinate n-tuples, represented as points, areas, and volumes [26, 49, 96]. Basically, this type is based on reference coordinate systems (RCS) such as TB-tree, 3D R-tree and so on. On the other hand, the location of a moving object is indicated by abstract symbols or cells in cellular space. In cellular space, the location descriptions are represented by sets, and a located object in that case is considered as a member of these sets. Next, we illustrate how a moving object is represented in cellular space (see definition 4.6).

Definition 4.6. The moving object O in cellular indoor space is represented as $\{O, (C, t) \,|C \in S, t = [Time], F\}$, where O is the moving object, t indicates the current time, C is the cell, S is space and F is the floor number.

The space is divided into grids based on the coverage provided by its positioning devices (Wi-Fi)(see positioning systems background in Chapter 2). The division of the cells will depend on the technical resources of the positioning

devices' coverage and on the partitions (walls and obstacles). Figure 4.4 gives an example of the coverage cells' distribution.

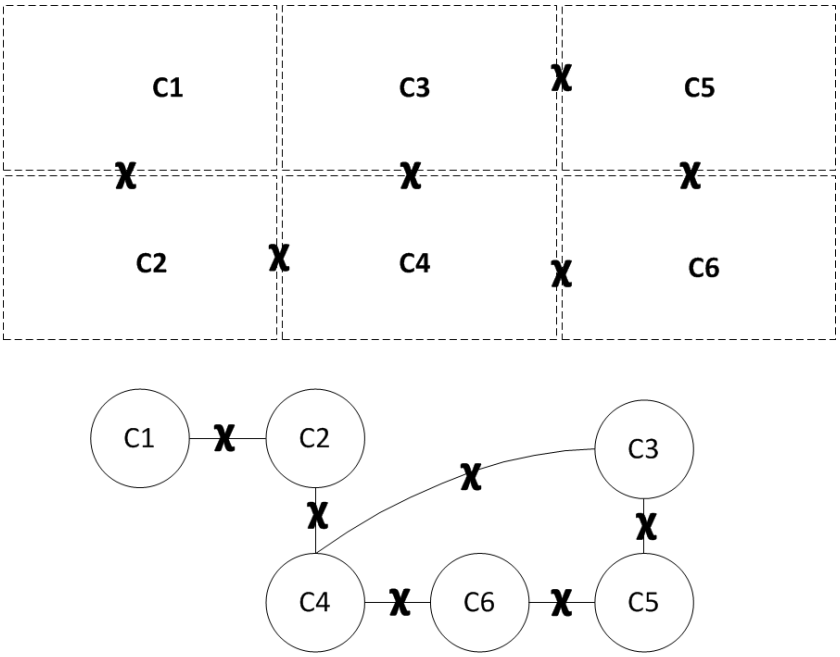


FIGURE 4.4: An example of the coverage cells' distribution, where χ means 'connected'

4.2 Spatial Indexing Moving Objects in Cellular Space

This section starts by explaining the representation of cellular indoor space. Then, it defines the uniqueness of our structure which consists of the *connections* (adjacency) in the indoor space cells with its structure and algorithm. Then, the *Indoor Filling Space Representation algorithm*, which is the structure that is based on the connections, facilitates the tree construction and the insert and delete algorithms. Subsequently, it represents the construction of the tree and the maintaining algorithms.

For illustration purposes, the next example consists of a floor plane of 7 rooms including stairs R_7 and corridor R_6. Figure 4.5 illustrates the floor space. Each venue is divided based on the sensors' coverage. For example, Room 6 which is the corridor is divided into 5 cells (C_{14}, C_{10}, C_5, C_2 and C_1) as shown in Figure 4.6. Note the stairs are treated as a room ($R7$ is the stairs). The result of the new cells' distribution is as shown in Figure 4.6.

FIGURE 4.5: An example of floor plane of 7 rooms

The indoor space is an overlapped environment which has many constraints such as rooms, doors and hallways. This work assumes that the overlap in the positioning devices' coverage is addressed by creating a new individual cell for any cells overlap.

From the cells' distributions, it is clear that for any object located in a particular cell, its next location must be in one of its adjacent cells. For example, (Figure 4.6) for objects located in C_{12}, their next location must be C_{16}. Therefore, we can establish connections between the cells which indicate the possible movements between the cells.

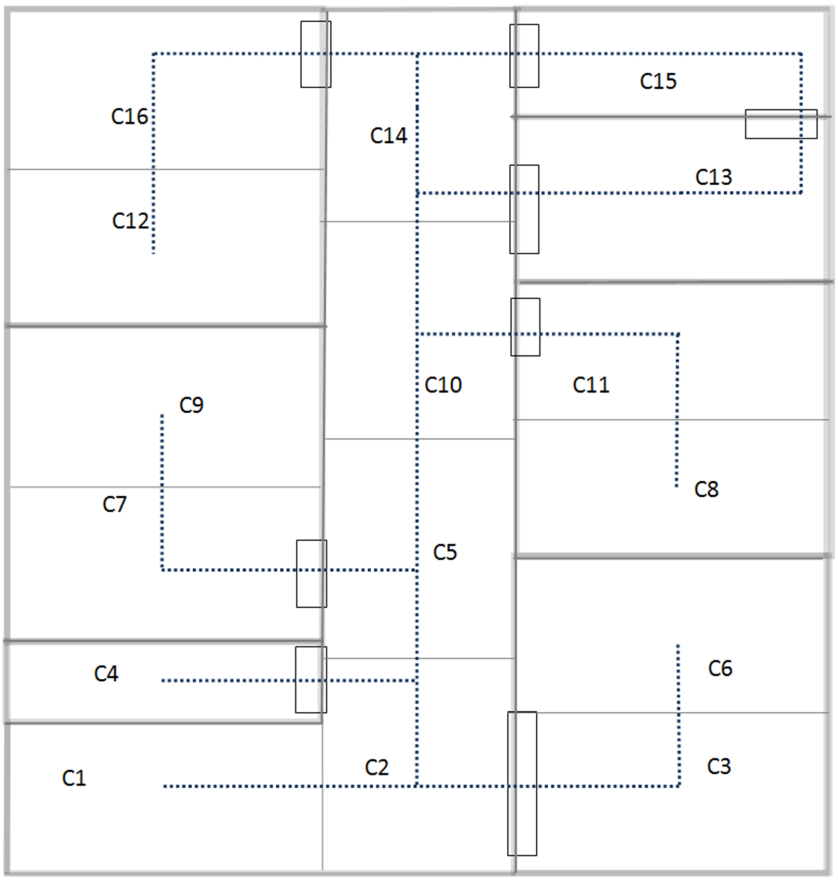


FIGURE 4.6: Illustration of the connectivity between the cells

The connection between the cells is stored in a neighbours' distance lookup table to facilitate the grouping in the data structure. In this table, a pre-computed neighbours' distances between cells is given. The distance is not metric, but is based on the number of hops, where 0 means the cell itself, 1 means neighbour number one, 2 means neighbour number two and so on. We assume that the positioning devices' coverage and the neighbours' distance lookup are pre-fixed. The neighbours' distance lookup table is established to assist in building *the Indoor Filling Space Representation algorithm*. Moreover, it will be used to compare the adjacency of the cells in order to obtain the adjacent cells in case of inserting, deleting or updating (to determine the suitable nodes). Note that the size of the neighbours' distance lookup table is small (Figure 4.7). Moreover, the cells' connections are stored as a multi-dimensional array list where the search complexity is $O(n)$ [97]. The checking algorithm that will use the table is called the Adjacency

Comparison algorithm, which returns the nearest connected cell (the cell that has the MIN value in the neighbours' distances lookup table).

	C1 [0]	C2 [1]	C3 [2]	C4 [3]	C5 [4]	C6 [5]	C7 [6]	C8 [7]	C9 [8]	C10 [9]	C11 [10]	C12 [11]	C13 [12]	C14 [13]	C15 [14]	C16 [15]
C1 [0]	0	1	2	2	2	3	3	5	4	3	4	6	4	4	5	5
C2 [1]	1	0	1	1	1	2	2	4	3	2	3	5	3	3	4	4
C3 [2]	2	1	0	2	2	1	3	5	4	3	4	6	4	4	5	5
C4 [3]	2	1	2	0	2	3	3	5	1	3	4	6	4	4	5	5
C5 [4]	2	1	2	2	0	3	1	3	2	1	2	4	2	2	3	3
C6 [5]	3	2	1	3	3	0	4	6	5	4	5	7	5	5	6	6
	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..

FIGURE 4.7: The neighbours' distances

Algorithm 1 Adjacency Comparison algorithm

```
/* check the inserted to Ci with set of C. */
/* Adjacency Comparison Algorithm will return the cell that has the MIN value */
R = initial value
for Ci adjacent cells AC.i do
    if AC.i < R then
        R = AC.i
    end if
end for
Return R // MIN value
```

4.2.1 Indoor Filling Space Representation

Indoor space is a filling space based on the connection between the cells [24, 70]. Since indoor space is usually based on cellular notation, unlike outdoor space which is based on coordinates (xi, yi), we need to establish a pre-computed indoor filling space algorithm (extracted from the neighbours' distance table) to represent overlapping between the cells. Since, the indoor space cannot be precisely transferred to a straight line, the expansion idea can assist us to represent the filling indoor space cells to determine the higher cell (has more connection) and the lower cell (has less connection). Note that the default cell DC is the cell that is chosen to be the main cell on the indoor floor.

Definition 4.7. Given a set of cells $C = \{C_1, C_2, ..., C_n\}$, C_j is considered as an expand point *EX-P* iff C_j --➝ C_n as C_1 --➝ C_2, C_2 --➝ C_j,...,C_j --➝ C_n where C_1 is DC and C_n is the last cell, $\forall$ $C_1 \chi C_2 .... C_j \chi C_n$.

Definition 4.8. Given a set of cells $C = \{C_1, C_2, ..., C_n\}$, the expand points' order are $C_1 > C_2$,...,$> C_j$, iff:

- C_1 is the DC and C_n is the last cell, and

- C_1 --➝ C_2, C_2 --➝ C_x,...,C_j --➝ C_n, $\forall$ $j \neq x$ and $x \neq n$.

Since the cells in the indoor space overlap, as shown in Figure 4.4, the cells connection is represented as an acyclic graph; from the default cell DC, based on definition 4.7, 4.8 the acyclic graph is converted to tree [98]. Figure 4.8 shows as example of a tree acyclic graph converted to tree. The indoor expansion will be explained next.

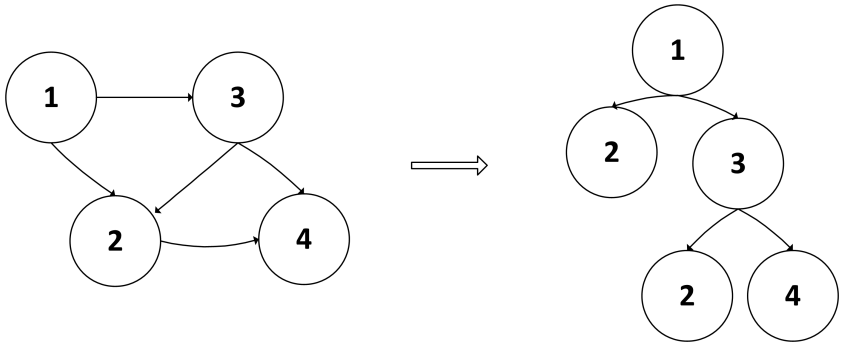


FIGURE 4.8: Converting acyclic graph to tree

Our objective is to use the data of the expansion idea (stored similarly to the neighbours' distance table) in the construction of the tree and the maintaining operations such as insertion and deletion. It is clear that each cell must be connected to an expand cell (point); therefore, advantage is taken of this to record the two expand points with non-leaf nodes as range and the largest expand point is recorded in the leaf nodes (for comparison and to choose node goals). The steps for the indoor filling space (expansion) algorithm are as follows:

- The algorithm establishes *expand points* which are the points (cells) based on definition 4.7, $C_1 \dashrightarrow C_2$, $C_2 \dashrightarrow C_j$,...,$C_j \dashrightarrow C_n$.
- The expansion starts from the default cell DC (assigned as an expand point) to the adjacent cells, where the algorithm will start to list the cells that have fewer adjacent cells. E.g. based on Figure 4.9, $C_4(N\chi) < C_3(N\chi)$.
- Then the algorithm will list the cells that have more adjacent cells (more connections); for example, those in Figure 4.9, $C_5(N\chi) > C_3(N\chi)$.
- The process is repeated up to the *last cell* (each cell that continues the connection will be considered as an *EX-P*).

Referring to Figure 4.9, assuming that C_1 is the default cell, the algorithm will start from C_1 and assign it as an *EX-P*. Then this will expand to the adjacent cells starting from the cells that have fewer connections (C_4) to the cells that have more connections (C_3 and C_5). Cells that continue the connection will be assigned as expand points (e.g. C_3 and C_5). The highest expand points start from the left in descending order to the right. The ordering is logical because the moving objects start the movement from the default cell (the highest (C_1)) to the last cell on the floor (in our case C_{12}). Note that this expansion does not restrict the movement of the objects; they can still freely move between any cells. The objective of this is to have a better understanding of the cells' overlap and connection. Then, we will take advantage of expand points and record the two expand points (RC) with non-leaf nodes as range, and the largest expand point (LE) will be recorded in the leaf nodes (for comparison and to choose nodes' goals). Figure 4.9 represents the expand steps of the floor example.

As mentioned, an indoor environment is characterized by entities that enable or constrain the movement (doors and hallways). Hence, based on this a data

structure, the connectivity between the cells is built. Traditional data structures such as R-trees and their followers based their comparison on Euclidean space, so the objects will be grouped together based on MBR least enlargement [8]. Moreover, other traditional data structures such as Hilbert R-trees and their followers based their comparison on the Hilbert value, so the objects are grouped together based on MBR based on *LHV* (*Largest Hilbert Value*) [99]. These techniques cannot be applied to an indoor environment which is not based on Euclidean distance. Furthermore, the indoor environment has overlapping areas, which cannot be treated as straight lines as are the Hilbert R-trees. Therefore, the expansion idea provides an optimal representation of the indoor space, which helps us to compare cells and group the objects based on their connections.

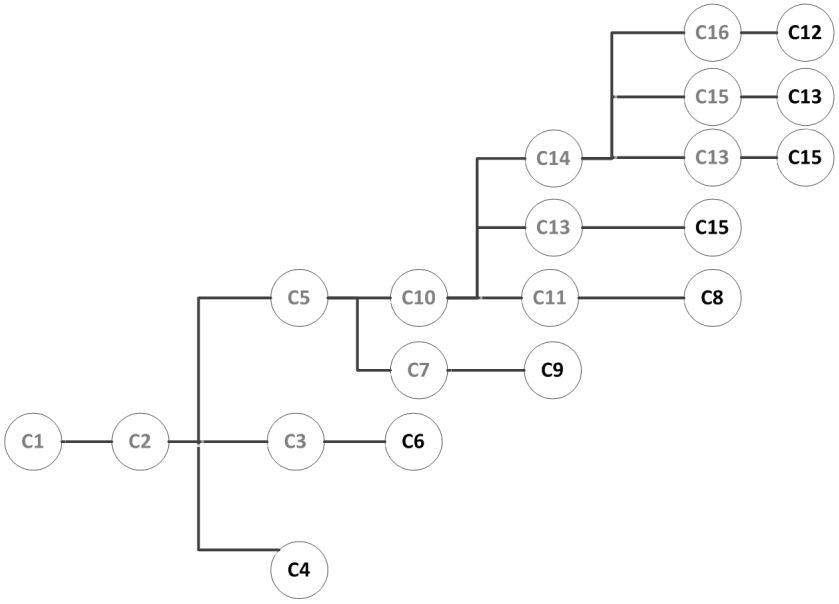


FIGURE 4.9: The expand steps of the floor example (grey indicates the expand points)

4.2.2 Tree Construction

Based on the adjacency concept mentioned in the previous section, the data structure groups the entries based on their adjacency cells. This is the best means of

measuring the indoor environment which is based on the adjacency and connections between venue and rooms. Conversely, the outdoor environment is based on metric measurement. Therefore, the idea is to start by grouping the objects inside the same cell. In the case of overflowing MBR, splitting will be performed to group it with one of the objects in adjacent cells based on the adjacency comparison algorithm. The idea is to check the *RC* (range cells) at the non-leaf node or the *LE* (the largest expand point) at the leaf node by comparing them with the inserted cells (by the adjacency comparison algorithm) and choosing the cell that has the MIN value (the nearest connected cell).

Definition 4.9. Given a set of cells $C = \{C_1, C_2, ..., C_n\}$, and set of non-leaf nodes $N = \{N_1, N_2, ..., N_n\}$, the *RC (Range cells)* in N_i is the two expand cells as follows:

- $\lfloor C_i \rfloor$ is highest expand point and $\lceil C_j \rceil$ is lowest expand point, where $C_i >$ C_x,...,> C_j, $\forall$ C_i, C_x,...,$C_j \in N_i$.

Definition 4.10. Given a set of cells $C = \{C_1, C_2, ..., C_n\}$, and set of leaf nodes $N = \{N_1, N_2, ..., N_n\}$, the *LE* is highest expand point $\lfloor C_i \rfloor$, where $C_i > C_x$,...,> C_j, $\forall$ C_i, C_x,...,$C_j \in N_i$.

Example: For the non-leaf nodes, using the data in Figure 4.11, the non-leaf node N_1 has three leaf nodes N_{1a}, N_{1b} and N_{1c}, and six cells = $\{C_{12}$, C_{16}, C_{14}, C_{13}, C_{10} and $C_8\}$. The *RC* for N_1 is $(C_{10}; C_{16})$(where C_{10} is the highest expand cell contained in N_1 and C_{16} is the lowest). For the leaf nodes, using the data in Figure 4.11, the leaf node N_{1a} has two cells = $\{C_{12}$, $C_{16}\}$. The *LE* for $N1a$ is (C_{10})(where C_{10} is the highest expand cell contained in N_{1a}).

In our indoor tree, the data then is ordered according to the connectivity between the cells. There are two main properties of an indoor-tree:

1. Non-leaf nodes contain (RC and $ChildPTR$) where RC is defined in definition 4.9. The RC is stored in each non-leaf node, which is based on the indoor filling space algorithm (grey in Figure 4.9). Thus, each non-leaf node will have a range of two expand points as the maximum and minimum cells. $ChildPTR$ is the pointer to the child node. A non-leaf node contains at most O_n entries, which is the maximum capacity of the non-leaf nodes.
2. Leaf nodes contain (LE, obj, and PTR) where LE is defined in definition 4.10. Therefore, one expand cell only is in the leaf node which is the largest connected cell at the node. For example, assume that a leaf node has three cells C_i, C_j and C_x where $C_i > C_j > C_x$ (three cells that contain the objects at that leaf node), C_i is recorded at the LE, because C_i is the largest expand point at that leaf node. The objects that are contained in the MBR at that time are denoted as obj, and PTR is the pointer. A leaf node contains at most O_n entries, which is the maximum capacity of the leaf. The structure is illustrated in Figure 4.10.

Using the sample data in Figure 4.11, suppose that the five MBRs are clustered into a larger MBR, where moving objects $= 1, 2, 3, 4, ..., 12$ and $M = 3$ and $m = 2$. Note that the objects' positions are shown only for simplification purposes. Furthermore, $N1$ RC is (C_{10}, C_{16}) (where C_{10} is the highest expand cell contained in N_1 and C_{16} is the lowest) and N_2 RC is (C_1, C_3) based on the indoor filling space (expansion) algorithm. In the leaf node N_{1a} LE is (C_{16}), N_{1b} LE is (C_{14}), $N1c$ LE is (C_{10}), and N_{2a} LE is (C_1), N_{2b} LE is (C_3). The indoor-tree is shown in Figure 4.12.

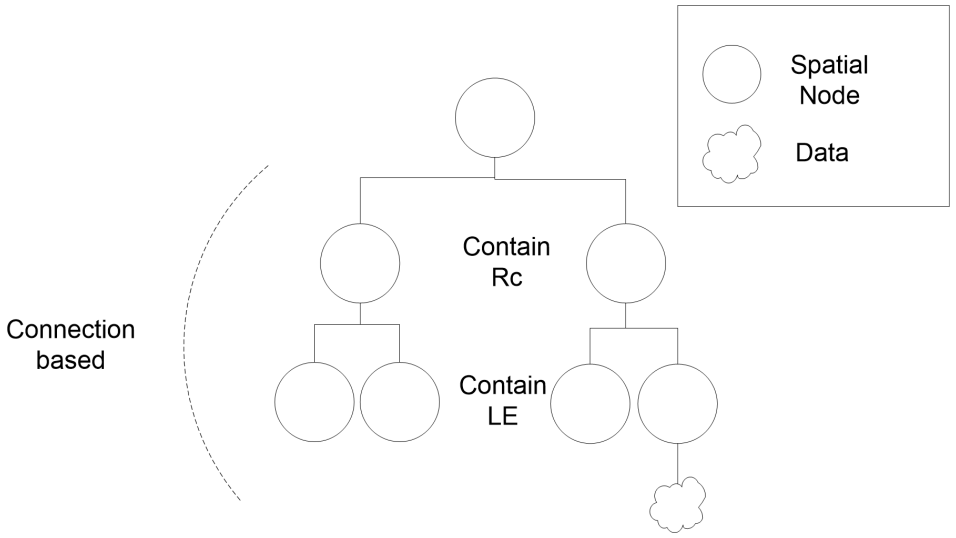


FIGURE 4.10: Indoor-tree

4.2.3 Insertion, Deletion and Updating

In our data structure, the splitting policy that will be used is called "immediate split policy". When an entry is inserted into a cell and the node overflows (see definition 4.11), the node has to be split immediately but needs to be grouped with the adjacent cells.

Definition 4.11. Given a set of Nodes $N = \{N_1, ..., N_n\}$, and spatial objects O_1,O_2,...,O_n, where N_i contain at most m entries. N_i is indicated as an *overflow* node if N_i contain $> m$ entries.

Definition 4.12. Given a set of Nodes $N = \{N_1, ..., N_n\}$, and spatial objects O_1, O_2,...,O_n, where N_i contains at most m entries. N_i is indicated as an *underflow* if N_i contain $< m/2$ entries.

The *choose leaf node algorithm* starts to check the range of the cells (RC) (for the non-leaf nodes) and compares it with the inserted cell. If the inserted cell is one

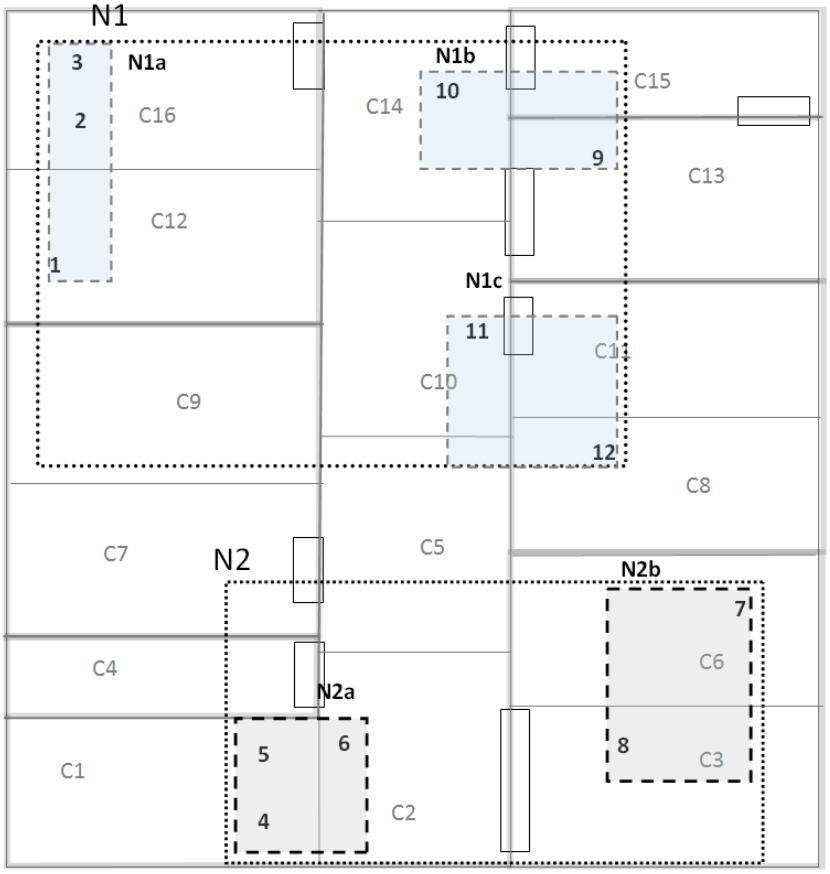


FIGURE 4.11: The MBRs grouping based on the connection

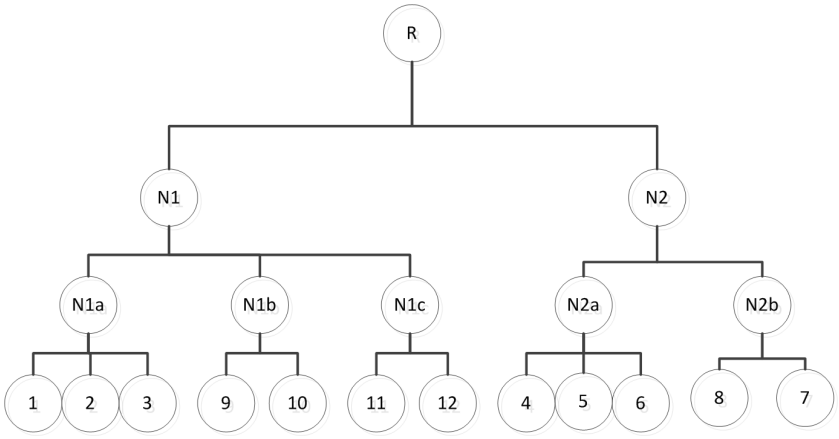


FIGURE 4.12: An example of an indoor-tree

of the range cells, the node that has this RC is chosen; otherwise, the algorithm will check the inserted cell with each range (RC) (called Adjacency Comparison Algorithm) and choose the one that has a MIN value cell. For the leaf node, the algorithm checks the inserted cell with the LE and chooses the node that has the MIN value cell. For example, using the sample data in Figure 4.11, assume

that an entry '13' will be inserted to C_{16}. The algorithm starts with the *choose leaf algorithm*, where it has to identify which non-leaf node is suitable for the new entity. Then *choose leaf algorithm* will call the adjacency comparison algorithm to compare C_9 with the RC for each non-leaf node; hence, it chooses N_1. However, the non-leaf node N_1 is overflowing which will lead to an immediate split. N_1 will be split to $N1'$ and $N1''$. Moreover, the algorithm will choose the leaf node N_{1a} (result of comparing LE with C_{16}) which is overflowing as well. N_{1a} will be $N_{1a'}$ and $N_{1a''}$. The non-leaf node splitting and the leaf node splitting are shown in Figure 4.13. If the RC comparison results and LE comparison results are equal, the algorithm will choose the node that has more neighbours (connections)(algorithm 2).

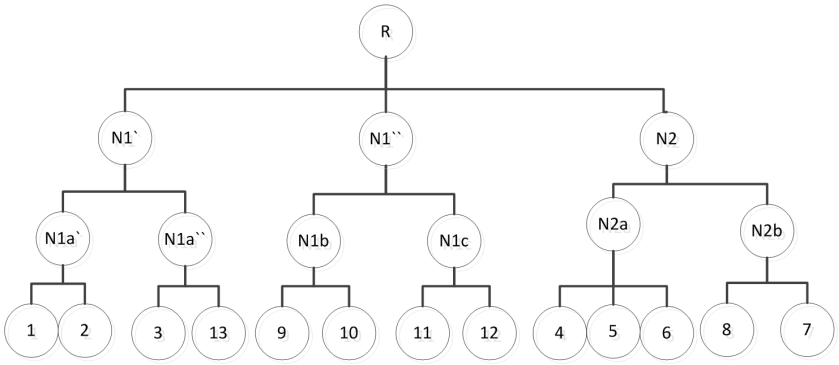


FIGURE 4.13: Inserting 13 to $N1a''$

In the deletion operation, first it checks for the underflowing node, instead of the overflowing node. If a node is underflowing (see definition 4.12), then it is needed to to check the sibling nodes that are based on the connection to participate in solving the underflow. The deletion algorithm will be explained in the next example. Suppose that we want to delete object 7 from C_6 where $m = 2$ and $M = 3$ (Figure 4.13). The algorithm will first call the FindLeaf algorithm which iteratively searches for the target node from the root node to the leaf node [10, 56]. After discovering that N_{2b} is underflow, the delete algorithm will determine the sibling-connected node based on the adjacency comparison algorithm (Figure 4.14). Then re-insert the remain entities into that node (if it is overflowing a split is performed,

Algorithm 2 ChooseLeaf algorithm

```
1: /* Return the leaf node that has the inserted n */
2: Set N to the root node
3: if N is non-leaf node then
4: |  Call adjacency comparison algorithm(ACA) compare with RC
5: |  Node(N_i) has MIN Value
6: |  Choose the (N_i,ptr)
7: |  if N_i.value = N_j.value then
8: |  |  Call adjacency comparison algorithm(ACA)
9: |  |  Check RC at N_i AND RC at N_j
10: |  |  if RC.N_i adjacents > RC.N_j adjacents then
11: |  |  |  Choose the (N_i)
12: |  |  else
13: |  |  |  Choose the (N_j)
14: |  |  end if
15: |  end if
16: end if
17: if N is leaf node then
18: |  Call ACA compare with LE
19: |  Node(N_x) has MIN Value
20:    Choose the (Nx, ptr)
21: |  if N_x.value = N_y.value then
22: |  |  Call adjacency comparison algorithm(APA)
23: |  |  Check LE at N_x AND LE at N_y
24: |  |  if LE.N_x adjacents > LE.N_y adjacents then
25: |  |  |  Choose the (N_x)
26: |  |  else
27: |  |  |  Choose the (N_y)
28: |  |  end if
29: |  |  if LE.N_x = LE.N_y then // (overlapping case)
30: |  |  |  Choose the Node that has fewer entities
31: |  |  end if
32: |  end if
33: end if
```

as explained in the insertion algorithm). Hence, in the deletion, if underflow occurs, we deal with the adjacent cells.

Although the nodes have some sort of adjacency (connection) in an indoor-tree, in the search process, the adjacency is not used. Therefore, the search process

Algorithm 3 Insert algorithm

```
1: /* Input: o is the entry to be inserted. */
2: procedure INSERT(o into C)
3:     Insert o into the C
4:     call ChooseLeaf to find the suitable leaf node N
5:     if N overflows then
6:         Choose the new N' based on ACA
7:         Split N
8:         Group o in N'
9:         Update RC,LE at N,N'.
10:    end if
11:    if node split propagated to the root node and root node to split then
12:        Create a new root node with new resulting child nodes
13:    end if
14: end procedure
```

Algorithm 4 delete algorithm

```
1: /* Input: o is the entry to be deleted. */
2: procedure DELETE(o from C)
3:     delete o from the C
4:     FindLeaf to find the leaf node N
5:     if N underflows then
6:         find N sibling adjacent based on ACA
7:         re-insert o into N_i
8:         if N_i overflows then
9:             HandleOverflow(N_i, o)
10:        end if
11:    end if
12:    Update RC,LE at N,N_i.
13:    if node split propagated to the root node and root node to split then
14:        Create a new root node with new resulting child nodes
15:    end if
16: end procedure
```

in the indoor-tree follows the same search process as in the original R-tree [8]. So, basically the search starts from the root node, and descends the tree, examining all nodes that intersect the query scope. Clearly then, one essential advantage of our data structure is that the grouping of the objects is based on the connectivity between the cells. Therefore, it is more likely that objects will move from one cell

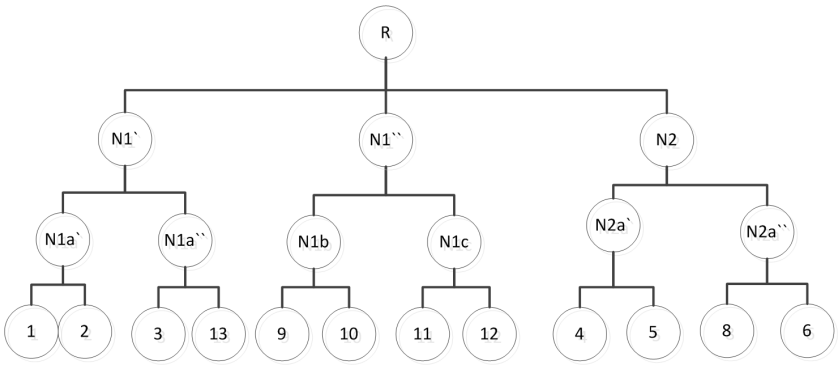


FIGURE 4.14: After deleting 7, object 8 is reinserted to N_{2a}

to another without any updating in the nodes (since a node contains the adjacent cells).

Algorithm 5 update algorithm

```
1: /* Input: o is the entry to be updated. */
2: procedure UPDATE(o from C)
3: |  FindLeaf to find the leaf node N
4: // the FindLeaf function
5: |  if o move to adjacent cells then
6: |  |  positioning device updating
7: |  end if
8: |  if N effected then
9: |  |  if N underflows then
10: |  |  |  ChooseLeaf function to find N sibling adjacent
11: |  |  |  re-insert o into N_i
12: |  |  end if
13: |  |  if N_i overflows then
14: |  |  |  HandleOverflow(N_i, o)
15: |  |  end if
16: |  end if
17: |  if node split propagated to the root node and root node to split then
18: |  |  Create a new root node with new resulting child nodes
19: |  end if
20: end procedure
```

4.3 Spatiotemporal Indexing Moving Objects in Cellular Space

Here we aim to reduce the tracking and the updating by efficient temporal techniques. Therefore, the temporal part of the index has been presented using three different techniques. The first is based on the non-leaf nodes' timestamping where the FindPredecessor algorithm will be performed to connect the objects with its last location. The second is for monitoring the objects that change their cells and delay the others, and the third is based on the deltas for each cell to reduce the update cost. Table 4.2 illustrates the four types of queries that can be supported by our proposed data structures.

TABLE 4.2: Aim Queries types in indoor Adjacency data structures

Query Type	**Example**
Spatial Queries	Return the 3NN of object X in C_8?
Topological Queries	Which Object enter C_6?
Adjacency Queries	Return the adjacent objects of object O_3 ?
Aggregation Queries	Return how many objects in C_5?
Interval Queries	Return the moving objects which have been in C_12 between t_2 and t_7?
Historical Queries	Return the cell whose object O_12 was in t_3?

4.3.1 Technique 1: Trajectories Indoor-tree (TI-tree)

4.3.1.1 Tree Construction

Based on the adjacency method mentioned in the previous section, the data structure will group the entries based on their adjacent cells at the current time linked with their predecessors. Therefore, this technique starts by grouping the objects

inside the same cell. Splitting will be performed to group it with one of the objects in the adjacent cells in case of any overflowing MBR. The idea is to check the RC (Range cells) at the non-leaf node or the LE (the largest expand point) at the leaf node by (by the adjacency comparison algorithm) to choose the MIN value cell (the nearest connected cell) (check *definition 4.9* and *definition 4.10*). Furthermore, when the leaf node is determined, the FindPredecessor algorithm will be performed to connect the objects with its last location. There are two main properties of the Trajectories Indoor-tree (TI-tree):

1. Non-leaf nodes contain (RC and $ChildPTR$) where RC is the range of the cells. We store the range of cells in each non-leaf node, which is based on the expansion algorithm (Figure 4.9). Moreover, the lower non-leaf nodes which are linked to the leaf nodes will have an additional attribute which is the $TimeID$ (RC, $TimeID$ and $ChildPTR$) where $TimeID$ which indicates the time of that non-leaf node. The $TimeID$ is stored in the lower non-leaf nodes (not the higher levels of the non-leaf nodes (in case the non-leaf nodes become multi-levels)) to include the temporal aspects in the data structure which better meets the demands of the temporal queries. $ChildPTR$ is the pointer to the child node. Note that the non-leaf node contains at most O_n entries, which is the maximum capacity of the non-leaf nodes.
2. Leaf nodes contain (LE, MO, C_i and ptr) based on the expansion algorithm, where each cell is connected to the expand cell (point); hence, the LE is recorded as the largest expand point (one expand cell). The moving objects are denoted as MO, C_i is the cell and ptr is the pointer. The leaf node contains at most O_n entries, which is the maximum capacity of the leaf. The structure is illustrated in Figure 4.15.

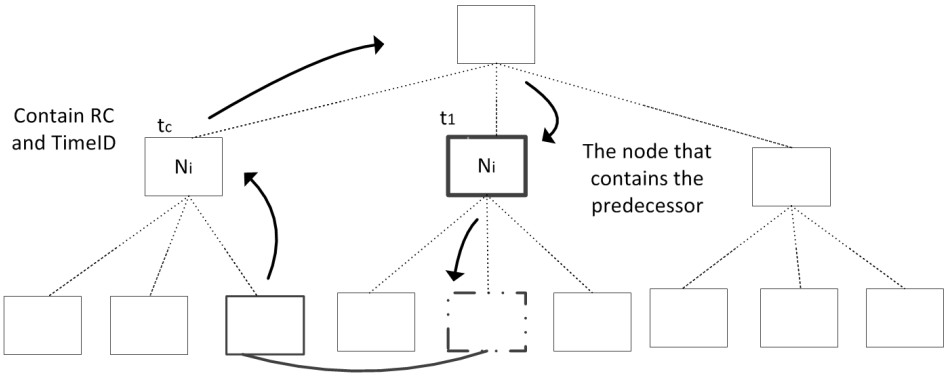


FIGURE 4.15: Trajectories Indoor-tree (TI-tree)

Using the sample data in Figure 4.16, current time t_c, suppose that the four $MBRs$ are grouped into larger $MBRs$, where moving objects $= 1, 2, 3, 4, ..., 11$ and $M = 3$ and $m = 2$. Note that the objects' positions are shown only for simplification purposes. Furthermore, N_1 RC is (C_1, C_6)(where C_1 is the highest expand cell contained in N_1 and C_6 is the lowest) and N_2 RC is (C_{10},C_{11}) based on the expansion algorithm. In the leaf node N_{1a}, LE is (C_1), N_{1b} LE is (C_7), and N_{2a} LE is (C_{10}), N_{2b} LE is (C_{11}). The current time indoor tree is shown in Figure 4.17.

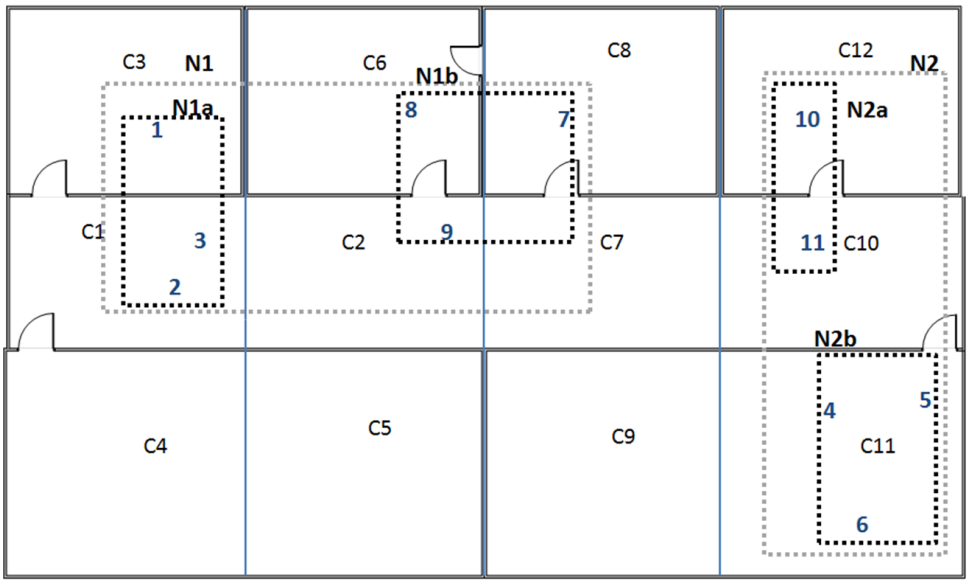


FIGURE 4.16: Illustrates the MBRs grouping based on the connection (current time)

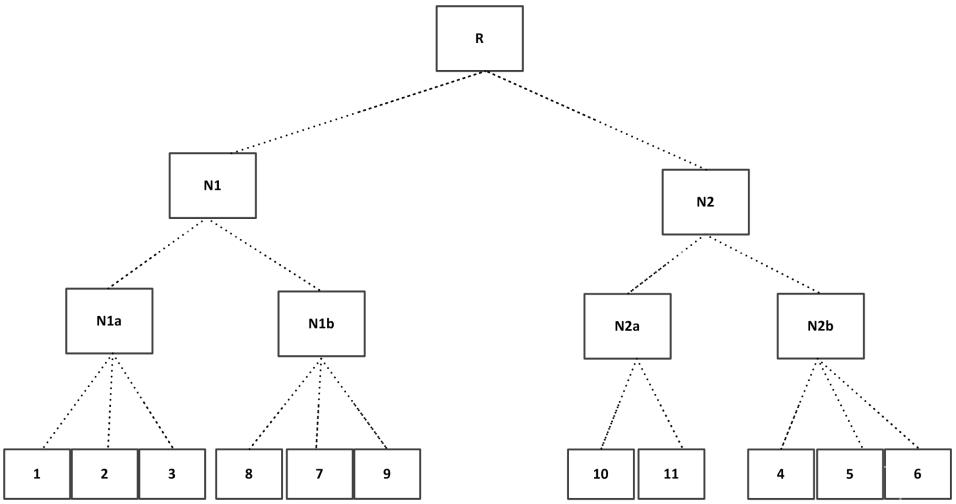


FIGURE 4.17: An example of TI-tree at t_c

Assume that the following objects change their location in t_2, where Object "2" checked out from C_1 and checked in C_2, Object "1" checked out from C_3 and checked in C_1, Object "8" checked out from C_6 and checked in C_8, Object "5" checked out from C_{11} and checked in C_{10} and the objects "3", "4", "6", "7", "9", "10" and "11" remained in their cells. The TI-tree will be updated as follows: for "2" the chooseleaf algorithm compared the new cell C_2 with RC and the LE in all nodes, and the adjacency comparison algorithm found that the suitable non-leaf node is N_1. Moreover, "2" will be grouped in N_{1b}, which led to the splitting of $N_{1b'}$ and $N_{1b''}$. Note that the non-leaf node has been updated to the current time and the previous time recorded is t_1. *FindPredecessor* algorithm now will be performed to connect the objects with their last location, and initiate a linked list between them. Figure 4.18 illustrates the updating of the previous scenario. We notice that only "2" caused a splitting, but the rest of the moving objects did not cause any splitting. Therefore, we argue that the connectivity index is efficient to reduce the cost, where it is more likely that a moving object will move to its adjacent cells, which will require updating only the cell number.

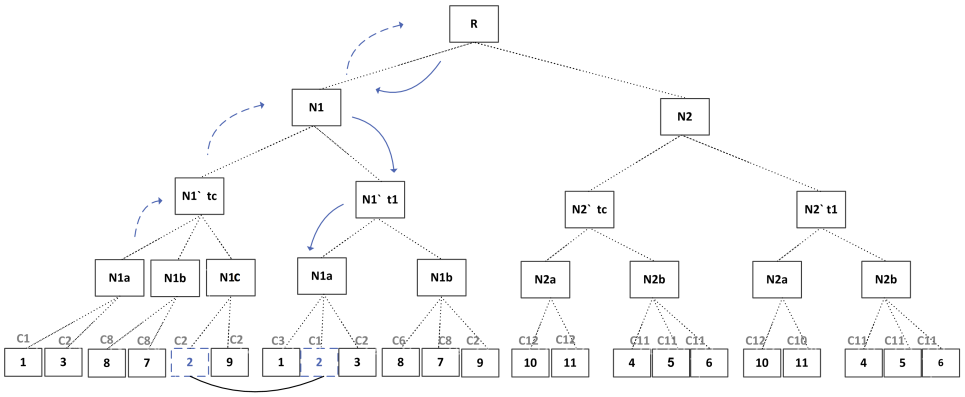


FIGURE 4.18: Updating the TI-tree from t_1 to t_c (linked list is shown only for "2", the remaining objects are treated similarly)

4.3.1.2 Insertion, Deletion and Update

The TI-tree used an immediate split policy. When a node overflows when an entry is inserted, the node has to be split immediately to group connected cells' entries. The *chooseleaf* algorithm starts to check the range of the cells (RC) (for the non-leaf nodes) and compares it with the inserted cell. If the inserted cell is one of the range, the chooseleaf algorithm chooses the node that has these RC; otherwise, we check the inserted cell with the each range (RC) (call *Adjacency Comparison* algorithm) and it chooses the one that has the MIN value cell. For the leaf node, the algorithm will check the inserted cell with the LE (called the *Adjacency Comparison* algorithm) and will choose the node that has the MIN value cell. Note that when inserting a new object the *chooseleaf* algorithm checks only the current time non-leaf node (t_c), and the old $timeIDs$ will be ignored. In cases where the RC comparison results and LE comparison results are the same, the insert algorithm will choose the node that has more neighbours (connections).

In deletion, when the moving object is deleted from the indoor floor (or checks out of the floor), it checks for the underflow node, instead of the overflow node. If a node is an underflow, the sibling nodes that are based on the connection must

participate in solving the underflow. The deletion algorithm will be illustrated in the next example. Suppose that we want to delete object "9" from C_2 where $m = 2$ and $M = 3$ (based on the data in Figure 4.18). The algorithm first will call the FindLeaf algorithm which iteratively searches for the target node from the root node to the leaf node similar to R-tree [8]. After discovering that N_{1c} is underflow, the deletion algorithm will determine the sibling connected node based on the adjacency comparison algorithm. Then re-insert the remaining entities into that node (if it is an overflow, it will be splitted as explained in the insertion algorithm). Hence, in the deletion, if underflow occurs, we deal with the adjacent cells. Note that the last location of the deleted object still exists, which can be a target in security and historical queries.

Although the nodes are based on adjacency in TI-tree, in the search process, adjacency is not used. Thus, the search process in TI-tree follows the same search process as in the original R-tree. Note the important point in TI-tree is creating a new non-leaf node (lower level) with the current time (t_c). Note that a new non-leaf node (lower level) could cause a splitting due to the overflows in the higher levels or the leaf node. Figure 4.19 illustrates object "5" updated from C_{11} to C_{10} at t_4 (assume that the objects t_3 remain at same locations). In the updating process, it is clear that one essential advantage of our data structure is that it is based on connection cells. Therefore, it is more likely that objects move from a cell to another without any updating in the nodes (since the node contains the adjacent cells).

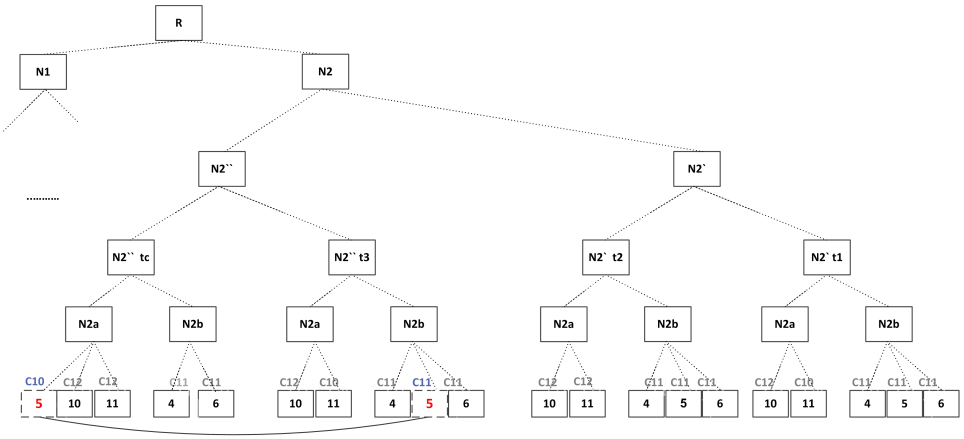


FIGURE 4.19: Updating the TI-tree from t_3 to t_c (linked list is shown only for "5", the remaining objects are treated similarly)

4.3.1.3 FindPredecessor algorithm

The ideal task for a FindPredecessor algorithm is performed after completing the insertion, where its mission is to obtain the trajectory by linking the current time with the old times of the objects (see definition 4.13). Algorithm 6 and Figure 4.20 illustrate the steps of the FindPredecessor algorithm.

Algorithm 6 FindPredecessor algorithm

```
1: /* Input: O_x is the entry to be linked with its predecessor. */
2: procedure (linklist) (O_x, N_x.t_c − N_y.t_j)
3:     FindLeaf to find the leaf node N.t_c which contains O_x // current time
4:     if O_x found then
5:         Return N_x.t_c
6:         FindLeaf to find the leaf node N.t_j which contains O_x // old time
7:         if Ox found then
8:             Return N_y.t_j
9:             establish linklist between (N_x.t_c, N_y.t_j)
10:        end if
11:    end if
12: end procedure
```

Definition 4.13. Given a set of nodes $N = \{N_1, N_2, ..., N_x\}$, and a spatial object O located at C_i at time t_i, the predecessor node of C_i is the leaf node that contains

the last location of O at t_{i-1}.

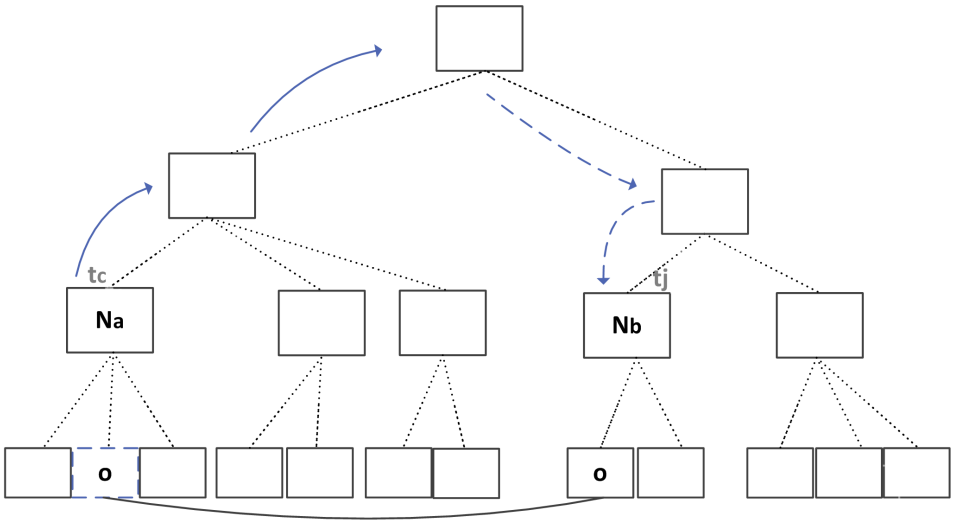


FIGURE 4.20: The linked list in the FindPredecessor algorithm

4.3.2 Technique 2: Moving Objects Timestamping-tree in an Indoor Cellular Space (MOT-tree)

In indoor environments, the objects' movements are slower than those of objects moving outdoors. As explained previously, the indoor environments are based on cellular notations, since it is pointless to report locations continuously. Therefore, the objects change their locations between the cells with a certain amount of stabilization in some cells. For example, an object O moves from C_i to C_j at t_1 (t_1 illustrates the timestamp), then stays at C_j from t_1 to t_9, and then checks out from C_j and checks into a new cell. Note that there is no need to track the object O between t_1 to t_9. As a result, we argue that this technique of monitoring and dealing with only the updated moving objects (checked out and checked in to a new cell) is an ideal means of reducing the updating cost. The MOT-tree is based on the adjacency method as well, as the data structure will group the entries based on their adjacency cells at the current time, linked with their predecessors.

The MOT-tree comprises the following steps. It starts by grouping the objects inside the same cell. Splitting is done to group them with one of the objects in connected cells in the case of any overflowing MBR. The RC (Range cells) is checked at the non-leaf node and the LE (the largest expand point) is checked at the leaf node (by the adjacency comparison algorithm) in order to choose the MIN value cell (the nearest adjacent cell) [38]. Furthermore, when the leaf node is determined, the FindPredecessor algorithm is applied to connect the object with its last location. The key idea in the MOT-tree is to update only the moving objects that updated their cells; objects that remain in their cells will not be processed until they check out or check in to new cells. There are two main properties of the MOT-tree:

1. Non-leaf nodes contain RC and $ChildPTR$ where RC is the range of the cells which is stored in each non-leaf node using the expansion algorithm. Hence, each non-leaf node has a range of two expand points as the highest cell and lowest cell. $ChildPTR$ is the pointer to the child node. Note that the non-leaf node contains at most O_n entries, which is the maximum capacity of the non-leaf nodes.
2. Leaf nodes contain $LE, obj, C_i, TimeID$ and PTR. Using the expansion algorithm, whereby each cell is connected with expand cells (points), the largest expand point (one expand cell) is recorded in the leaf node as LE (see definition 4.10). The objects that are contained in the MBR are denoted as obj, C_i denotes the cell. Note that in this Technique, the $TimeID$ is included in each leaf node. The leaf node contains at most O_n entries, which is the maximum capacity of the leaf. The structure is illustrated in Figure 4.21.

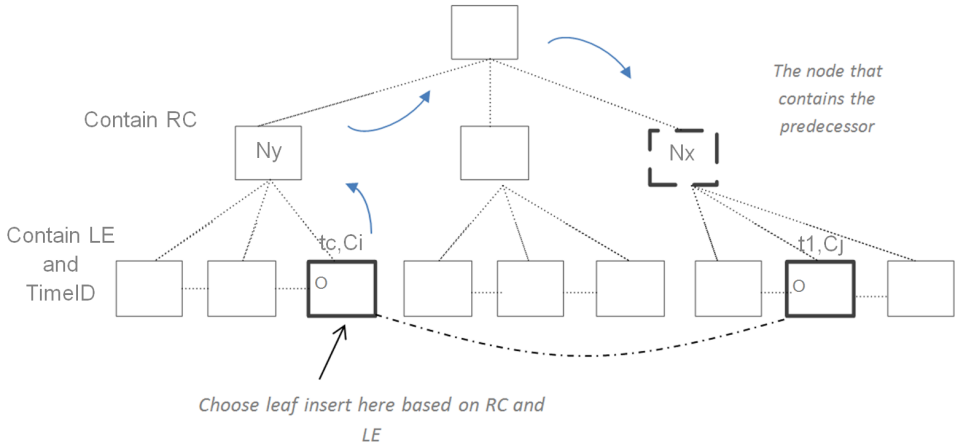


FIGURE 4.21: Moving Objects Timestamping tree (MOT-tree)

Using the sample data in Figure 4.22 at t_c, suppose that a five $MBRs$ are grouped into larger $MBRs$, where $1, 2, 3, 4, ..., 13$ are the moving objects and $M = 3$ and $m = 2$. Based on the MOT-tree, we show the RC and LE for each non-leaf nodes and leaf nodes. N_1 RC is (C_1,C_4) (where C_1 is the highest expand cell contained in N_1 and C_4 is the lowest) and N_2 RC is (C_7, C_{11}) based on the expansion algorithm. In the leaf node N_{1a} LE is (C_1), N_{1b} LE is (C_4), and N_{2a} LE is (C_7), N_{2b} LE is (C_{11}) and N_{2c} LE is (C_{10}). The MOT-tree is shown in Figure 4.23 (current time and current cells' locations).

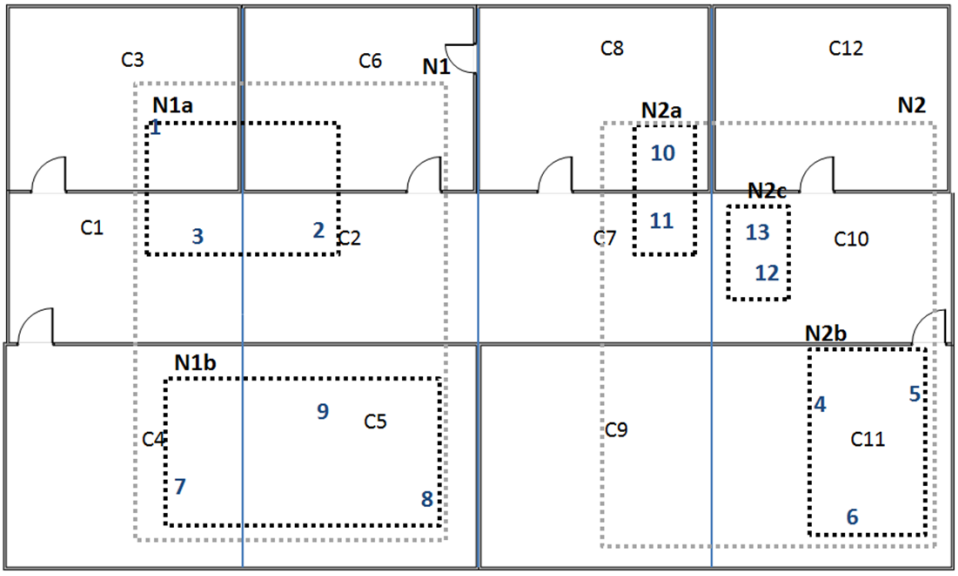


FIGURE 4.22: Illustrates the $MBRs$ grouping based on the connection (current time)

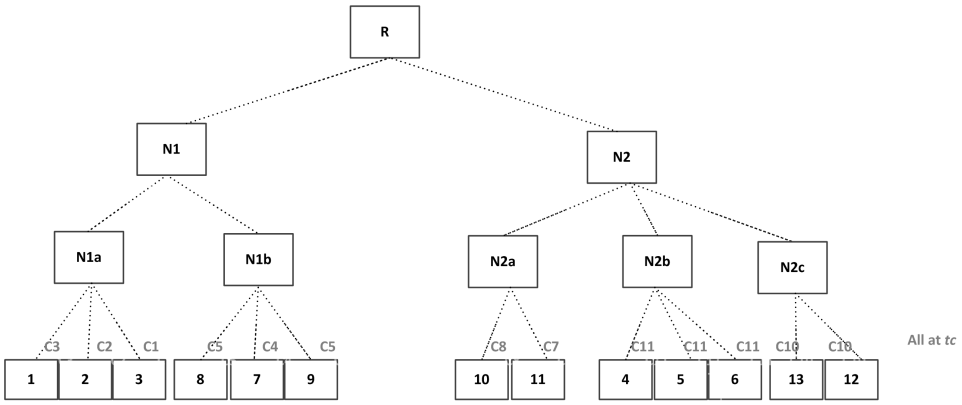


FIGURE 4.23: An example of MOT-tree at t_c

For further illustration, assuming that the following objects changed their location at t_2, where Object "1" checked out from C_3 and checked into C_1, Object "9" checked out from C_5 and checked into C_4, Object "13" checked out from C_{10} and checked into C_7, and the objects "2", "3", "4", "5", "6", "7","8", "10","11" and "12" remained in their cells. Here, the advantages of the MOT-tree become clear because we need to track only the updated objects, which reduces the updating cost.

The tree will be updated as follows: For "1", a new object will be inserted as $(1, C_1, t_c)$, the choose leaf algorithm will compare the new cell C_1 with RC and the LE in all nodes. The adjacency comparison algorithm finds that the suitable non-leaf node is N_1. Hence, the new "1" will be grouped in N_{1a}, which leads to a split of N_{1a}‘ and N_{1a}“. For object "9" a new object will be inserted as $(9, C_4, t_c)$, and grouped in N_1 and leaf node N_{1b}. Again, N_{1b} is overflown which leads to a split of N_{1b}‘ and N_{1b}“. For object "13" a new object will be inserted as $(13, C_7, t_c)$, grouped in N_2 and leaf node N_{2a}. N_{2a} is not overflown which leads to group the new "13" with N_{2a}.

Figure 4.24 illustrates the updating of the previous scenario (note that N_1 become overflown which causes the splitting of N_1). It can be noticed that the

remaining objects still have their timestamps, and when any of them moves to a new cell, the gap time will be recognized as the time spent in the previous cell. Note that after inserting the objects with their new timestamps, the FindPredecessor algorithm is used to return the node that contains the predecessor and initiates a link list between them.

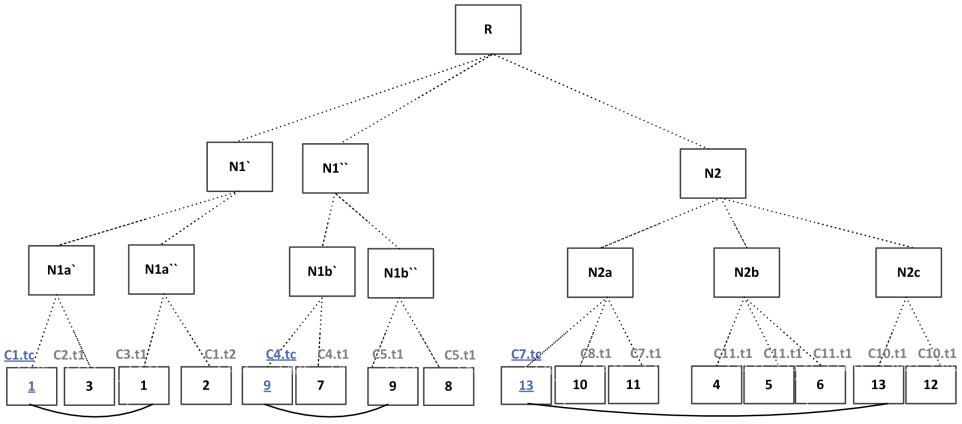


FIGURE 4.24: Updating the MOT-tree for the modified objects ("1","9", and "13")

4.3.3 Technique 3: Indoor Trajectories Deltas Index based on Connectivity Cellular Space (ITD-tree)

This technique is based on the idea of a delta store. Basically the main idea here is to use the trajectory delta as a temporary recorder of any updating of each cell. Therefore, the updating can be sent to the tree structure after a fixed time which reduces the update costs. The trajectory delta is defined as follows:

Definition 4.14. Considering the collection data of moving objects on the floor plane $(O_1, O_2, ..., O_n)$, and cells $\{C_i, C_2, ..., C_n\}$, a *Cellsdelta* (noted $\triangle$) records the updating of the objects from one cell to another. It consists of a set of basic operations (update, insert, delete).

Definition 4.15. Given a cell C_i which contains objects on the indoor floor, *CellDeltaForward* is defined as the *Cellsdelta* which records the changes made to objects in C_i at a certain time (Checked In).

Definition 4.16. Given a cell C_i which contains objects on the indoor floor, *CellDeltaBackward* is defined as the *Cellsdelta* which records the check-out made by objects in C_i at a certain time.

Definition 4.17. Considering the collection data of moving objects on the floor plane $(O_1, O_2, ..., O_n)$, and set of cells C, $C_i.\triangle.t_i$ represent as
$C_i.\triangle.t_i = \{O_1.\vartheta, O_2.\vartheta, ...\}$ *and* $\{O_4.\theta, O_5.\theta, ...\}$, where $Ci \in C$, t_i is the timestamp, ϑ indicates CheckIn and θ indicates CheckOut.

In this data structure, the index focuses on the deltas of the data in each cell. Technique 3 comprises the following steps. After the tree has been constructed, the updating of the moving objects will be monitored by the *cellsdelta* to determine the modification of the moving objects at each time. Similar to the previous technique, connectivity is the basis of the tree, where we check the RC (Range cells) at the non-leaf node or the LE (the largest expand point) at the leaf node (by the adjacency comparison algorithm) by choosing the MIN value cell (the nearest adjacent cell). However, the modification of each cell will be stored in the *cellsdelta* first, and will be updated on the tree after a fixed time. Note that the *cellsdelta* can be queries. Here, there is no need for the *FindPredecessor* algorithm since it is possible to determine the old locations of the data by the *CellDeltaForward* and *CellDeltaBackward*. The ITD-tree has three main properties:

1. Non-leaf nodes contain RC and $ChildPTR$ where RC is the range of cells (similar to Technique 1).

2. Leaf nodes contain LE, obj and PTR. LE is similar to the previous technique (based on the expansion algorithm). The objects that are contained in the MBR at that time are denoted as obj, and PTR is the pointer. Note that both the non-leaf node and leaf nodes contain at most O_n entries, which is the maximum capacity of the nodes.
3. *Cellsdelta* is the recorder of the node changes, which can be updated after several changes in nodes, in order to reduce the number of updating nodes. *Cellsdelta* as *CellDeltaForward* and *CellDeltaBackward* (based on Definition 4.15 and 4.16) (for example $CellDeltaForward\mathrm{C}_i = \{\triangle t_1, \triangle t_2,...,\triangle t_c\}, where\ \mathrm{t}_i$ indicates the time). The structure is illustrated in Figure 4.25.

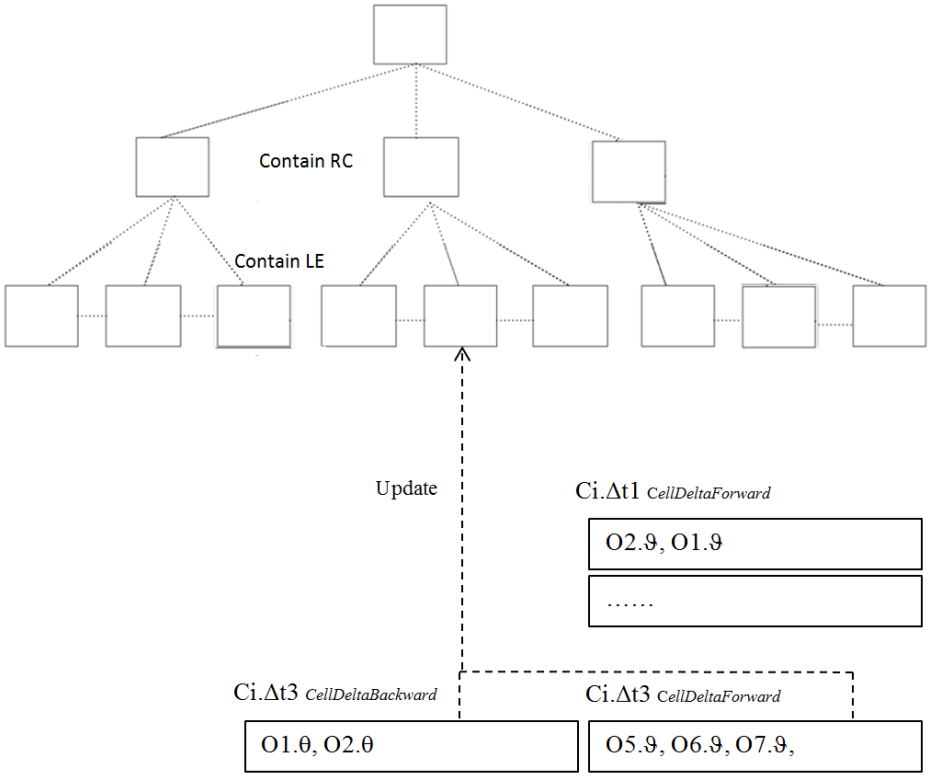


FIGURE 4.25: Indoor trajectories delta tree (ITD-tree)

4.4 Experimental Results and Performance Analysis

This section presents experimental results to evaluate the proposed index via the query performance of the indoor-tree index structure. The experiment has been carried out on an Intel Core i5-2400S processor 2.50GHz PC, with 4 GB of RAM running on 64-bit Windows 7 Professional. The maximum number of entries per node, M, was 60 and the minimum m 30. The data structure has been implemented in Java. The data set size ranges from 10 to 1000 moving objects on the indoor floor.

In this experiment, due to the lack of real data for indoor environments, a synthetic dataset of moving objects on an indoor space floor is used. We generate the location of the objects based on the number of cells. For example, in the 12-cells case, the moving objects to be covered by all the cells are generated, then the number of objects in each cell is increased. As mentioned earlier, the movement of the objects will be unspecific inside the cells (treated as static) and will be updated only when the object moves out of the cell and checks into a new cell. For the indoor environments, a several indoor spaces are used, starting from a 12-cells real-case floor environment, then extending the floor to 50, 100 and 150 cells. The cells expansion is based on realistic scenarios. It is clear that each case will have different expand cells, where the 12-cells case will have fewer expand cells than the others.

4.4.1 Indoor-tree Performance Evaluations

This section investigates the following: tree construction costs and search and insert performance. For the former, the execution time is measured for each test. For the tree construction test, this section measures the costs for the different number of moving objects and the indoor space overlapping or the connection complexity. For the query performance and maintaining operations test, this section measures the complexity and the efficiency for different intensities of moving objects, the influence of the indoor connection complexity and the expand points complexity. Note that the operations are performed 5-7 times and the average is calculated. The parameters used are summarized in Table 4.3.

TABLE 4.3: Parameters and Their Settings

Parameter	**Setting**
Node capacity	60
Number of moving objects	(10 to 1000)
Indoor space	12,50,100,150 cells
Expand cells	increase with the increasing of the indoor cells
Operations	Insert, find, update
Dataset	synthetic

4.4.1.1 Tree Construction

Tree construction costs are illustrated in Figures 4.26, 4.27. The moving objects range from 10 to 1000 on each floor; then, the structure tree is performed to group the moving objects together based on the indoor connectivity index. Figure 4.26 illustrates the increase of the number of moving objects for each indoor cell. As the number of moving objects increases, the construction cost increases accordingly. This behaviour is natural in an indexing tree where, with the increase in the

number of moving objects, the construction cost will increase, and the number of the nodes and splits will increase; moreover, the comparison and checking of the lookup table for the MIN values will be increased. Figure 4.27 illustrates the increase in the number of cells. It can be noticed that the tree construction cost increases slightly in some of the cases, which indicates that the indoor tree index strategy is effective for the indoor tree construction.

Figure 4.28 shows the effect of the cell connections' complexity. Basically, we tested the construction on different complexities of connections. Apart from the actual connection on the floor, the worst case connection is included which is a complete graph connection, and the best case is ascending connections [98]. Note that the increase of the complexity of the cells' connections does not impact on the tree construction costs. This behavior can be explained as follows: when we insert any objects into any cell, with the increase of the connections, it will be more likely to have an adjacent cell to be grouped with, which does not usually occur in the ordinary case. Consequently, the proposed index tree construction cost remains steady and stable.

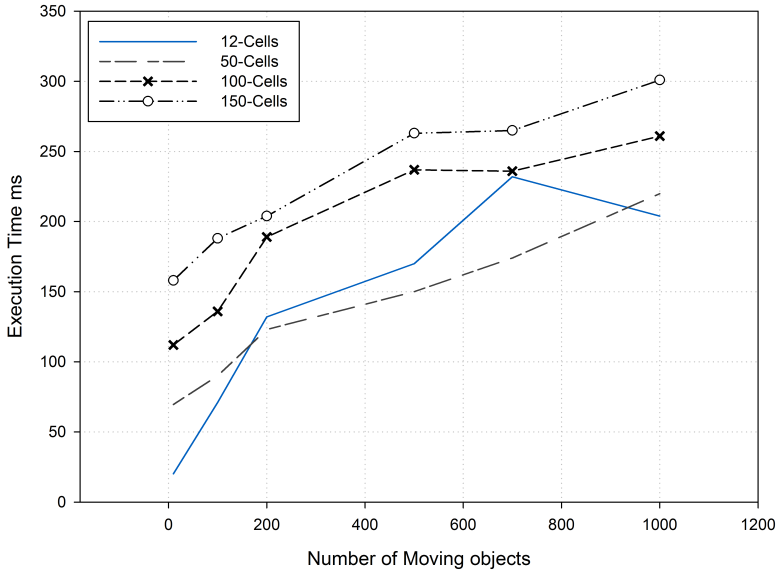


FIGURE 4.26: Effect of Objects Number

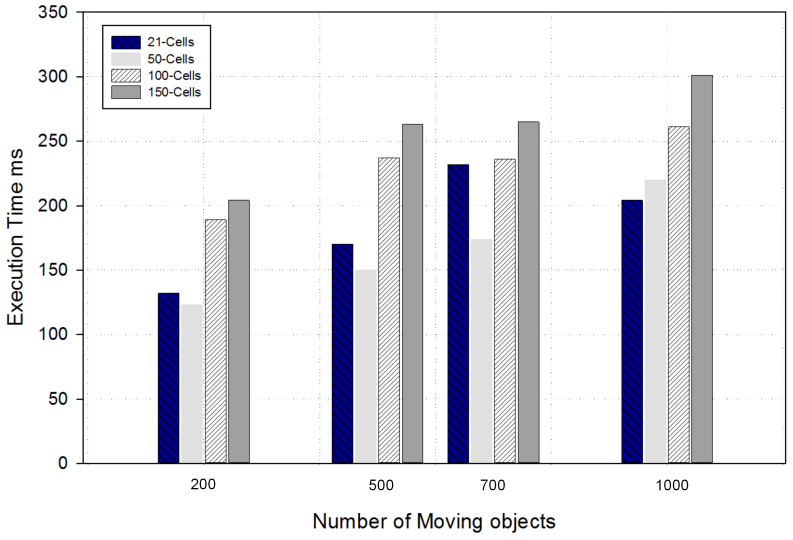


FIGURE 4.27: Effect of Cells Numbers

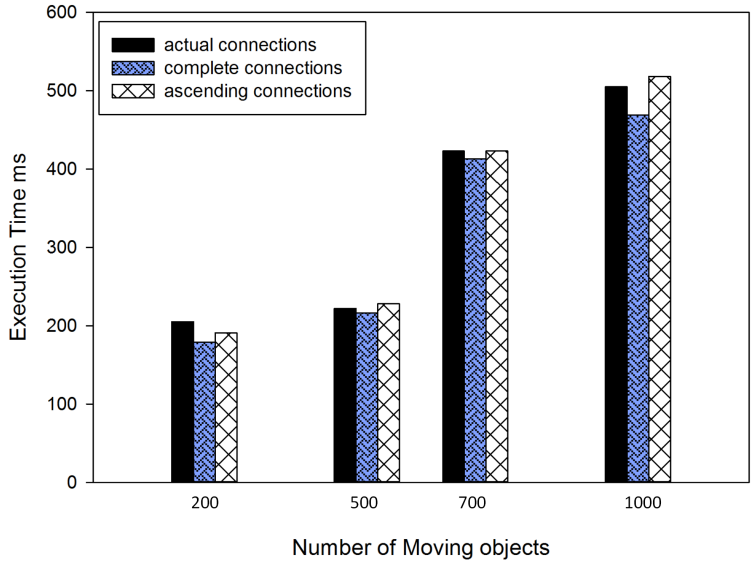


FIGURE 4.28: Illustrates the effect of connection complexity

4.4.1.2 Search and Insert Costs

This section studies the search and the insert costs. For the search, we test the typical descending search which is basically a search from the root to the leaf nodes searching for a particular object. For the insert, this section measures the proposed insert algorithm with its associated operations cost. Moreover, we consider the

effects of increasing the number of moving objects and the complexities of the connections and expand points. In this section, the test of the search and the insert is conducted after constructing the moving objects, where we search for particular objects in 10, 100, 200, 500, 700 and 1000 objects case. The insertion case is similar. In order to be able to report meaningful outcomes, we optimized the system's configuration and warmed up the system's caches by executing queries 5-7 times, and the stabilized runtime is used.

Effect of Number of Objects. A different number of cells are used in order to monitor the influence of the increase in the number of moving objects for different cases. First, there are the search query measurements, then the insertion. For the search (find) query, we can notice in Figure 4.29, that with the increase in the number of moving objects, the query cost increases slightly around 1.4ms. Moreover, we can notice that for the 12-cells case, when we perform a query at 1000 objects, the cost is almost similar to that for other number of objects cases. Note that the search query cost is usually less than the insertion cost, due to the related operations that can be produced from the insertion (such as split, compare,...,etc).

For the insertion, we insert an object into a particular cell several times, using 10 - 1000 moving objects, and then the average time is recorded. Figure 4.30 shows that in some cases, with the increase in the number of moving objects, the cost increases around 8ms. Figure 4.30, shows that the insertion cost in the 12-cells case increases slightly with the increase in the number of moving objects, due to the increase of the related operations and nodes. However, we can also see that for the 200-object case in 12 cells, the performance is close to that of the 200-object for 50 cells.

Furthermore, Figure 4.31 illustrates the updating of the indoor-tree of 50 cells at 1000 objects. We update 100 objects first, then 500 objects, then all moving objects. We note that our indoor-tree still performs efficiently when updating high number of moving objects.

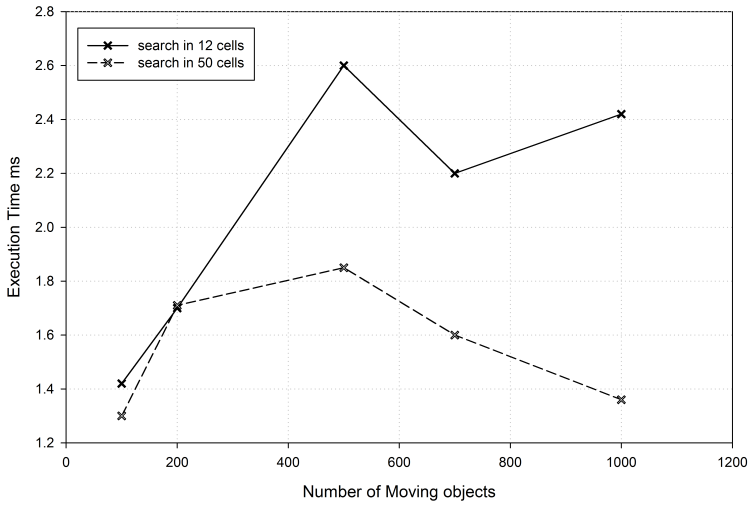


FIGURE 4.29: Effect of number of objects (search performance)

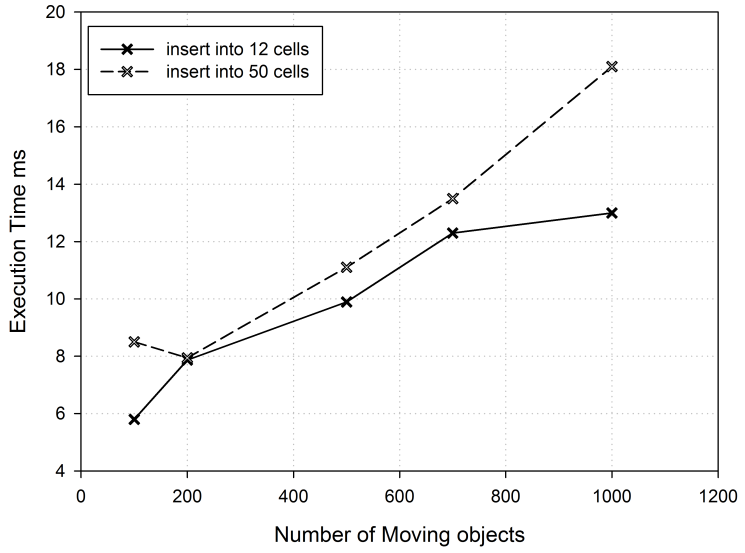


FIGURE 4.30: Effect of number of objects (insert performance)

Effect of Connections Number. Here, the complexity based on the cells' connections is measured, which means that we increase the cells' neighbours in order to monitor the effect. Moreover, it illustrates the effect of the distributions

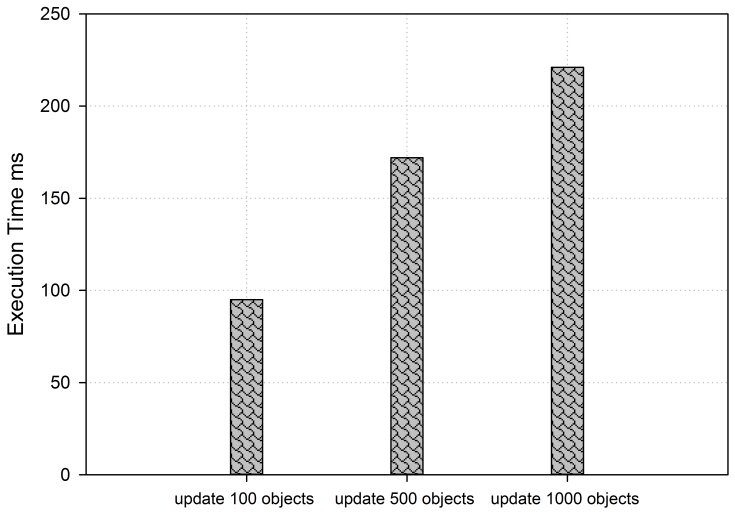


FIGURE 4.31: Illustrates the effect of updating different number of moving objects

of the cells in different densities. In this section, after examining the original connection case (it is called *actual connection*), also the worst case and the best case theoretically of the cells connection are tested. The theoretical worst connection case is a complete graph where each cell is connected with all the other cells. The best theoretical case is the ascending connection where each cell is connected with one other cell.

For the find query, the number of connections will not affect the search (find) query, because the indoor-tree is already constructed and the connection number will not be used in the search queries. Therefore, we tested only the insert, where the connection is essentially important. This begins by inserting an object into a particular cell several times, and then, the average time is recorded (this will be done in all connections cases). As shown in Figure 4.32 that with the increase in the number of moving objects, the insert cost in the worst case and the best case scenario is less than the actual connection. This can be explained as follows: When a floor has more connections between its cells, this facilitates the insertion in our indoor-tree. The reason is that when an object is inserted into the floor,

the choose leaf algorithm will check the inserted cell and compare it with the RC and LE of other nodes, where it is more likely to have many options, when it is grouped with any of its many neighbours. Moreover, the splitting can also be facilitated in environments with more connections; where any node has to be split (because of overflow or underflow), the objects will be grouped with the connected cells' entries. Figure 4.33 illustrates the updating of a high number of moving objects. Here we can see that the indoor-tree still performs well in the worst case connection.

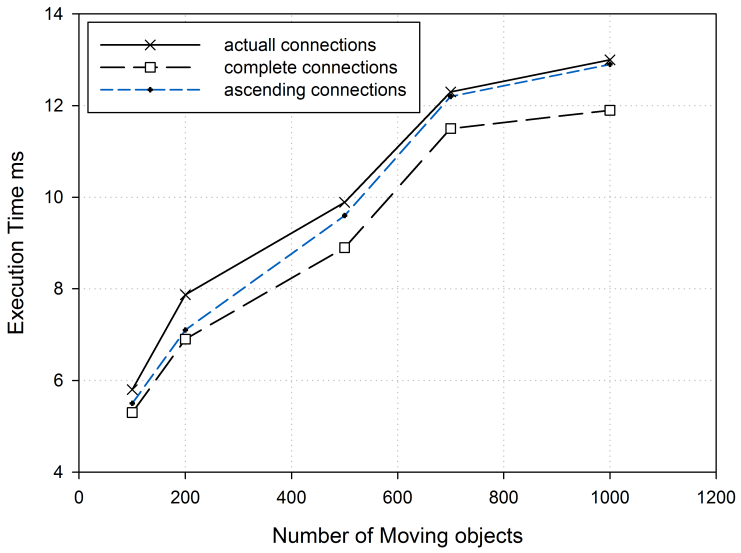


FIGURE 4.32: The effect of connection complexity on inserting

Effect of Number of Expand Points. This section measures the expand points' complexity. In this section, after examining the original expand points case (called *actual expansion*), the *high expansion* is tested where all of the cells are considered as expand points.

The effect of the expansion complexity is tested for the 50-cells. Figure 4.34 shows that with the updating of high number moving objects, the indoor-tree still performs very well in the high expansion case.

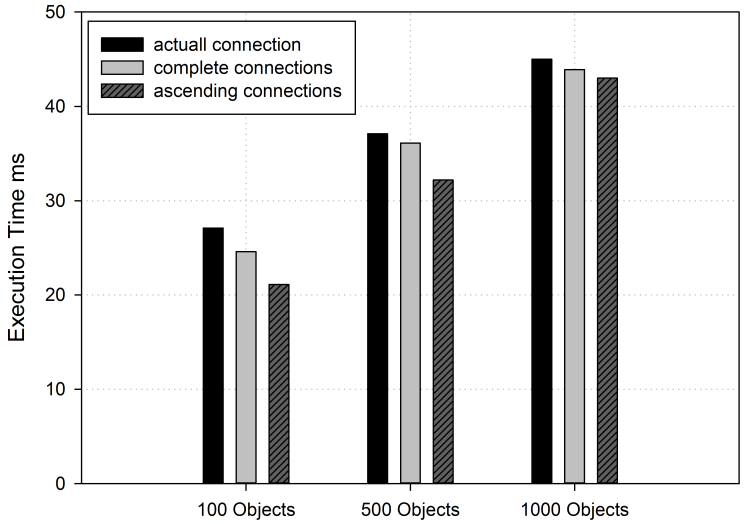


FIGURE 4.33: The effect of connection complexity on updating different number of objects

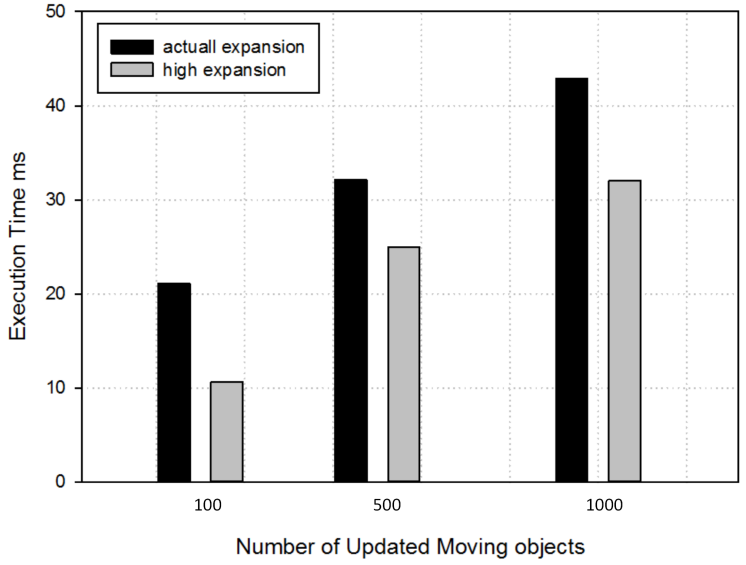


FIGURE 4.34: The effect of expansion complexity on updating different number of objects

Summary. The evaluation experiments evaluated the existing parameters in our indoor-tree in order to test the proposed indoor index. It examined the unique parameters in our data structure in order to determine the effect on the data structure (e.g. moving objects number, connection complexity, expansion complexity and cells number). The results can be summarized as follows: Three

important observations follow from the conducted experiments. First, increasing the number of moving objects and increasing the complexity of cell connections (high adjacency) have no explicit effect on the tree constructions. Second, the query costs and the insertion costs perform efficiently with different cases having different numbers of moving objects, cell connections and expand points. Third, the indoor-tree can successfully obtain a reliable and a robust tree index based on the novel idea of indoor connections and adjacency.

4.4.2 Temporal Techniques Performance Evaluations

This section examines our experimental results to evaluate the proposed data structures. It compared the three structures with each other (TI-tree, MOT-tree and ITD-tree). The experiment was carried out on same features in 4.4.1.

For the search, a typical descending search to locate moving objects at a certain time was conducted. For the update, we compared the proposed structures to determine which one incurred less update costs. Moreover, this section considers the effects of increasing the number of moving objects and the complexity of the floor connections. Note that the search and the update tests were conducted after the moving objects had been constructed. Note that the search and the update of particular objects was done with the number of objects ranging from 100, 300, 500, 700 to 900.

For the search, the search performance of the TI-tree, MOT-tree and ITD-tree are tested for different numbers of moving objects and cell connections. Figure

4.35 shows that with the increase in the number of the moving objects, the query cost of the MOT-tree is less than for the TI-tree and ITD-tree. For example, in the 900-objects case, the MOT-tree reduced the search cost by around 69% lower than the TI-tree. Moreover, the TI-tree had the highest search cost which it can be explained as follows: the TI-tree is based on the idea of non-leaf node timestamping, which requires searching in more nodes. On the other hand, the MOT-tree and the ITD-tree have fewer nodes, which deliver efficient results in the search performance. Moreover, Figure 4.36 shows the effect of the increase in the number of indoor connections, when the search query was conducted in the worst case connections of 12 cells. The query cost of the MOT-tree and the ITD-tree still performed faster than TI-tree. In some cases, the search using the proposed techniques was 71% faster than the TI-tree.

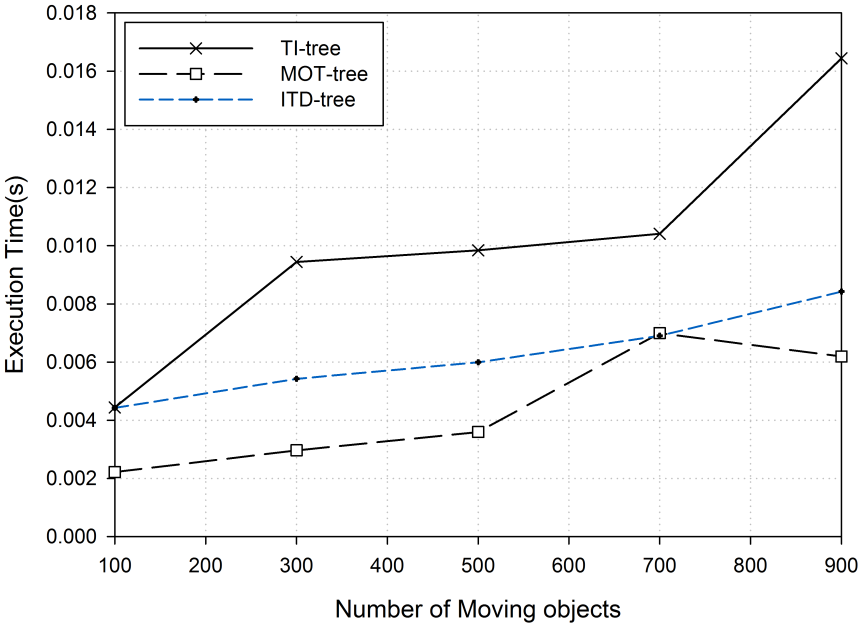


FIGURE 4.35: The effect of objects number on search performance

For the updating, first the experiments updated an object in the TI-tree, MOT-tree and ITD-tree for different ranging from 100 - 900 objects. As shown in Figure 4.37, in all objects number cases, the MOT-tree and ITD-tree performed

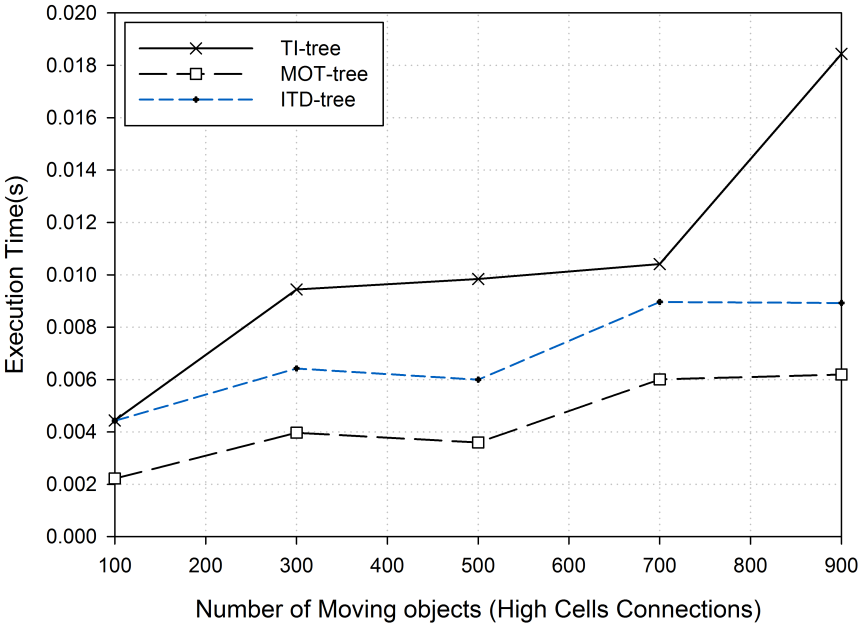


FIGURE 4.36: The effect of cell connections on search performance

with less cost than the TI-tree. Furthermore, as shown in Figure 4.38, the updating of the 12-cells case (the worst case number of connections), the MOT-tree and the ITD-tree are still cost-efficient despite the increase in the cell connections. In the TI-tree, each non-leaf node will be updated, regardless of whether or not the moving objects changed their location, which increases the updates costs. On the other hand, the MOT-tree updated only the objects that change their location which obviously requires less updating compared to the TI-tree. The ITD-tree also has lower update costs since the delta store plays an important role in managing the update operations.

Figure 4.39 illustrates the updating of a high number of objects at the TI-tree, MOT-tree and ITD-tree with 12 cells. Here, a large number of the moving objects are updated at the same time in 100, 500 and 900 case. Clearly, the update costs of the ITD-tree and MOT-tree were less than those of the TI-tree. ITD-tree has the lowest cost because the delta store delays the ITD-tree updating, which reduces the update costs.

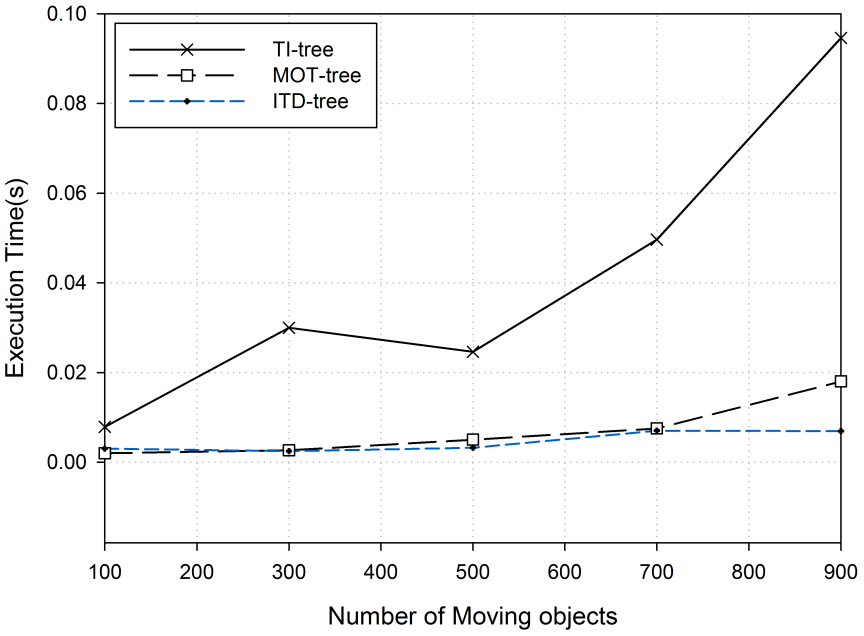


FIGURE 4.37: The effect of number of objects on update performance

The results in this section can be summarized as follows: Three important observations followed from the experiments. First, increasing the number of moving objects and increasing the complexity of cell connections (high adjacency) have no particular effect on the MOT-tree and ITD-tree. Second, the update costs of MOT-tree and ITD-tree are just as efficient regardless of different cases having different numbers of moving objects and cell connections. Third, the MOT-tree and ITD-tree can successfully produce a reliable and robust tree index for indoor spatial and temporal data.

4.5 Chapter Summary

This chapter addresses the challenge of building an index data structure that is appropriate for indoor spaces. The measurement of indoor space is different from

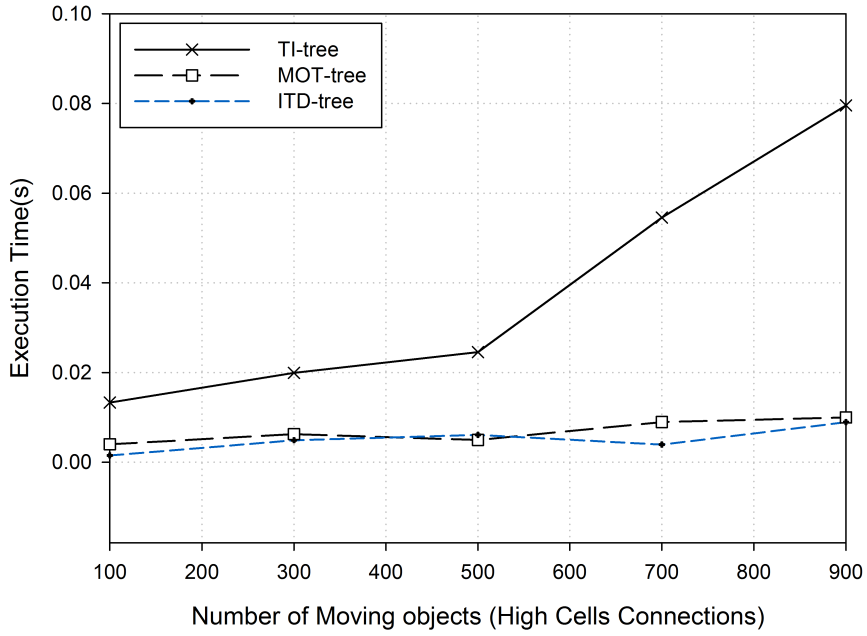


FIGURE 4.38: The effect of cell connections on update performance

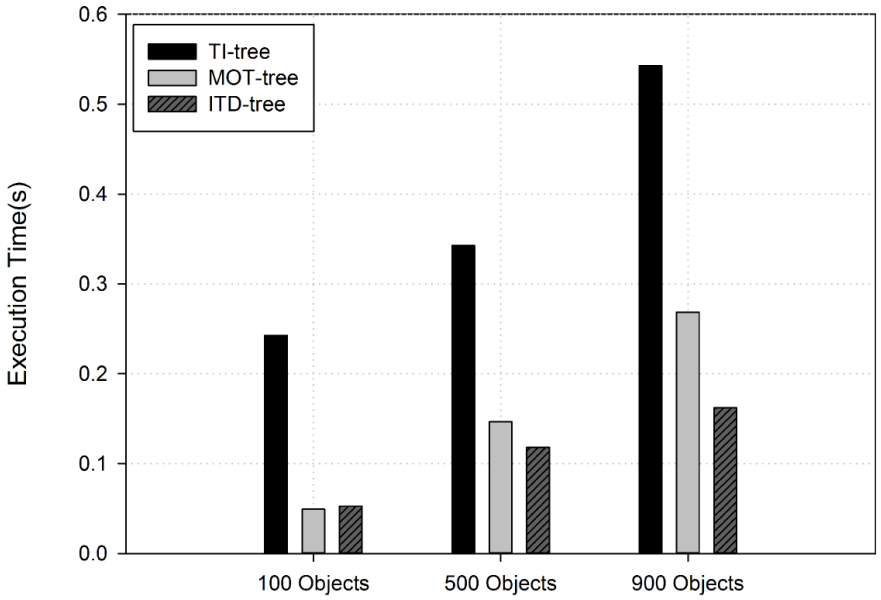


FIGURE 4.39: The effect of updating a large number of moving objects

that of outdoor space that is based on Euclidean space or a spatial network. Indoor space is related to the notion of cellular space that contains different entities such as rooms, doors and hallways that enable or constrain movement. The proposed index takes into account the entities that enable or constrain the movement in an

indoor environment (doors and hallways). Our tree structure is based on adjacency and cell connections, which allows us to answer spatial queries more efficiently. In addition, our data structure is based also on the indoor expansion, which is the way that the indoor floor is divided from its main entrance to the last cell. This enables us to answer future queries more efficiently. The expansion algorithm enables us to use the expand points and record the two expand points with non-leaf nodes as range, and the largest expand point is recorded in the leaf nodes (for comparison and to choose nodes' goals). The temporal part of the index used three different techniques. The first technique is based on the non-leaf nodes timestamping. The second one tracked only the updated objects and delayed the others. The last technique is based on the delta store, where we used the delta as a temporary recorder of the object modifications on the floor. Extensive performance studies were conducted and results indicate that the index structures are both robust and efficient for the targeted queries.

CHAPTER 5

Density and Adjacency-Based Indexing for Moving Objects in Multi-Floor Indoor Spaces

Chapter Plan:

Publications and Submissions:

- Sultan Alamri, David Taniar, Haidar Al-Khalidi and Kinh Nguyen: "Density and Adjacency Based Indexing for Moving Objects in Multi-Floor Indoor Spaces". *IEEE Transactions on Knowledge and Data Engineering* (2014) (Under Review).

In indoor space, moving objects are located inside the rooms or cells contain of these entities. However, these cells should not be treated equally when indexing the moving objects. Some cells such as classrooms usually contain a high density of moving objects, whereas teachers' offices usually contain a low density of moving objects. Thus, the data structure of the moving objects in indoor spaces should distinguish between the cells that have a high density and those that have a low density of moving objects [31, 32]. Therefore, this chapter presents a density-based index for moving objects in indoor spaces. The index structure distinguishes between the dense cells and the low density cells, thereby resulting in a faster performance of the data structure.

In addition, data density plays an essential role in the data structure performance. Therefore, by considering the dense cells in an indoor environment in data structures, this can increase the tree's performance. For example, consider that dense cell at a certain time is expected to reach 150 moving objects, where the number of objects will increase by 3 objects every 10 seconds. Here, if the maximum

per node is $M = 3$, and the minimum $m = 2$, the data structure is maintained 150 times, through 450 seconds, and around 66 splittings are performed. As is clear, here for each update operation, the location server will update the locations of the moving objects which maintain the spatial index. Therefore, this works focuses on minimizing the above-mentioned update cost for each update which is an essential issue in various indoor applications. In this work, when considering the dense cells in the index structure, the cost of maintaining the index will be efficiently decreased as a result of the decrease in the number of splitting operations; hence, the lower the number of nodes the lower is the access cost for any of the insert or delete queries. This chapter provides a new technique which considers the data density in the tree structure in order to produce high performance and support the adjacency queries and density-based queries more efficiently.

This chapter proposes a new index structure, density and adjacency-based, for moving objects in indoor spaces. This work concentrates on indexing the moving objects based on the notion of cellular space. The indoor-density-tree or Indoor^d-tree takes into account the indoor walls and partitions that control movement in an indoor environment and index the moving objects based on the connectivity/adjacency of the indoor cells. Moreover, since the moving object density in indoor spaces greatly affects the efficiency of the index structure, we distinguish between cells of different densities in our index structure. Moreover, indoors usually consist of multi-levels which have a vertical transit such as stairs. Hence, this chapter considers the challenges of an object's vertical movement (e.g. via stairs or elevator) within the multi-floor indoor space. The contributions in this chapter can be summarized as follows:

- Indoor^d-tree is a density-based data structure which takes into account the

high-density cells separately from the low density-cells to avoid any performance issues.

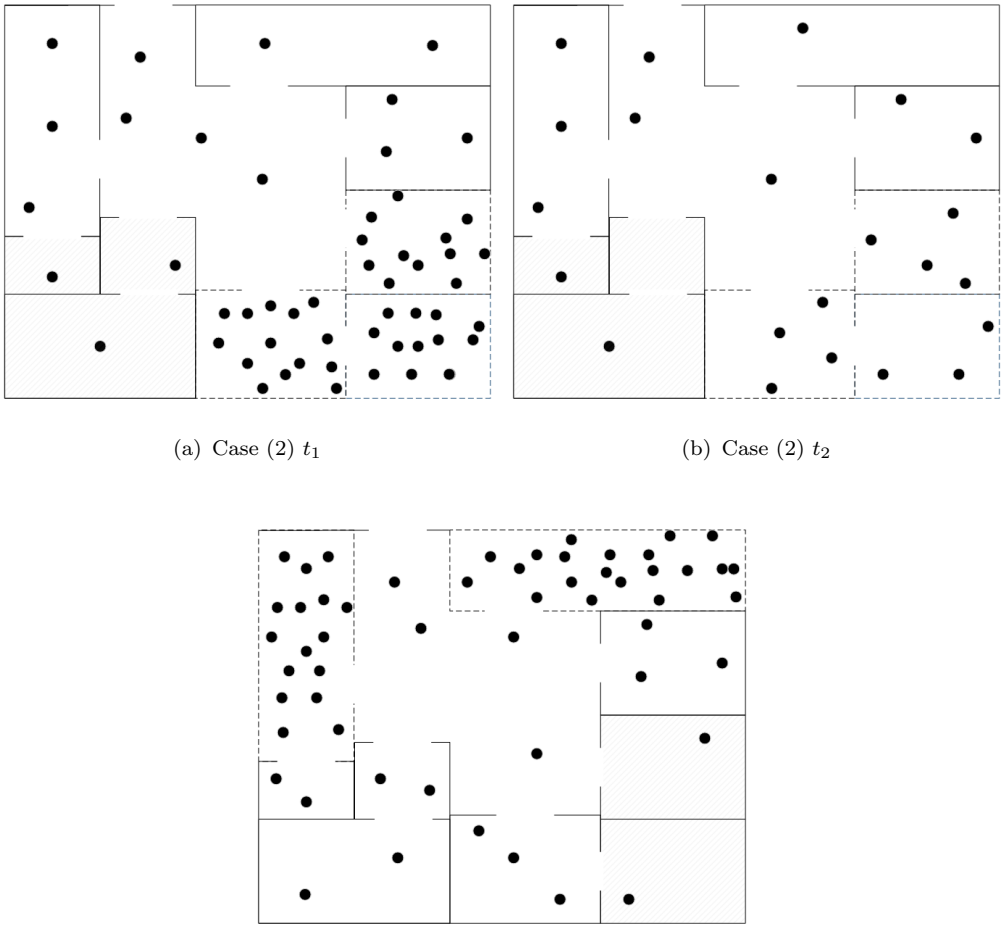
- It considers the vertical transit issues of multi-level indoors (e.g. stairs and elevator) in order to index all moving objects at different indoor levels.
- An analytical cost study is provided of the search cost of the low-density tree and high-density tree in addition to the storage costs.
- Experiments are conducted in a simulated environment in order to evaluate the proposed index structure by studying the construction, insert and query performances of the Indoor^d-tree.

5.1 Impact of Data Density in Indoor spaces

Basically, indoor moving objects are spatial objects whose locations change over time. This section discusses three cases which illustrate the different impacts of the density in indoor spaces.

Density in Space: In general, the moving objects' densities vary in different indoor spaces. Buildings such as business centers and shopping centers usually have higher object densities than do other indoor spaces. These situations are illustrated in Figure 5.1 (a). At time t_1, we have three high-density cells (dashed cells), and the other cells with relatively lower moving object densities (solid cells).

Density with Time: In an indoor environment, the number of moving objects changes with time. For example, in train stations we expect a high density of moving objects in the rush hours, causing heavy overcrowding on the stations.



(a) Case (2) t_1

(b) Case (2) t_2

(c) Case (3) t_3

FIGURE 5.1: Data densities in indoor spaces

In contrast, the density of the moving objects is relatively light at night time. Referring to our example (Figure 5.1 (a)), from time t_1 (Figure 5.1 (a)) to t_2 (Figure 5.1 (b)), busy cells are still hotspots; however, a significant decrease occurs in all cells.

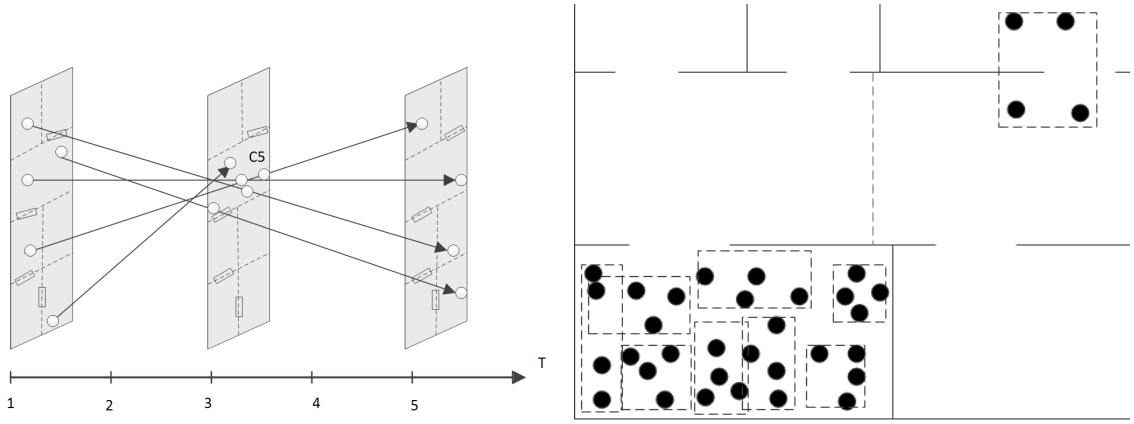
Density in Space with Time: This type focuses on the density of moving objects and the busy cells change with time. In the previous example, from t_2 (Figure 5.1 (b)) to time t_3 (Figure 5.1 (c)), we notice that with the increase in the total number of moving objects, the busy cells move as well. Sparse cells may become busy cells while busy cells may become sparse due to factors such as rush

hour activities. As a practical scenario, between 1pm to 2pm, students are in the campus food court which makes it a high density area; whereas, classrooms become low density cells. During classes hours, most students move around the classrooms which makes these high density areas.

In summary, moving object density in indoor spaces greatly influences the efficiency of an index. Considering the three scenarios mentioned above, an index suffers from lower performance when the construction of the moving objects is a uniform grouping, with no consideration given to the density. The reasons for this are:

- If we consider the grouping of the dense cells to be split as $M + 1$ entries of a node which overflows into two groups, then the number of splitting operations will be highly increased which definitely will affect the tree's performance and the tree construction. Moreover, due to the increase in the splitting operations, the number of the nodes will be increased which increases the access cost, and the maintenance operations (e.g. insert, delete and update) will be strongly influenced (see Figure 5.2 (b)) [31, 32].
- There are many advantages of taking into account the dense cells separately in the spatial index structure. First, the density-based queries will be supported easily if the dense cells are considered in the data structure (see Figure 5.2 (a)). For example, "return the dense cells that is located in the second floor". Furthermore, an aggregation query about a dense cell will also be served efficiently such as "return the number of the moving objects at cell C_2".

As a result, an efficient indoor moving object index must: (1) distinguish between cells of different densities, and (2) consider the density and the busy cells that changes over time.



(a) Indoor density-based query (b) High splitting may affect the performance

FIGURE 5.2: Objects' densities in indoor spaces

5.2 The Indoor^d-tree

5.2.1 Mapping domain concepts to modeling concepts

We need to be clear about the intended meaning of "indoor space". Essentially, an *indoor space* can be perceived as consisting of one or more buildings. An indoor space can contain things (or elements) such as *rooms, doors, corridors, floors, stairs, elevators* and *pathways* between buildings. Given an indoor space with its associated elements, our first step is to map it into a logical model. The logical model is an *undirected graph* with *cells (nodes)* and *edges (connections)*.

The mappings from domain concepts to modeling concepts are summarized in Table 5.1. A room is mapped to a cell, and a door to an edge connecting two

cells. A corridor, or other kind of pathway, also known as a circulation area, is typically mapped to several cells with several edges. If a corridor is not too long, we can model it as one cell. Otherwise, we partition it into several cells for the logical model. This is done manually, based on the judgment of the domain expert or the modeler. Note that the stairway is treated similarly where it can be mapped as single cell or multiple cells as needed, and the elevator is mapped to a single cell. Floors and buildings are not modeled explicitly. However, if we wish, they can be included as properties of cells (formally as a function from cells to floors, for example) or by a separate supporting model (for example, a graph where the nodes represent the floors).

domain concept	modeling concepts
room	a cell
door	an edge
corridor	one or more cells with one or more edges
stair	one or more cells with one or more edges
elevator	one cell with several edges
pathway	one or more cells with several edges

TABLE 5.1: Mapping domain concepts to modeling concepts

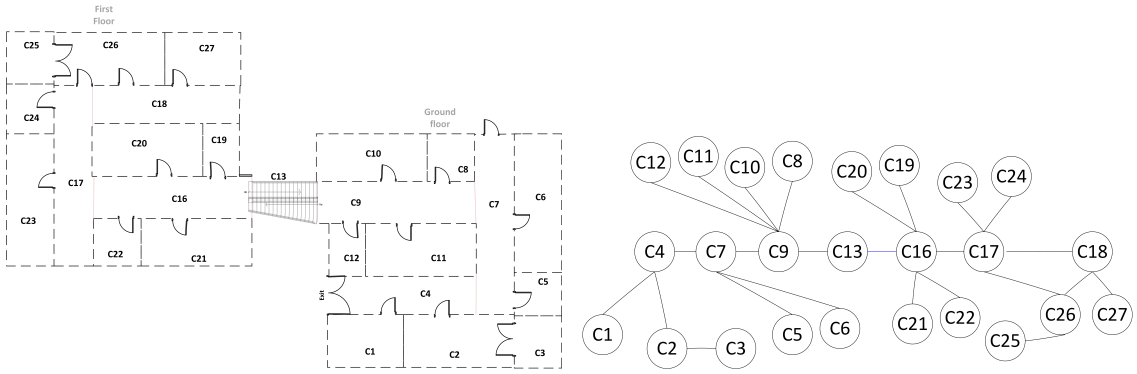
Once the mapping is done, the indoor space is represented as an undirected graph of cells and edges. Formally, we have:

Definition 5.1: An indoor space is a connected undirected graph $(Cells, Edges)$, where $Cells = \{C_1, C_2, \ldots, C_n\}$ is a set of cells, and $Edges = \{E_1, E_2, \ldots, E_m\}$ is set of edges, each of which is a set of two cells, that is, each edge connects two distinct cells.

The graph representing an indoor space will be called the *connectivity graph*[1].

[1]The condition that the connectivity graph is connected does not necessarily limit the range of applications. If we have 2 disconnected buildings, we can add a pseudo path between them.

Example: Figure 5.3 (a) shows the layout of an indoor space. This layout is mapped to the connectivity graph shown in Figure 5.3 (b).



(a) An example of multi-floor indoor cells

(b) The connectivity graph based on (a)

FIGURE 5.3: Indoor Connectivity Graph

5.2.2 Low-density tree

As mentioned previously, the indoor environments are based on cellular notations and the exact location of objects in indoor environments is not important; therefore, the moving objects will be grouped in the adjacency tree based on the connectivity between the cells. The basic idea is to delay the update of the moving objects' locations until they reach a new cell. Thus, the exact location of the moving objects in the indoor space is not tracked in the system, since the cell is known. Then, the moving objects inside the same cell are grouped together as the basic step.

The input of the index tree here will be the connectivity tree that resulted from the graph that represents the indoor space. The connectivity tree is constructed as explained in Chapter 4, Section 4.2.1 (Indoor Filling Space Representation). Figure 5.4 shows the connectivity tree (or the expansion algorithm).

Note that in the case of overflowing MBR, splitting will be performed to group it with one of the objects in adjacent cells based on the adjacency priority algorithm. In this algorithm, when inserting a new object in a certain cell, we check the RC (range cells) at the non-leaf node and the LE (the largest expand cell) at the leaf node, and choose the nearest non-leaf node and nearest leaf node that has the MIN RC and LE (see definition 5.2) [30]. Therefore, the clustering of the moving objects will be based on the connectivity between the cells. Adjacency clustering is illustrated in Figure 5.5.

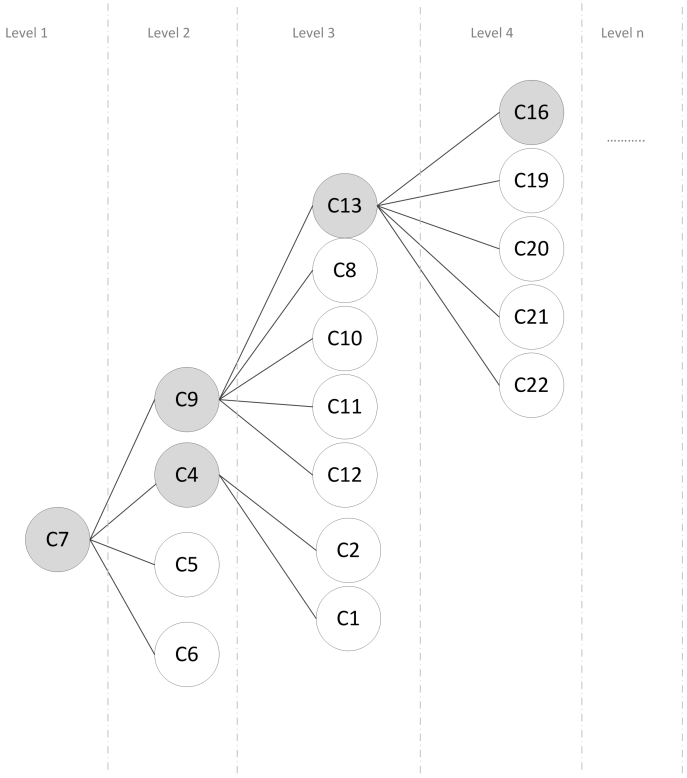


FIGURE 5.4: Connectivity tree of the graph in Figure 5.3

Definition 5.2: Given a set of moving objects $P = \{O_1, O_2, ..., O_n\}$, $CellDist(O_l, O_k)$ is the number of hops of the cells, and calculated as:

$CellDist(O_l,O_k) = |\{C_1, C_2, ..., C_n\} - 1|$, where $O_l \xrightarrow{in} C_1$ and $O_k \xrightarrow{in} C_n$. $\forall$ O_l, O_k $\in$ P.

Algorithm 7 Adjacency Priority Algorithm

```
/* check the inserted to Ci with set of C. */
/* Adjacency Priority Algorithm will return the cell that has the MIN value */
R = initial value
N non leaf node or leaf node
for Ci adjacent cells adc.i do
    if adc.i < R then
        R = adc.i
    end if
end for
Return R // nearest connected cell
Check RC and LE
R ∈ (N_i) and (LN_i)
Choose the (N_i,ptr)
Choose the (LN_i,ptr)
if N_i.R = N_j.R then // similar for the leaf nodes LN_i
    if RC.N_i adjacents > RC.N_j adjacents then
        Choose the (N_i)
    else
        Choose the (N_j)
    end if
end if
```

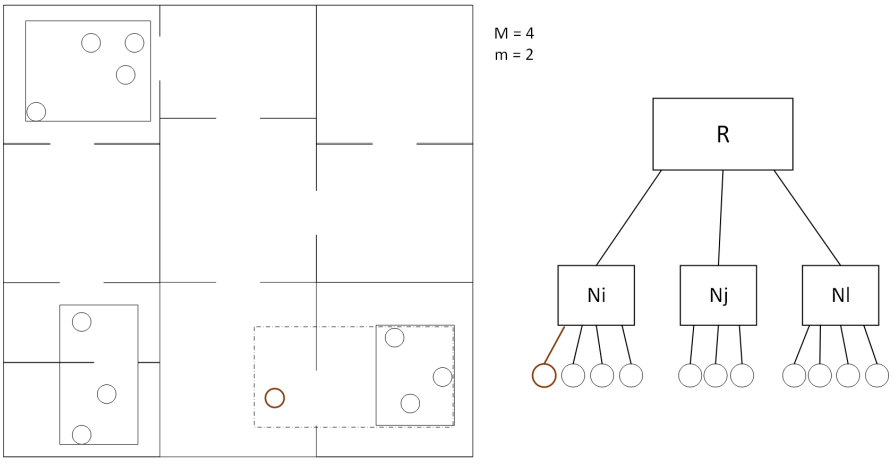


FIGURE 5.5: Adjacency clustering (Low-density tree)

5.2.3 Indoor Density Grouping

Considering the distribution of the moving object on the indoor floor, the density of the moving objects in each cell varies. Therefore, our Indoor^d-tree cluster the

moving objects considering the dense cell.

Definition 5.3: The density of a cell C_i at a time t_i is defined as follows: Density(C_i,t_i) = $\ell N / Area\mathrm{C}_i$, where ℓN is the number of moving objects in C_i at t_i and, $Area\mathrm{C}_i$ is area of the cell.

Definition 5.4: C_i is considered as a dense cell at a time t_i If: Density(C_i,t_i) > threshold ∂.

Definition 5.5: (Density Query): Given a set of P moving objects in indoor multi-floor space, a horizon H and thresholds $\partial 1$, find cells $C = C_1$,..., C_1 and time $T = t_1$,...t_n where, $t_i \subset [\mathrm{T}_{now}, T_{now} + H]$ and, Density(C_i, t_i) > $\partial 1$ $i \in$ [1, n].

The basic idea of our data structure is to use the adjacency method and group the moving objects based on their density. The cells that have high density will be grouped in the high-density tree, which includes only dense cells. For example, for a set of cells $C = C_1$, C_2,..., C_n, If C_2 and C_3 at $t_2 > \partial$, which means density(C_2,t_2) > ∂ and density(C_3,t_2) > ∂. Then, C_2 and C_3 at t_2 are included in the high density-tree and excluded from the low-density tree. The high-density tree is different from the low-density tree (as will be explained next) where the objects are inserted one by one, or a bulk loading operation is performed. On the other hand, with the high-density tree, instead of building the index using the objects, we build the index using dense cells. These cells can be determined from an analysis of the indoor floor (definitions 5.3, 5.4), or we can determine it from the past behavior of moving objects. The fact that the moving object settles for a while inside the indoor cell prompted us to build the high-density tree for

the dense cells. These busy, dense cells represent reasonable bounds within which objects remain for long periods of time. The basic idea is to treat the dense cells from the preceding step as MBR one level above the leaf nodes where the moving objects are stored.

This high-density cell tree, as shown in Figure 5.6, has the following features:

- The level above the leaf node stores dense cells. Note that at this level, there will be no overlapping issues between the nodes.
- The leaf nodes of the tree store the moving objects within the dense cells. At this level, we use the idea of no maximum/minimum objects per node restriction. Therefore, all moving objects inside a dense cell are stored below that cell's leaf level. Note that this technique facilities the insertion and makes it an easy procedure. Whereas, in the traditional spatial trees, the object insertion requires a large amount of computation (such as least enlargement nodes computation in R-trees and its successors). We will link this tree with the other using the adjacency method as explained next.

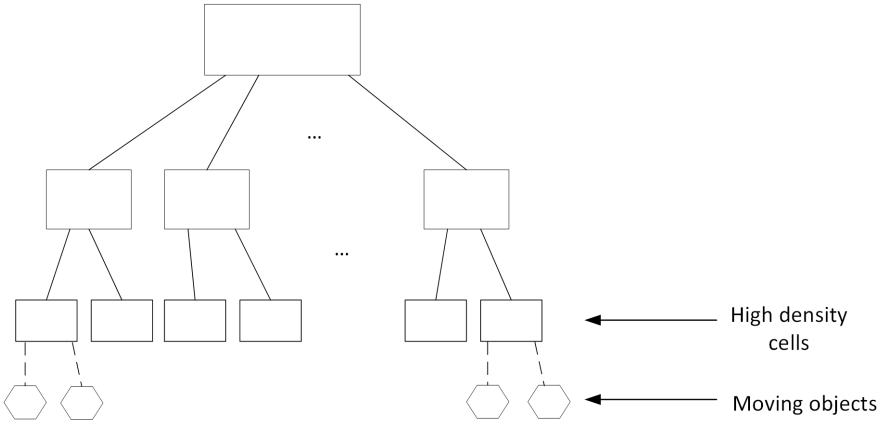


FIGURE 5.6: High density-tree

The second structure will focus on grouping the moving objects themselves based on their adjacency clustering which is mentioned in Section 5.2.2. This

structure (*low-density tree*) focuses on grouping the moving objects in low density cells C_1, C_2,..., C_n, where density(C_i,t_x) $\leqslant \partial$, $\forall\ t_x \subset$ [T$_n ow$, $T_n ow + H$] and $i \in$ [1, n]. Therefore, the moving objects inside the same cell will be grouped together as the basic step. Moreover, the splitting will be performed in order to group an object with its adjacent cells based on the *adjacency priority algorithm*. Note that the maximum/minimum objects per node restriction is applied here.

5.2.4 Combining the High-density tree with the Low-density tree

Since our objective is to maintain the moving objects in indoor spaces based on their connectivity to each other (the adjacency between the cells), we build a link list for each high-density cell with pointers to its adjacent MBR in the low-density tree which makes the adjacency still applicable in the combinations. The adjacency between the high-density tree is considered first, followed by the adjacency between the high-density tree and its adjacency in the low density tree. Because some of the moving objects (e.g. O_i, O_j) might move from the dense cell to a lower-density cell, the update from the high-density tree to the low density tree uses the link list which facilitates the process and ensures that the objects (O_i, O_j) are still grouped based on the connectivity between the cells. An example of combining the high-density tree with the low-density tree is shown in Figure 5.7 corresponding to the floor shown in Figure 5.1 (a). Assuming that $M = 4$ and $m = 2$, we can see, the dense level at the high density tree is linked with the non-leaf node level at the low density level. In our example, N_1 and N_3 are linked with the nearest connected node at the low adjacency tree which is N_b. Note that high-and low-density trees are not overlapping spaces.

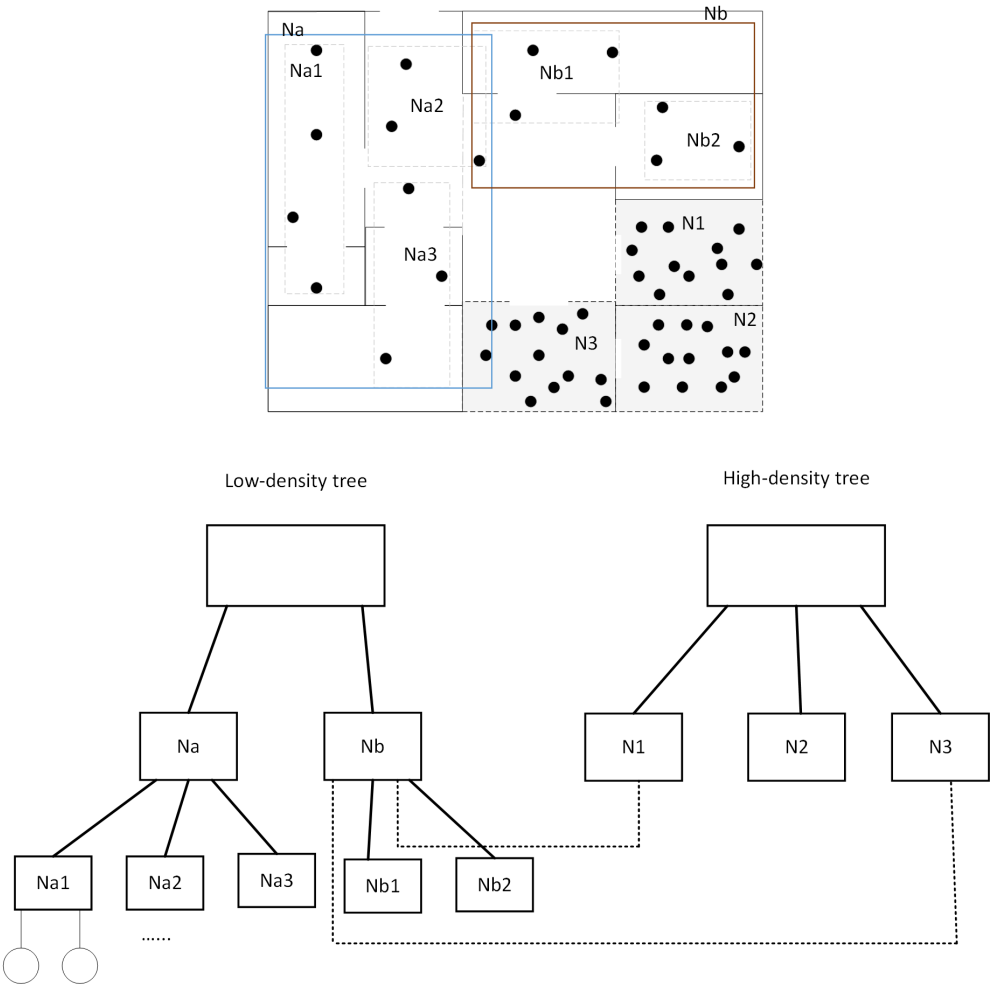


FIGURE 5.7: Combine the high-density tree with the low-density tree

5.2.5 Updating the High-density tree with the Low-density tree

The update of the combined trees could be one of the following cases:

(a) The moving object O_i is currently at the high-density tree, and its new location will still be at the high-density tree. Therefore, O_i changes its dense cell C_x to another dense cell C_y. The update algorithm will simply delete the object O_i from C_x and insert it at the new dense cell C_y. Moreover, in this case of updating, the high-density tree has no restrictions in terms of the maximum/minimum entities;

thus, no splitting is expected and there are no overflow and underflow cases. Note that tracking and updating take place only if the object changes its location cell.

(b) The moving object O_i is currently at the high-density tree, and its new location is not at the high-density tree. The new location cell O_i will be at the low density tree. Therefore, O_i changes its dense cell C_x to another low cell C_l. The update algorithm will simply delete the object O_i from C_x where there is no underflow, and uses the link list of the combined trees to insert O_i to C_l. Here, the moving object O_i will be inserted to a linked non-leaf node, and then the insert process is completed by choosing the adjacent leaf node suitable for the new cell of the O_i. Note that in this case, overflow and splitting is possible at the low density tree.

(c) The moving object O_i is currently at the low-density tree, and its new location is the low-density tree. Therefore, O_i changes its cell C_i to another cell C_j. The update algorithm will simply delete the object O_i from C_i where there is a possibility of underflow, and insert O_i to C_j where there is also a possibility of overflow. Simply, the update algorithm will check the RC of the non-leaf nodes and LE of the leaf nodes in order to group O_i with its nearest adjacent cells.

(d) The moving object O_i is currently at the low-density tree, and its new location is not the low-density tree. The new location cell O_i will be at the high-density tree. Therefore, O_i changes its cell C_i to another a dense cell C_x. The update algorithm will simply delete the object O_i from C_i where there is a possibility of underflow, and use the link list of the combine trees to insert O_i to C_x. Here, the moving object O_i will be inserted to link MBR to the high-density tree where there are no restrictions regarding the maximum/minimum entities.

We argue that indoor updating does not result in a large update overhead for the following reasons: From the fact that moving objects inside indoor spaces are

slow-moving objects, and they tend to settle for long periods of time in their cells. Moreover, we track and update the moving objects only if they change their cells since no tracking is applicable when they are inside the cells. Figure 5.8 shows an example of the Indoor^d-tree.

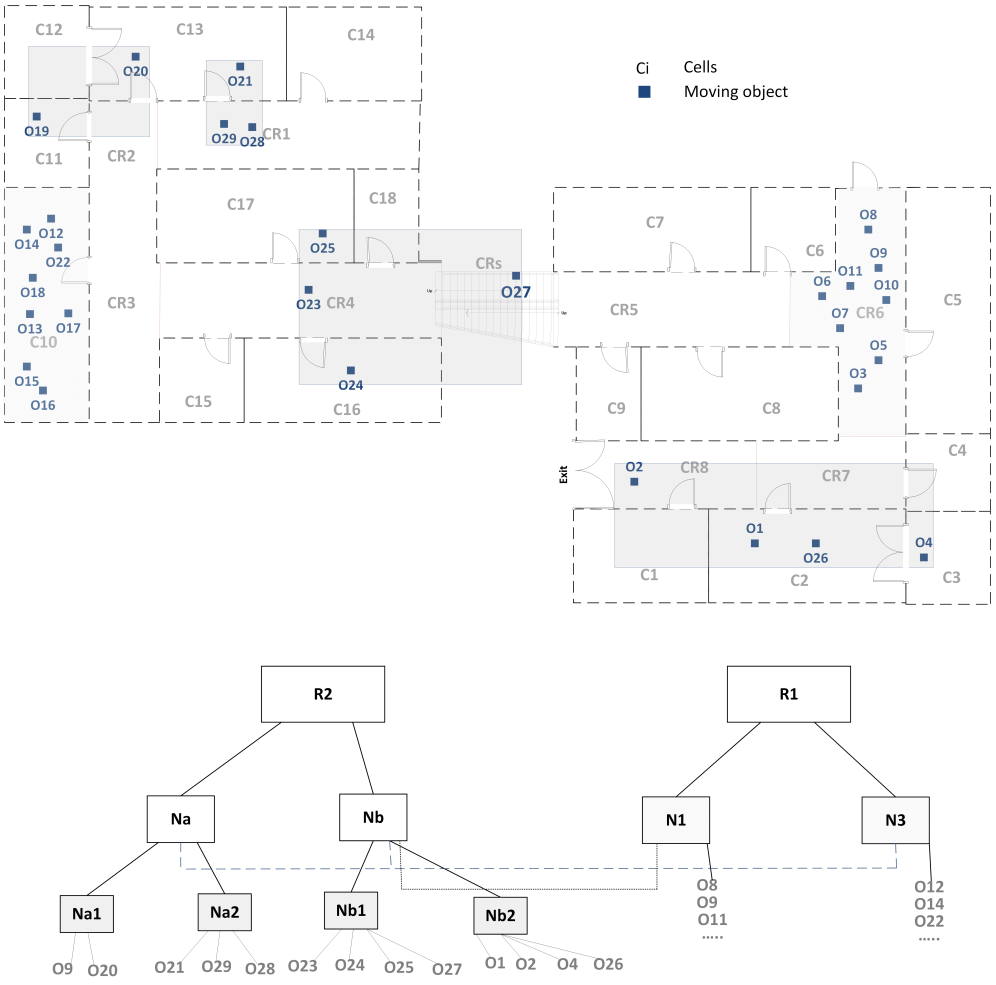


FIGURE 5.8: Grouping the moving objects in a multi-floor space including the vertical transits (stairs)

Note that in the high-density tree, the cells that considered as dense cells is already pre-assigned based on the analysis of the indoor floor, or past behavior of moving objects in these cells. Therefore, these cells in the high-density tree will continue to group the moving objects since there are assumed to be dense cells for majority of time. However, in a few cases, if the database updates some of the cell so that they become dense cells, then, the high-density tree and low-density tree

as work follows: First, the high-density tree will create an above level which will store that dense cell. Then, the moving objects in that cell will be stored in the leaf node. Second, in the low-density tree any objects located in that cell will be deleted and a case of underflow here is possible. Note that the new dense cell will be linked with the adjacent node in the low-density tree as explained previously. For example, based on Figure 5.7, assume that the cell that is located in the north west corner is changed so that it becomes a dense cell based on the analysis of the indoor floor. Here a new $N4$ will be created in the high-density tree and the objects will be inserted there. Also, the moving objects in that cell will be deleted from N_{a1} and since underflow has occurred in the N_{a1}, then the remaining object there will be re-inserted in N_{1b}. Figure 5.9 illustrates the result of this process.

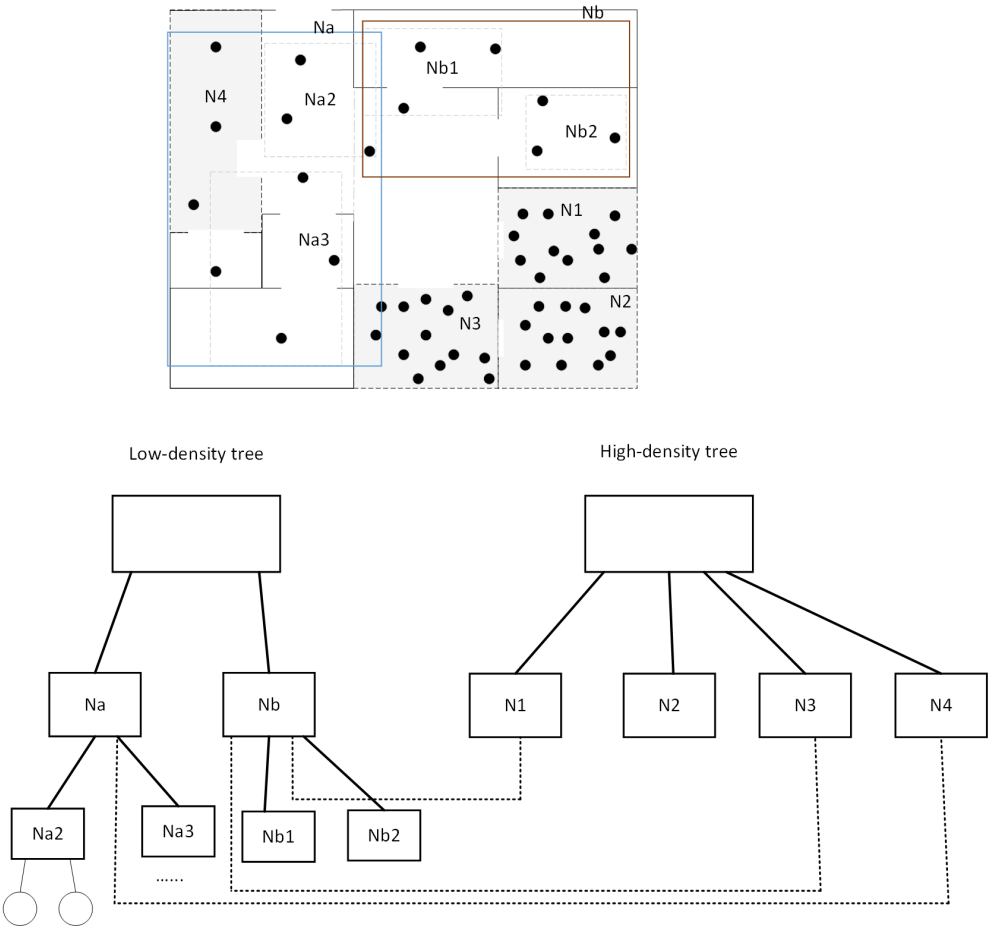


FIGURE 5.9: The high-density tree with the low-density tree after modifying the dense cell

5.3 Analytical study

We analyze the cost of the search for the low-density tree and high-density tree and the storage cost as well. We discuss the case of data space DS = single floor, where the dimension $d = 2$, and it can be applicable to other floors. The average disk page accesses of general queries on the low-density tree are given by equation 1. Note that a disk page is a node, so page and node are used synonymously in this section.

$$DP(N) = \sum_{h-1}^{i=0} \text{ni.P}_{DP}i \qquad (1)$$

h is the height of the low-density tree (until level 0 (the leaf node)). $P_{DP}i$ is the probability that a node at level i of the tree is accessed. ni is the total number of nodes at level i of the tree.

$$h = 1 + (log_f N/b)$$

$$ni = N/f^{i+1}$$

Where b indicates the maximum capacity of a node, and f is the average node fanout (usually, f = 69% . b) [25, 100].

Lamma 5.1. Let ι be a find query which will check over the entire data space. Then the probability of an object ι is $P_{DP}i = \sum_{h-1}^{i=0} P(x)$,where $p(x) = x/ni$ for each level.

Proof: Suppose we construct a uniform low-density tree which contains 2 non-leaf nodes and contains 2 leaf nodes. The probability of the tree root is $P(x = \text{right} \bigcup x = \text{left}) = 0.5 + 0.5 = 1$. Therefore, the tree probability is adding of each level of nodes as $p(x) = x/ni$. Where the probability of reaching a leaf node will be P(non-leaf node) + P(leaf node) = 1.

In addition, the average cost of the high-density tree is given by equation 2. Let a list L, of n objects and a search key x. Moreover, Let $ki = i$ be the number of comparisons when the object $x = L[i]$ and Let $k0 = n$ be the number of comparisons when $x \notin L$. The average cost of the high-density tree is calculated as follows:

$$\begin{aligned} F(n) &= 1/2 * ki + 1/2 * k0 \\ &= 1/2 * (n/2) + 1/2 * (n) \end{aligned} \qquad (2)$$

Note that the worst case cost of the high-density tree is :

$$\text{f(n)} = \begin{cases} \partial & \text{n} = \partial \\ f(n-1)+1 & \text{n} > \partial \end{cases}$$

For the storage cost, we assume that the disk is filled with the indexed objects. Moreover, we assume that caching is for all levels of the low-density tree except the lower level (the leaf nodes with the data). The number of data pages is d/p, where is d indicates the disk size (bytes) and p indicates the page size (bytes). Moreover, for the number of index signs ($<RC, ptr>$) in the higher level of the leaf

nodes, each index sign can take up (e/u) bytes, where e indicates index entry size (bytes) and u indicates the storage utilization for the low-density tree. In order to determine the amount of main memory that is taken up by the low-density tree, we multiply the data pages by the number of index signs.

$$\text{Low-density tree (memory)} = (d/p) * (e/u) \tag{3}$$

In addition, we cache the low-density tree memory in order to save on accessing costs.

$$\text{Cache Low-density tree (memory)} = C_l((d/p) * (e/u)) \tag{4}$$

For the high-density tree, we cache all the dense cells' level. The dense cells level $<dense - cell, pointer>$, each index sign can take up (e/u) bytes, where e indicates index entry size (bytes) and u indicates the storage utilization for high-density. Moreover, for the data level here (list of the objects e.g. array) the number of data pages is ev*es, where is ev indicates the entry number and es indicates each enter size (bytes). Note that we assume the access rate is sustained. Therefore, the cost of the low/high density trees are the main memory cost of caching the tree in addition to the cost of the disk.

$$\text{Cost} = C_l((d/p) * (e/u))+((e/u) * (ev * es)) \tag{5}$$

5.4 Experimental Results and Performance Analysis

This section examines the experimental results in order to evaluate the proposed data structures (Indoor^d-tree). We compared our new structures with Indoor-tree [30]. The experiment was carried out on an Intel Core i5-2400S processor 2.50GHz PC, with 4 GB of RAM running on 64-bit Windows 7 Professional. The maximum number of entries per node, M, was 80 and the minimum m 40. The data structure was implemented in Java. The data set size ranged from 200 to 1000 moving objects on the indoor floors. We used synthetic datasets of moving objects on an indoor environment floor due to the lack of real data for indoors. In the experiment, we use real-case 12-cell and 50-cell indoor floors.

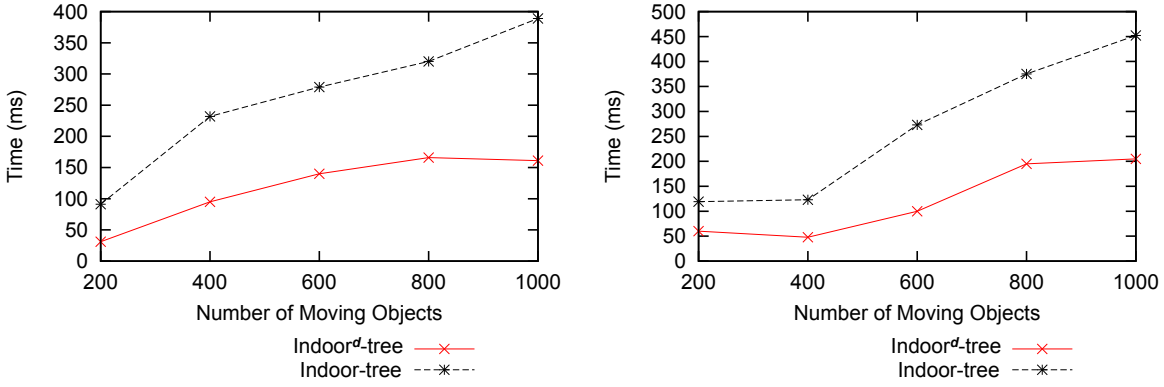
We investigated the construction, insert and search performance costs. For all operations, the execution time was measured. For both the insert and search query, we took into account the effect of the complexity of connections for different densities of moving objects. Note that each operation was performed 7 times and the average was calculated. The parameters used are summarized in Table 5.2.

TABLE 5.2: Parameters and their settings

Parameter	**Setting**
Node capacity	80
Number of moving objects	200 to 1000
Indoor space	12/50 cells
Operations	construction, insert and find
Dataset	synthetic

We examined the construction, insert and search costs. For the construction, we construct moving objects in both Indoor-tree and the Indoor^d-tree for various

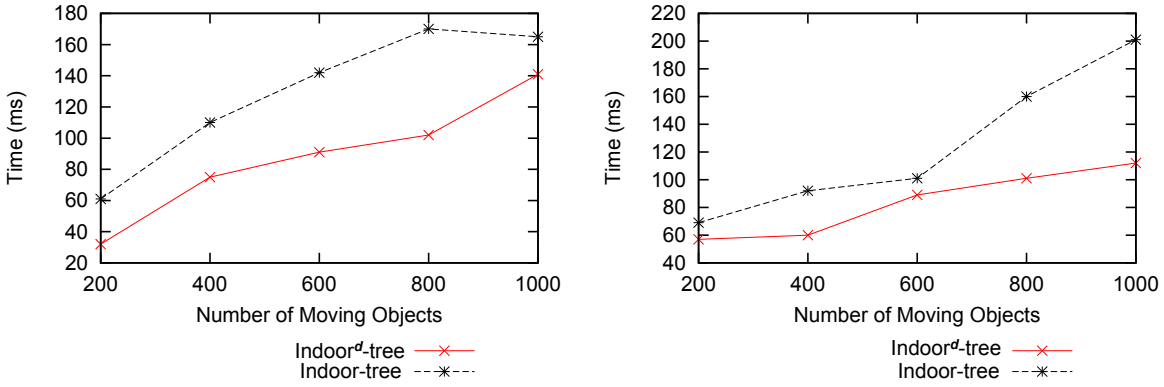
numbers of cells and different numbers of objects ranges from 200 to 1000 moving objects. For the search, we conducted a descending search to locate a moving object on the Indoor-tree. For the insert, we compared the proposed structure to the Indoor-tree in which one incurred less insert costs. Moreover, for the insertion, we considered the influence of increasing the number of moving objects and increasing the number of the indoor connections (connections complexity). Note that the insert and the search operations tests were conducted after the objects had been constructed.



(a) Trees construction costs of 12-cells case

(b) Trees construction costs of 50-cells case

FIGURE 5.10: Trees construction costs



(a) Insert performance of 12-cells case

(b) Insert performance of 50-cells case

FIGURE 5.11: Insert performance costs

For the construction, we tested the construction performance costs of the Indoor-tree and Indoor^d-tree for different densities (numbers) of moving objects.

Figure 5.10 (a) shows the 12-cell case, where we note a significant cost decrease using our proposed index. In some cases, the cost is decreased by about 52% lower than the Indoor-tree. Figure 5.10 (b) shows the 50-cells case, where construction costs are continuing to decrease. We can notice in the case of 600 moving objects that the cost is approximately 57% less than for the Indoor-tree. The Indoor^d-tree had less grouping cost which we can explain as follows: the Indoor^d-tree takes into account the difference in the cells' densities which make the insertion operation in the high-density cells less costly (no splitting is required). On the other hand, the Indoor-tree requires a splitting as $M + 1$ entries in case of overflow nodes which increase the number of splitting operations that affect the construction performance.

For the insertion, we began by inserting an object for both the Indoor-tree and Indoor^d-tree in different densities. We can see from Figure 5.11 (a) (12-cells case) that in all density cases, the Indoor^d-tree performed with less cost than the Indoor-tree. Furthermore, in Figure 5.11 (b) we can see that in the insertion of the 50-cell case, the Indoor^d-tree is still cost-efficient despite the increase in the cell number. For example, in Figure 5.11 (b), the insert cost in the 1000 objects case is decreased approximately 43% in the Indoor^d-tree. Moreover, we tested the Indoor^d-tree in high-connections environments (worst case of connections number) as shown in Figure 5.12 which indicates that the Indoor^d-tree performed with less cost in all density cases than did the Indoor-tree.

In addition, we tested the search performance of the Indoor^d-tree and Indoor-tree in the 50-cell case. Figure 5.13 shows that Indoor^d-tree can reduce the search cost by between 4% to 45% for all densities of the moving objects. For example, in the 800-objects case, the Indoor^d-tree reduced the search cost by about 9% lower than the Indoor-tree.

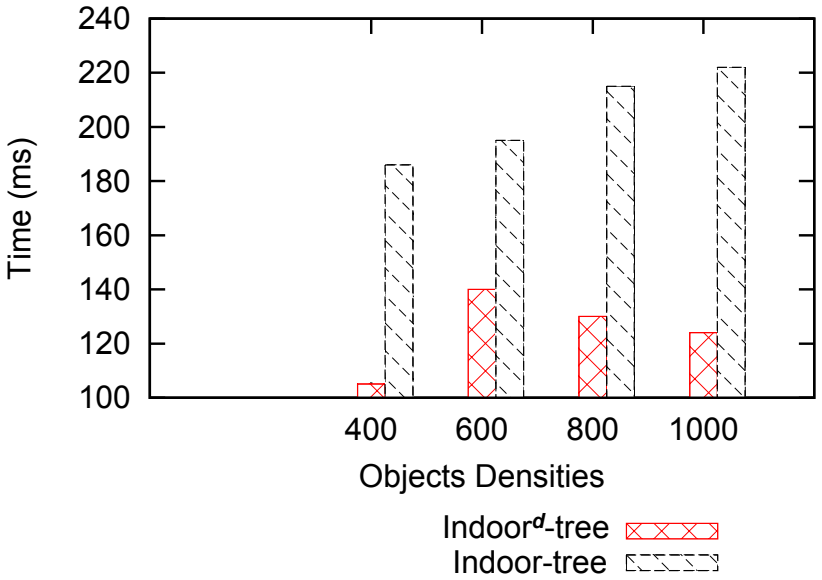


FIGURE 5.12: The effect of cell connections on insert performance

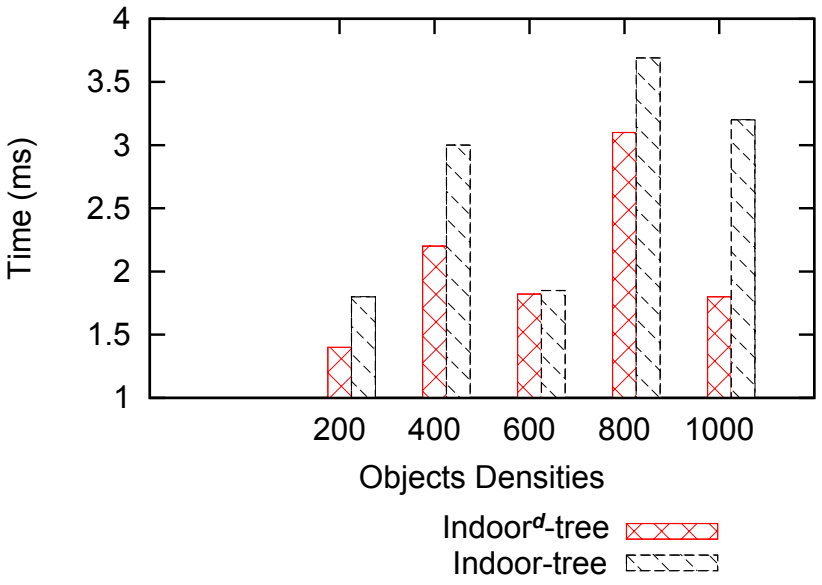


FIGURE 5.13: Search performance

We can summarize the results as follows: First, increasing the number of objects and increasing the cell connections (worst case connections) has no particular influence on the Indoor^d-tree. Second, the index-maintaining operations such as insert and search are efficient and more cost-effective compared with the Indoor-tree. Third, the proposed index structure Indoor^d-tree can successfully produce a reliable and robust tree index for multi-floor indoor spaces.

5.5 Chapter Summary

This chapter addresses the challenge of building an index structure for multi-floor indoor spaces. The measurements for indoor space and outdoor space are significantly different. Outdoor spaces are typically based on Euclidean space or a spatial network. On the other hand, the indoor spaces are related to the symbolic/notion of cellular space which clearly consists of unique entities (e.g. walls and doors) that play an important roles in preventing and enabling the object's movement. Our proposed index structure (Indoor^d-tree) takes into consideration the obstacles to indoor movements (e.g. walls) . Therefore, the Indoor^d-tree is based on the adjacency/connections of the cells. Thus, we group the moving objects according to their connection to each other, not based on their metric closeness. In addition, since the moving object density in indoor spaces greatly affects the efficiency of the data structure, we distinguish between cells of different densities in the index structure. The basic idea is to use the adjacency method and group the moving objects based on their density. The cells with high density will be grouped in the high-density tree, whereas, the cells with a low density of objects will be grouped in the low-density tree. This technique will easily and more efficiently support the density-based queries and the aggregation queries of the dense cells. Moreover, it will improve the performance since the splitting operations will be not applied to the high-density tree. Moreover, we considered the challenges of objects' vertical movement (e.g. via stairs and elevator) in a multi-floor indoor space. Extensive performance studies were conducted and results indicate that the Indoor^d-tree is both robust and efficient.

CHAPTER 6

Indexing Moving Objects in Multi-Floor Indoor Environments

Chapter Plan:

6.3 Chapter Summary

Publications and Submissions:

- Sultan Alamri, David Taniar, Haidar Al-Khalidi and Kinh Nguyen: "Indexing Mobile Objects in Multi-Floor Indoor Environments". *ACM Transactions on Spatial Algorithms and Systems* (2014) (Under Review).

The importance of the location of mobile objects can be different in some cases. Some buildings (such as shopping centers) contain wings which affects the success of the positioning queries. From observation, in some buildings the positioning of the mobile objects can be based on their section [42, 45]. For instance, "Where is object O_3?" The exact location here is not important since the query can be stratified with the answer "north section/wing". Also, here we note that the exact floor in this case is not needed. This is clearly observed in shopping centers and buildings that have several wings. Not only is the exact cell not needed; it might be useless to the user. For example, for the query "Where is object O_1?", answers such as "room 133" might not be useful to the user who has no idea where room 133 is located. However, an answer using the location of the wing/section (e.g. he is in the north wing) might be more useful to the user.

Here, the spatial index of indoor space usually groups the moving objects in indoor spaces based on the closeness of the objects. Therefore, the moving objects will be grouped based on the adjacency of the cells such as the Indoor-tree. It is not well-understood how the index structure will perform if it groups the moving objects horizontally and vertically. In some indoor multi-floor environments, the layout of the building comprises different wings and sections. In these buildings,

the moving objects in a certain wing should be grouped together in that wing. Therefore, the aim is to group the moving objects that are close to each other horizontally (on same floor) and vertically (the floor above), which places them eventually in the same wing.

If the index considers grouping the moving objects based only on the horizontal adjacency aspects (based on the Indoor-tree), then, the index will result in many cross-over nodes within the indoor building. For example, based on Figure 6.1, grouping the moving objects based on the Indoor-tree is produced two Crossed-Over nodes between the building's wings. However, grouping the moving objects based on horizontal/vertical adjacency, the index in that case will produce 0 Crossed-Over nodes between the building's wings (see Figure 6.2). Keeping the moving objects that close to each other horizontally and vertically will help to process the cell queries and wing queries more efficiently. For example, if a query asks about a building section/wing which contains moving objects, the index structure (the Indoor-tree) will need to engage in several verification steps in order to reach the targeted answer (see Figure 6.3). This is obvious since the idea is to group the moving objects close to each other, and the index needs to check the nodes on each floor individually which will incur high access costs. Therefore, we argue that it is possible to group the moving objects based on multidimensional grouping in each section/wing, which is useful in wing/section queries and aggregation queries. Our major contributions are:

- We propose a model to represent indoor spaces.
- We extend the basic indoor space model for multidimensional grouping (e.g. based on cells and wings) and identify typical situations where such a model can be usefully applied.

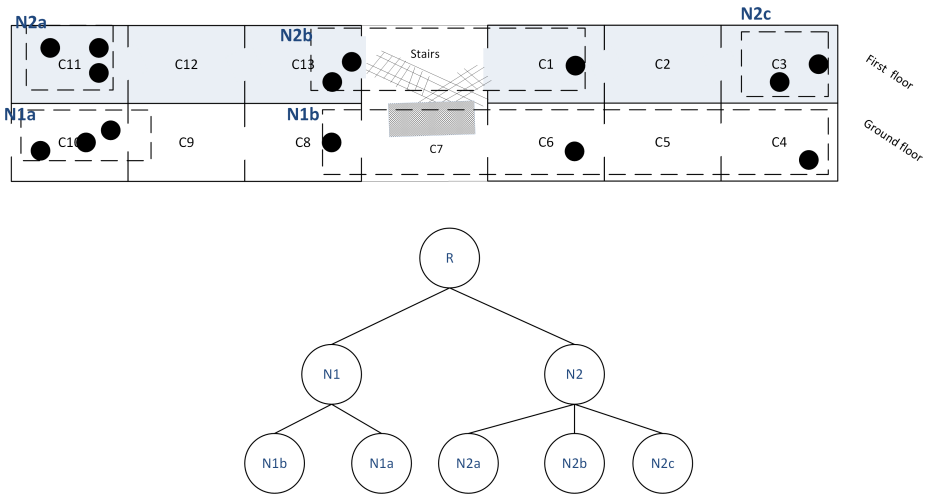


FIGURE 6.1: N2b and N1b is Crossed-over nodes between the wings

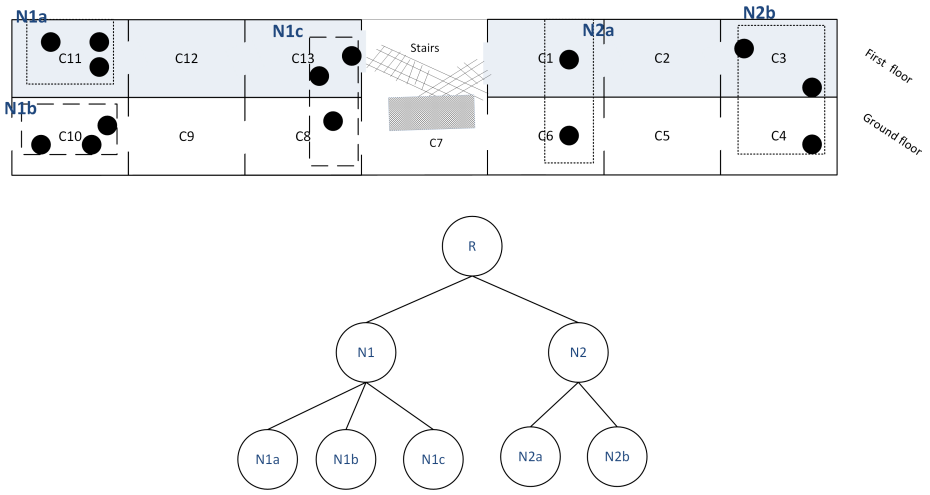


FIGURE 6.2: The moving objects are grouped in each wing

- We develop a multidimensional index structure: the GMI-tree.
- Under a simulation environment, we conduct experiments in order to evaluate the proposed index structure GMI-tree by studying construction, insert, and query performances.

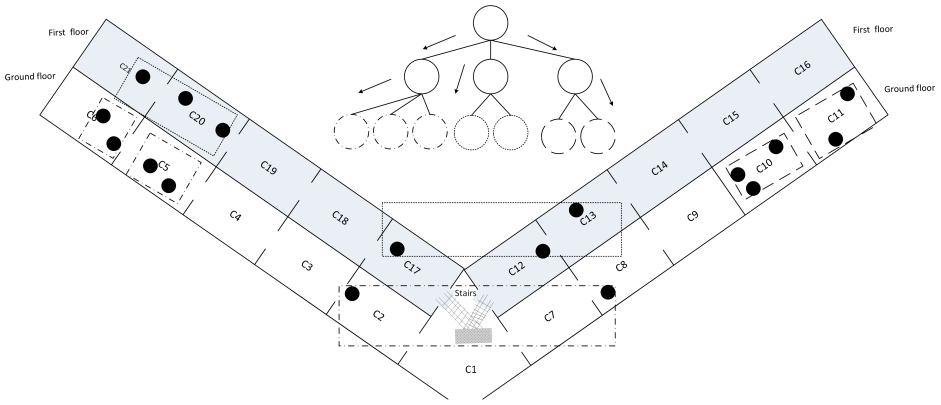


FIGURE 6.3: Grouping the objects on each floor individually will incur high access costs when processing the Wing's direction queries

6.1 Graph-based Multidimensional Indoor-Tree

6.1.1 Indoor Connectivity Graph

The connectivity is one of the common concepts of graphs determining the number of nodes/vertices or edges which are needed to connect a certain graph [98, 101]. The graph theory has been use in many researches concerned with spatial databases such as road network flow problems [52, 102]. Moreover, the shortest path algorithm and Dijkstra's algorithm have been used in many of the spatial query processing (e.g. KNN, range queries) [21, 50, 51]. Figure 6.4 shows an example of an indoor connected graph.

Based on the methodology of the positioning system that is used in this work, the cells of the indoor space are connected with each other. Indoor space contains elements such as *rooms, doors, corridors, floors, stairs, elevators* and *pathways* between buildings. The mappings from domain concepts to modeling concepts are summarized in Table 6.1. The indoor will be an *undirected graph* with *cells (nodes)* and *edges (connections)* (see definition 6.1). For example, Figure 6.4 illustrates

the connectivity of a single floor, and shows how the connectivity graph of that floor is represented. Here, the indoor-connectivity essentially is converted to a graph connectivity where if a cell has a connection (e.g. door) to another cell, then both will be linked with an edge in the graph. In Figure 6.4, C_{10}, C_9, C_8, C_7, C_4, C_6, C_5, C_3, C_2 and C_1 are connected, where C_{10} represents stairs which connect the two floors.

Definition 6.1: An indoor space is a connected undirected graph $(Cells, Edges)$, where $Cells = \{C_1, C_2, \ldots, C_n\}$ is a set of cells, and $Edges = \{E_1, E_2, \ldots, E_m\}$ is set of edges, each of which is a set of two cells, that is, each edge connects two distinct cells.

Definition 6.2: (Adjacent Nodes) Let $G = (N,C)$ be a graph, and two Nodes $n_1,n_2 \in N$ of G are adjacent if: $\exists$ n_1,n_2 are connected.

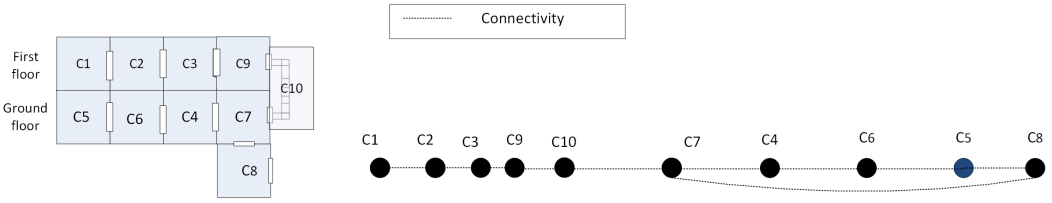


FIGURE 6.4: An example indoor connected graph

domain concept	modeling concepts
room	a cell
door	an edge
corridor	one or more cells with one or more edges
stair	one or more cells with one or more edges
elevator	one cell with several edges
pathway	one or more cells with several edges

TABLE 6.1: Mapping domain concepts to modeling concepts

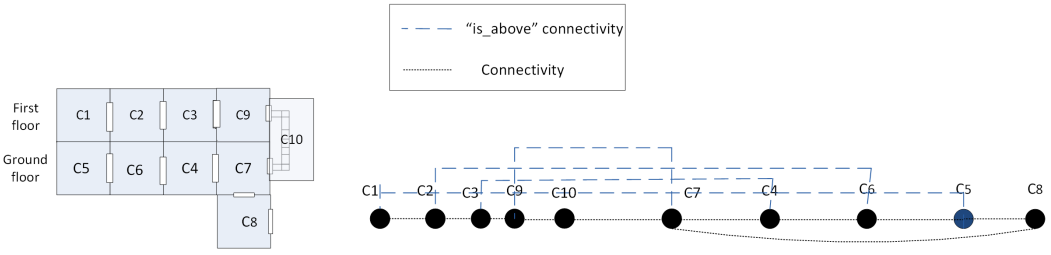


FIGURE 6.5: The Multidimensional Connectivity Graph for the indoor space in Figure 6.4

6.1.2 Indoor Multidimensional Connectivity Graph

Given a multidimensional indoor space, i.e. one for which we can conceive of two or more types of edges, our first step is to convert it into a multidimensional connectivity graph. Figure 6.5 shows the multidimensional connectivity graph that represents the indoor space in Figure 6.4. The graph has primary edges, shown as solid lines between the cells, and secondary edges, shown as dotted lines. The primary edges are derived from the indoor space (Figure 6.5) as described in the previous section (see Table 6.1). The secondary edges between two cells are obtained by virtue of one cell being directly above the other.

Note that the secondary "is_above" connection considers the cells that are located above each other. This includes the case where a cell is connected vertically with more than one cell (vertically overlapped); here the "is_above" connectivity will be made to all the overlapped cells regardless of the size of the overlap. The reason for this choice is because the grouping here focuses on the wing/section, where all overlapped cells are located in the same side (wing).

Definition 6.3: (Multidimensional Connectivity Graph) Given a set of cells C_i, C_j, C_f, C_l in the ground floor and a set of cells C_x, C_y,C_r, C_t the multidimensional connectivity graph refers to the multiple edges that connect the single floor cells

and the above cells in the multi-floor space.

6.1.3 The Multidimensional Connectivity Tree

Given a graph representing an indoor space, the connectivity tree is constructed as follows:

1. First we select one cell to be the default cell such as C_1 in the Figure 6.6.
2. Then we perform a breadth-first search on the connectivity graph, starting from the default cell. In this way, we obtain a tree as a result.
3. We then order the siblings (nodes at the same level) by the number of descendants. Those with more descendants are said to be higher. For siblings with the same number of descendants, we select an order arbitrarily. In this way, all the nodes in the trees are ordered [30].
4. We then establish the "is_above" (secondary) link for the cell that has any cell that are located above it (see Figure 6.6).

Example: For the indoor connectivity in Figure 6.3, we get the connectivity tree shown in Figure 6.6. C_1 is on level 1: it is the highest node according to our definition. On level 2, we have cells C_{12}, C_{17}, C_2 and C_7. Note that the connectivity tree will be *the input* to our GMI index tree.

Note that the multidimensional connectivity tree is constructed from the multidimensional connectivity graph by ignoring the secondary edges, and applying the same procedure as described in Chapter 4 Section 4.2.1 for constructing

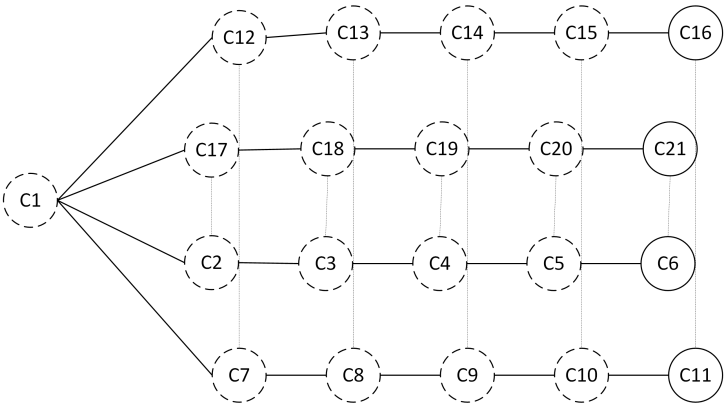


FIGURE 6.6: Multidimensional Connectivity Tree based on the indoor space in Figure 6.3

the basic (or 1D) connectivity tree. Figure 6.6 shows a connectivity tree for the multidimensional connectivity graph shown in Figure 6.3. Here, when using that connectivity tree in the index tree construction, we take into account the secondary edges which will have an effect on what we regard as the distance between pairs of cells.

6.1.4 Tree Creation

The construction of GMI-tree is essentially the process of inserting an object into an existing GMI-tree. This task involves two major subtasks: (i) navigating down the GMI-tree to get to the leaf node in which the new object is to be inserted; and (ii) in the case where we need to split a node, choosing which objects (in the node) are to be grouped with the new object.

For navigation down the tree, we use exactly the same strategy as we did in the Indoor-tree. That is, suppose we need to insert an object occupying the cell C_{new}, we examine the RC and LE of the nodes, and at any particular stage,

- Choose the non-leaf node whose RC covers C_{new} according to the multidimensional connectivity tree, or
- Choose the leaf node which has a cell of closest distance to the cell C_{new}

However, the distance between any pair of cells is now modified according to the definition below.

Definition 6.4: Let $d_P(x, y)$ and $d_S(x, y)$ denote the primary and secondary distances between cells x and y based on the number of primary edges and secondary edges between them. The distance between x and y is defined to be

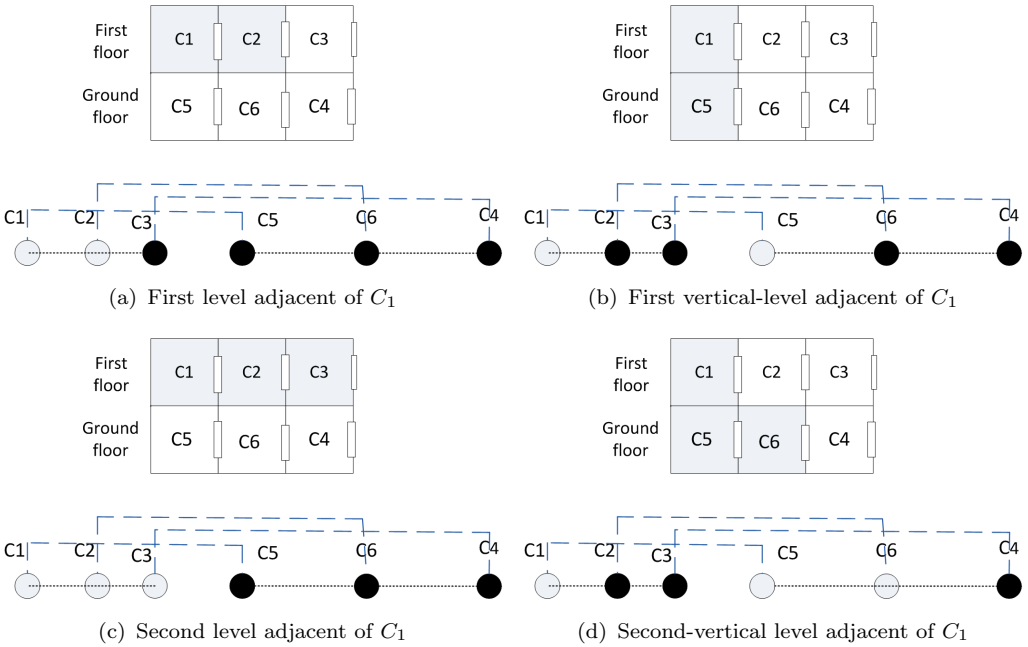
$$\text{d(x, y)}=\begin{cases} d_P(x,y), & if \quad d_P(x,y) \leq d_S(x,y) \\ d_S(x,y) & otherwise \end{cases}$$

The procedure for constructing the GMI-tree is given in Algorithm 8 and Algorithm 9. Examples of the GMI-tree grouping are given in Figure 6.7.

6.1.5 Multidimensional Indexing-Applicable Indoor Spaces

The notion of cut node defined below will be useful for identifying and indexing indoor spaces to which the multidimensional indexing method is applicable.

Consider a graph G of nodes (cells) and edges. Suppose we have a subset S of node of G such that when we remove the nodes in S from the graph, what we get is a number of unconnected subgraphs. Then we say that S is a set of cut nodes and each element of S is called a cut node.



(a) First level adjacent of C_1 (b) First vertical-level adjacent of C_1

(c) Second level adjacent of C_1 (d) Second-vertical level adjacent of C_1

FIGURE 6.7: The adjacency levels that are used in the GMI-tree construction

Definition 6.5: (Graph with Cut Nodes) Let $G = (Cells, Edges)$ be a connected undirected graph. Let S be a subset of nodes in $Cells$. Then we say that S is a set of cut nodes if the graph $G' = (Cells', Edges')$ is an unconnected graph, where

- $Cells' = Cells \setminus S$ and
- $Edges' = \{e \in Edges : \forall x \in S, x \notin e\}$. (Note that an edge e is an unordered pair (i.e. a set) of two nodes).

Theorem 6.1: The Node n is a cut node of a graph G if and only if there exist two nodes v and u in G such that
(i) $n \neq v$, $n \neq u$ and $v \neq u$, but
(ii) n is on every $u - v$ path.

Proof. Let us consider the case where n is a cut-node of a graph G and the case where G-n is not connected. There are at least two elements $G_1 = (N_1, E_1)$ and

Algorithm 8 *Insert_Object*
/* To Insert an object into the current index tree.
The tree is of order M = maximum number of child nodes */

Data: The connectivity tree $CONNECT\_TREE$; the Current index tree $INDEX\_TREE$: the object to be inserted $OBJECT$; the cell that is occupied by the object, $OBJECT\_CELL$ **Result:** The updated index tree

1: /* Find the leaf node to insert the new object */
2: Let $p = root\ of\ INDEX\_TREE$;
3: **while** p is not a leaf node **do**
4: | Construct the set $EXPANDING\_CELLS$ from all the RC and LE of the children of p;
5: | Choose from $EXPANDING\_CELLS$, a cell $NEAREST\_CEll$ that is nearest to the $OBJECT\_CELL$, according to $d(x,y) = IF\ d_P(x,y) \leq d_S(x,y)\ THEN\ d_P(x,y)\ ELSE\ d_S(x,y)$;
6: | Choose a node $Node$ that contains $NEAREST\_CELL$ as a bound of RC or LE;
7: | Let $p = Node$;
8: **end while**
9: /*Insert the object into the leaf node p */
10: **if** Node p is not full, i.e. number of objects is less than m **then**
11: | /* NOTE: Add this condition to definition of Index Tree */
12: | Insert $OBJECT$ into node p;
13: **else**
14: | Choose a set of $\lceil \frac{M}{2} \rceil$ existing objects in the node p, that are nearest to $OBJECT\_CELL$.
15: | Call this set $GROUPED\_CELLS$;
16: | Insert $OBJECT$ and $GROUPED\_CELLS$ into a sibling node of p;
17: | Split the node, using the standard node-splitting procedure for B+Tree, and insert $OBJECT$ and $GROUPED\_CELLS$ into a sibling node of p;
18: **end if**
19: For nodes that contain new objects, update RC and LE

$G_2 = (N_2,E_2)$. We choose $v \in N_1$ and $u \in N_2$. The path of $u - v$ is in G because it is connected. Therefore, if n is not on this path, then the path is also in $G - n$. For the same reason, this is used for all the $u - v$ paths in G. Therefore, if n is in every $u - v$ path, then the nodes u and v are not connected in $G - n$.

An example of a cut node is shown in Figure 6.8. The Figure 6.8 represents the connectivity graph of Figure 6.3. Other examples of cut nodes are shown in Figure 6.9.

Algorithm 9 $CONSTRUCT\_INDEX\_TREE$
/* To construct an index tree for a set of objects */
Data: The connectivity tree $CONNECT\_TREE$; set of objects $OBJECTS$; function $OCCUPIES : OBJECTS \rightarrow CELLS$ **Result:** The index tree

1: Let $Index\_Tree = \emptyset$;
2: **for** each object O_i of $OBJECTS$ **do**
3: | Call algorithm $INSERT\_OBJECT$ to insert O_i into $INDEX\_TREE$;
4: **end for**
5: **return** $Index\_Tree$;

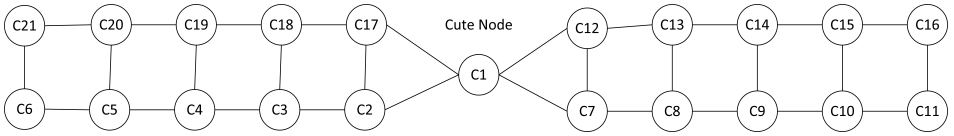


FIGURE 6.8: C_1 is the cut node

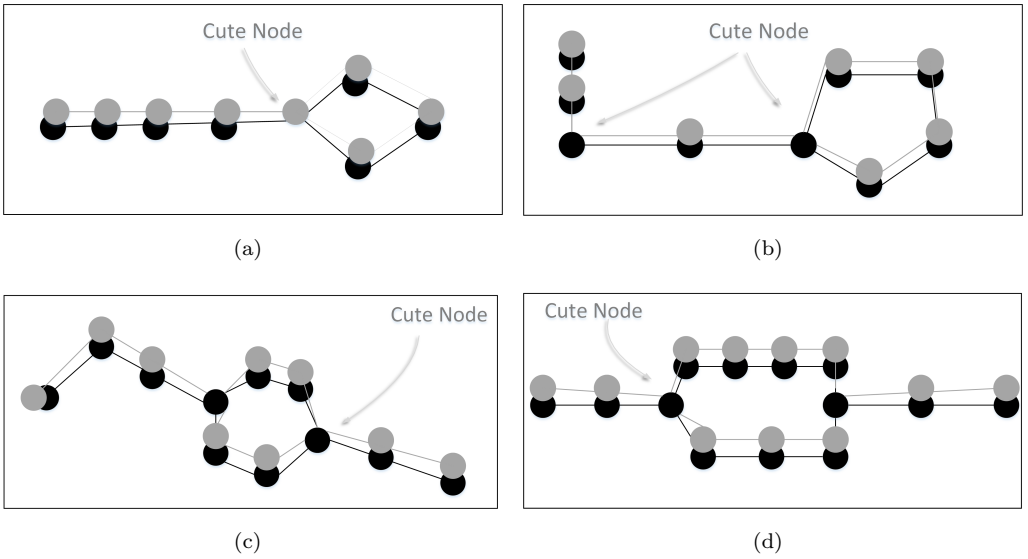


FIGURE 6.9: Cut nodes of connectivity graphs

In principle, we can apply the GMI Index Construction Procedure to any indoor spaces that is represented by an multidimensional connectivity. However, in practice, as will be seen in the results section, the types of indoor spaces to which the multidimensional procedure can be applied successfully are those with the following properties:

1. *It is a graph with cut nodes.*

2. *The graph after the removal of cut nodes forms a number of islands (isolated subgraphs) of reasonable sizes.*
3. *There are no secondary edges among the cut nodes (this is reasonable because cut nodes should not be part of a wing: they separate wings from one another).*
4. *There are no secondary edges between nodes that belong to different islands (secondary edges are confined to separate islands).*

Under those conditions, we can formalize the concept of a wing as follows.

Definition 6.6: Assuming the context of the conditions above, two cells X and Y are in the same wing iff they belong to the same island.

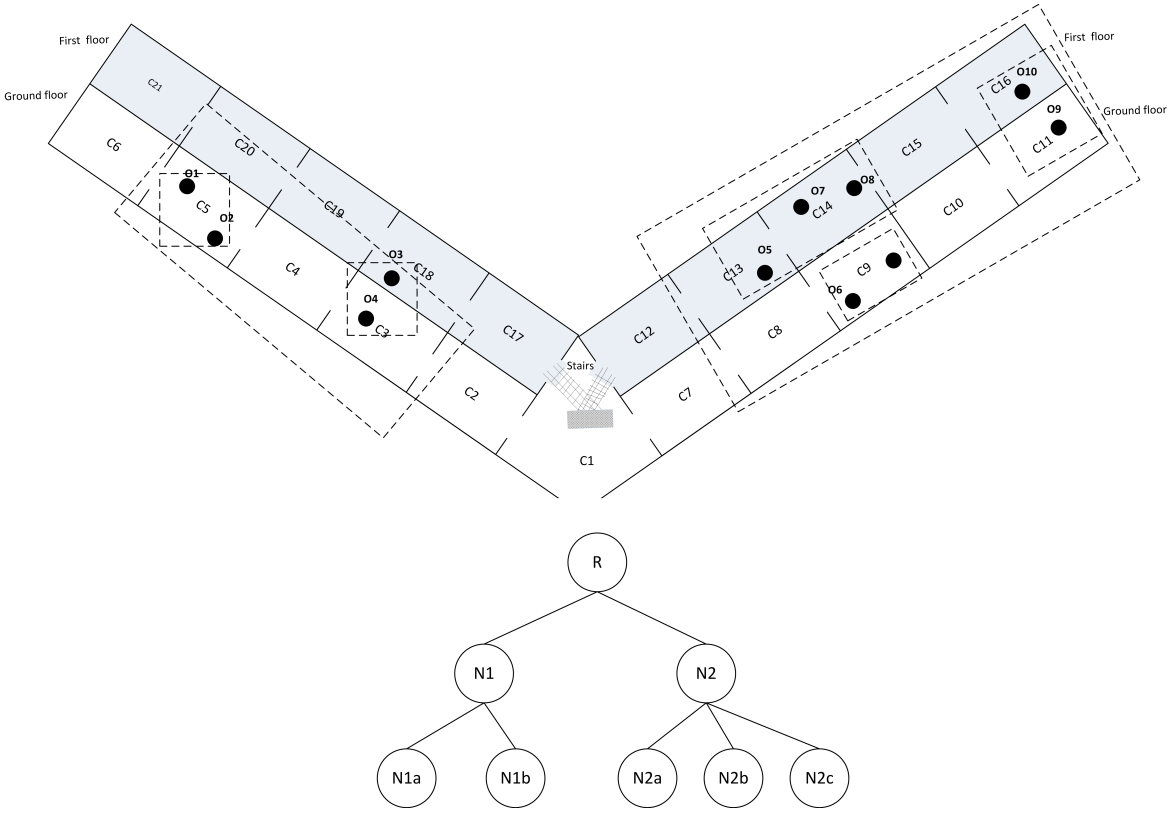


FIGURE 6.10: An example of Graph-based Multidimensional Indoor-Tree

For example, Figure 6.3, gives an example of a two-floor building and Multidimensional Connectivity tree is shown in Figure 6.6. Figure 6.10 represents a set of moving objects O_1, O_2,...,O_{11} in a multi-floor indoor space. The GMI-tree will group the objects based on their section/wing as follows: The GMI-tree starts by grouping the moving objects that are located in the same cell, then starts to evaluate the adjacency levels in case of overflow or underflow. For example, O_4 is grouped with the above cell entities which is O_3. Note that the objects in the West wing are grouped together (O_1, O_2, O_3 and O_4), and East wing has the objects (O_5, O_6, O_7, O_8, O_9, O_{10} and O_{11}). In this grouping technique, the distance will be based on the number of the hops between the cells (on both horizontal and vertical adjacent levels). Moreover, in this technique, we argue that queries concerning the wing objects are processed easily with low access costs.

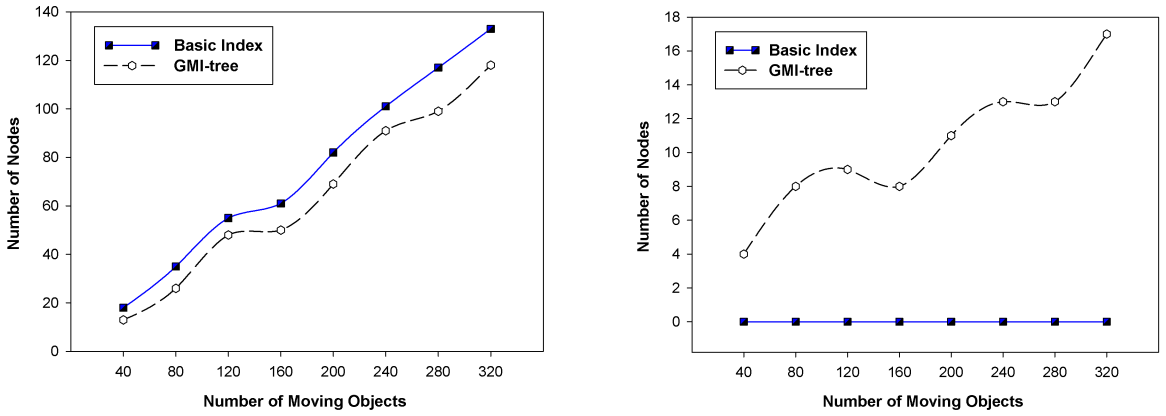
6.2 Experimental Results and Performance Analysis

In this section, under a simulation environment, we conduct experiments in order to evaluate the proposed index structures Graph-based Multidimensional Indoor-Tree (GMI-tree). We compared our new structure with the Basic-Index (which only groups the moving objects horizontally) [30]. The experiment was carried out on an Intel Core i5-2400S processor 2.50GHz PC, with 4 GB of RAM running on 64-bit Windows 7 Professional. The maximum number of entries per node, $M = 80$ and the minimum $m = 40$. The data structure was implemented in Java. The data set size ranges from 20 to 1000 moving objects on the indoor floors. We used synthetic datasets of moving objects on an indoor environment floor due to the lack of real data for indoors. In the experiment, we use a real 20-cell case.

We will focus on the following operations: First, the number of the nodes that grouped the moving objects vertically and horizontally in the GMI-tree and the Basic-Index. Here it will show the ability of the data structure to keep together the moving objects in each wing. Second, the number of the Crossed-Over-Nodes between the wings of the building, which means how many nodes that the construction of the GMI-tree and the Basic-Index caused to be crossed over between the building wings which is considered as false hits based on the individual wing/-section grouping. Third, the data structure's performance in both the construction costs in addition to the search performance costs. For all operations, the execution time was measured. Note that each operation was performed several times and the average was calculated.

Figure 6.11 (a) shows that the GMI-tree groups the moving objects horizontally proximally to the Basic-Index which has only the horizontal methodology (through the basic link of the connectivity tree) to group the moving objects. However, Figure 6.11 (b) shows that the GMI-tree groups the moving objects vertically (which means it considers the vertical grouping through the additional link of the connectivity tree (beside the horizontal methodology)) much higher than the Basic-Index. Here, Figure 6.11 (b) illustrates that the GMI-tree is successful in grouping the moving objects that are horizontally and vertically close to each other in each wing (based on the graph definitions of the cute node and the cycle connection). Moreover, Figure 6.12 shows that the GMI-tree significantly groups the moving objects in each wing without any cross-over between the building wings which is considered as false hits.

In addition, we examined the construction of the GMI-tree. We can note from Figure 6.13 (a) that the construction cost is no different from that of the Basic-Index. The basic point here is that with the new methodology in indexing



(a) Number of nodes that horizontally grouped the moving objects

(b) Number of nodes that vertically grouped the moving objects

FIGURE 6.11: Horizontal and vertical grouping

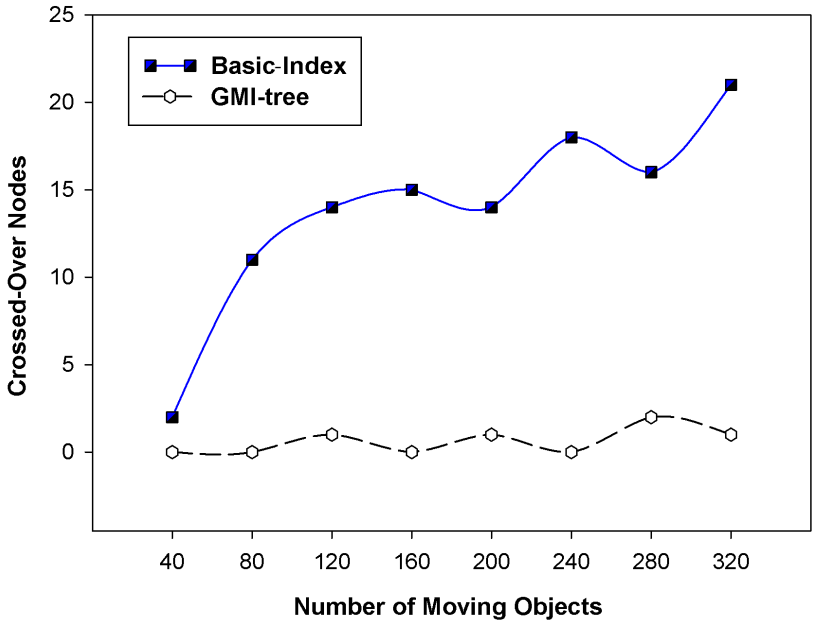
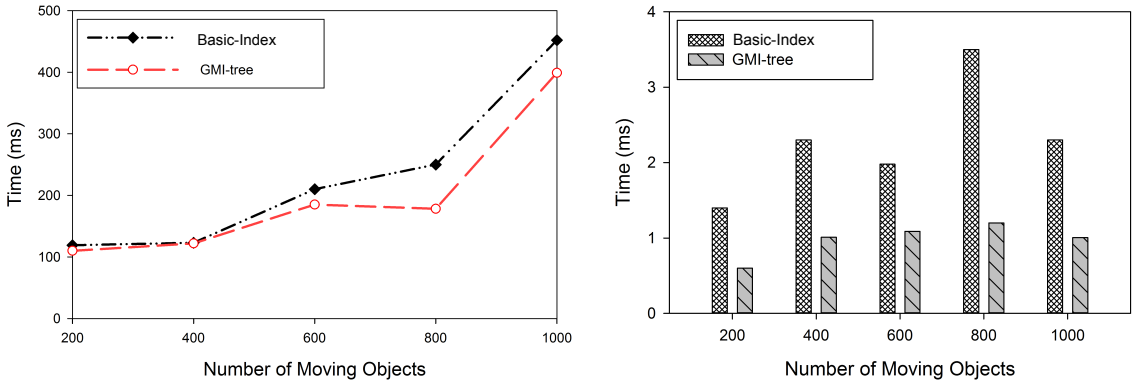


FIGURE 6.12: Crossed-Over Nodes between the building wings

the mobile objects on the basis of two links of the indoor graph, we find that construction is still good. In fact, in most cases, the construction costs of the GMI-tree is less than that of the Basic-Index. For example, in the case of 800 objects, the GMI-tree is decreased by around 28%.

However, for the search performance, we can also see a big improvement in our



(a) Construction Performance

(b) Search Performance

FIGURE 6.13: Construction and Search Performance

new index structure. The development that resulted from the search performance can be explained by as follows: In the GMI-tree, the moving objects are separately grouped based on their section through two links of the connectivity tree which makes the search more likely to be faster in each wing. For example, in Figure 6.13 (b) shows that in the case of 600 densities, the search cost is reduced by over 44%.

Therefore, the proposed index structure Graph-based Multidimensional Indoor-Tree can successfully produce a reliable and robust tree index for multi-floor indoor spaces.

6.3 Chapter Summary

This chapter addresses the challenge of indexing moving objects as a multidimensional grouping regardless of their exact locations in multi-floor indoor environments. It is clear that the exact positions of moving objects in indoor spaces are

not needed. The indoor environments are related to the symbolic/notion of cellular space which contains restriction entities (e.g. walls) that play important roles in restricting an object's movements. Hence, the indoors locationing is based on the symbolic of the cellular space in order to locate them. It is clear that in some interiors of buildings, the locationing of the moving objects can be based on their section/wing. Therefore, this chapter presents a new index structure that considers multidimensional grouping of mobile objects in the multi-floor indoor space. The basic idea is to determine the indoor spaces that can feasibly be considered for multidimensional grouping. The indoor environment is a connectivity environment where the rooms/venues are connected with each other through doors. Therefore, this chapter takes advantage of the graph that results from the indoor connectivity. If the graph represents our proposed graph elements, then the indoors will be considered to be constructed as multidimensional for each section/wing (Multidimensional Indexing-Applicable Indoor Spaces). The grouping of the moving objects is based on their adjacency in the same section horizontally and vertically.

CHAPTER 7

Indexing Moving Objects in Outdoor Cellular Space

Chapter Plan:

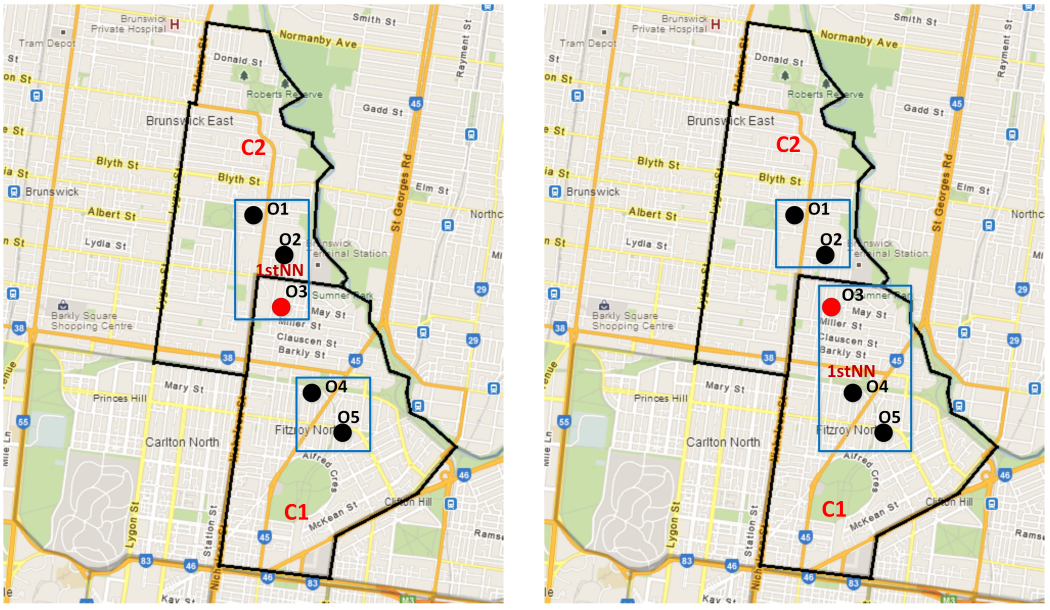
Publications and Submissions:

- Sultan Alamri, David Taniar, Maytham Safar, and Haidar Al-Khalidi. "Tracking moving objects using topographical indexing". *Concurrency and Computation: Practice and Experience*, pages n/an/a, 2013. ISSN 1532- 0634. doi: 10.1002/cpe.3169.

This chapter extended the regional/cellular indexing in indoor spaces to be applicable in topographical outdoor spaces. Many applications involving moving objects concentrate on the area or the cells that contain the moving objects, where exact locations are not important. The locations of moving objects are frequently and continuously changing, thereby producing a high volume of updates [18]. However, with adjacency-based applications, the updates are performed only when the moving object leaves its territory cell and checks into new cell. An example of an adjacency-base application, is the taxi systems in many cities around the world that distribute city areas as cells already fixed by a wide wireless network system [33, 34]. When a client contacts the taxi center to request a taxi, most of these taxi centers check the location of the client and then check the available taxis in that cell and send a request to the first one in the queue. If there is no taxi in that cell, the taxi center will look at the adjacent cell and send a request to the first available taxi. As is clear, the exact location of the taxi is not important in this application, since the cells are the key points.

Moreover, this is evident in the city council areas and suburbs, where requests are usually internal (the same territory), or in some cases from adjacent territories. Therefore, we can notice that the lazy update is used in this application, where inside the cells the location is not updated; the updating is performed only if the taxi checks out of a certain cell and checks in to a new cell. Tracking the moving

objects in this application is not useful since the moving objects can supposedly reach any place inside the cell in a very short time. Other examples are item location applications (e.g. search of individual or items), fleet tracking applications and many others where the locations are determined as regions/cells, not as exact locations. Figure 7.1 illustrates this issue using the metric distance structure and cellular-based structure.



(a) The $1stNN$ of O_3 in the metric-based structures is O_2

(b) The $1stNN$ of O_3 in the cellular-based is O_4

FIGURE 7.1: An example of using metric structures compared with cellular-based applications

This chapter proposes an index adjacency-based structure for moving objects in outdoor environments. Our work focuses on the construction of the moving objects based on the notion of cellular space. The key idea of our moving objects indexing is to take advantage of the adjacency and the connections within topographical outdoor environments. Therefore, we obtain the optimal representation of the outdoor cells and deal with the outdoor cells environment based on adjacency and connections. The adjacency index method can enable us to answer the spatial queries more efficiently. The uniqueness of our data structure is that we do not

index the regions/cells as containers of the moving objects such as [13, 25]. We index the moving objects by grouping the moving objects in the same cell first, and if a split is needed, we group them with moving objects in the adjacent cells. Therefore, we argue that the best index for outdoor cell environments is based on adjacency (connections), which can serve the following query types: spatial queries, topological queries and adjacency queries [22, 24].

7.1 Topographical Outdoor-tree (TO-tree)

This section presents our proposed index structure: Topographical Outdoor-tree (TO-tree), which facilitates spatial queries in outdoor cellular applications and also reduces the update cost. Then, it illustrates the maintaining algorithms (insert, delete and update algorithms). Table 7.1 explains the notations used throughout this chapter.

7.1.1 Topographical Outdoor Cellular Space

The main difference between outdoor Euclidean space and outdoor cellular space is the reliance on the geometric representation of spatial property [25, 54]. A query in a typical outdoor space is given with coordinates such as $(x_i; y_i)$. On the other hand, queries in cellular space usually rely on a cellular notation such as: "return the moving objects in cell (or region) 85". To further explain, in outdoor geometric spaces, a representation as coordinate n-tuples will be done for both the located objects and locations (represented as points, volumes, and areas) [19, 26, 51]. On the other hand, the location of a moving object is represented by abstract symbols

TABLE 7.1: Notations used throughout this chapter

Notation	Definition
C	Cell
$Ci\ \chi\ Cj$	Ci is connected to Cj
$ChildPTR$	The pointer to the child node
$d_e(q, O)$	The Euclidean distance between query q and object O
$LNPosition$	The MBR position of the node (determined by the center of the related cells)
$NPosition$	The position of the node determined by the leaf nodes $LNPosition$
N	Leaf node
$NX(Oi)$	Next cell of object Oi
MBR	Minimum Boundary Rectangle
MIN	Minimum values
$O_i \overset{in}{\rightarrow} C_j$	O_i located inside C_j
O_n	The maximum capacity of a leaf node
PTR	The pointer
$(x; y)$	x and y coordinates of a point
η	Set of moving objects

or cells in cellular space. In our outdoor adjacency data structure, we use both Euclidean space and cellular space. Figure 7.2 represents outdoor cellular space. We propose that the plane be divided into a cells-based Voronoi diagram (VD) [25]. The Voronoi diagram partitions the Euclidean plane into a set of convex polygons (cells), named Voronoi cells or Voronoi regions [19, 51]. Each Voronoi cell has a generator (point in center), where the Euclidean distance d_e from any object in its cell is smaller than that of any other cell's generators [51, 103]. The formal definition of Voronoi cells in our work is as follow:

Definition 7.1: Given a set of moving objects $\eta = (O_1, O_2, ..., O_n)$, n Integer In, the Voronoi cell of C_i, $V(C_i)$, is a universal set of points that fulfil the following:

$$\forall\ d_e(O, C_i) <= d_e(O, C_j), \text{ where } i, j \in In, i \neq j$$

The Voronoi diagram for η is:

$$\bigcup_{i=1}^{n} V(C_i)$$

The function $d_e(O, C_j)$ denotes the distance from a point O to C_j, which can be calculated by:

$$d_e(O, C_j) = \sqrt{(x_O^2 - x_{C_j}^2) + (y_O^2 - y_{C_j}^2)}$$

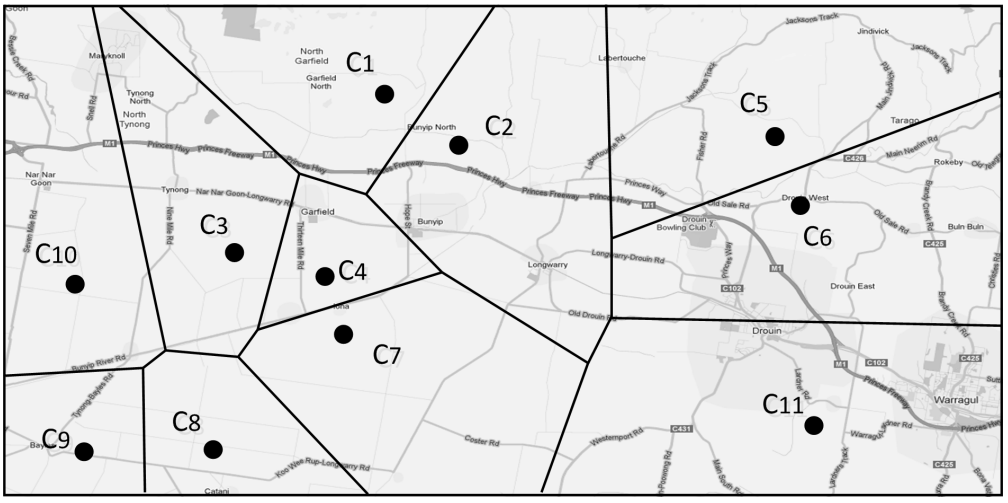


FIGURE 7.2: Outdoor cellular space based on a Voronoi diagram

Definition 7.2: Let C be a set of cells $C = \{C_1, C_2, ..., C_n\}$, and a moving object O_x, C_i is *an adjacent cell* to C_j ($C_i \ \chi \ C_j$) iff $O_x \xrightarrow{in} C_j$ and next location of O_x $NX(O_x)$ is C_i.

Next, this chapter introduces the *Connection Cells Algorithm* which is responsible for determining the topographical cell levels for an object. Therefore, basically it retrieves the adjacent levels for the targeted cell.

Algorithm 10 Connection Cells Algorithm

```
1: /* check the inserted to C_i with set of C. */
2: /* Connection Cells Algorithm will return the nearest level */
3: for C_i adjacent cells AC.x do
4: |  if AC.x χ C_i then
5: |  |  R = AC.x
6: |  end if
7: end for
8: Return R // the nearest level of C_i
```

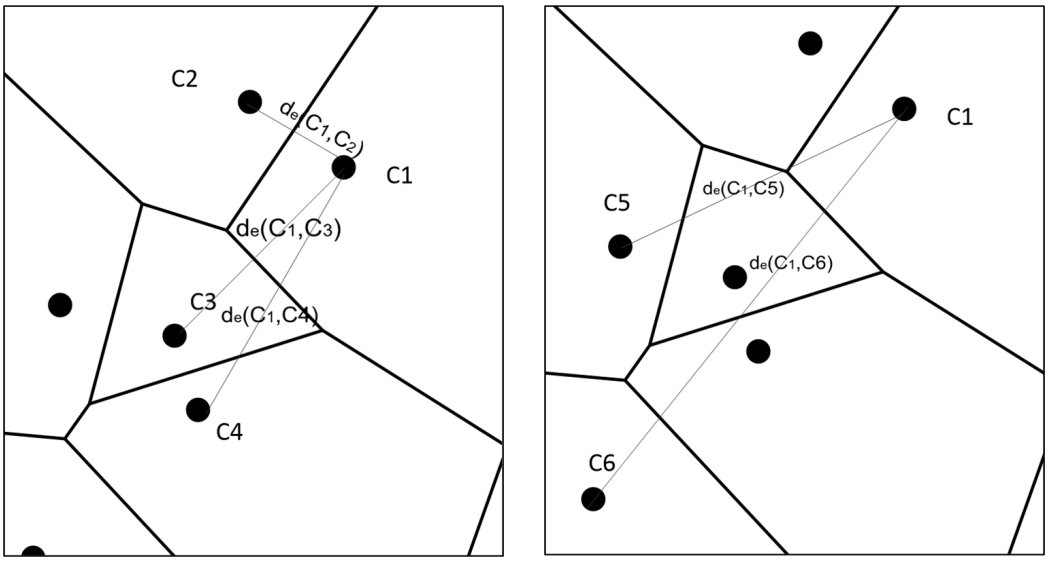
Figure 7.2 illustrates a cellular space which contains 11 cells (e.g. C_1, C_2,...,C_{11}). Note that each cell has a first level adjacent and a second level adjacent and so on. For example, for C_4 the first level adjacent cells are C_1, C_2, C_7 and C_3. The second adjacent level are C_8, C_{10}, C_5, C_6 and C_{11} (which requires crossing another cell to reach the second adjacent level). Note that the cellular application system is already fixing any case of overlapping between the cells' coverages.

7.1.2 Adjacent Level Algorithm

In our data structure, we argue that treating the outdoor environment as cellular is essential for many applications. In many cases, the outdoors is usually based on coordinates (x_i; y_i), whereas in our case, we treat the outdoor space as cellular notations. Besides the previous connection which is explained in the previous section, the outdoor cell can have more than one adjacent cell; therefore, choosing any of its adjacent cells to be grouped with is not essential. Thus, we need to establish a pre-computed outdoor Adjacent Level Algorithm, to narrow down the best adjacent cells for the purpose of insertion or deletion.

The outdoor cellular applications have a fixed distribution of the cells, where the center location-coordinate of each cell is known by the system. The Adjacent Level

Algorithm is intended to calculate the distance between target cells (Voronoi cell's generators) with each of its adjacent cells and to choose the MIN distance (see Figure 7.3). Our objective is to use the result from the Adjacent Level Algorithm to construct the tree and maintain the insertion and deletion operations. It is clear that the calculation of the distance between the cell centres can assist in narrowing the number of suitable nodes where the *Connection Cells Algorithm* is insufficient. This is different from the typical metric indexes where comparisons are based on Euclidean space so the objects will be grouped together base on MBR least enlargement [8, 10]. Moreover, other traditional data structures such as Hilbert R-trees and their followers base their comparison on the Hilbert value so the objects are grouped together in the $MBRs$ based on LHV (Largest Hilbert Value) [99]. Algorithm 11 illustrates the outdoor Adjacent Level Algorithm.



(a) Calculating the distance to the first level adjacent (b) Calculating the distance to the second level adjacent

FIGURE 7.3: Adjacent Level Algorithm determining the nearest Voronoi cell at each level

Algorithm 11 Adjacent Level Algorithm

```
1: /* check the Objects O cell C_i. */
2: Call Connection Cells Algorithm CCA (C_i)
3: for CCA.x(C_i) do // for all adjacent cells to C_i
4: |  if d_e(C_i, C_x) <= d_e(C_i, R) then
5: |  |  R = C_x
6: |  end if
7: |  x++;
8: end for
9: Return R // nearest connected cell.
```

7.1.3 Tree Creation

Using the adjacency method that was mentioned in the previous section, the data structure will group the entries based on their adjacency cells at the current time. In our data structure, we start by grouping the objects inside the same cell. Splitting will be performed *internally* to group it with the objects in the same cell in the case of any overflowing MBR. In the case of underflow, we group the moving object with one of its adjacent cells. The idea is to start to retrieve the adjacent cells of the target cells using the *Connection Cells Algorithm*. Then the *Adjacent Level Algorithm* is used in order to start narrowing down to the most suitable one; this algorithm checks the nearest adjacent cell by calculating the distance between the cells' centers (Voronoi generators) which is already known in the system. Note that the objects' locations are not determined by the system; the system recognizes only the locations of the cells' centers which will be used in the *chooseleaf* algorithm.

Non-leaf nodes contain ($NPosition$, and $ChildPTR$) where $NPosition$ is the position of the node determined by the leaf nodes $LNPosition$. For example, a non-leaf node has three leaf nodes N_i, N_j and N_x. $NPosition$ is the MBR position of N_i,N_j and N_x. $ChildPTR$ is the pointer to the child node. *Leaf nodes* contain

($LNPosition$, obj, $cell$ and PTR) where $LNPosition$ is the MBR position of the node which will be checked by the Adjacent Level Algorithm in order to choose the suitable leaf node. $LNPosition$ here will be determined by the center of the cells (Voronoi generators) that are contained in this leaf node. For example, a non-leaf node has three cells C_i,C_j and C_x. $LNPosition$ is the MBR that grouped the cell centers of C_i,C_j and C_x. The objects and the cells that are contained in the MBR at that time are denoted as obj and $cell$, and PTR is the pointer. Note that non-leaf nodes/leaf nodes contain at most O_n entries, which is the maximum capacity of the leaf. The Topographical Outdoor-tree is illustrated in Figure 7.4.

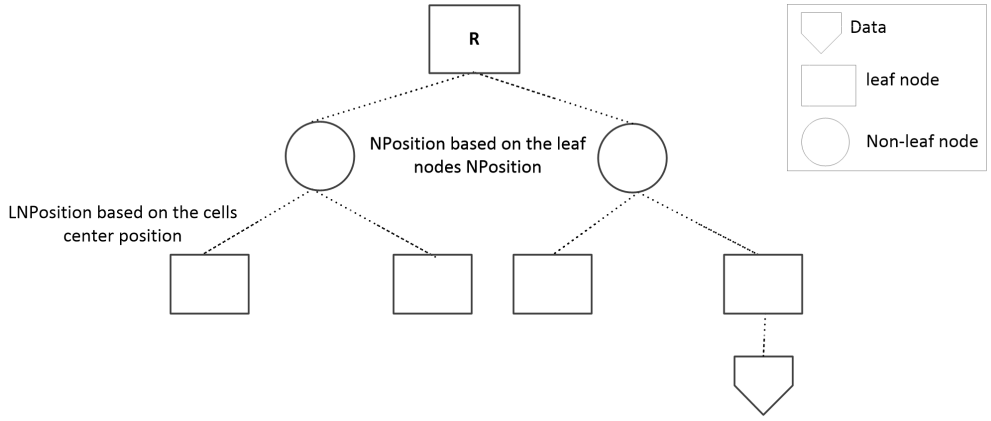


FIGURE 7.4: Summary of the topographical outdoor-tree or TO-tree

Using the sample data in Figure 7.5, suppose that the four $MBRs$ are clustered into larger two $MBRs$, where moving objects $= a, b, c, d, e, f, g, y, h, r$ and x and $M = 3$ and $m = 2$. The TO-tree is shown in Figure 7.6. Note that the locations of the objects are shown only for illustrative purposes.

7.1.4 Maintaining Operations

In the Topographical Outdoor data structure, when a node overflows, it has to be split immediately but must be grouped within the same cell (without any attention

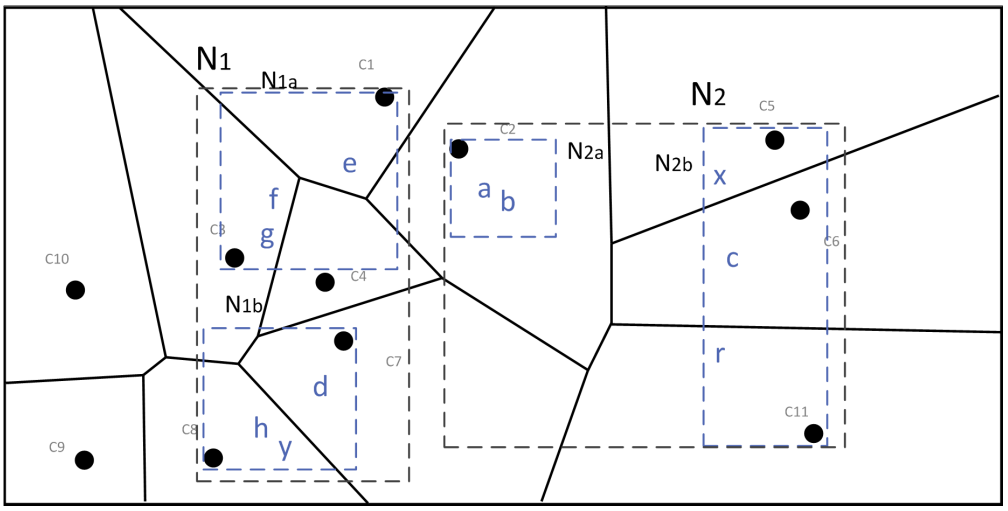


FIGURE 7.5: The MBRs grouping based on the adjacency

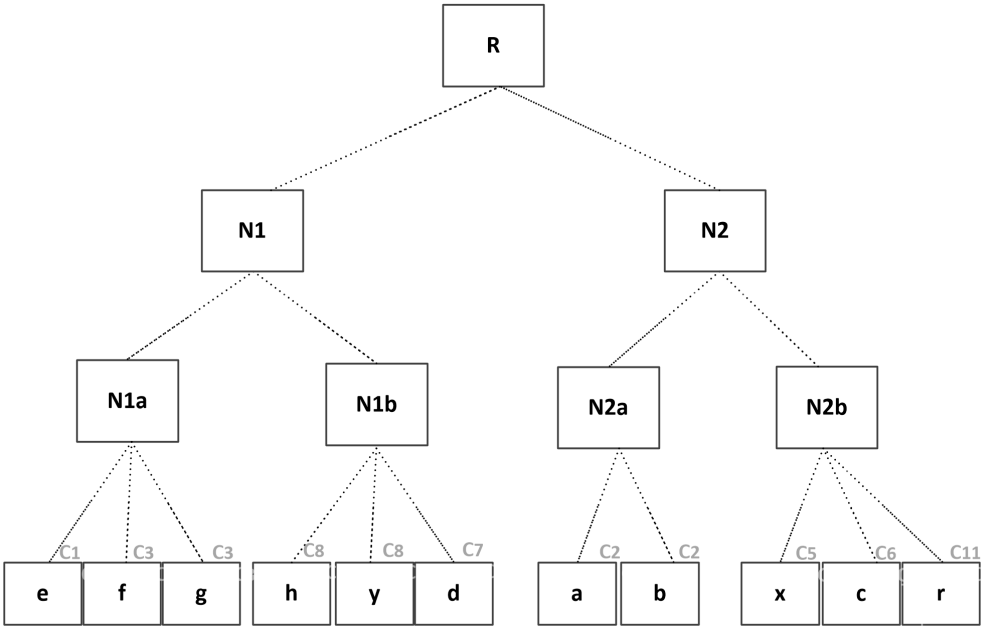
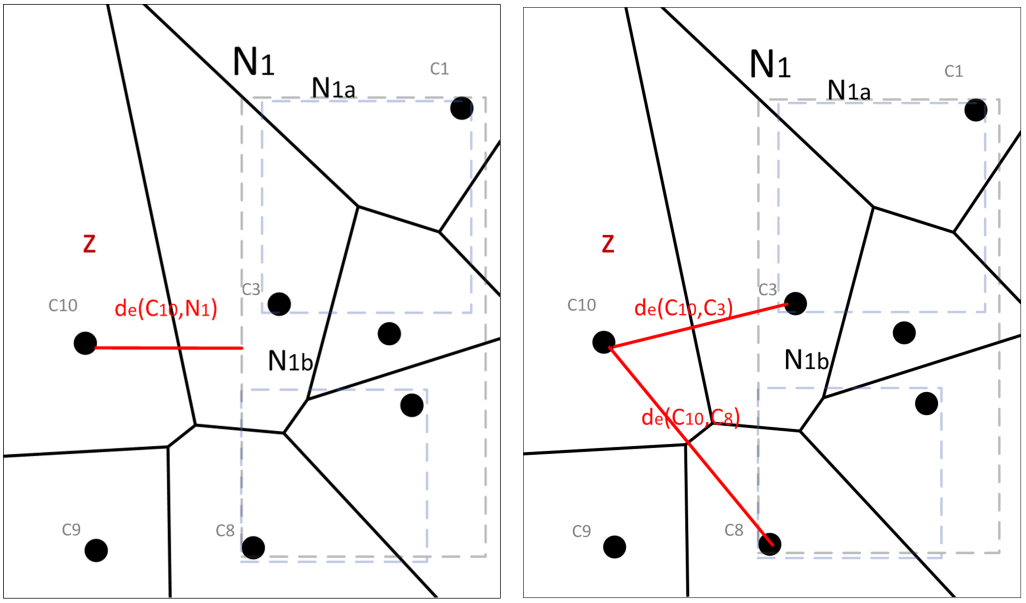


FIGURE 7.6: The TO-tree based on data in Figure 7.5

to the object location inside the cell, because the object movement is not tracked here). In the case where the number of moving objects in a certain cell is fewer than the minimum limit, (the cases of the underflow issue), the chooseleaf algorithm is responsible for grouping the moving objects with the suitable adjacent cell. The chooseleaf node algorithm starts to check the $NPosition$ of the non-leaf nodes by checking the distance from the cell's generator (in the targeted cell) to the MBR of the non-leaf nodes (see Figure 7.7 (a)). Note that when inserting objects to a



(a) Checking the suitable non-leaf node

(b) Calculating the distance to the first level adjacent to choose the suitable leaf node

FIGURE 7.7: The chooseleaf algorithm for inserting object "z" to C_{10}

target cell, the target cell center is used, not the object location. For the leaf node, the chooseleaf algorithm starts by calling the Adjacent Level Algorithm which will retrieve the adjacent cells to the target inserted cell (the first level neighbours) by the Connection Cells Algorithm. Then, the Adjacent Level Algorithm is used to check the nearest distance from the center of the targeted cell to the adjacent cells.

For example, based on data in Figure 7.5, assume that an object "z" is inserted to C_{10}. We note that C_{10} has no other objects; therefore, the chooseleaf is responsible for grouping the moving objects with the suitable adjacent cell. The chooseleaf algorithm steps are illustrated in algorithm 12 and Figure 7.7. Note that if the Adjacent Level Algorithm results are equal, (returned similar distance), the algorithm will choose the node that has more neighbors (connections). The insert algorithm steps are illustrated in algorithm 13. The insertions of "z" to C_{10} and "w" to C_2 are shown in Figure 7.8.

Algorithm 12 ChooseLeaf algorithm

```
/* Return the leaf node that has the inserted n */
Set N to the root node
if N is non-leaf node then
    for each non-leaf nodes N_i do
        check d_e(C_x, N_i),
        Node(N_i) has MIN Value
        Choose the (N_i,ptr)
    end for
end if
if N is leaf node then
    Call Adjacent Level Algorithm(C_x)
    Node(N_x) has MIN Dist
    Choose the (Nx, ptr)
    if N_x.d_e = N_y.d_e then // (overlapping case )
        Choose the Node that has fewer entities
    end if
end if
```

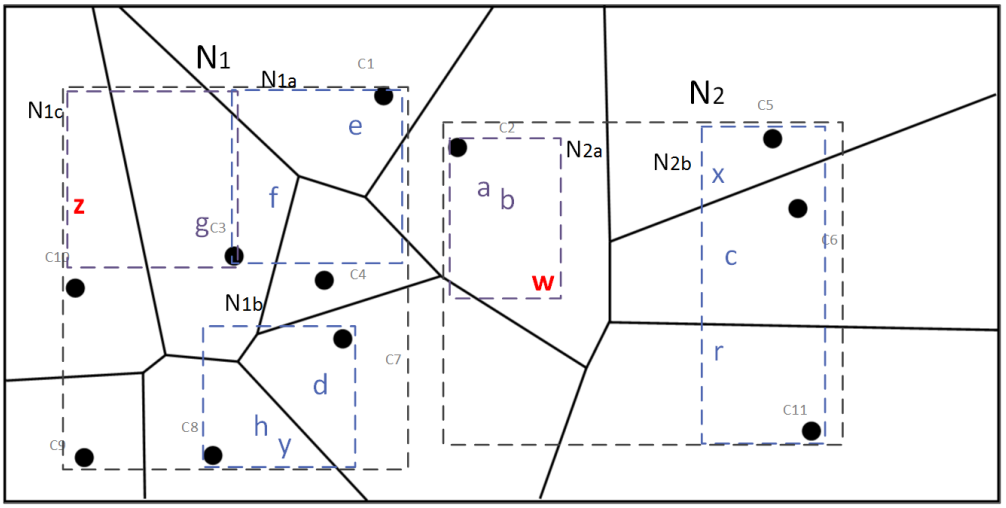


FIGURE 7.8: Inserting "z" to C_{10} and "w" to C_2

In the deletion, when the moving object is no longer in the plane (or checked out), this means that it has been deleted from the system. With the delete algorithm, we check for the underflow node instead of the overflow node (see algorithm 14). If a node is underflow, we need to get the sibling nodes that are based on the connection to participate in resolving the underflow. The deletion algorithm will be illustrated and can be explained as follows: The algorithm first calls the

FindLeaf algorithm, after discovering that N_i is underflow, which will determine the sibling connected Node based on the Adjacent Level Algorithm. Then the remaining entities are inserted into that Node. Hence, in the deletion, if underflow occurs, we deal with the adjacent cells.

Algorithm 13 Insert algorithm

```
1: /* Input: O is the entry to be inserted. */
2: procedure INSERT(O into C_i)
3:     Insert O into C_i
4:     if N_i overflows then
5:         Split N_i
6:         Group O in N_ia
7:         Update NPosition,LNPosition at N_i, N_ia and N_ib
8:     end if
9:     if N_i underflows then
10:        call ChooseLeaf to find the suitable leaf node
11:        Update NPosition,LNPosition
12:    end if
13:    if node split propagated to the root node and root node to split then
14:        Create a new root node with new resulting child nodes
15:    end if
16: end procedure
```

Algorithm 14 delete algorithm

```
1: /* Input: O is the entry to be deleted. */
2: procedure DELETE(O from C_j)
3:     delete O from C_j
4:     FindLeaf to find the leaf node N_j
5:     if N_j underflows then
6:         call ChooseLeaf to find the adjacent suitable nodes N_i
7:         re-insert O_j into N_i
8:         if N_i overflows then
9:             HandleOverflow(N_i, O_j)
10:        end if
11:    end if
12:    Update NPosition,LNPosition at N_i
13:    if node split propagated to the root node and root node to split then
14:        Create a new root node with new resulting child nodes
15:    end if
16: end procedure
```

7.2 Experimental Results and Performance Analysis

In this section, we present our experimental results to evaluate the proposed index TO-tree and compare it with the TPR-tree [6] which is the most common metric data structure for moving objects in outdoor spaces. The basic idea is to compare our data structure with a metric data structure (a data structure that based on distance grouping) and see the results. The experiment was carried out on an Intel Core i5-2400S processor 2.50GHz PC, with 4 GB of RAM running on 64-bit Windows 7 Professional. The TO-tree data structure has been implemented in Java. The data set size ranges from 100 to 9000 moving objects on the outdoor plane.

In this experiment, due to the lack of real data, we use synthetic datasets of moving objects on a plane of outdoor space. We generated the location of the objects based on the number of cells. As mentioned earlier, the movement of the objects will not be tracked inside the cells and we update the location of the object when it moves out of the cell and checks into a new cell.

We investigate the following: in addition to measuring the tree construction and insert performance, we calculate the false hits of the moving objects in the adjacency application for both of the TPR-tree and the TO-tree which illustrate the significant of using the topographical index compare with any metric data structure. For the former, the execution time is measured for each test. For the query performance and maintaining operations tests, we compared the proposed index efficiency for different numbers of moving objects and different numbers of

cells. Note that the operations are performed several times and the average is calculated. The parameters used are summarized in Table 7.2.

TABLE 7.2: Parameters and Their Settings

Parameter	**Setting**
Number of moving objects	(100 to 9000)
Outdoor space	11, 50, 100 cells
Operations	Construction, insert and false hit
Dataset	synthetic

We calculate the approximate false hits of moving objects in an adjacency application for both the TPR-tree and TO-tree. The moving objects that are grouped incorrectly (false hits) are illustrated in Figure 7.9. Moreover, we tested the grouping for 100 to 1000 moving objects. We found that by using the TPR-tree in adjacency applications and topographical applications can result many objects being forced and grouped based on the distance method. In some cases, in excess of 19% of the moving objects can be grouped based on the distance, which is not useful for topographical applications. Figure 7.9 shows that using the TPR-tree, a number of objects can be forced on distance construction and this might produce a false result.

Tree construction costs are illustrated in Figure 7.10. We generate moving objects from 1000 to 9000; then we use the structure trees to construct the moving objects together based on the TO-tree connectivity and TPR-tree. As the number of moving objects increases, the construction cost increases accordingly. This behaviour is natural in an indexing tree where, with the increase in the number of the moving objects, the construction cost will increase, and the number of the nodes and splits will increase. Moreover, we notice in Figure 7.10 that the TO-tree construction cost is lower than the TPR-tree construction cost in most cases. The aim here is to illustrate that with the new construction algorithms in our

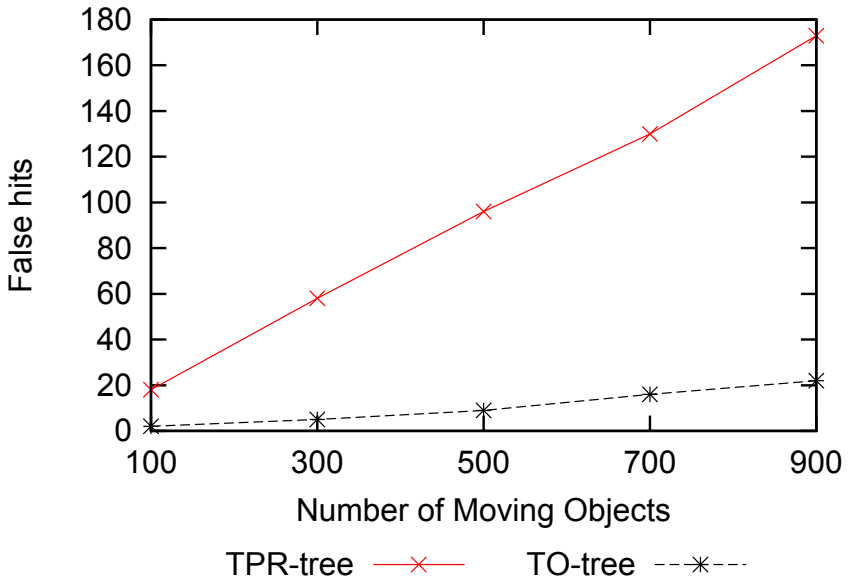


FIGURE 7.9: High False hits using metric structure on topographic applications

TO-tree, the construction cost is still very acceptable compared to the common metric construction. In our data structure, the grouping of the moving objects will be based on the connectivity between the cells, whereas the TPR-tree or any metric structure will calculate the distance in order to group the moving objects. Moreover, the fact that the TO-tree does not track the moving objects inside the cell will reduce the costs. Therefore, we can say that the construction cost in the TO-tree is efficient.

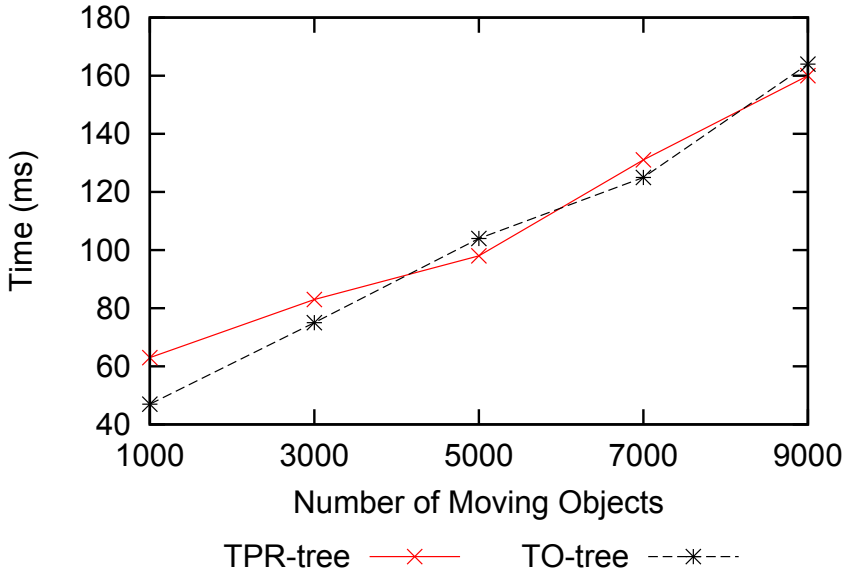


FIGURE 7.10: Construction costs

Figure 7.11 demonstrates the effect of the number of cells. Here we evaluate the insertion of the moving objects into 11-cell, 50-cell and 100-cell densities. We notice that with the increase in the number of moving objects in each different cell density, the insertion costs increase. However, we can see that with the increase in the cell densities, the insertion costs are comparable. For example, in the 900-objects density, there is a cost increase of around 16 ms between 11 cells and 50 cells. Therefore, we can say that the increase in the number of the cells does not impact significantly on tree insertion costs.

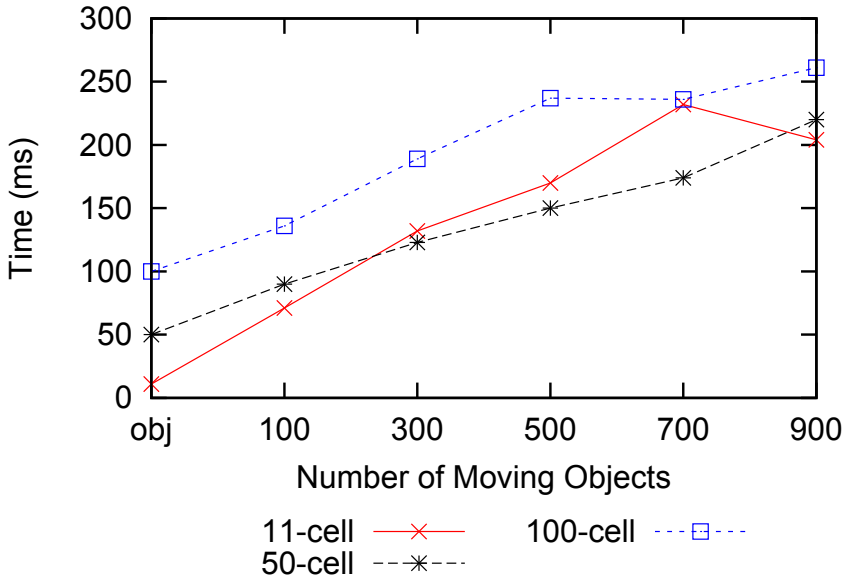


FIGURE 7.11: Illustrates the effect of the cells number

In our evaluation experiments, we evaluated the side that shows the importance of the adjacency structure and how a structure such as a TPR-tree can produce many false hits in an adjacency application. Moreover, we compared the construction costs for both the TO-tree and the TPR-tree for different numbers of moving objects. Also, we evaluated the existing parameters in our index tree in order to test the proposed index (the effect of the number of cells). We summarise the results as follows: First, the effect of increasing the number of moving objects and the increasing of the cell number have no explicit influence on the tree constructions. Second, the TO-tree creates an optimal data structure for the adjacency

applications and performs more efficiently than the metric structures such as the TPR-tree. Third, the TO-tree can successfully produce a reliable and robust tree index based on the novel idea of topographical connections and adjacency.

7.3 Chapter Summary

This chapter addresses the challenge of building an index data structure that is appropriate for topographical outdoor applications. The measurement of topographical space is different from that of typical outdoor space that is based on Euclidean space or a spatial network. Topographical space is related to the notion of cellular space. Our tree structure is based on adjacency and cell connections which allows us to answer spatial queries more efficiently for topographical applications. Therefore, we obtain an optimal representation of the outdoor cellular space in order to handle the outdoor cells as adjacency and connections-based areas. Therefore, we argue that the best index for outdoor cellular spaces is based on adjacency (connections) between the cells, which can serve the following query types: spatial queries, topological queries and adjacency queries. In addition, our data structure uses the Adjacent Level Algorithm which facilitates the calculation of the distance between target cells (using the cells center) with each of its adjacent cells and chooses the MIN distance (nearest adjacent cell). To the best of our knowledge, using adjacency/connectivity in *ChooseLeaf* comparison for topographical space is unique to our tree data structure. Extensive performance studies were conducted and results indicated that the TO-tree is both robust and efficient for targeted queries. The results show that using any metric data structures such as a TPR-tree can produce inaccurate construction which may lead to incorrect query results.

CHAPTER 8

Directions and Velocities Indexing for Moving Objects

Chapter Plan:

Publications and Submissions:

- Sultan Alamri, David Taniar, Maytham Safar: "Indexing moving objects for directions and velocities queries". *Information Systems Frontiers* 15(2): 235-248 (2013).

In this chapter, we propose a novel index structure that includes the direction queries and the velocity queries for moving objects. The data structure in TPR-tree and its successors are based on the space domain, without considering any distribution of the moving objects' direction and velocity. This work focuses on the construction of the moving objects based on the spatial, direction and velocity domains. The key idea of our moving objects indexing is to cover new types of queries, namely directions and velocity queries (DV queries), which are not covered by the traditional moving objects indexes (more details in Section 8.2). Moreover, the existing queries in the traditional moving objects indexes will be supported. Insertions are performed in traditional TPR*-tree groups top-down through new direct access tables, while deletions are performed in groups bottom-up via the lookup-table. Note that the insertion and deletion techniques are cooperating with a hash-based table which enables fast utilization of main memory [60]. Also this chapter compared the performance of our index with that of the TPR-tree and its successors on queries related to direction and velocity.

8.1 Background

Several recent works have focused on the moving query and consider only the movement of the query [19, 50, 51]. TPR-Tree is the most common moving-object index based on the R-tree concepts. Three types of queries are supported by TPR-Tree and its successors to retrieve points with positions within specified regions [6]. The three query types are distinguished according to the regions that are specified by the query. The Timeslice query Q = *(R, t)*, indicates objects that intersect a given a hyper rectangle R at time point t. The second type is the window query: Q = (R, t_i, t_j), $t_i < t_j$ which indicates objects that intersect a given rectangle some time from t_i to t_j. The third type is the moving query: Q = (R_1,R_2,, t_i, t_j), $t_i < t_j$ obtained by connecting R_1 at time t_i to R_2 at time t_j. In other words, these are objects that intersect a given moving rectangle at some time between t_i and t_j.

There is a new type of query which we term Motion-Dependent Query. These queries are based on the object's velocity and direction. *Velocity queries* are the ones that show the relevance of different speeds of each moving object. There is no doubt that each moving object will have a different velocity range for its own movement. For example, a transport company has a number of moving points (such as taxis); hence, the velocity range between the moving objects will be different. Moreover, the velocity range of each single object trip will be different as well. Therefore, object velocity queries can be classified as *similar range velocities* or *different range velocities*.

On the other hand, *Directions queries* are queries that depend explicitly on the moving objects' directions, whether or not they have a similar direction.

Therefore, object direction queries can be classified as *similar directions* or *different directions*. *Similar directions* means that the moving objects on the map are moving on the same road and in the same direction (e.g. north, south..etc.), whereas *different directions* means that the moving objects on the map are moving on different roads and in different directions (Figure 8.1). Next, velocity/direction queries will be illustrated by examples.

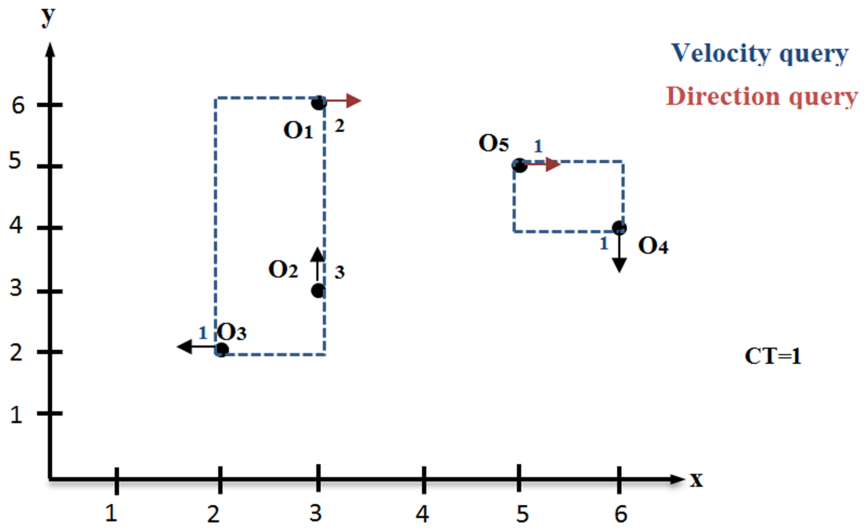


FIGURE 8.1: Five moving objects on two-Dimensional Data

q_i. Return all vehicles in the suburb of Clayton that are moving within a similar velocity range. (similar velocity).

q_{ii}. Taxi management center wants to return all taxis on the Monash Freeway that are moving in northerly direction. (similar direction).

8.2 Problem Setup

The TPR-tree and its successors are constructed based on the space domain, without considering any distribution of the moving objects' direction and velocity. In

this section, we employ the *DV* queries in a TPR-tree base, in order to analyze the query performance. Consider the example in Figure 8.2, where three leaf nodes A=$\{O_1, O_2\}$, $B = \{O_3, O_4\}$ *and* C= $\{O_5, O_6\}$ *at time* $t = 0$.

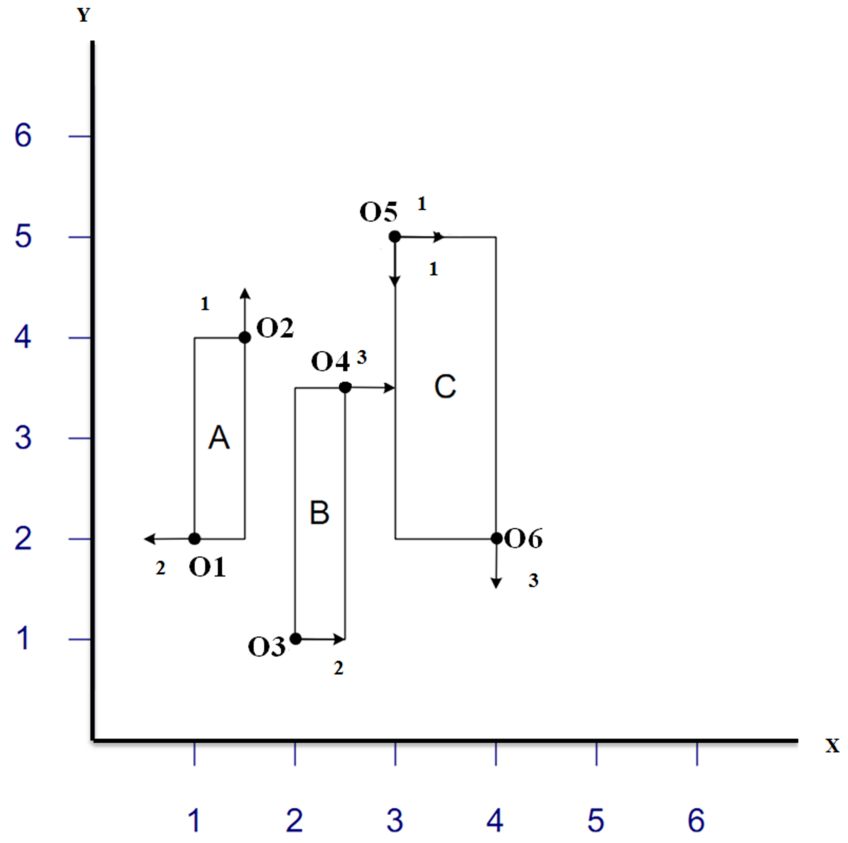


FIGURE 8.2: Three leaf nodes A, B and C at time $t = 0$

Assuming that we run the next query q_i at $t = 0$, "Return all objects that are moving in *North* direction", how will the TPR-Tree process this query? The TPR-Tree index has been designed to fulfil the timeslice queries, window queries and moving queries with efficient performance. Performing direction and velocity queries with the current TPR-Tree will force the index to search through the whole index until the direction and velocity queries' conditions are obtained [2, 66]. In Figure 8.2 the q_i will be processed as follows:

- Check leaf node A and notice that the O_2 velocity parameter indicates the north direction to (first objects obtained); however the O_1 velocity parameter

does not indicate the north direction. Note that the search process will not stop at this stage (1 node accessed).

- Check leaf node B and note that O_3 and O_4 velocity parameters do not indicate a north direction for both of them (both moving east (1 node accessed).
- Check leaf node C and notice that O_5 and O_6 velocities parameters do not indicate a north direction for both of them (O_5 moving south east, o_6 moving south (1 node accessed).

The total cost incurred is therefore h + *3 nodes accessed* where h is the height of the TPR-tree. Note that 3 = the total number of leaf nodes.

Let us consider the next scenario. Suppose we are to track the locations of 500,000 cars in the Great Los Angeles area (population of about 4 million). Assume that these moving objects are distributed on 1000 leaf nodes for indexing (TPR-Tree based). An application runs a query (for the whole map, *absolute query)* to determine the cars heading north once every 10 minutes on average. It should be noticed that in extreme cases when the query frequency is very high, it is easy to see that each DV-query will incur an extreme cost. In our scenario, the whole tree (h+1000 leaf nodes) will be accessed every 10 minutes, which is the best case that can be obtained based on the query.

Thus, this method results in the deterioration of the TPR-tree, and is therefore not suitable for DV queries applications. Motivated by this, we improve the indexing strategy in [10] in addition to the bottom-up update in [104] to improve the DV query performance.

8.3 The DV-TPR*-tree

This Section discusses the structure of the DV-TPR*-tree, and provides the construction algorithm, illustrates the insertion and deletion algorithms, and describes the update algorithm and the query algorithm.

8.3.1 The DV-TPR*-tree Structure

To overcome the problem with the DV-queries, we kept the TPR*-tree intact, and build two-dimensional auxiliary structures to avoid unnecessary tree traversals. We call the resultant structure Direction and Velocity queries TPR*-tree (DV-TPR*-tree). The DV-TPR*-tree consists of multiple dimensions. The first dimension is the basic TPR*-Tree (Rs) [10]. Each moving object will store the velocities of objects along with their positions in the nodes. The next linear functions will be used to capture the present and future positions. Note that $\bar{x}(t_0)$ is the spatial position of $\bar{x}$ at some time t_0 and $\bar{v}$ is the velocity vector.

$$\bar{x}(t) = \bar{x}(t_0) + \bar{v}(t - t_0), \quad \text{where } t \geq now \tag{8.1}$$

On the other dimension, another linear Directions bucket structure is introduced. Each Direction bucket will hold objects that share the same direction. In other words, the moving objects will be constructed based on the direction only (Rd). Maximum non-leaf nodes in this dimension will be only *nine* values which are (*static , north , south, east, west , southeast , northeast , northwest and southwest),* whereas, the minimum number of non-leaf nodes is *one* (the dimensions

root). The upper and the lower boundaries of the direction $MBRs$ will be based on:

$$U^d = R^u(tref) + V^u(t - tref) \tag{8.2}$$

$$L^d = R^l(tref) - V^l(t - tref) \tag{8.3}$$

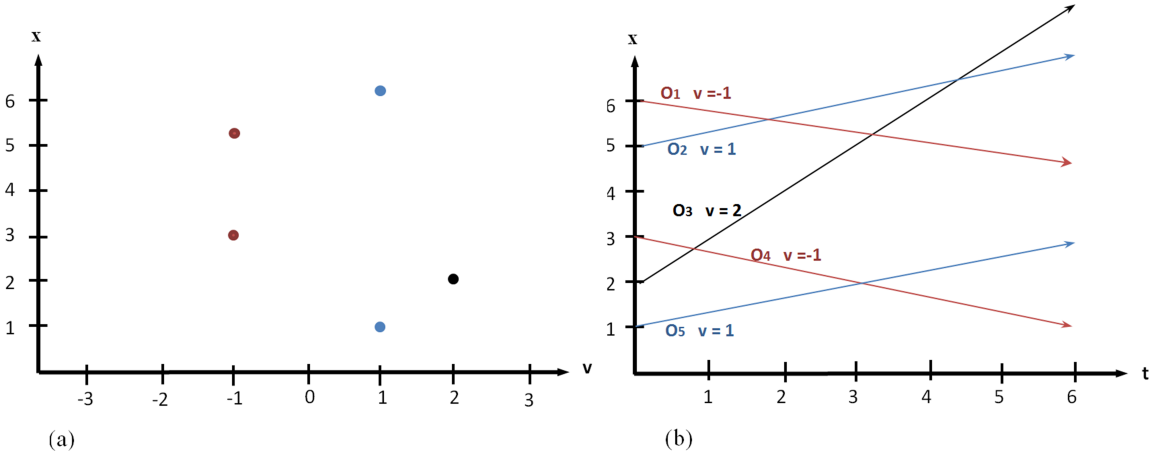


FIGURE 8.3: (a) Direction dimension at time 0 (CT=0), (b) Five moving objects on One-Dimensional Data (spatial dimension)

Where U^d is the upper boundary based on the direction and L^d is the lower boundary based on the direction. $tref$ denotes some reference time, R^u and R^l indicate the spatial upper and lower positions. Note that by subtracting the current spatial position of the bucket from the velocity of the moving object at R^l, lower boundary will keep tightening with movement. Moreover, V^u and V^l are R's velocity vectors for its upper and lower bound respectively. The Direction bucket is visited only when a DV-query is presented, which works as an auxiliary dimension to reduce frequent disk I/Os. The third dimension consists of two auxiliary table structures:

- A *lookup table* is adopted from [94] to map object *ids* to the leaf to which they belong. It prevents the expensive query on the tree and assists in locating the leaf node during any updating of the tree.
- A *velocity access table* is the second auxiliary structure, where any table's entry corresponds to one or more >=1 internal nodes of the index. An entry in the velocity access table is a three tuple of the form (nodes queue, velocity, nodesPTR) where nodesPTR is a pointer to the direction nodes themselves. Essentially, we obtain essentially a cache of all internal nodes by building the velocity access table which determines the nodes that are congruent in the velocity with less disk cost.

Note that the size of the velocity access table and the lookup table is only a small fraction of the index, because they store only internal nodes, without storing its children's bounding boxes. Therefore, the compact size of the velocity access table and the lookup table is negligible. Moreover, the axes x, v and t indicate the spatial, velocity and the time coordinates of the moving objects. Figure 8.4 illustrates the DV-TPR*-tree structure.

For simplification, we explain a one-dimensional data example. Consider the example in Figure 8.3 of five moving objects (O_1, O_2, O_3, O_4, *and* O_5) assuming that $m = 2$ and $M = 3$ in one-dimensional data. As shown, objects O_2 and O_5 are moving with velocity 1, whereas the velocity of $O_3 = 2$ and O_4 and O_1 =-1 (negative velocity indicates an opposite direction).

The main idea is that the moving objects will be scanned in sequence and inserted into the corresponding TPR-tree (direction dimension) pointed to by a bucket according to their direction. Note that the algorithm will already insert the

object in the spatial Dimension (Rs). Furthermore, the auxiliary tables' structures will be filled at the end.

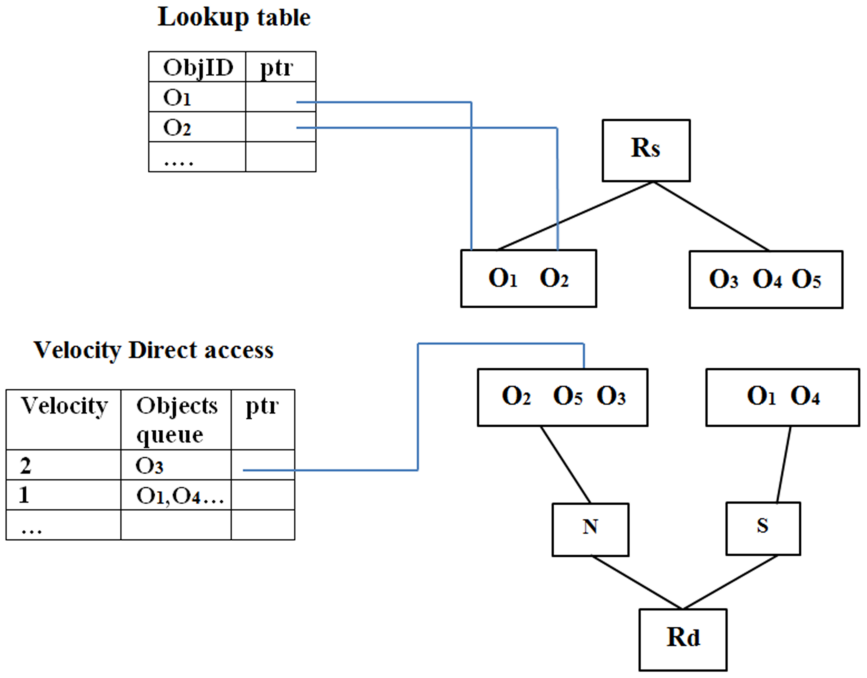


FIGURE 8.4: Summary of the DV-TPR*-tree structure

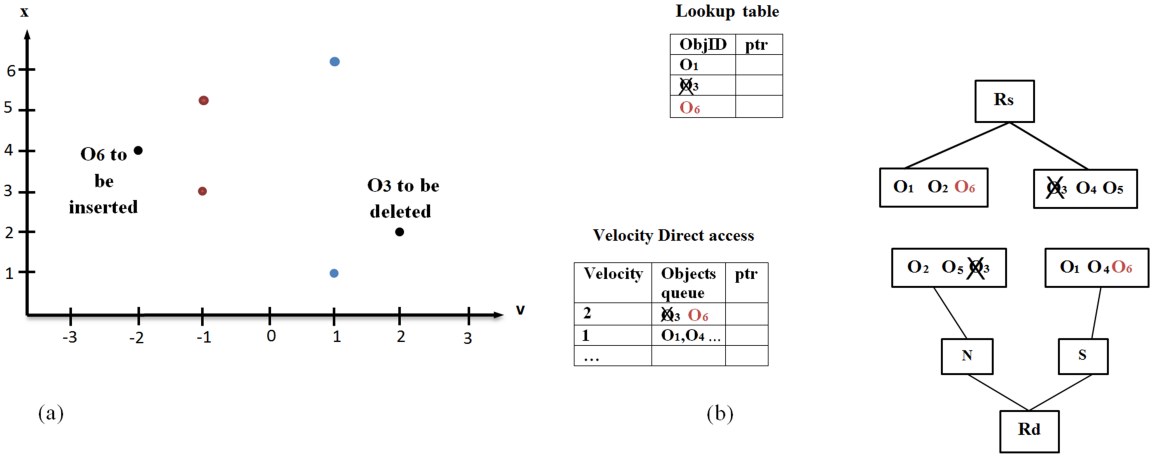


FIGURE 8.5: (a) object O_6 v = -2 to be inserted and O_3 v = 2 to be deleted , (b) the deletion and the insertion in Rs,Rd and the auxiliary tables' structures

Figure 8.4 illustrates the DV-TPR*-tree structure. The top tree is a spatial dimension based on the basic TPR-Tree (TPR*-Tree hold the basic structure of TPR-Tree); the bottom tree is a Direction bucket queue, where objects are clustered based on direction (note that in our example we have only two directions

N=north and S=South). The third dimension consists of a hash index (lookup table) constructed on IDs of moving objects, the hash index is <ObjID, *ptr*>, where *ObjID* denotes the identifier of moving objects; whereas, *ptr* denotes physical offset of the object entry in the leaf node; velocity direct access which is a 3 tuple of the form (Velocity, objects queue, *nodesPTR*) where velocity denotes the velocity value range, objects queue denotes the moving objects on that velocity range, *nodesPTR* are pointers to the nodes themselves.

8.3.2 Insertion and Deletion

The insertion in the DV-TPR*-tree is done in three stages. First, the algorithm receives the input objects to be inserted into the *Rs* which are responsible for clustering the objects based on the TPR*-tree. Then the algorithm scans the direction leaf nodes *Rd* to find the leaf node that contains this object. Then a new object entry is produced and inserted into a suitable direction leaf node. The algorithm scans the main-memory velocity direct access queue to obtain the bucket that contains this object, and then insert it. Finally, a new object will be inserted into the hash index (lookup table). Note that if overflow occurs while insertion is *Rs* or *Rd*, the TPR*-tree node will be split based on the method mentioned in [10] which considers all possible divisions that are sorted based on the starting/ending objects of their extents on this dimension. Figure 8.5 illustrates the insertion of O_6, v = -2 and the deletion of O_3, v = 2.

The deletion algorithm will first locate the object entry's leaf node N by the lookup table, and then deletes the entry directly from *Rs* and *Rd*. Then the algorithm ascends the branches of *Rs* and *Rd* until it reaches the root node in

Algorithm 15 Insertion in DV-TPR*-tree

```
1: /* Input: o is the entry to be inserted. */
2: procedure INSERT(o into Rs)
3: |  Stage 1: Insert o into the Rs
4: |  Choose Path to find the leaf node N to insert o
5: |  A priority queue PQ, containing the explored path(s) A TPR*-(sub) Tree
   rooted at N
6: |  Calculate the cumulative cost degradation
7: cost(q) = ∑_{Every node o} ASR(o', qT)    o← PQ
8: |  Set the least cumulative cost degradation at the top N
9: |  Choose the path N that has the least cumulative cost degradation
10: |  if N overflows then
11: |  |  if re-inserted 0 = false then
12: |  |  |  Pick Worst to select a set Sworst of entries
13: |  |  |  Entries in Sworst from N
14: |  |  |  Add them to Lreinsert
15: |  |  |  Re-inserted 0 = true
16: |  |  end if
17: |  else
18: |  |  Invoke Node Split to split N into N1 and N2 (TPR-Tree based)
19: |  end if
20: end procedure
21: procedure INSERT(o into Rd)
22: |  Check o coordinators x, y
23: |  Determine the direction based on the coordinators x, y
24: |  Scans the Direction leaf nodes Rd with similar D
25: |  Insert o leaf node.D
26: end procedure
27: procedure INSERT(o into auxiliary tables' structures)
28: |  The main-memory scans velocity direct access queue
29: |  Obtain the bucket suitable for o
30: |  Insert o to the velocity direct access
31: |  o inserted into the hash index (lookup table)
32: end procedure
```

order to modify the MBR and VBR of nodes along the path. Note that tightening of both *Rs* and *Rd* occurs after each deletion. Finally, the corresponding object item in velocity direct access and the hash index will be deleted.

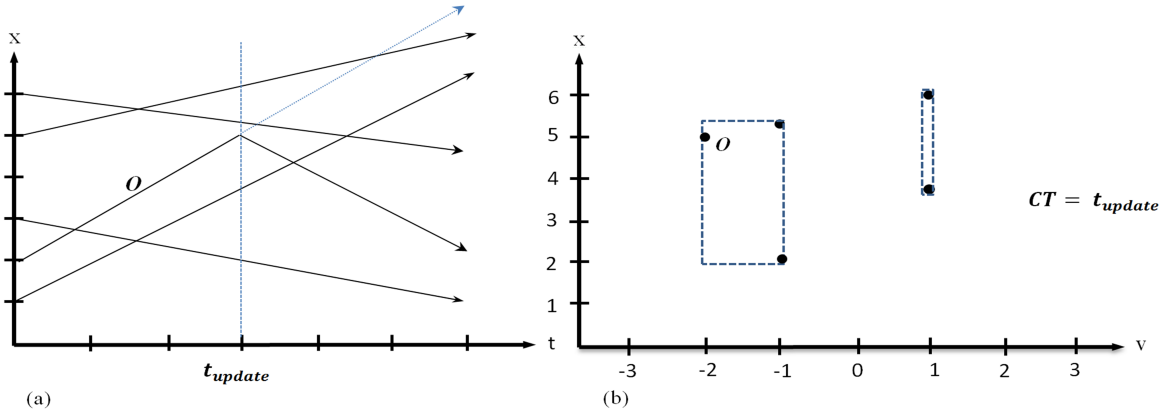


FIGURE 8.6: (a) update o direction on one-dimensional data, (b) o is grouped with new direction leaf node

Algorithm 16 Deletion in DV-TPR*-tree

```
1: /* Input: o is the entry to be deleted. */
2: procedure DELETE(o)
3:     Lookup table locates o's the leaf node
4:     Delete o from N
5:     Change the MBR and VBR of intermediate nodes
6:     Tightening N until the root
7:     if Rs underfull then
8:         force-reinsert
9:     end if
10:    if Rd underfull then
11:        force-reinsert
12:    end if
13:    The main-memory scans velocity direct access queue
14:    Obtain the bucket of o
15:    Delete o from the velocity direct access
16:    Delete o from the hash index (lookup table)
17: end procedure
```

8.3.3 Update Algorithm

The main purpose of the update algorithm is to locate the objects by the hash index, then perform a direct updating function on the three dimensions (Rs, Rd and the auxiliary tables' structures). For example, if object o is presented to be updated, the lookup table will locate o's leaf node N at Rs. If the object o is located in the current N's MBR, the algorithm will update directly; otherwise, o

will be deleted and reinserted. At *Rd*, the algorithm will judge whether or not the object o remains in the same direction. Note that no updating will be performed only if the object is still moving in the same direction. Otherwise, the object o will be deleted and reinserted in the suitable leaf node direction. Note that to reflect the change, the lookup table and velocity direct access will be updated as well. To illustrate, Figure 8.6 shows an example of updating an object o direction on one-dimensional data and grouping it with a new direction leaf node.

Algorithm 17 Updating in DV-TPR*-tree

```
1: /* Input: o is the entry to be updated. */
2: procedure UPDATE(o)
3:     Lookup table locates o's the leaf node
4:     if o updating in N then
5:         Update o
6:         Tightening o path
7:     else
8:         Delete o
9:         Reinsert o in new leaf node N‘
10:        Tightening N.N‘ path
11:    end if
12:    if o in same direction then
13:        Do not update o.d
14:    else
15:        update o.d
16:    end if
17:    The main-memory scans velocity direct access queue
18:    Obtain the bucket of o
19:    Update o in the velocity direct access
20:    Update ptr of o in the hash index (lookup table)
21: end procedure
```

8.3.4 DV-TPR*-tree Querying

Our proposed data structure successfully added two new types of moving object queries which are the direction queries and the velocity queries. The standard

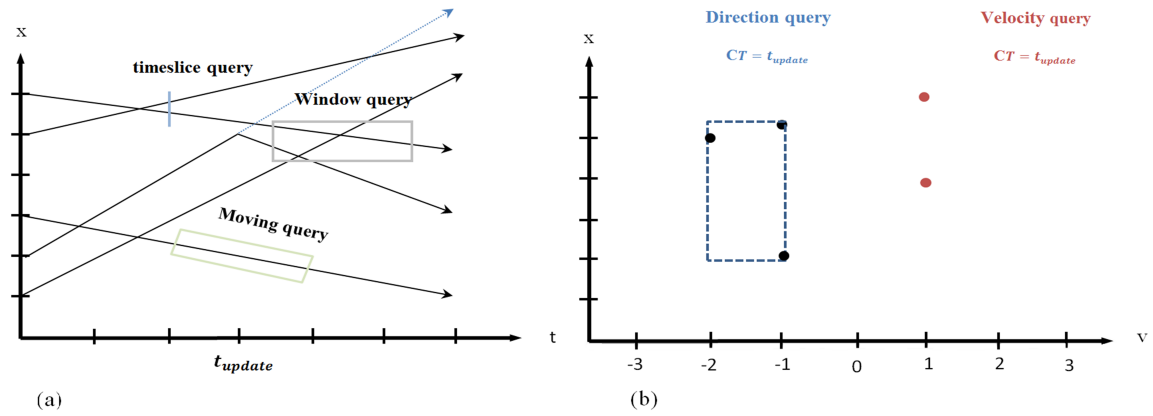


FIGURE 8.7: Five query types supported on DV-TPR*-tree (a) timeslice, window and moving queries, (b) direction query and velocity query

three types of queries in TPR-tree and its successors remain functional in our data structure. The main idea is that the Rs is visited only when timeslice query $Q = (R, t)$, window query: Q = (R, t_i, t_j), $t_i < t_i$ and moving query: Q = (R_1, R_2, t_i, t_j), $t_i < t_j$. On the other hand, the Rd is visited only when direction query Q=(R,D) is introduced. The velocity direct access is visited when velocity query Q=(R,V) is introduced. The combination of timeslice, window and moving queries with the DV-queries is supported in our proposed data structure. Next we present an example of a window query combined with a direction query:

Q. Between 5:00 pm and 6:00 pm, return all vehicles in the suburb of Clayton that are moving in a northerly direction.

Note that when we combined timeslice, window and moving queries with the DV-queries, both Rs and Rd can be visited beside the second auxiliary structure. Figure 8.7 illustrates the five types of queries that are supported in our data structure. (a) Represents the three possible queries which are timeslice, window and moving queries. On the other side, (b) represents a possible direction query that indicates moving objects which are moving south (negative positions); whereas, the moving objects at velocity $v = 1$ represent a possible velocity query for objects with similar velocity.

8.4 Experimental Results and Performance Analysis

In this section, we present our experimental results comparing the *DV* queries performance of the TPR-Tree and DV-TPR*-tree under different object agilities. The experiment has been carried out on an Intel Core 2 Duo 2.53GHz PC with 4 GB of RAM running on 64-bit Windows 7 Home Premium. The page size (and tree node size) was set to 14kb and the maximum number of entries per node, M, was 34 resulting in a data set size of 100,000 objects.

In this experiment, we use synthetic datasets of moving objects with positions in a one-dimensional space (9000), based on uniform data, where object locations are chosen randomly, and the objects move in a random velocity and direction. Note that the velocity ranges from 0 to 30. For our datasets, the index has been constructed at time 0, and we measured the velocity and direction query costs for all similar time units. Note that we chose random direction queries, random velocity queries and random direction and velocity queries. The parameters used are summarized in Table 8.1.

TABLE 8.1: Parameters and Their Settings

Parameter	**Setting**
Page size	14K
Node capacity	34
Number of moving objects	≈ (10000 to 100000)
Space domain	9000
Queries	direction, velocity, direction with velocity
Dataset	synthetic

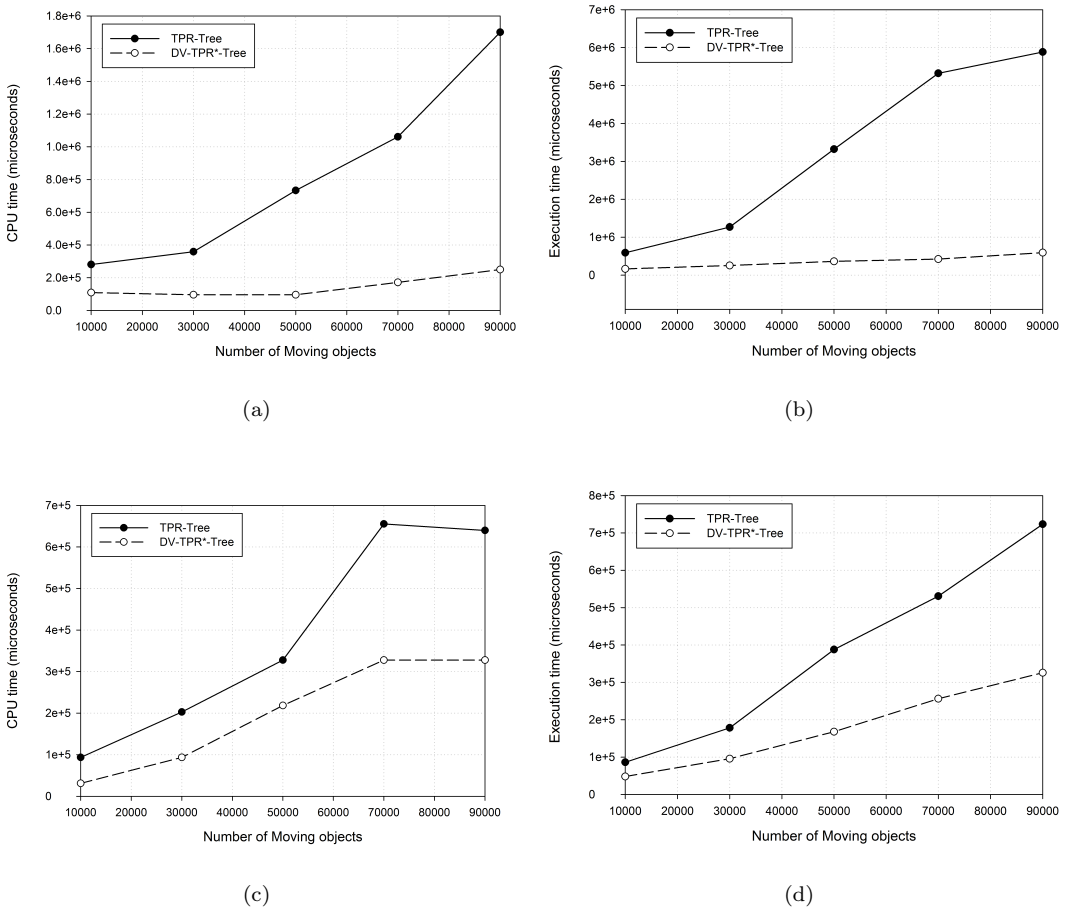


FIGURE 8.8: Effect of CPU and execution time on similar direction query performance

8.4.1 Direction query

In the first set of experiments, we study the direction query performance of the TPR-tree and the DV-TPR*-tree while varying the number of moving objects from ($\approx$ 10000 to 100000). As mentioned in Table 8.1, four random direction queries have been executed. Figure 8.8 shows the CPU and the execution time per direction query for each index. Figure 8.8 represents the result of a query to return the moving objects that are moving in the same direction.

We observe that DV-TPR*-tree variants maintain consistent performance and scale very well; whereas, the TPR-tree's performance cost increases with the

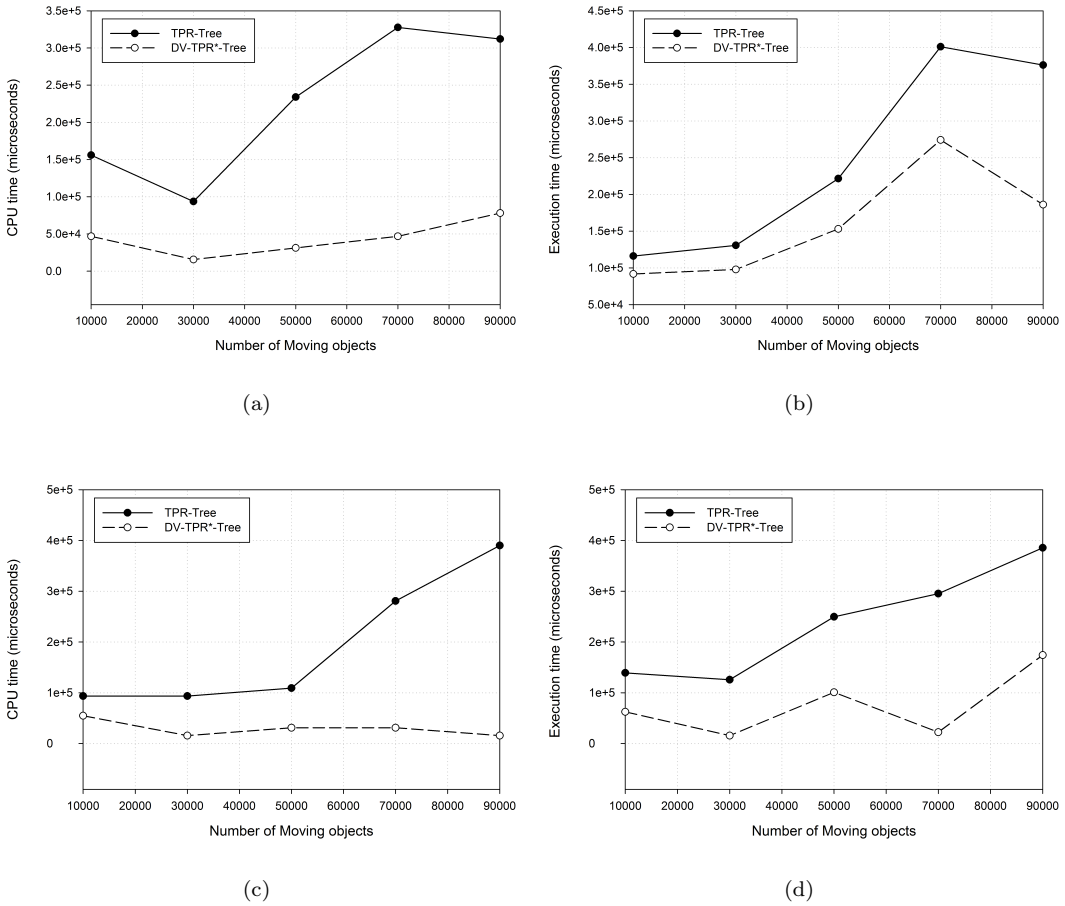


FIGURE 8.9: Effect of CPU and execution time on similar both (direction and velocity) query performance

increase of the dataset size. When the dataset reaches 50000 moving objects and above, the DV-TPR*-tree performs nearly four times better than the TPR-tree. This behaviour can be explained as follows. In the DV-TPR*-tree, each direction has been determined by a separate domain and is relatively independent of the number of moving objects. As the dataset grows, the direction query performance cost of the DV-TPR*-tree increases, mainly due to the increase in the number of objects [26, 105].

However, the structure of the TPR tree is more affected by increasing the number of moving objects. For direction queries, the TPR-tree needs to go through the velocity parameter in each object because the TPR-tree and its successors

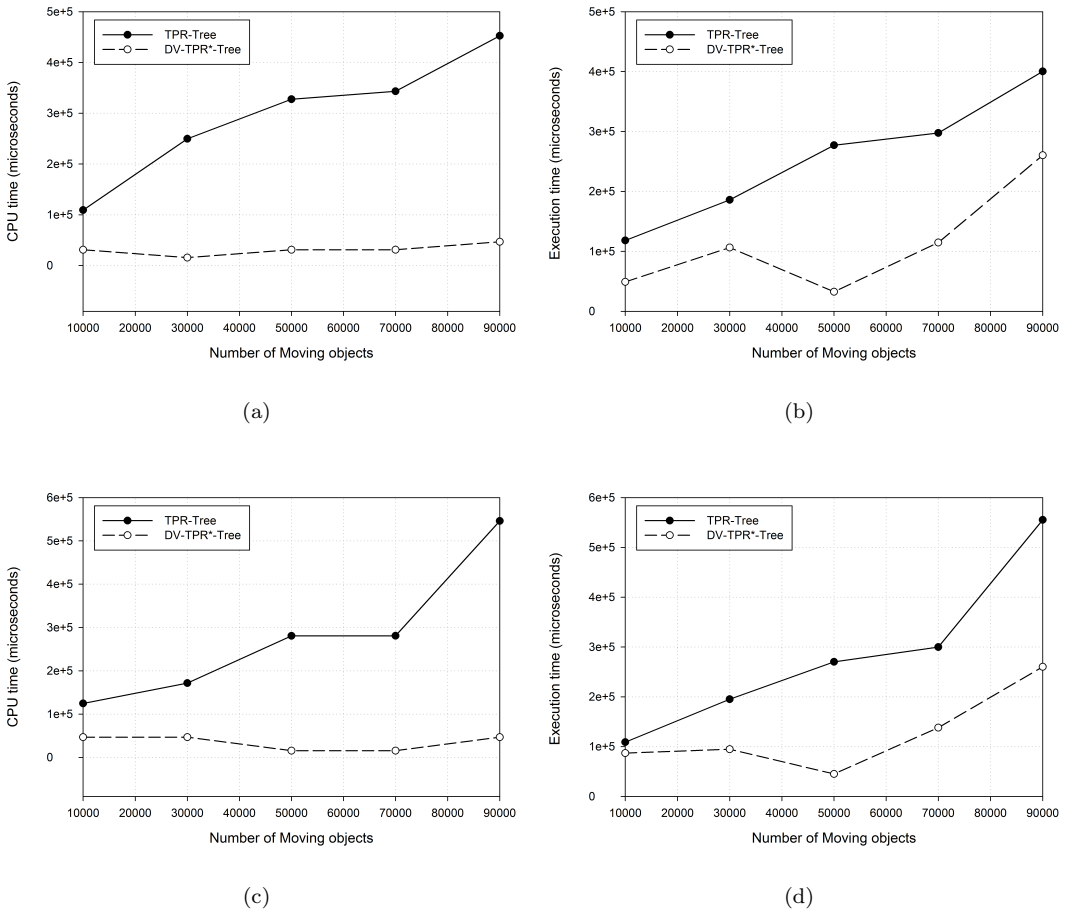


FIGURE 8.10: Effect of CPU and execution time on similar velocity queries performance

are constructed based only on the space domain. Moreover, when the dataset size increases, the MBRs in the TPR-tree lead to a higher probability of overlapping, which increases the query performance cost. Therefore, The DV-TPR*-tree achieves better direction query performance than does the TPR-tree and its successors, because the DV-TPR*-tree more effectively considers the queries that are based on direction. Note that Figure 8.9 shows the results of queries cost when the moving objects of both similar direction and velocity are considered.

8.4.2 Velocity query

In the second set of experiments, we study the velocity query performance of the TPR-tree and the DV-TPR*-tree while varying the number of moving objects from 10000 to 100000. Four random velocity queries have been performed. Figure 8.10 shows the CPU and the execution time per velocity query for each index. Figure 8.10 represents the result of the fifth query to return the moving objects that are moving at similar velocities (ignoring the direction).

Similar to the direction query, we can notice that DV-TPR*-tree variants maintain consistent performance and scale very well, compared with the TPR-tree. When the dataset reaches 50000 moving objects and above, the velocity query performance of the DV-TPR*-tree are nearly four times better than that of the TPR-tree. We explain this behaviour as follows. In the DV-TPR*-tree, each velocity has been determined by a separate domain and is relatively independent of the number of moving objects. As the dataset grows, the velocity query performance cost of the DV-TPR*-tree increases mainly due to the increase in the number of objects. However, the structure of the TPR tree is more affected by the increasing number of the moving objects. For a velocity query, the TPR-tree needs to go through the velocity parameter in each object because the TPR-tree and its successors are constructed based only on the space domain. Moreover, when the dataset size increases, the MBRs in the TPR-tree lead to higher probabilities of overlapping, which increase the query performance cost. Therefore, The DV-TPR*-tree achieves better velocity query performance than do the TPR-tree and its successors, because the DV-TPR*-tree generates a better distribution for the queries that are based on velocity.

8.5 Chapter Summary

This chapter addresses the challenge of supporting new types of queries namely, direction and velocity queries (*DV* queries). We propose a novel index structure to include the direction queries and the velocity queries for moving objects. The data structure in the TPR-tree and its successors are based on the space domain, without considering any distribution of the moving objects' direction and velocity. In this chapter, we focus on construction based on the spatial, direction and velocity domains. This data structure is based on the TPR*-tree. New dimensions have been provided, and a directions bucket structure is introduced which will hold objects that share the same direction. The second dimension is the auxiliary table's structures, which include a lookup table which is used for bottom-up updates and a velocity access table which determines the nodes that are congruent in the velocity. Algorithms are provided for insertion, deletion and updating of the indexed objects, as well as for illustrative examples. Extensive performance studies were conducted which indicate that the DV-TPR*-tree is both robust and efficient for *DV* queries more so than the TPR-tree and its successors. In fact, it is capable of performing with the new query types (*DV* queries) nearly four times better than the TPR-tree and its successors.

CHAPTER 9

Conclusions and Future Work

Chapter Plan:

9.1 Conclusions

Moving-object applications deal with dynamic objects that continuously change their locations. Therefore, these applications involve queries and space structures that are different from those which are in the traditional spatial databases. The majority of traditional spatial databases focus on the moving objects in outdoor spaces with no consideration given to indoor spaces. Therefore, this thesis started by presenting a clear explanation about the moving objects databases and the possible variety of queries that can be raised in moving-object databases from five perspectives which are: Location perspective, Motion perspective, Object perspective, Temporal perspective and Patterns perspective. Then, it addresses the challenges of indexing moving objects in indoor spaces, introducing a new direction of indexing called **adjacency-based indexing of cellular space**. Moreover, it indexes the moving objects in indoor spaces, taking into account the data density and its influences on the performances of the data structure. For the multi-floor indoor spaces, it presents a unique index structure to group the moving objects based on multidimensional grouping. This thesis also extended the regional-adjacency indexing in indoor spaces to be applicable to topographical outdoor spaces. The overview study of this thesis is illustrated in Figure 9.1.

9.1.1 Indexing Moving Objects In Indoor Spaces

This thesis proposes a cells-adjacency-based index structure for moving objects (called Indoor-tree). The novel index structure focuses on the moving objects based on the notion of cellular space, in contrast to the outdoor space structures which are based on the space domain, Euclidean or spatial network. Moreover,

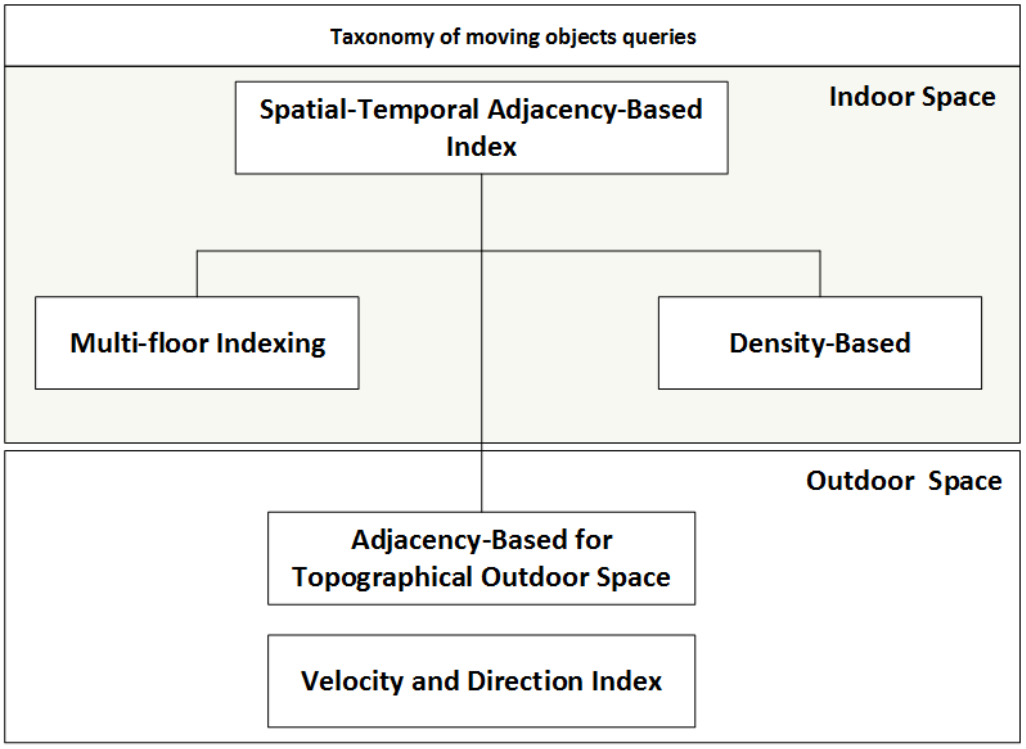


FIGURE 9.1: An overview of this Thesis Study

the key idea behind our moving objects indexing is to take advantage of the entities such as doors and hallways that both enable and constrain movement in an indoor environment. Therefore, we obtain an optimal representation of the indoor environment that is different from the outdoor environment. Indoor-tree structure is based on adjacency and cell connections, which allows us to answer spatial queries more efficiently (**Chapter 4**). We evaluated the existing parameters of the indoor-tree to determine the effect on the data structure (e.g. moving objects number, connection complexity, expansion complexity and cells density). In addition, we extended our indoor-tree index structure to include the temporal side. Therefore, the temporal part of the index has been presented using three different techniques. The first technique is based on the non-leaf nodes timestamping (TI-tree); the second technique is for monitoring the objects that change their cells and delay the others (MOT-tree), and the last technique is based on the deltas for each cell to reduce the update cost (ITD-tree). The results show that the indoor-tree and MOT-tree and ITD-tree can successfully obtain a reliable and

a robust tree index based on the novel idea of indoor connections and adjacency.

Furthermore, data density plays an essential role in the data structure performance (**Chapter 5**). Therefore, considering indoor space high density cells in any data structure can produce many benefits to increase the tree performance. Hence, we proposed a new index structure, density and adjacency-based, for moving objects in indoor spaces. This work concentrates on indexing the moving objects based on the notion of cellular space. Indoor^d-tree takes into account the indoor walls and partitions that control movement in an indoor environment and index the moving objects based on the connectivity/adjacency of the indoor cells. Moreover, since the moving object density in indoor spaces greatly affects the efficiency of the index structure, we distinguish between cells of different densities in the index structure. An analytical cost study is provided for the search cost of the low-density tree, and high-density tree and the storage costs.

For multi-floor indoor spaces, in some cases it is possible to group the moving objects based on multidimensional grouping in each section/wing, which is useful for wing/section positioning queries and aggregation queries. Therefore, we proposed a unique index structure which is based on section/wing multidimensional grouping (**Chapter 6**). The GMI-tree takes advantage of the graph that results from the connectivity of the indoor cells. If the graph meets our proposed graph elements, then the indoor objects will be considered to be grouped as a multidimensional methodology. The idea is to group the mobile objects based on their adjacency on the same floor and same section (based on the connectivity between the cells) and extend that to group the objects in each section as multidimensional grouping for the multi-level. In this data structure, we proposed a multidimensional connectivity tree which presents the connectivity of the indoor cells horizontally and vertically. We evaluated the construction performance,

search performance, and section/wing search performance of the GMI-tree. The results show that the GMI-tree can successfully produce a reliable and robust tree index for multi-floor indoor spaces.

9.1.2 Indexing Moving Objects in Outdoor Spaces

For outdoor spaces, we extended the regional-adjacency indexing in indoor spaces to be applicable to topographical outdoor spaces. A Topographical Outdoor-tree (TO-tree) has been proposed which is based on the adjacency technique for moving objects in outdoor environments. The TO-tree focuses on the construction of the moving objects based on the notion of cellular space (**Chapter 7**). The key idea of our moving objects indexing is to take advantage of the adjacency and the connections within topographical outdoor environments. Note that the adjacency index method can enable us to answer the spatial queries more efficiently. The uniqueness of the TO-tree is that it does not index the regions/cells as a containers; rather, it indexes the moving objects by grouping the moving objects in the same cell first, and if a split is needed, they are grouped with moving objects with the adjacent cells. Therefore, we argue that the TO-tree is an optimal index for outdoor cell environments which can serve the spatial queries, topological queries and adjacency queries. We conducted experiments to compare the TO-tree with the common metric data structure (TPR-tree). The results show that the TO-tree creates an optimal data structure for the adjacency applications and performs more efficiently than the metric structures such as the TPR-tree.

In addition, we proposed a novel index structure to include the direction queries and the velocity queries for moving objects. The data structure in TPR-tree and

its successors are based on the space domain, without considering any distribution of the moving objects' direction and velocity. Our proposed index (DV-TPR*-Tree) focuses on the construction of the moving objects based on the spatial, direction and velocity domains (**Chapter 8**). The key idea of our moving objects indexing is to cover new types of queries, namely directions and velocity queries (DV queries). Extensive performance studies were conducted which indicate that the DV-TPR*-tree is both robust and efficient for DV queries more than are the TPR-tree and its successors.

9.2 Future Work

There are several promising directions for future work in the research area presented in this thesis. The directions range from direct extensions of the research to applying the fundamental ideas to other applications.

For the queries side, we intend to extend our taxonomy model to include existing moving-objects query processing. Moreover, we intend to classify the queries of the moving objects based on their underlain structures such as indoor and outdoor space and presenting the differences and challenges of supporting these queries in different space structures. Moreover, indoor query processing has attracted little attention which makes queries such as adjacency, KNN, range, multi-floor range queries very interesting.

In indoor spaces, several directions can be extended from our existing index structures. First, by improving the temporal queries and the trajectories. The temporal aspects are one of the challenges that we intend to investigate for the

indoor space index. Moreover, we aim to extend our indoor trees to include different spatial queries and different navigational queries. Second, movement in indoor space is different from the outdoor, where the moving objects are expected to be dynamic in the circulation areas (such as corridors and stairs), whereas in the rooms/venues they tend to be less dynamic. Therefore, an index based on temporal stabilization is an interesting future work direction. Moreover, the multi-floor indoor environments have many challenges that need to be studied in the future. For example, some buildings have different shapes (e.g. spiral) where the levels are not located exactly above each other. Also the different movements in some buildings have different flows which means that each building must be considered individually in order to obtain an efficient index structure for it. Another avenue of research is the consideration of the index structure of moving objects in relation to new patterns of movement inside buildings. Therefore, we aim to extend our tree to include different movement patterns of the moving objects in indoor spaces.

For outdoor spaces, several directions can be extended from our existing index structures. First, the temporal queries and the trajectories based on the adjacency techniques. As we did for the indoors, the outdoors need to be considered for the temporal queries where the update is delayed as the regional indexing techniques. Second, is to combine the indoor and the outdoor spaces in order to build a data structure that serves both spaces. Moreover, movement outdoors seem to have the flow of a Levy flight which makes it interesting to index the moving objects outdoors based on the Levy flight movement patterns which might reduce the updating costs significantly.

Bibliography

[1] 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, 2012. doi: 10.1109/ICDE.2012.44.

[2] Yong-Jin Choi and Chin-Wan Chung. Selectivity estimation for spatio-temporal queries to moving objects. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 440–451, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5.

[3] Yufei Tao and Dimitris Papadias. Range aggregate processing in spatial databases. *IEEE Trans. on Knowl. and Data Eng.*, 16(12):1555–1570, 2004.

[4] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches to the indexing of moving object trajectories. In *International Conference on Very Large Databases*, pages 395–406, 2000.

[5] Sultan Alamri, David Taniar, and Maytham Safar. Indexing moving objects for directions and velocities queries. *Information Systems Frontiers*, 15(2): 235–248, 2013.

[6] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Rec.*, 29:331–342, May 2000. ISSN 0163-5808.

[7] Wenting Liu, Zhijian Wang, and Jun Feng. Continuous clustering of moving objects in spatial networks. In *KES (2)*, pages 543–550, 2008.

[8] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *international conference on management of data*, pages 47–57. ACM, 1984.

[9] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Lugrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, page 13, 2006.

[10] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.

[11] Elias Frentzos. Indexing objects moving on fixed networks. In *SSTD*, pages 289–305, 2003.

[12] Bin Lin and Jianwen Su. On bulk loading tpr-tree. In *Mobile Data Management, 2004. Proceedings. 2004 IEEE International Conference on*, pages 114 – 124, 2004. doi: 10.1109/MDM.2004.1263049.

[13] Yuni Xia and Sunil Prabhakar. Q+rtree: Efficient indexing for moving object database. In *DASFAA*, pages 175–182, 2003.

[14] Jae-Woo Chang, Jung-Ho Um, and Wang-Chien LeeP. A new trajectory indexing scheme for moving objects on road networks. In *Flexible and Efficient*

Information Handling, volume 4042 of *Lecture Notes in Computer Science*, pages 291–294. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-35969-2.

[15] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with $SETI$. In *CIDR*, 2003.

[16] Panfeng Zhou, Donghui Zhang, Betty Salzberg, Gene Cooperman, and George Kollios. Close pair queries in moving object databases. In *Proceedings of the 13th annual ACM International Workshop on Geographic Information Systems*, GIS '05, pages 2–11, New York, NY, USA, 2005. ACM. ISBN 1-59593-146-5. doi: 10.1145/1097064.1097067.

[17] Mario A. Nascimento and Jefferson R. O. Silva. Towards historical r-trees. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, SAC '98, pages 235–240, New York, NY, USA, 1998. ACM. ISBN 0-89791-969-6. doi: 10.1145/330560.330692.

[18] Haidar Al-Khalidi, David Taniar, John Betts, and Sultan Alamri. On finding safe regions for moving range queries. *Mathematical and Computer Modelling*, 58(5-6):1449–1458, 2013.

[19] Geng Zhao, Kefeng Xuan, W. Rahayu, D. Taniar, M. Safar, M.L. Gavrilova, and B. Srinivasan. Voronoi-based continuous k nearest neighbor search in mobile navigation. *Industrial Electronics, IEEE Transactions on*, 58(6):2247 –2257, june 2011. ISSN 0278-0046. doi: 10.1109/TIE.2009.2026372.

[20] Guohui Li, Ping Fan, YanHong Li, and Jianqiang Du. An efficient technique for continuous k-nearest neighbor query processing on moving objects in a road network. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 627–634. IEEE Computer Society, 2010.

[21] Haidar Al-Khalidi, Zainab Abbas, and Maytham Safar. Approximate range query processing in spatial network databases. *Multimedia Syst.*, 19(2):151–161, 2013.

[22] Sultan Alamri. Indexing and querying moving objects in indoor spaces. In *ICDE Workshops*, pages 318–321, 2013.

[23] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, and Christian S. Jensen. Path prediction and predictive range querying in road network databases. *The VLDB Journal*, 19(4):585–602, 2010.

[24] Sultan Alamri, David Taniar, and Maytham Safar. A taxonomy for moving object queries in spatial databases. *Future Generation Computer Systems*, 2014. doi: 10.1016/j.future.2014.02.007.

[25] Dan Lin, Rui Zhang, and Aoying Zhou. Indexing fast moving objects for knn queries based on nearest landmarks. *GeoInformatica*, 10(4):423–445, 2006. ISSN 1384-6175. doi: 10.1007/s10707-006-0341-9. URL http://dx.doi.org/10.1007/s10707-006-0341-9.

[26] Dan Lin. *Indexing and Querying Moving Objects Databases*. PhD thesis, National University of Singapore, Singapore, 2006.

[27] Kyoung-Sook Kim, Si-Wan Kim, Tae-Wan Kim, and Ki-Joune Li. Fast indexing and updating method for moving objects on road networks. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops*, WISEW'03, pages 34–42, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2103-7.

[28] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pages 189–207, 2009.

[29] Xike Xie, Hua Lu, and T.B. Pedersen. Efficient distance-aware query evaluation on indoor moving objects. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 434–445, 2013. doi: 10.1109/ICDE.2013.6544845.

[30] Sultan Alamri, David Taniar, Maytham Safar, and Haidar Al-Khalidi. A connectivity index for moving objects in an indoor cellular space. *Personal and Ubiquitous Computing*, pages 1–15, 2013. ISSN 1617-4909. doi: 10.1007/s00779-013-0645-3.

[31] Christian S. Jensen, Dan Lin, Beng Chin Ooi, and Rui Zhang 0003. Effective density queries on continuously moving objects. In *ICDE*, page 71, 2006.

[32] Marios Hadjieleftheriou, George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *Proc. SSTD*, pages 306–324, 2003.

[33] Hajar Mousannif, Ismail Khalil, and Stephan Olariu. Cooperation as a service in vanet: Implementation and simulation results. *Mobile Information Systems*, 8(2):153–172, 2012.

[34] Jason C. Hung. The smart-travel system: utilising cloud services to aid traveller with personalised requirement. *IJWGS*, 8(3):279–303, 2012.

[35] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: issues and solutions. In *Scientific and Statistical Database Management, 1998. Proceedings. Tenth International Conference on*, pages 111 –122, jul 1998. doi: 10.1109/SSDM.1998.688116.

[36] Sultan Alamri, David Taniar, and Maytham Safar. Indexing moving objects in indoor cellular space. In *NBiS*, pages 38–44, 2012.

[37] Sultan Alamri, David Taniar, Maytham Safar, and Haidar Al-Khalidi. Spatiotemporal indexing for moving objects in an indoor cellular space. *Neurocomputing*, 122:70–78, 2013.

[38] Sultan Alamri, David Taniar, and Maytham Safar. Indexing of spatiotemporal objects in indoor environments. In *AINA*, pages 453–460, 2013.

[39] Sultan Alamri, David Taniar, Haidar Al-Khalidi, and Kinh Nguyen. Density and adjacency based indexing for moving objects in indoor spaces. In *IEEE Transactions on Knowledge and Data Engineering*, 2014. Under review.

[40] Sultan Alamri, David Taniar, Haida Al-Khalidi, and Kinh Nguyenr. Indexing moving objects in multi-floor indoor environments. In *ACM Transactions on Spa- tial Algorithms and Systems*, 2014. Under review.

[41] Sultan Alamri, David Taniar, Maytham Safar, and Haidar Al-Khalidi. Tracking moving objects using topographical indexing. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013. ISSN 1532-0634. doi: 10.1002/cpe.3169.

[42] Toms Ruiz-Lpez, Jos Garrido, Kawtar Benghazi, and Lawrence Chung. A survey on indoor positioning systems: Foreseeing a quality design. In *Distributed Computing and Artificial Intelligence*, volume 79 of *Advances in Intelligent and Soft Computing*, pages 373–380. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-14882-8.

[43] Yan Luo, Orland Hoeber, and Yuanzhu Chen. Enhancing wi-fi fingerprinting for indoor positioning using human-centric collaborative feedback. *Human-centric Computing and Information Sciences*, 3(1):1–23, 2013. doi: 10.1186/2192-1962-3-2.

[44] F. Lassabe, P. Canalda, P. Chatonnay, and F. Spies. Indoor wi-fi positioning: techniques and systems. *Annals of Telecommunications - Annales*

des Telecommunications, 64(9-10):651–664, 2009. ISSN 0003-4347. doi: 10.1007/s12243-009-0122-1.

[45] F. Forno, G. Malnati, and G. Portelli. Design and implementation of a bluetooth ad hoc network for indoor positioning. *Software, IEE Proceedings* - , 152(5):223 – 228, oct. 2005. ISSN 1462-5970. doi: 10.1049/ip-sen:20045027.

[46] System Networks. Pats people and asset tracking system, 2013. URL http://www.shyamnetworks.com/security/infrastructure-protection/pats-people-and-asset-tracking-system.

[47] cisco. Location tracking approaches, 2013. URL http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Mobility/WiFiLBS-DG/wifich2.html.

[48] Navizon. Indoor gps and indoor tracking solutions by navizon, 2013. URL http://www.navizon.com/indoors-solutions.

[49] Beom-Ju Shin, Kwang-Won Lee, Sun-Ho Choi, Joo-Yeon Kim, Woo Jin Lee, and Hyung Seok Kim. Indoor wifi positioning system for android-based smartphone. In *Information and Communication Technology Convergence (ICTC), 2010 International Conference on*, pages 319–320, 2010. doi: 10.1109/ICTC.2010.5674691.

[50] Kefeng Xuan, Geng Zhao, David Taniar, and Bala Srinivasan. Continuous range search query processing in mobile navigation. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 361–368, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3434-3. doi: 10.1109/ICPADS.2008.69.

[51] Kefeng Xuan, Geng Zhao, David Taniar, Wenny Rahayu, Maytham Safar, and Bala Srinivasan. Voronoi-based range and continuous range query processing in mobile databases. *J. Comput. Syst. Sci.*, 77:637–651, July 2011. ISSN 0022-0000.

[52] S. Nutanong, Rui Zhang, E. Tanin, and L. Kulik. V*-knn: An efficient algorithm for moving k nearest neighbor queries. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1519–1522, 2009. doi: 10.1109/ICDE.2009.63.

[53] Goetz Graefe. A survey of b-tree locking techniques. *ACM Trans. Database Syst.*, 35(3), 2010.

[54] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, Sung Nok Chiu, and D. G. Kendall. *Spatial Tessellations:Concepts and Applications of Voronoi Diagrams*, pages 585–655. John Wiley Sons, Inc., 2008. ISBN 9780470317013. doi: 10.1002/9780470317013.refs.

[55] Yanfei Lv, Jing Li, Bin Cui, and Xuexuan Chen. Log-compact r-tree: An efficient spatial index for ssd. In *DASFAA Workshops*, pages 202–213, 2011.

[56] 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*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM. ISBN 0-89791-365-5. doi: 10.1145/93597.98741.

[57] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[58] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[59] David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. The skip quadtree: A simple dynamic data structure for multidimensional data. *CoRR*, abs/cs/0507049, 2005.

[60] Bin Lin and Jianwen Su. Handling frequent updates of moving objects. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, pages 493–500, New York, NY, USA, 2005. ACM. ISBN 1-59593-140-6.

[61] VictorTeixeira de Almeida and RalfHartmut Gting. Indexing the trajectories of moving objects in networks*. *GeoInformatica*, 9(1):33–60, 2005. ISSN 1384-6175. doi: 10.1007/s10707-004-5621-7.

[62] Yufei Tao and Dimitris Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 431–440, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4.

[63] Yannis Theodoridis, Michael Vazirgiannis. Timos Sellis, Michael Vazirgiannis, and Timos Sellis. Spatio-temporal indexing for large multimedia applications. In *ICMCS*, pages 441–448, 1996.

[64] Yufei Tao and D. Papadias. Efficient historical r-trees. In *Scientific and Statistical Database Management, 2001. SSDBM 2001. Proceedings, Thirteenth International Conference on*, pages 223–232, 2001. doi: 10.1109/SSDM.2001.938554.

[65] David Lomet and Betty Salzberg. The performance of a multiversion access method. *SIGMOD Rec.*, 19(2):353–363, May 1990. ISSN 0163-5808. doi: 10.1145/93605.98744.

[66] Yong-Jin Choi, Jun-Ki Min, and Chin-Wan Chung. A cost model for spatio-temporal queries using the TPR-tree. *Journal of Systems and Software*, 73 (1):101 – 112, 2004. ISSN 0164-1212. doi: DOI:10.1016/S0164-1212(03) 00209-7.

[67] Wei Liao, Guifen Tang, Ning Jing, and Zhinong Zhong. Vtpr-tree: An efficient indexing method for moving objects with frequent updates. In *Advances in Conceptual Modeling - Theory and Practice*, volume 4231 of *Lecture Notes in Computer Science*, pages 120–129. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-47703-7. doi: 10.1007/11908883_15.

[68] Hoang Thanh Tung, Young Jin Jung, EungJae Lee, and KeunHo Ryu. Moving point indexing for future location query. In *Conceptual Modeling for Advanced Application Domains*, volume 3289 of *Lecture Notes in Computer Science*, pages 79–90. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23722-8. doi: 10.1007/978-3-540-30466-1_8.

[69] Christian Jensen, Hua Lu, and Bin Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *Advances in Spatial and Temporal Databases*, volume 5644 of *Lecture Notes in Computer Science*, pages 208–227. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-02981-3.

[70] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, March 2000. ISSN 0362-5915.

[71] Talel Abdessalem, José Moreira, and Cristina Ribeiro. Movement query operations for spatio-temporal databases. In *BDA*, 2001.

[72] Maribel Yasmina Santos, José Mendes, Adriano J. C. Moreira, and Monica Wachowicz. Towards a spatio-temporal information system for moving objects. In *ICCSA (1)*, pages 1–16, 2011.

[73] Haibo Hu, Dik Lun Lee, and Victor C. S. Lee. Distance indexing on road networks. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 894–905, 2006.

[74] K. Raptopoulou, A. N. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *Geoinformatica*, 7 (2):113–137, 2003.

[75] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.

[76] José Moreira, Cristina Ribeiro, and Talel Abdessalem. Query operations for moving objects database systems. In *ACM-GIS*, pages 108–114, 2000.

[77] Natalia V. Andrienko and Gennady L. Andrienko. Designing visual analytics methods for massive collections of movement data. *Cartographica*, 42(2): 117–138, 2007.

[78] AngelD. Sappa, David Gernimo, Fadi Dornaika, Mohammad Rouhani, and AntonioM. Lpez. Moving object detection from mobile platforms using stereo data registration. In *Computational Intelligence Paradigms in Advanced Pattern Classification*, volume 386 of *Studies in Computational Intelligence*, pages 25–37. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-24048-5. doi: 10.1007/978-3-642-24049-2_3.

[79] Xianfeng Fei, Yasunobu Igarashi, and Koichi Hashimoto. Parallel region-based level set method with displacement correction for tracking a single moving object. In *Advanced Concepts for Intelligent Vision Systems*, volume

5807 of *Lecture Notes in Computer Science*, pages 462–473. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04696-4. doi: 10.1007/978-3-642-04697-1_43.

[80] Peter Spoer. Displacement estimation for objects on moving background. In *Image Sequence Processing and Dynamic Scene Analysis*, volume 2 of *NATO ASI Series*, pages 424–436. Springer Berlin Heidelberg, 1983. ISBN 978-3-642-81937-7. doi: 10.1007/978-3-642-81935-3_20.

[81] Yunyao Qu, Changzhou Wang, Like Gao, and Xiaoyang Sean Wang. Supporting movement pattern queries in user-specified scales. *IEEE Trans. Knowl. Data Eng.*, 15(1):26–42, 2003.

[82] Anat Levin, Lior Wolf, and Amnon Shashua. Time-varying shape tensors for scenes with multiply moving points. In *CVPR (1)*, pages 623–630, 2001.

[83] Martin Erwig, Ralf Hartmut Güting, Markus Schneider, and Michalis Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

[84] Goce Trajcevski, Ouri Wolfson, Klaus Hinrichs, and Sam Chamberlain. Managing uncertainty in moving objects databases. *ACM Trans. Database Syst.*, 29(3):463–507, September 2004. ISSN 0362-5915. doi: 10.1145/1016028.1016030.

[85] Mindaugas Pelanis, Simonas Saltenis, and Christian S. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM Trans. Database Syst.*, 31(1):255–298, 2006.

[86] Jinfeng Ni and Chinya V. Ravishankar. Pa-tree: a parametric indexing scheme for spatio-temporal trajectories. In *Proceedings of the 9th international conference on Advances in Spatial and Temporal Databases*, SSTD'05,

pages 254–272, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28127-4, 978-3-540-28127-6. doi: 10.1007/11535331_15.

[87] Hsiao-Ping Tsai, De-Nian Yang, and Ming-Syan Chen. Mining group movement patterns for tracking moving objects efficiently. *Knowledge and Data Engineering, IEEE Transactions on*, 23(2):266 –281, feb. 2011. ISSN 1041-4347. doi: 10.1109/TKDE.2009.202.

[88] Donny K. Sutantyo, Serge Kernbach, Valentin A. Nepomnyashchikh, and Paul Levi. Multi-robot searching algorithm using levy flight and artificial potential field. *CoRR*, abs/1108.5624, 2011.

[89] Injong Rhee, Minsu Shin, Seongik Hong, Kyunghan Lee, and Song Chong. On the levy-walk nature of human mobility. In *INFOCOM*, pages 924–932, 2008.

[90] Seongik Hong, Injong Rhee, Seong Joon Kim, Kyunghan Lee, and Song Chong. Routing performance analysis of human-driven delay tolerant networks using the truncated levy walk model. In *Mobility Models*, pages 25–32, 2008.

[91] Marta C. González, Cesar A. Hidalgo R., and Albert-László Barabási. Understanding individual human mobility patterns. *CoRR*, abs/0806.1256, 2008.

[92] Dan Lin, Christian S. Jensen, Beng Chin Ooi, and Simonas Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *Mobile Data Management*, pages 59–66, 2005.

[93] Huiping Cao, Nikos Mamoulis, and David W. Cheung. Mining frequent spatio-temporal sequential patterns. In *ICDM*, pages 82–89, 2005.

[94] Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the current positions of moving objects using the lazy update R-tree. In *Mobile Data Management, MDM*, pages 113–120, 2002.

[95] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 611–622. ACM, 2004.

[96] David Taniar and J. Wenny Rahayu. A taxonomy of indexing schemes for parallel database systems. *Distrib. Parallel Databases*, 12:73–106, July 2002. ISSN 0926-8782.

[97] Arne Andersson, Torben Hagerup, Johan Håstad, and Ola Petersson. The complexity of searching a sorted array of strings. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 317–325, New York, NY, USA, 1994. ISBN 0-89791-663-8.

[98] Jay Yellen Jonathan L. Gross. *Graph Theory and Its Applications*, pages 585–655. Chapman and Hall/CRC, 2005. ISBN 158488505X.

[99] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 500–509, San Francisco, CA, USA, 1994. ISBN 1-55860-153-8.

[100] Moon-Bae Song and H. Kitagawa. Managing frequent updates in r-trees for update-intensive applications. *Knowledge and Data Engineering, IEEE Transactions on*, 21(11):1573–1589, 2009. ISSN 1041-4347. doi: 10.1109/TKDE.2008.225.

[101] Alan Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985. ISBN 978-0521288811.

[102] 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*, VLDB '03, pages 802–813. VLDB Endowment, 2003. ISBN 0-12-722442-4.

[103] Maytham Safar, Dariush Ebrahimi, and David Taniar. Voronoi-based reverse nearest neighbor query processing on spatial networks. *Multimedia Syst.*, 15 (5):295–308, 2009.

[104] Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 608–619. VLDB Endowment, 2003. ISBN 0-12-722442-4.

[105] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient B+-tree based indexing of moving objects. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 768–779. VLDB Endowment, 2004. ISBN 0-12-088469-0.

Appendix A

Selected codes, and screen shots of **Indoor-tree , MOT-tree, TO-tree**.

First, we illustrate the code that responsible to build the indoor-tree. The idea is how the comparison done when inserting a new object to the tree. Basically it will start by comparing the inserted object's cell to the RC of the non-leaf node and run the algorithm that can determine the nearest expand cell that connected to the inserted object's cell. Also to determine the suitable leaf node, the algorithm will check the LE to obtain the leaf node that has an expand cell that considered as the nearest expand cell.

```
/*
 */
import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
/**
 *
 * @xxxx
 */
public class Tree {
    RootNode root = null;
    public int[][] table = null;
    ArrayList<String> cellArray = new ArrayList<String>();
    ArrayList<String> expandCellArray = new ArrayList<String>();
    public Tree() {
        makeTable();
        expandCell();
        RootNode rootNode = new RootNode();
        NonLeafNode nonLeafNode = null;
        LeafNode leafNode = null;
         for (int i = 0; i < Node.min; i++) {
           nonLeafNode = new NonLeafNode();
          for (int j = 0; j < Node.min; j++) {
              leafNode = new LeafNode();
              nonLeafNode.pointers.add(leafNode);
              }
            rootNode.pointers.add(nonLeafNode);
         }
     }
    public void searchObject(int value) {
        if (root == null) {
            System.out.println("No Element in the Tree");
        } else {
            boolean found = false;
            for (int i = 0; i < root.pointers.size(); i++) {
                NonLeafNode get = root.pointers.get(i);
                for (int j = 0; j < get.pointers.size(); j++) {
                    LeafNode get1 = get.pointers.get(j);
                    for (int k = 0; k < get1.data.size(); k++) {
                        if (get1.data.get(k) == value) {
               System.out.println("Object is found ");
```

```
            System.out.println("Non Leaf Node" + j);
            System.out.println("Leaf Node" + k);
            System.out.println("RC Max : " + get.Rc_max);
            System.out.println("RC Min : " + get.Rc_min);
            System.out.println("Leaf Data Size : " + get1.data.size());
            System.out.println("LE Val : " + get1.LE);
                        found = true;
                        break;
                    }
                }
            }
        }
        if (!found) {
            System.out.println("Object is not found");
        }
    }
}
public String findNearestCell(String cell) {
    int index = getIndex(cell);
    String ret = "";
    for (int i = index; i > 0; i--) {
        if (expandCellArray.contains("c" + i)) {
            ret = "C" + i;
            break;
        }
    }
    if (ret.equals("")) {
        for (int i = index; i < expandCellArray.size(); i++) {
            if (expandCellArray.contains("c" + i)) {
                ret = "C" + i;
                break;
            }
        }
    }
    return ret;
}

public void insert(String cell, int value) {
    if (root == null) { // if root is null
        RootNode rootNode = new RootNode();
        NonLeafNode nonLeafNode = new NonLeafNode();
        rootNode.pointers.add(nonLeafNode);
```

```
            LeafNode leafNode = new LeafNode();
            nonLeafNode.pointers.add(leafNode);
            if (expandCellArray.contains(cell.toLowerCase())) {
                nonLeafNode.Rc_max = cell;
                nonLeafNode.Rc_min = cell;
                leafNode.LE = cell;
            } else {
                String findNearestCell = findNearestCell(cell);
                nonLeafNode.Rc_max = findNearestCell;
                nonLeafNode.Rc_min = findNearestCell;
                leafNode.LE = findNearestCell;
            }
            leafNode.data.add(value);
            root = rootNode;
        } else { //if root is not null
            ArrayList<Integer> findLeafNode = findLeafNode(cell);
            NonLeafNode nonLeafNode = root.pointers.get(findLeafNode.get(0));
            LeafNode leafNode = nonLeafNode.pointers.get(findLeafNode.get(1));
            if (leafNode.data.size() == Node.max) { // if leadNode is full
                if (nonLeafNode.pointers.size() == Node.max)
                // if NonLeafNode is full
                {
           NonLeafNode newnonLeafNode = new NonLeafNode();
                     for (int i = 1; i < Node.min; i++) {
newnonLeafNode.pointers.add(nonLeafNode.pointers.remove
(nonLeafNode.pointers.size() - i));
                    }
                    root.pointers.add(newnonLeafNode);
                    LeafNode newleafNode = new LeafNode();
                    newnonLeafNode.pointers.add(newleafNode);
                    if (expandCellArray.contains(cell.toLowerCase())) {
                        newnonLeafNode.Rc_max = cell;
                        newnonLeafNode.Rc_min = cell;
                        newleafNode.LE = cell;
                    } else {
                        String findNearestCell = findNearestCell(cell);
                        newnonLeafNode.Rc_max = findNearestCell;
                        newnonLeafNode.Rc_min = findNearestCell;
                        newleafNode.LE = findNearestCell;
                    }
                    newleafNode.data.add(value);
                } else // make another leaf node
```

```
            {
                LeafNode newNode = new LeafNode();
                nonLeafNode.pointers.add(newNode);
                //add min node to this leaf node
                for (int i = 1; i < Node.min; i++) {
 newNode.data.add(leafNode.data.remove(leafNode.data.size() - i));
                }
                newNode.data.add(value);
                if (expandCellArray.contains(cell.toLowerCase())) {
                    int rcmax = 0;
                    int rcmin = 0;
                    int cellindex = 0;
rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
cellindex = expandCellArray.indexOf(cell.toLowerCase());
                    if (cellindex > rcmin) {
                        nonLeafNode.Rc_min = cell;
                    } else if (cellindex < rcmax) {
                        nonLeafNode.Rc_max = cell;
                    }
                    newNode.LE = cell;
                } else {
                    String findNearestCell = findNearestCell(cell);
                    newNode.LE = findNearestCell;
                    int rcmax = 0;
                    int rcmin = 0;
                    int cellindex = 0;
rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
cellindex = expandCellArray.indexOf(findNearestCell.toLowerCase());
                    if (cellindex > rcmin) {
                        nonLeafNode.Rc_min = findNearestCell;
                    } else if (cellindex < rcmax) {
                        nonLeafNode.Rc_max = findNearestCell;
                    }
                }
            }

        } else { //add value to leaf node
            leafNode.data.add(value);
            if (expandCellArray.contains(cell.toLowerCase())) {
                int rcmax = 0;
```

```
                int rcmin = 0;
                int cellindex = 0;
    rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
    rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
    cellindex = expandCellArray.indexOf(cell.toLowerCase());
                if (cellindex > rcmin) {
                    nonLeafNode.Rc_min = cell;
                } else if (cellindex < rcmax) {
                    nonLeafNode.Rc_max = cell;
                }
     if (cellindex < expandCellArray.indexOf(leafNode.LE.toLowerCase())) {
                    leafNode.LE = cell;
                }
            } else {
                String findNearestCell = findNearestCell(cell);
                if (nonLeafNode.Rc_min.equals("")) {
                    nonLeafNode.Rc_max = findNearestCell;
                    nonLeafNode.Rc_min = findNearestCell;
                    leafNode.LE = findNearestCell;
                } else {
                    int rcmax = 0;
                    int rcmin = 0;
                    int cellindex = 0;
     rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
     rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
     cellindex = expandCellArray.indexOf(findNearestCell.toLowerCase());
                    System.out.println("    cell index "+cellindex);
                    System.out.println("    max cell index "+rcmax);
                    System.out.println("    min cell index "+rcmin);
                    if (cellindex > rcmin) {
                        nonLeafNode.Rc_min = findNearestCell;
                    } else if (cellindex < rcmax) {
                        nonLeafNode.Rc_max = findNearestCell;
                    }
   if (cellindex < expandCellArray.indexOf(leafNode.LE.toLowerCase())) {
                        leafNode.LE = findNearestCell;
                    }
                }
            }
        }
    }
}
```

```
public ArrayList<Integer> findLeafNode(String cell) {
    List<NonLeafNode> nonLeafs = root.pointers;
    int nonLeafIndex = 0;
    int nonLeafValue = 0;
    for (int i = 0; i < nonLeafs.size(); i++) {
        NonLeafNode get = nonLeafs.get(i);
        if (get.Rc_max == null || get.Rc_min == null) {
            continue;
        }
        //System.out.println("Min : " + cell + " , " + get.Rc_min);
        //System.out.println("Max : " + cell + " , " + get.Rc_max);
        int rcmin = compare_method(cell, get.Rc_min);
        int rcmax = compare_method(cell, get.Rc_max);
        int minvalue = rcmax;
        if (rcmin <= rcmax) {
            minvalue = rcmin;
        }
        if (i == 0) {
            nonLeafValue = minvalue;
        } else {
            if (minvalue < nonLeafValue) {
                nonLeafValue = minvalue;
                nonLeafIndex = i;
            }
        }
    }
    NonLeafNode selectedNonLeafNode = nonLeafs.get(nonLeafIndex);
    List<LeafNode> leafNodes = selectedNonLeafNode.pointers;
    int leafIndex = 0;
    int leafValue = 0;
    for (int i = 0; i < leafNodes.size(); i++) {
        LeafNode get = leafNodes.get(i);
        if (get.LE == null) {
            continue;
        }
        int lemin = compare_method(cell, get.LE);
        if (lemin <= leafValue) {
            leafValue = lemin;
            leafIndex = i;
        }
    }
    ArrayList<Integer> leafNonLeafIndex = new ArrayList<Integer>();
```

```
        leafNonLeafIndex.add(nonLeafIndex);
        leafNonLeafIndex.add(leafIndex);
        return leafNonLeafIndex;
    }
    public int compare_method(String cell1, String cell2) {
        int i = getIndex(cell1);
        int j = getIndex(cell2);
        return this.table[i][j];
    }
    public int getIndex(String cell) {
        int index = -1;
        try {
index = Integer.parseInt(cell.substring(cell.toLowerCase().indexOf("c") + 1));
        } catch (Exception ex) {
System.out.println(ex.toString());
        }
 return index - 1;
    }
    private void makeTable() {
        String filename = "table.txt";
        FileInputStream fstream = null;
        try {
            int count = count(filename);
            count++;
            table = new int[count][count]; ‘
            fstream = new FileInputStream(filename);
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String strLine;
            int i = 0;
            while ((strLine = br.readLine()) != null) {
                String[] split = strLine.split(" s+");
                for (int j = 0; j < split.length; j++) {
                    table[i][j] = Integer.parseInt(split[j].trim());
                }
                i++;
            }
            for (int j = 1; j <= count; j++) {
                cellArray.add("C" + j);
            }
        } catch (IOException ex) {
        } finally {
```

```
            try {
                fstream.close();
            } catch (IOException ex) {
            }
        }
    }
    private int count(String filename) throws IOException {
   InputStream is = new BufferedInputStream(new FileInputStream(filename));
        try {
            byte[] c = new byte[1024];
            int count = 0;
            int readChars = 0;
            while ((readChars = is.read(c)) != -1) {
                for (int i = 0; i < readChars; ++i) {
                    if (c[i] == '\n') {
                        ++count;
                    }
                }
            }
            return count;
        } finally {
            is.close();
        }
    }
    private void expandCell() {
        String filename = "expandcell.txt";
        FileInputStream fstream = null;
        try {
            fstream = new FileInputStream(filename);
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String strLine;
            while ((strLine = br.readLine()) != null) {
                expandCellArray.add(strLine.trim().toLowerCase());
            }
        } catch (IOException ex) {
        } finally {
            try {
                fstream.close();
            } catch (IOException ex) {
            }
        }
```

```
    }
    public void printTable() {
        for (int i = 0; i < table.length; i++) {
            for (int j = 0; j < table[i].length; j++) {
                if (j == 0) {
                    System.out.print(table[i][j]);
                } else {
                    System.out.print("\t" + table[i][j]);
                }
            }
            System.out.println("");
        }
    }
    public void printTree() {
        for (int i = 0; i < root.pointers.size(); i++) {
            NonLeafNode get = root.pointers.get(i);
            System.out.println("_____________________________________");
            System.out.println("Non Leaf " + i);
            System.out.println("RC Max : " + get.Rc_max);
            System.out.println("RC Min : " + get.Rc_min);
            for (int j = 0; j < get.pointers.size(); j++) {
                LeafNode get1 = get.pointers.get(j);
                System.out.println("==================================");
                System.out.println("Leaf " + j);
                System.out.println("Leaf Data Size : " + get1.data.size());
                System.out.println("LE Val : " + get1.LE);
                for (int k = 0; k < get1.data.size(); k++) {

                    System.out.println("Data" + k + " : " + get1.data.get(k));
                }
                System.out.println("==================================");
            }
            System.out.println("_____________________________________");
        }
    }
}
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
import java.util.Scanner;
/**
```

```
 *
 * @xxxx
 */
public class TreeDemo {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        insertNode();
        Tree tree = new Tree();

        Scanner sc = new Scanner(System.in);
        String input;
        boolean run = true;
        do {
            System.out.println("*********************************************");
            System.out.println("For insertion Press : 1");
            System.out.println("For Searching Press : 2");
            System.out.println("For printing  Tree  : 3");
            System.out.println("Press Any other key to quit");
            System.out.println("*********************************************");
            System.out.println();
            input = sc.next().trim();
            if (input.equals("1")) {
                int value = -1;
                String cell = "";
                boolean isnumeric = false;
                do {
                    try {
                        System.out.println("Enter Cell Value");
                        value = Integer.parseInt(sc.next().trim());
                        System.out.println("Enter Cell Name");
                        cell = sc.next().trim().toUpperCase();
                        if (tree.cellArray.contains(cell)) {
                            tree.insert(cell, value);
                            isnumeric = true;
                        } else {
                            System.out.println("Cell Name should be in ");
                            for (int i = 0; i < tree.cellArray.size(); i++) {
                                System.out.print(tree.cellArray.get(i) + "\t");
                            }
```

```
                    System.out.println();
                }
            } catch (Exception ex) {
System.out.println("Value should be numeric"+ex.getMessage());
            }
        } while (!isnumeric);
    } else if (input.equals("2")) {
        System.out.println("Enter Value to Search");
        int value = -1;
        boolean isnumeric = false;
        do {
            try {
                value = Integer.parseInt(sc.next());
                isnumeric = true;
            } catch (Exception ex) {
System.out.println("Value should be numeric"+ex.getMessage());
            }
        } while (!isnumeric);
        long start = System.nanoTime();
        tree.searchObject(value);
        double elapsedTimeInSec = (System.nanoTime() - start) * 1.0e-9;
        System.out.println(elapsedTimeInSec);
    } else if (input.equals("3")) {
        tree.printTree();
    } else {
        run = false;
    }
} while (run);
 tree.printTable();
 tree.insert("C1", 1);
 tree.printTree();
 tree.insert("C2", 1);
 tree.insert("C3", 3);
 tree.insert("C2", 4);
 tree.insert("C5", 5);
 tree.insert("C2", 6);
 tree.insert("C7", 7);
 tree.insert("C2", 8);
 tree.insert("C9", 9);
 tree.insert("C8", 0);
 tree.insert("C6", 12);
 tree.insert("C2", 13);
```

```
        tree.insert("C2", 31);
        tree.insert("C9", 33);
        tree.printTree();
        tree.searchObject(36);
    }
    public static void insertNode()
    {
        //Tree tree = new Tree();
        //tree.insert("C1", 1);
       //tree.searchObject(36);
    }
}
```

Listing 1: Some Code

Note that The insertion will be done as :

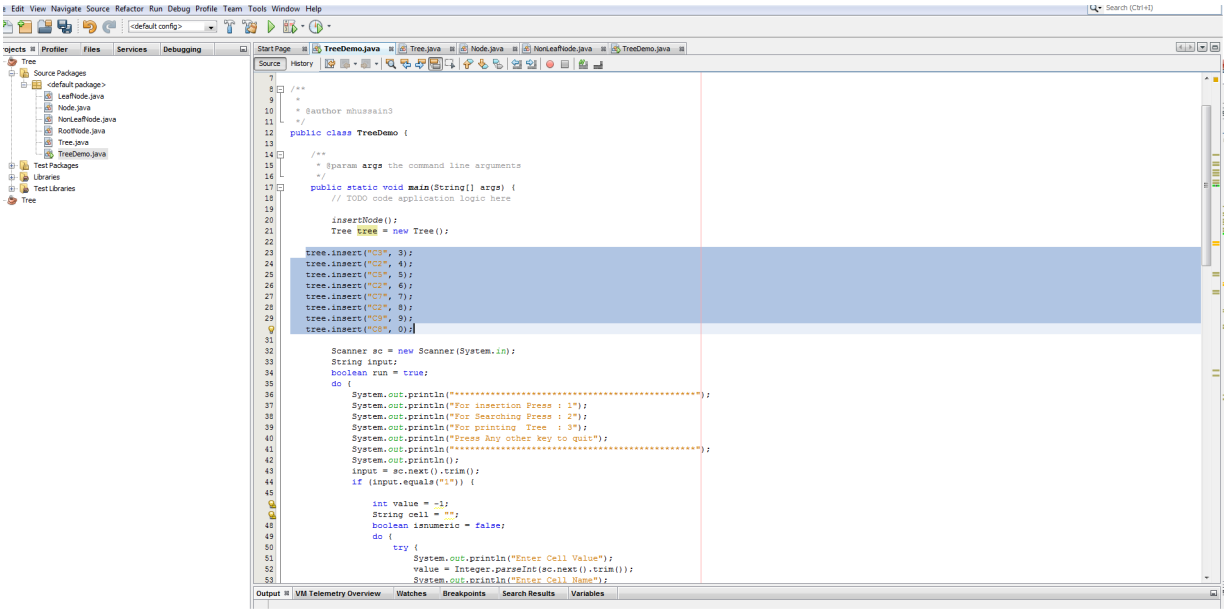
tree.insert(“C2”, 1);

tree.insert(“C3”, 3);

tree.insert(“C2”, 4);

tree.insert(“C2”, 6);

tree.insert(“Cx”, i); where i is the object id, and Cx is the cell number.



```
/**
 *
 * @author mhussain3
 */
public class TreeDemo {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        insertNode();
        Tree tree = new Tree();

        tree.insert("C3", 3);
        tree.insert("C2", 4);
        tree.insert("C5", 5);
        tree.insert("C2", 6);
        tree.insert("C7", 7);
        tree.insert("C2", 8);
        tree.insert("C9", 9);
        tree.insert("C8", 0);

        Scanner sc = new Scanner(System.in);
        String input;
        boolean run = true;
        do {
            System.out.println("************************************************");
            System.out.println("For insertion Press : 1");
            System.out.println("For Searching Press : 2");
            System.out.println("For printing  Tree  : 3");
            System.out.println("Press Any other key to quit");
            System.out.println("************************************************");
            System.out.println();
            input = sc.next().trim();
            if (input.equals("1")) {

                int value = -1;
                String cell = "";
                boolean isnumeric = false;
                do {
                    try {
                        System.out.println("Enter Cell Value");
                        value = Integer.parseInt(sc.next().trim());
                        System.out.println("Enter Cell Name");
```

FIGURE 2: Screenshot 1

0	1	2	2	3	2	1	5	3	4	4	4
1	0	1	1	2	1	2	4	2	3	3	3
2	1	0	2	3	2	3	5	3	4	4	4
2	1	2	0	1	2	3	5	3	4	4	4
3	2	3	1	0	1	4	4	2	3	3	3
2	1	2	2	1	0	3	3	1	2	2	2
1	2	3	3	4	3	0	6	4	5	5	5
5	4	5	5	4	3	6	0	2	3	3	1
3	2	3	3	2	1	4	2	0	1	1	1
4	3	4	4	3	2	5	3	1	0	1	2
4	3	4	4	3	2	5	3	1	1	0	2
4	3	4	4	3	2	5	1	1	1	2	0

FIGURE 3: The neighbors' distances table

The tree construction of the MOT-tree.

We illustrate the code that responsible to build the MOT-tree. The comparison of the RC and LE as usual. However, here the time is involved in this process. Here the object is updated only when it moves from cell to another. Also it will be treated as individual new object.

```
/*

 */

import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author xxxx
 */
public class Tree {

    RootNode root = null;
    public int[][] table = null;
    ArrayList<String> cellArray = new ArrayList<String>();
    ArrayList<String> expandCellArray = new ArrayList<String>();
    int counter = 0;

    public Tree() {
        makeTable();
        expandCell();
    }
```

```
 public void searchObject(int value) {
     if (root == null) {
         System.out.println("No Element in the Tree");

     } else {


         boolean found = false;
         for (int i = 0; i < root.pointers.size(); i++) {
             NonLeafNode get = root.pointers.get(i);
             for (int j = 0; j < get.pointers.size(); j++) {
             LeafNode get1 = get.pointers.get(j);
       for (int k = 0; k < get1.data.size(); k++) {
  if (get1.data.get(k) == value) {
System.out.println("Object is found ");
System.out.println("Non Leaf Node" + j);
System.out.println("Leaf Node" + k);
System.out.println("RC Max : " + get.Rc_max);
System.out.println("RC Min : " + get.Rc_min);
System.out.println("Leaf Data Size : " + get1.data.size());
System.out.println("LE Val : " + get1.LE);
found = true;
break;
                     }
                 }

             }

         }
         if (!found) {
             System.out.println("Object is not found");
         }
     }
 }

 public String findNearestCell(String cell) {
     int index = getIndex(cell);
     String ret = "";
     for (int i = index; i > 0; i--) {
         if (expandCellArray.contains("c" + i)) {
             ret = "C" + i;
             break;
```

```
            }
        }
        if (ret.equals("")) {
            for (int i = index; i < expandCellArray.size(); i++) {
                if (expandCellArray.contains("c" + i)) {
                    ret = "C" + i;
                    break;
                }
            }
        }
        return ret;
    }

    public void insert(String cell, int value, boolean showCounter) {




        writeTreeToFile(cell, value);
        if (root == null) { // if root is null
            emptyTreeValueFile(); //empty the tree value file
            RootNode rootNode = new RootNode();
            NonLeafNode nonLeafNode = new NonLeafNode();
            rootNode.pointers.add(nonLeafNode);
            LeafNode leafNode = new LeafNode();
            nonLeafNode.pointers.add(leafNode);

            if (expandCellArray.contains(cell.toLowerCase())) {
                nonLeafNode.Rc_max = cell;
                nonLeafNode.Rc_min = cell;
                leafNode.LE = cell;
            } else {
                String findNearestCell = findNearestCell(cell);
                nonLeafNode.Rc_max = findNearestCell;
                nonLeafNode.Rc_min = findNearestCell;
                leafNode.LE = findNearestCell;
            }
            leafNode.data.add(value);
            root = rootNode;
            if (showCounter) {
                System.out.println("2 nodes are modified in this update");
            }
        } else { //if root is not null
```

```
            ArrayList<Integer> findLeafNode = findLeafNode(cell);
            NonLeafNode nonLeafNode = root.pointers.get(findLeafNode.get(0));
            LeafNode leafNode = nonLeafNode.pointers.get(findLeafNode.get(1));

            if (leafNode.data.size() == Node.max) {
            // if leadNode is full

                if (nonLeafNode.pointers.size() == Node.max)
                 // if NonLeafNode is full
                {

                    NonLeafNode newnonLeafNode = new NonLeafNode();
                    for (int i = 1; i < Node.min; i++) {
newnonLeafNode.pointers.add(nonLeafNode.pointers.remove(
nonLeafNode.pointers.size() - i));
                    }

                    root.pointers.add(newnonLeafNode);
                    LeafNode newleafNode = new LeafNode();
                    newnonLeafNode.pointers.add(newleafNode);
                    if (expandCellArray.contains(cell.toLowerCase())) {
                        newnonLeafNode.Rc_max = cell;
                        newnonLeafNode.Rc_min = cell;
                        newleafNode.LE = cell;
                    } else {
                        String findNearestCell = findNearestCell(cell);
                        newnonLeafNode.Rc_max = findNearestCell;
                        newnonLeafNode.Rc_min = findNearestCell;
                        newleafNode.LE = findNearestCell;
                    }
                    newleafNode.data.add(value);
                    if (showCounter) {
                        System.out.println("2 nodes are modified in this update");
                    }

                } else // make another leaf node
                {
                    LeafNode newNode = new LeafNode();
                    nonLeafNode.pointers.add(newNode);


                    //add min node to this leaf node
```

```
                for (int i = 1; i < Node.min; i++) {
      newNode.data.add(leafNode.data.remove(leafNode.data.size() - i));
                }
                newNode.data.add(value);

                if (expandCellArray.contains(cell.toLowerCase())) {

                    int rcmax = 0;
                    int rcmin = 0;
                    int cellindex = 0;

   rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
   rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
   cellindex = expandCellArray.indexOf(cell.toLowerCase());
                    if (cellindex > rcmin) {
                        nonLeafNode.Rc_min = cell;
                    } else if (cellindex < rcmax) {
                        nonLeafNode.Rc_max = cell;
                    }
                    newNode.LE = cell;

                } else {
                    String findNearestCell = findNearestCell(cell);
                    newNode.LE = findNearestCell;

                    int rcmax = 0;
                    int rcmin = 0;
                    int cellindex = 0;

  rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
  rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
  cellindex = expandCellArray.indexOf(findNearestCell.toLowerCase());
                    if (cellindex > rcmin) {
                        nonLeafNode.Rc_min = findNearestCell;
                    } else if (cellindex < rcmax) {
                        nonLeafNode.Rc_max = findNearestCell;
                    }

                }
                if (showCounter) {
System.out.println("1 nodes are modified in this update");
                }
```

```
            }

         } else { //add value to leaf node
            leafNode.data.add(value);
            if (expandCellArray.contains(cell.toLowerCase())) {

                int rcmax = 0;
                int rcmin = 0;
                int cellindex = 0;

      rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
      rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
      cellindex = expandCellArray.indexOf(cell.toLowerCase());
                if (cellindex > rcmin) {
                    nonLeafNode.Rc_min = cell;
                } else if (cellindex < rcmax) {
                    nonLeafNode.Rc_max = cell;
                }
   if (cellindex < expandCellArray.indexOf(leafNode.LE.toLowerCase())) {
                    leafNode.LE = cell;
                }

            } else {
                String findNearestCell = findNearestCell(cell);
                if (nonLeafNode.Rc_min.equals("")) {
                    nonLeafNode.Rc_max = findNearestCell;
                    nonLeafNode.Rc_min = findNearestCell;
                    leafNode.LE = findNearestCell;
                } else {
                    int rcmax = 0;
                    int rcmin = 0;
                    int cellindex = 0;

       rcmax = expandCellArray.indexOf(nonLeafNode.Rc_max.toLowerCase());
       rcmin = expandCellArray.indexOf(nonLeafNode.Rc_min.toLowerCase());
       cellindex = expandCellArray.indexOf(findNearestCell.toLowerCase());
                    System.out.println("    cell index "+cellindex);
                    System.out.println("    min cell index "+rcmin);
                    if (cellindex > rcmin) {
                        nonLeafNode.Rc_min = findNearestCell;
                    } else if (cellindex < rcmax) {
                        nonLeafNode.Rc_max = findNearestCell;
```

```
                    }
  if (cellindex < expandCellArray.indexOf(leafNode.LE.toLowerCase())) {
                        leafNode.LE = findNearestCell;
                    }
                }

            }
            if (showCounter) {
                System.out.println("1 nodes are modified in this update");
            }
        }
    }

}

public ArrayList<Integer> findLeafNode(String cell) {
    List<NonLeafNode> nonLeafs = root.pointers;
    int nonLeafIndex = 0;
    int nonLeafValue = 0;
    for (int i = 0; i < nonLeafs.size(); i++) {
        NonLeafNode get = nonLeafs.get(i);
        if (get.Rc_max == null || get.Rc_min == null) {
            continue;
        }
        //System.out.println("Min : " + cell + " , " + get.Rc_min);
        //System.out.println("Max : " + cell + " , " + get.Rc_max);
        int rcmin = compare_method(cell, get.Rc_min);
        int rcmax = compare_method(cell, get.Rc_max);
        int minvalue = rcmax;
        if (rcmin <= rcmax) {
            minvalue = rcmin;
        }
        if (i == 0) {
            nonLeafValue = minvalue;
        } else {
            if (minvalue < nonLeafValue) {
                nonLeafValue = minvalue;
                nonLeafIndex = i;
            }
        }
    }
    NonLeafNode selectedNonLeafNode = nonLeafs.get(nonLeafIndex);
```

```
        List<LeafNode> leafNodes = selectedNonLeafNode.pointers;
        int leafIndex = 0;
        int leafValue = 0;
        for (int i = 0; i < leafNodes.size(); i++) {
            LeafNode get = leafNodes.get(i);
            if (get.LE == null) {
                continue;
            }
            int lemin = compare_method(cell, get.LE);
            if (lemin <= leafValue) {
                leafValue = lemin;
                leafIndex = i;
            }
        }


        ArrayList<Integer> leafNonLeafIndex = new ArrayList<Integer>();
        leafNonLeafIndex.add(nonLeafIndex);
        leafNonLeafIndex.add(leafIndex);


        return leafNonLeafIndex;
    }

    public int compare_method(String cell1, String cell2) {

        int i = getIndex(cell1);
        int j = getIndex(cell2);

        return this.table[i][j];
    }

    public int getIndex(String cell) {
        int index = -1;
        try {
index = Integer.parseInt(cell.substring(cell.toLowerCase().indexOf("c") + 1));
        } catch (Exception ex) {
            System.out.println(ex.toString());
        }
        return index - 1;
    }
```

```
 private void makeTable() {
     String filename = "table.txt";
     FileInputStream fstream = null;

     try {
         int count = count(filename);
         count++;
         table = new int[count][count];
         fstream = new FileInputStream(filename);
         DataInputStream in = new DataInputStream(fstream);
         BufferedReader br = new BufferedReader(new InputStreamReader(in));
         String strLine;
         int i = 0;
         while ((strLine = br.readLine()) != null) {
             String[] split = strLine.split("\\s+");
             for (int j = 0; j < split.length; j++) {
                 table[i][j] = Integer.parseInt(split[j].trim());
             }

             i++;

         }
         for (int j = 1; j <= count; j++) {
             cellArray.add("C" + j);
         }

     } catch (IOException ex) {
     } finally {
         try {
             fstream.close();
         } catch (IOException ex) {
         }
     }

 }

 private int count(String filename) throws IOException {
InputStream is = new BufferedInputStream(new FileInputStream(filename));
     try {
         byte[] c = new byte[1024];
         int count = 0;
```

```
            int readChars = 0;
            while ((readChars = is.read(c)) != -1) {
                for (int i = 0; i < readChars; ++i) {
                    if (c[i] == '\n') {
                        ++count;
                    }
                }
            }
            return count;
        } finally {
            is.close();
        }
    }

    private void expandCell() {
        String filename = "expandcell.txt";
        FileInputStream fstream = null;

        try {
            fstream = new FileInputStream(filename);
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String strLine;
            while ((strLine = br.readLine()) != null) {
                expandCellArray.add(strLine.trim().toLowerCase());
            }

        } catch (IOException ex) {
        } finally {
            try {
                fstream.close();
            } catch (IOException ex) {
            }
        }
    }

    public void printTable() {
        for (int i = 0; i < table.length; i++) {
            for (int j = 0; j < table[i].length; j++) {
                if (j == 0) {
                    System.out.print(table[i][j]);
                } else {
```

```
                System.out.print("\t" + table[i][j]);
            }
        }
        System.out.println("");
    }
}

public void writeTreeToFile(String cell, int value) {
    try {
         // Create file

        String filename = "saveTree.txt";
        FileWriter fwo = new FileWriter(filename, true);
        // false means we will be writing to the file
        //BufferedWriter bwObj = new BufferedWriter(fwo);
        fwo.write(cell + "," + value + "\r\n");

        //Close the output stream
        fwo.close();
    } catch (Exception e) {//Catch exception if any
        //System.err.println("Error: " + e.getMessage());
    }
}

void loadFromFile() {
    if (root != null) {
        root.pointers.clear();
        root = null;
    }
    String filename = "saveTree.txt";
    FileInputStream fstream = null;
    List<NodeElementData> list = new ArrayList<NodeElementData>();
    try {
        fstream = new FileInputStream(filename);
        DataInputStream in = new DataInputStream(fstream);
        BufferedReader br = new BufferedReader(new InputStreamReader(in));
        String strLine;
        while ((strLine = br.readLine()) != null) {
            String[] split = strLine.trim().toLowerCase().split(",");
            if (split.length == 2) {
     list.add(new NodeElementData(split[0], Integer.parseInt(split[1])));
            }
```

```
            }
            for (int i = 0; i < list.size(); i++) {
                insert(list.get(i).data, list.get(i).value, false);
            }

        } catch (IOException ex) {
        } finally {
            try {
                fstream.close();
            } catch (IOException ex) {
            }
        }
    }

    public void emptyTreeValueFile() {
        String filename = "saveTree.txt";
        try {
            FileWriter fwo = new FileWriter(filename, false);
            fwo.write("");
            fwo.close();
        } catch (Exception e) {//Catch exception if any
            //System.err.println("Error: " + e.getMessage());
        }
    }

    public void printTree() {
        if (root != null) {
            for (int i = 0; i < root.pointers.size(); i++) {
                NonLeafNode get = root.pointers.get(i);
System.out.println("_____________________________________");
System.out.println("Non Leaf " + i);
System.out.println("Time " + (new java.util.Date(get.creationTime).toGMTString()));
System.out.println("RC Max : " + get.Rc_max);
System.out.println("RC Min : " + get.Rc_min);
                for (int j = 0; j < get.pointers.size(); j++) {
          LeafNode get1 = get.pointers.get(j);
          System.out.println("================================");
          System.out.println("Leaf " + j);
          System.out.println("Leaf Data Size : " + get1.data.size());
          System.out.println("LE Val : " + get1.LE);

                    for (int k = 0; k < get1.data.size(); k++) {
```

```
                        System.out.println("Data" + k + " : " + get1.data.get(k));

                    }
                    System.out.println("================================");
                }
                System.out.println("_____________________________________");
            }
        } else {
            System.out.println("Tree is empty");
        }
    }
}
```

Listing 2: Some Code

The tree construction of the TO-tree.

Based on the adjacency method, It starts by grouping the objects inside the same cell. Splitting will be performed internally to group it with the objects in the same cell in the case of any overflowing node. The idea is to start to retrieve the adjacent cells of the target cells (using the Connection Cells Algorithm). Then the Adjacent Level Algorithm is used in order to start narrowing down to the most suitable one; this algorithm checks the nearest adjacent cell by calculating the distance between the cell's centers (Voronoi generators) which is already known in the system. Note that the compassion will use the NPosition in the non-leaf-node to determine the suitable non-leaf nodes. Also will use LNPosition in the leaf-node to indicated the suitable leaf nodes.

```
/*

 */
```

```
import java.io.BufferedInputStream;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

/**
 *
 * @author xxx
 */
public class Tree {

    RootNode root = null;
    public int[][] table = null;
    public int[][] distanceTable = null;
    ArrayList<String> expandCellArray = new ArrayList<String>();
    int tableSize = 11;

    public Tree() {
        makeTable();
        makeDistanceTable();
        expandCell();
    }

    public void searchObject(int value) {
        if (root == null) {
            System.out.println("No Element in the Tree");

        } else {
            boolean found = false;
            for (int i = 0; i < root.pointers.size(); i++) {
                NonLeafNode get = root.pointers.get(i);
                for (int j = 0; j < get.pointers.size(); j++) {
                    LeafNode get1 = get.pointers.get(j);
                    for (int k = 0; k < get1.data.size(); k++) {
```

```
                    if (get1.data.get(k) == value) {
            System.out.println("Object is found ");
            System.out.println("Non Leaf Node" + j);
            System.out.println("Leaf Node" + k);
            System.out.println("NPosition : " + get.NPosition.toString());
            System.out.println("Leaf Data Size : " + get1.data.size());
            System.out.println("LNPosition : " + get1.LNPosition.toString());
                        found = true;
                        break;
                    }
                }

            }
        }
        if (!found) {
            System.out.println("Object is not found");
        }
    }
}

public void insert(String cell, int value) {
    if (root == null) { // if root is null
        RootNode rootNode = new RootNode();
        NonLeafNode nonLeafNode = new NonLeafNode();
        rootNode.pointers.add(nonLeafNode);
        LeafNode leafNode = new LeafNode();
        nonLeafNode.pointers.add(leafNode);

        nonLeafNode.NPosition.add(cell);
        leafNode.LNPosition.add(cell);

        leafNode.data.add(value);
        root = rootNode;
    } else { //if root is not null
        ArrayList<Integer> findLeafNode = findLeafNode(cell);
        NonLeafNode nonLeafNode = root.pointers.get(findLeafNode.get(0));
        LeafNode leafNode = nonLeafNode.pointers.get(findLeafNode.get(1));

        if (leafNode.data.size() == Node.max) { // if leadNode is full

            if (nonLeafNode.pointers.size() == Node.max)
            // if NonLeafNode is full
```

```
                {

                    NonLeafNode newnonLeafNode = new NonLeafNode();
                    for (int i = 1; i < Node.min; i++) {
newnonLeafNode.pointers.add(nonLeafNode.pointers.remove
(nonLeafNode.pointers.size() - 1));
                        if (newnonLeafNode.NPosition.size() - 1 > 0) {
newnonLeafNode.NPosition.add(nonLeafNode.NPosition.remove
(nonLeafNode.NPosition.size() - 1));
                        } else if (leafNode.LNPosition.size() - 1 == 0) {
newnonLeafNode.NPosition.add(nonLeafNode.NPosition.get
(nonLeafNode.NPosition.size() - 1));
                        }
                    }
                    newnonLeafNode.NPosition.remove(cell);
                    newnonLeafNode.NPosition.add(cell);

                    root.pointers.add(newnonLeafNode);
                    LeafNode newleafNode = new LeafNode();
                    newnonLeafNode.pointers.add(newleafNode);
                    newleafNode.data.add(value);

                    newleafNode.LNPosition.remove(cell);
                    newleafNode.LNPosition.add(cell);

                } else // make another leaf node
                {
                    LeafNode newNode = new LeafNode();
                    nonLeafNode.pointers.add(newNode);

                    nonLeafNode.NPosition.remove(cell);
                    nonLeafNode.NPosition.add(cell);
                    //add min node to this leaf node
                    for (int i = 1; i < Node.min; i++) {
newNode.data.add(leafNode.data.remove(leafNode.data.size() - 1));
                        if (leafNode.LNPosition.size() - 1 > 0) {
newNode.LNPosition.add(leafNode.LNPosition.remove(leafNode.LNPosition.size() - 1));
                        } else if (leafNode.LNPosition.size() - 1 == 0) {
newNode.LNPosition.add(leafNode.LNPosition.get(leafNode.LNPosition.size() - 1));
                        }
                    }
                    newNode.data.add(value);
```

```
                newNode.LNPosition.remove(cell);
                newNode.LNPosition.add(cell);

            }

        } else { //add value to leaf node
            nonLeafNode.NPosition.remove(cell);
            nonLeafNode.NPosition.add(cell);

            leafNode.LNPosition.remove(cell);
            leafNode.LNPosition.add(cell);

            leafNode.data.add(value);

        }
    }
}

public ArrayList<Integer> findLeafNode(String cell) {
    //this.printTree();
    HashMap<Integer, Integer> nonleafslevel = new HashMap<Integer, Integer>();
    List<NonLeafNode> nonLeafs = root.pointers;
    int nonLeafIndex = 0;
    int nonLeafValue = 0;
    int minnonLeafValue = 0;
    int minLeafValue = 0;
    //compare level
    for (int i = 0; i < nonLeafs.size(); i++) {
        NonLeafNode get = nonLeafs.get(i);
        for (int j = 0; j < get.NPosition.size(); j++) {
            int level = getLevel(cell, get.NPosition.get(j));
            if (j == 0) {
                nonLeafValue = level;
            } else {
                if (level < nonLeafValue) {
                    nonLeafValue = level;

                }
            }
        }
```

```
            if (i == 0) {
                minnonLeafValue = nonLeafValue;
                nonLeafIndex = i;
                nonleafslevel.put(i, nonLeafValue);
            } else {
                if (nonLeafValue < minnonLeafValue) {
                    nonleafslevel.clear();
                    nonleafslevel.put(i, nonLeafValue);
                    nonLeafIndex = i;
                } else if (nonLeafValue == minnonLeafValue) {
                    nonleafslevel.put(i, nonLeafValue);
                    nonLeafIndex = i;
                }
            }
        }
//now compare distance if hashmap has more than one entries
// which has same min level
        if (nonleafslevel.size() > 1) {
            int mindistance = 0;
            int i = 0;
            for (Map.Entry<Integer, Integer> entry : nonleafslevel.entrySet()) {
                NonLeafNode get = nonLeafs.get(entry.getKey());
                for (int j = 0; j < get.NPosition.size(); j++) {
                    int distance = getDistance(cell, get.NPosition.get(j));
                    if (i == 0) {
                        mindistance = distance;
                        nonLeafIndex = i;
                    } else {
                        if (distance < mindistance) {
                            mindistance = distance;
                            nonLeafIndex = i;
                        }
                    }
                }
                i++;
            }
        } else {
            for (Map.Entry<Integer, Integer> entry : nonleafslevel.entrySet()) {
                nonLeafIndex = entry.getKey();
            }
        }
        NonLeafNode selectedNonLeafNode = nonLeafs.get(nonLeafIndex);
```

```
HashMap<Integer, Integer> leafslevel = new HashMap<Integer, Integer>();
List<LeafNode> leafNodes = selectedNonLeafNode.pointers;
int leafIndex = 0;
int leafValue = 0;
for (int i = 0; i < leafNodes.size(); i++) {
    LeafNode get = leafNodes.get(i);
    for (int j = 0; j < get.LNPosition.size(); j++) {
        int level = getLevel(cell, get.LNPosition.get(j));
        if (j == 0) {
            leafValue = level;
        } else {
            if (level <= leafValue) {
                leafValue = level;
                leafIndex = i;
            }
        }
    }
    if (i == 0) {
        minLeafValue = leafValue;
        leafIndex = i;
        leafslevel.put(i, nonLeafValue);
    } else {
        if (nonLeafValue < minLeafValue) {
            leafslevel.clear();
            leafslevel.put(i, nonLeafValue);
            leafIndex = i;
        } else if (nonLeafValue == minLeafValue) {
            leafslevel.put(i, nonLeafValue);
            leafIndex = i;
        }
    }

}
//now compare distance if hashmap has more
//than one entries which has same min level
if (leafslevel.size() > 1) {
    int mindistance = 0;
    int i = 0;
    for (Map.Entry<Integer, Integer> entry : leafslevel.entrySet()) {
        LeafNode get = leafNodes.get(entry.getKey());
        for (int j = 0; j < get.LNPosition.size(); j++) {
            int distance = getDistance(cell, get.LNPosition.get(j));
```

```
                    if (i == 0) {
                        mindistance = distance;
                        leafIndex = i;
                    } else {

                        if (distance < mindistance) {
                            mindistance = distance;
                            leafIndex = i;
                        }
                    }
//System.out.println("distance is "+distance+" min dis
//"+mindistance+" leaf index "+leafIndex);
                }
                i++;
            }
        } else {
            for (Map.Entry<Integer, Integer> entry : leafslevel.entrySet()) {
                leafIndex = entry.getKey();
                //System.out.println("I am here leaf Index : " + leafIndex);
            }
        }


        ArrayList<Integer> leafNonLeafIndex = new ArrayList<Integer>();
        leafNonLeafIndex.add(nonLeafIndex);
        leafNonLeafIndex.add(leafIndex);
        return leafNonLeafIndex;
    }

    public int getDistance(String cell1, String cell2) {
        int i = getIndex(cell1);
        int j = getIndex(cell2);
        return this.distanceTable[i][j];
    }

    public int getLevel(String cell1, String cell2) {
        int i = getIndex(cell1);
        int j = getIndex(cell2);
        return this.table[i][j];
    }

    public int getIndex(String cell) {
```

```
        int index = -1;
        try {
index = Integer.parseInt(cell.substring(cell.toUpperCase().indexOf("C") + 1));
        } catch (Exception ex) {
            System.out.println(ex.toString());
        }
        return index - 1;
    }

    private void makeTable() {
        table = new int[tableSize][tableSize];
        for (int i = 0; i < tableSize; i++) {
            for (int j = 0; j < tableSize; j++) {
                table[i][j] = 0;
            }
        }
        String filename = "table.txt";
        FileInputStream fstream = null;

        try {
            int count = count(filename);
            count++;
            table = new int[count][count];
            fstream = new FileInputStream(filename);
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String strLine;
            int i = 0;
            while ((strLine = br.readLine()) != null) {
                String[] split = strLine.split("\\s+");
                for (int j = 0; j < split.length; j++) {
                    table[i][j] = Integer.parseInt(split[j].trim());
                }
                i++;
            }

        } catch (IOException ex) {
        } finally {
            try {
                fstream.close();
            } catch (IOException ex) {
            }
```

```
        }
    }

    private void makeDistanceTable() {
        distanceTable = new int[tableSize][tableSize];
        for (int i = 0; i < tableSize; i++) {
            for (int j = 0; j < tableSize; j++) {
                distanceTable[i][j] = 0;
            }
        }
        String filename = "distancetable.txt";
        FileInputStream fstream = null;

        try {
            int count = count(filename);
            count++;
            distanceTable = new int[count][count];
            fstream = new FileInputStream(filename);
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String strLine;
            int i = 0;
            while ((strLine = br.readLine()) != null) {
                String[] split = strLine.split("\\s+");
                for (int j = 0; j < split.length; j++) {
                    distanceTable[i][j] = Integer.parseInt(split[j].trim());
                }
                i++;
            }

        } catch (IOException ex) {
        } finally {
            try {
                fstream.close();
            } catch (IOException ex) {
            }
        }
    }

    private void expandCell() {
        String filename = "expandcell.txt";
        FileInputStream fstream = null;
```

```
        try {
            fstream = new FileInputStream(filename);
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new InputStreamReader(in));
            String strLine;
            while ((strLine = br.readLine()) != null) {
                expandCellArray.add(strLine.trim().toUpperCase());
            }

        } catch (IOException ex) {
        } finally {
            try {
                fstream.close();
            } catch (IOException ex) {
            }
        }
    }

    private int count(String filename) throws IOException {
        InputStream is = new BufferedInputStream(new FileInputStream(filename));
        try {
            byte[] c = new byte[1024];
            int count = 0;
            int readChars = 0;
            while ((readChars = is.read(c)) != -1) {
                for (int i = 0; i < readChars; ++i) {
                    if (c[i] == '\n') {
                        ++count;
                    }
                }
            }
            return count;
        } finally {
            is.close();
        }
    }

    public void printLevelTable() {
        for (int i = 0; i < table.length; i++) {
            for (int j = 0; j < table[i].length; j++) {
                if (j == 0) {
```

```
                System.out.print(table[i][j]);
            } else {
                System.out.print("\t" + table[i][j]);
            }
        }
        System.out.println("");
    }
}

public void printDistanceTable() {
    for (int i = 0; i < distanceTable.length; i++) {
        for (int j = 0; j < distanceTable[i].length; j++) {
            if (j == 0) {
                System.out.print(distanceTable[i][j]);
            } else {
                System.out.print("\t" + distanceTable[i][j]);
            }
        }
        System.out.println("");
    }
}

public void printTree() {
    if (root != null) {
        for (int i = 0; i < root.pointers.size(); i++) {
            NonLeafNode get = root.pointers.get(i);
       System.out.println("_____________________________________");
       System.out.println("Non Leaf " + i);
       System.out.println("NPosition : " + get.NPosition.toString());
       for (int j = 0; j < get.pointers.size(); j++) {
          LeafNode get1 = get.pointers.get(j);
          System.out.println("================================");
          System.out.println("Leaf " + j);
          System.out.println("Leaf Data Size : " + get1.data.size());
          System.out.println("LNPosition : " + get1.LNPosition.toString());

           for (int k = 0; k < get1.data.size(); k++) {

         System.out.println("Data" + k + " : " + get1.data.get(k));

                }
       System.out.println("================================");
```

```
                }
            System.out.println("_____________________________________");
             }
        } else {
             System.out.println("Tree is Empty!");
        }
    }
}
```

Listing 3: Some Code