INAV: A SPATIAL MODEL SUPPORTING ROUTE PLANNING IN INDOOR SPACE

By

WENJIE YUAN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY UNIVERSITY OF FLORIDA 2011
ACKNOWLEDGMENTS

First, I would like to thank my supervisor, Dr. Markus Schneider, for providing the guidance and support throughout my research. I appreciate his knowledge and skill and his assistance in writing reports. Second, I would like to thank the other members of my committee (Dr. Howard Beck, Dr. Alin Dobra, Dr. Jeffrey Ho, and Dr. Joseph Wilson), for the assistance they provided.

I would also like to thank my family for the support they provide me through my entire life.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>3</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>11</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1  INTRODUCTION</td>
<td>13</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>16</td>
</tr>
<tr>
<td>1.3 Contributions and Approaches</td>
<td>16</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>18</td>
</tr>
<tr>
<td>2  RELATED WORK</td>
<td>19</td>
</tr>
<tr>
<td>2.1 Indoor Navigation Models</td>
<td>19</td>
</tr>
<tr>
<td>2.1.1 Symbolic Models</td>
<td>19</td>
</tr>
<tr>
<td>2.1.2 Semantic Models</td>
<td>20</td>
</tr>
<tr>
<td>2.1.3 Geometric Models</td>
<td>22</td>
</tr>
<tr>
<td>2.1.4 3D-based Models</td>
<td>25</td>
</tr>
<tr>
<td>2.2 Space Syntax Graph</td>
<td>26</td>
</tr>
<tr>
<td>2.3 Approaches to Supporting Range Queries in Outdoor Space</td>
<td>28</td>
</tr>
<tr>
<td>2.3.1 Stationary Range Queries</td>
<td>28</td>
</tr>
<tr>
<td>2.3.2 Continuous Range Queries</td>
<td>29</td>
</tr>
<tr>
<td>3  SUPPORTING THE SHORTEST PATH ROUTE PLANNING IN INDOOR SPACE</td>
<td>31</td>
</tr>
<tr>
<td>3.1 Problems in Existing Indoor Navigation Models</td>
<td>31</td>
</tr>
<tr>
<td>3.2 The Path Construction</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1 Simple Cells</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 Complex Cells</td>
<td>35</td>
</tr>
<tr>
<td>3.2.2.1 Flat complex cells</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2.2 Nested complex cells</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2.3 Implicit path segments in flat complex cells</td>
<td>40</td>
</tr>
<tr>
<td>3.2.2.4 Implicit path segments in nested complex cells</td>
<td>43</td>
</tr>
<tr>
<td>3.2.2.5 An algorithm for computing all shortest path segments in complex cells</td>
<td>46</td>
</tr>
<tr>
<td>3.2.3 Open Cells</td>
<td>51</td>
</tr>
<tr>
<td>3.2.4 Connecters</td>
<td>54</td>
</tr>
<tr>
<td>3.2.5 Accessibility</td>
<td>54</td>
</tr>
<tr>
<td>3.3 The iNav Model: Shortest Path Routing</td>
<td>56</td>
</tr>
</tbody>
</table>
6.2.2 Experiments

6.2.2.1 Experiments of route planning in 2D indoor spaces

6.2.2.2 Experiments of range query in 2D indoor spaces

6.2.2.3 Experiments of route planning for arbitrary-shaped users

7 CONCLUSIONS

A THE XSD FILE OF THE FORMATTED DATA FOR THE 2D SYSTEM

B AN EXAMPLE OF THE FORMATTED DATA FOR THE 3D SYSTEM

REFERENCES

BIOGRAPHICAL SKETCH
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>67</td>
</tr>
<tr>
<td>4-1</td>
<td>84</td>
</tr>
</tbody>
</table>

3-1 The path chain from room106 to room103

4-1 Interval representation for accesspoints in Figure 4-8B
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Models for representing indoor spaces.</td>
<td>26</td>
</tr>
<tr>
<td>2-2</td>
<td>Multiple configurations of convex spaces for one space.</td>
<td>28</td>
</tr>
<tr>
<td>2-3</td>
<td>Example of axial maps.</td>
<td>28</td>
</tr>
<tr>
<td>3-1</td>
<td>Indoor representation models.</td>
<td>32</td>
</tr>
<tr>
<td>3-2</td>
<td>Medial axis.</td>
<td>32</td>
</tr>
<tr>
<td>3-3</td>
<td>An example of Lorenz’s model.</td>
<td>33</td>
</tr>
<tr>
<td>3-4</td>
<td>Different types of cells.</td>
<td>36</td>
</tr>
<tr>
<td>3-5</td>
<td>Nested cells.</td>
<td>37</td>
</tr>
<tr>
<td>3-6</td>
<td>Path segments in different types of cells.</td>
<td>40</td>
</tr>
<tr>
<td>3-7</td>
<td>Examples of concave vertices and concave boundaries.</td>
<td>41</td>
</tr>
<tr>
<td>3-8</td>
<td>The shortest paths in a concave polygon.</td>
<td>41</td>
</tr>
<tr>
<td>3-9</td>
<td>Proof of Lemma 1</td>
<td>43</td>
</tr>
<tr>
<td>3-10</td>
<td>Layer structure of nested cells.</td>
<td>43</td>
</tr>
<tr>
<td>3-11</td>
<td>Proof cases for the inner polygon boundaries of a complex cell with a hierarchy of depth 2</td>
<td>45</td>
</tr>
<tr>
<td>3-12</td>
<td>Example of a face with two access points and two possible paths between them</td>
<td>45</td>
</tr>
<tr>
<td>3-13</td>
<td>Constructing path segments in open cells.</td>
<td>53</td>
</tr>
<tr>
<td>3-14</td>
<td>Maps and path segments.</td>
<td>55</td>
</tr>
<tr>
<td>3-15</td>
<td>An example of navigation.</td>
<td>57</td>
</tr>
<tr>
<td>3-16</td>
<td>Route planning where the starting node is an intermediate point.</td>
<td>59</td>
</tr>
<tr>
<td>3-17</td>
<td>The navigation descriptions</td>
<td>62</td>
</tr>
<tr>
<td>3-18</td>
<td>The calculation of angles and directions</td>
<td>64</td>
</tr>
<tr>
<td>3-19</td>
<td>The status transformation of the language generation and directions</td>
<td>66</td>
</tr>
<tr>
<td>4-1</td>
<td>R-tree.</td>
<td>70</td>
</tr>
<tr>
<td>4-2</td>
<td>Range Network Expansion.</td>
<td>70</td>
</tr>
</tbody>
</table>
4-3 Continuous query using split point ........................................ 73
4-4 Euclidean-based continuous range query .............................. 73
4-5 Network-based continuous range query ............................... 74
4-6 The Incremental Network Expansion ..................................... 75
4-7 The Indoor Range Network Expansion ................................. 80
4-8 Finding split points. ......................................................... 81
5-1 Examples of cubes. ......................................................... 87
5-2 The cube representation. ................................................... 88
5-3 Merging cubes to generate larger blocks. ............................ 91
5-4 Relationships between adjacent blocks. .............................. 96
5-5 The connectors with the maximum width between a plane and a stair cannot be found by using the basic merging Algorithm ............................ 97
5-6 Computing connectors between a plane and a stair. ............... 98
5-7 Examples showing difficulties to find the optimal routes. .......... 99
5-8 Demonstrations of checking accessibility in different scenarios .... 102
5-9 Refinement the approach for evaluating the maximum length for 90° corners .................................................. 104
5-10 The LEGO graph. ........................................................... 104
6-1 The map used to evaluating the efficiency. ............................ 108
6-2 Evaluation of the performance for route planning. ................. 109
6-3 Evaluation of the performance for route planning ................. 109
6-4 Evaluation of the performance for stationary range queries ...... 110
6-5 Evaluation comparison between IRNE and IRD. ................. 111
6-6 The architecture of the iNav system ................................. 112
6-7 The user interface ......................................................... 115
6-8 The visualized route from room 502 to 515 .......................... 118
6-9 The results of Example 1. .............................................. 118
6-10 The results of Example 2. ............................................. 119
6-11 The results of Example 3. .......................................................... 120
6-12 The results of Example 4. .......................................................... 121
6-13 The results of Example 5. .......................................................... 121
6-14 The results of Example 6. .......................................................... 122
INAV: A SPATIAL MODEL SUPPORTING ROUTE PLANNING IN INDOOR SPACE

By

Wenjie Yuan

August 2011

Chair: Markus Schneider
Major: Computer Engineering

With increasing requests for indoor navigation, route planning for indoor spaces is highly desirable especially when the indoor structure is large and complicated. This dissertation is to design a spatial model for supporting route planning. In our context, the primary goal of route planning is to provide a desired route in a specific environment from a certain source point. Our research mainly focuses on the following three open problems, which have not been solved in the related literature. First, distance-driven routing aims at finding the shortest path to a given target object, which is an essential problem in an indoor navigation system. Second, range-driven routing is to generate a set of routes to the qualifying objects that are in a specific walking-distance range from the source point. The source location can be static or dynamic, and thus, the determination of qualifying target objects can be classified to two types, stationary and continuous range queries. Third, when the object moving through the route (e.g., the wheelchairs or moving carts) cannot be approximated by a point, size-driven routing is necessary to offer a feasible path for the object with a particular size (i.e., the length, width and height) to reach a given target object.

Concentrating on the above three open problems, our research makes the following specific contributions: First, for discovering the shortest routes, we need to generate a distance-based graph from a cell-based indoor map, which is completely different from a network-based outdoor map. In our solution, a distance-based Direct Path Graph
(DPG) is introduced, where nodes denote the exits and edges represent the shortest path segments between the exits. The segment computation is based on the geometric shapes of different cells (e.g., rooms). Using Dijkstra’s (like) algorithm, the shortest path to a given target object can be effectively determined. Second, with the assistance of DPG, we also can address the stationary and continuous range queries in two separate solutions. For answering stationary range queries, we can obtain the qualifying objects by expanding the edges starting from the source point within a walking-distance range. However, this approach is not practical for continuous range queries (i.e., the source point keeps moving). After each movement, the set of qualifying objects need to be recalculated. We observe that the distances between the exits and the potentially qualifying target objects are relatively static. Based on this observation, we propose an efficient solution to continuous range queries by only considering the distance changes between the source point and the adjacent exits. Third, for resolving the size-driven routing, the challenge is how to decide the maximum allowable size for each route. A cube-based model is designed to represent the 3-dimensional information of the indoor environment, where the entire space is modeled using the same-size cubes. These cubes can be merged to form larger rectangular blocks. Based on this model, a new distance-based graph (called LEGO Graph) is constructed in which the blocks are edges and their intersection areas are nodes. The maximum accessible size of each node and edge can be determined from their sizes.
CHAPTER 1
INTRODUCTION

Research on navigation systems for an outdoor environment brings mature GPS navigation systems into daily use. This significantly affects our normal life. With the assistance of GPS systems, we can easily locate a desired place, obtain a detailed route from location A to location B, and learn information about the interesting locations nearby. However, indoor navigation is also highly demanded in some indoor environments. Imagine when we are in an unfamiliar huge building whose architectural structures are complex (e.g., an airport, a museum, a hospital, or a shopping mall), it is often difficult for us to find our desired target objects efficiently, especially if the destination name is not clear. We might take a circuitous way, go to a wrong floor, and even give up after a few attempts. Especially, for example, when you have to visit a hospital in an emergency, and the time available for finding the right place may be critical. Thus, navigation systems for indoor spaces are desired in our daily life.

1.1 Overview

The study of how humans find their ways around large buildings has a long tradition. Recent research on human spatial cognition [30] points out that if people do not know the exact route they desired, they’d prefer to stick as much as possible to the familiar parts of the building and head towards the goal as directly as possible. However, these strategies reveal a problem in that the routes people take may not be the most optimal ones. It is possible that a route containing some shortcuts is much shorter than the one that people can find by themselves. Thus, an indoor navigation system is required to provide users an ability to find better ways to navigate inside buildings.

In [9], navigation is defined as “the process of reading, and controlling the movement of a craft or vehicle from one place to another”. It involves two main aspects: localization and route planning. In outdoor spaces, the localization depends on the GPS signal; while with indoor spaces, sensors or mobile signals are often used to determine
the user’s current location and track their movements. Basic route planning refers to
the determination of appropriate routes from one place to another. The appropriate
routes can be the shortest routes, the routes for disabled persons in wheelchairs or
the routes for the blind persons. The route planning can also be extended to answer
some queries in relation to navigation. For example, range queries, which are used to
determine the interesting objects within a given distance, can be considered as a type
of extended route planning. This dissertation is to design a spatial model for supporting
route planning.

Route planning is highly dependent on the structure of the environment. In outdoor
spaces, most users who uses navigation systems are in vehicles, and in most cases
their movements are restricted to the constructed roads. Thus, the road network is the
most important structure in an outdoor space. The traditional route planning strategies
that are used for outdoor spaces are based on the road networks. With indoor spaces,
the entire environment is composed of various cells. Thus, the cells are the most
important structure of indoor spaces. Although there are some comparable concepts for
indoor and outdoor spaces, such as corridors and roads, with indoor spaces, there are
also concepts like rooms and lobbies for which we do not find counterparts in outdoor
space. Thus, the strategies designed for indoor route planning are very different from
those used in outdoor route planning. In addition, some other architectural constraints,
such as walls and doors, also play very important roles with indoor route planning, and
thus make indoor route planning more difficult.

In recent years, research on indoor route planning has attracted a lot of attention
in the academic community. As a result, more and more models for indoor navigation
have been proposed recently. Some of them use symbols to represent cells. The
generated routes are represented by a sequence of objects. Some of them try to find the
representative points for cells by analyzing their geometric shapes, and some of them
focus on building 3D models for the indoor space to capture the topological relationships
between different cells. However, compared to the matured GPS navigation systems for outdoor spaces, models for indoor spaces are far from practical in the aspect of route planning strategies. In outdoor spaces, these queries are often computed on the basis of a road network, which does not exist in indoor spaces. Constructing a network that can support route planning for indoor spaces is more challenging.

The process of network construction involves several difficulties. Firstly, there are a lot of architectural constraints such as walls and doors with indoor spaces. These constraints are important for modeling the indoor navigation systems. For example, some areas inside buildings are enclosed by walls, while some are adjacent to others without any barriers. Secondly, one cell cannot simply be represented by one edge. If one cell has more than two exits, users may take different exits to go through the cell. Even if there are only two exits in the cell, the number of implicit paths users may take is still infinite. Thirdly, the shortest route planning and range query calculation depend on the shortest walking distances between objects. Thus, the network for indoor spaces should be able to provide the shortest routes between any two objects. In fact, even if such a network with the minimum distances between any two objects exists, it is still not easy to design approaches to support continuous range queries, in which the location of the source point is dynamic. In outdoor spaces, the movements of vehicles are restricted by the directions and the speed limits of the roads. Unlike outdoor spaces, there is no particular rule to restrict the user’s movements in indoor spaces. That means people can roam anywhere and walk in any direction. Predicting a user’s movement is very difficult. Fourthly, the paths through indoor spaces might not be passed by all the users. Sometimes the paths that pedestrians can take are not suitable for persons in wheelchairs, as their size might prevent them from accessing narrow exits or turning around tight corners in corridors.

This dissertation contributes to the research of the indoor navigation systems with a specific focus on approaches to supporting indoor route planning. The intended
audience is broad and varied. Those interested in exploring the indoor environment will find a tool to model indoor space. Those interested in route finding for indoor spaces might get new insights into the approaches for converting cell-based structures into the networks for an indoor environment. Those interested in answering common navigational queries, such as range queries, could learn a feasible method to compute these queries.

1.2 Motivation

In our context, the primary goal of route planning is to provide a desired route in a specific environment from a certain source point. Our research mainly focuses on the following three open problems, which have not been solved in the related literature.

First, distance-driven routing aims at finding the shortest path to a given target object, which is an essential problem in an indoor navigation system.

Second, range-driven routing is to generate a set of routes to the qualifying objects that are in a specific walking-distance range from the source point. The source location can be static or dynamic, and thus, the determination of qualifying target objects can be classified to two types, stationary and continuous range queries.

Third, when the object moving through the route (e.g., the wheelchairs or moving carts) cannot be approximated by a point, size-driven routing is necessary to offer a feasible path for the object with a particular size (i.e., the length, width and height) to reach a given target object.

1.3 Contributions and Approaches

The overall goal of the presented dissertation is to analyze the indoor environment and design a model to support route planning through indoor spaces. The design of the model has three sub-goals:

1. Supporting shortest route planning.
2. Supporting range queries.
3. Providing feasible routes for different types of users.
In order to achieve the first sub-goal, we need to generate a distance-based graph from a cell-based indoor map, which is completely different from a network-based outdoor map. In our solution, a distance-based Direct Path Graph (DPG) is introduced, where nodes denote the exits and edges represent the shortest path segments between the exits. The segment computation is based on the geometric shapes of different cells (e.g., rooms). Using Dijkstra’s (like) algorithm, the shortest path to a given target object can be effectively determined. There are several facts that affect the network construction. First, users have to enter rooms through exits. As a result, doors and open boundaries of rooms must be taken into account during the construction. Second, a route provided to users cannot go across a barrier (e.g. a wall). Thus, the boundary and the shape of cells must be considered too. In this dissertation, we will explore the related facts and demonstrate how to construct path segments based on the geometric shapes of cells.

With the assistance of DPG, we also can address the second sub-goal in two separate solutions. For answering stationary range queries, we can obtain the qualifying objects by expanding the edges starting from the source point within a walking-distance range. However, this approach is not practical for continuous range queries (i.e., the source point keeps moving). After each movement, the set of qualifying objects need to be recalculated. We observe that the distances between the exits and the potentially qualifying target objects are relatively static. Based on this observation, we propose an efficient solution to continuous range queries by only considering the distance changes between the source point and the adjacent exits.

Third, for solving the third sub-goal, we have to determine the maximum allowable size for each route. A cube-based model is designed to represent the 3-dimensional information of the indoor environment, where the entire space is modeled using the same-size cubes. These cubes can be merged to form larger rectangular blocks. Based on this model, a new distance-based graph (called LEGO Graph) is constructed in which
the blocks are edges and their intersection areas are nodes. The maximum accessible size of each node and edge can be determined from their sizes.

1.4 Outline

The rest of the dissertation is organized as follows: Chapter 2 discussed the available models for indoor navigation systems and the existing approaches to supporting range queries in outdoor space. Chapter 3 introduced the iNav model that can support the shortest route planning for indoor space. Chapter 4 presents a range-driven routing approach to efficiently determine all the qualifying objects within a specific walking distance. a LEGO model is introduced in Chapter 5. Based on this model, the generation of a feasible route can be fully automated from the source point to the target object, when the size of a moving object cannot be approximated to a point. Chapter 6 evaluates our model and show some demos of our implemented system. Finally, we conclude with a summary of future work in Chapter 7.
2.1 Indoor Navigation Models

The earlier attempts of modeling indoor navigation systems focus on designing models for robots. The most famous ones are Polly, which is a mobile robot acting as a guide for MIT AT lab [32], and Minerva, which is an autonomous guide used in the National Museum of American History in Washington [95]. In fact, the ideas of designing navigation systems for robots and for human are different. Navigation systems for robots are based on local views, while a global view of the entire indoor spaces is more important for navigation systems used by humans. In recent years, research on human-oriented indoor navigation systems becomes a popular research area.

Considering the classification of location-based models in [6], the spatial models for indoor environment can be mainly classified into three categories: symbolic, semantic and geometric models.

2.1.1 Symbolic Models

Symbolic models define objects in terms of unique symbols. Most models focus on the representation of an indoor environment and the exploration of the topological relations between its objects. In fact, topological relations, such as meet, inside, and contains, are very useful in the process of route planning. By finding adjacent objects on the basis of the topological relations from the starting object to the target object, we are able to provide users with a rough route containing a sequence of objects. However, without geometric information like the distance between two objects, topological relations cannot be determined automatically. Thus, symbolic models have to manually explore and maintain the topological relations. Brumitt and Shafer [10] define symbols by creating a semantic space, in which each object has its unique identifier and its related information, such as the topological relations with its neighbors. Since the topological relations of one object are stored separately and there is no structure to
manage all the relations, the determination of the relations among multiple objects is not so straightforward. A better way to maintain and use these topological relations is to organize symbols by using hierarchical structures, such as trees ([79]) and lattices, which can reflect the topological relations among all the objects. Target objects can be found by referring to their symbols through the hierarchical structure, and the route to it can be obtained by computing the adjacent objects from the starting object.

The authors in [31, 70, 71] aim to develop a set of formal definitions for the symbols representing objects such as walls, doors, and angles in an indoor environment. Although their ideas have originally been used to instruct robots' action, these formalized symbols also can be used for human wayfinding. By using these symbols, possible paths and changes between paths can be recognized to achieve good navigation descriptions.

The aforementioned models are good for representing indoor space. However, it needs a lot of work to derive a sequence of paths to perform navigation by using these models. Raubal and Worboys [74, 75] process navigation queries in an indoor environment by performing spatial reasoning on a knowledge base. In the knowledge base, knowledge observed from the environment is described in terms of propositions that are either true or false. The reasoning is conducted by combining these propositions with some logic operators to derive new knowledge.

The primary advantage of symbolic models is that the semantic meaning of each object is understandable for users and the structure of the relationships among objects can be well designed. However, lacking geometric information prevents them to be applied to the applications that require precise locations. In addition, it requires a large amount of manual work to construct and maintain the structure of the symbols.

2.1.2 Semantic Models

In [44, 45], Kuipers and Levitt have observed the complexity of handling geographic data in real-world navigation. Some applications such as location-based pervasive
applications and intelligence applications require rich semantic information such as names, purposes, neighbors, and related features for representing indoor environments. Objects in the world all have geographic properties with respect to shapes, constraints, locations and neighborhood. However, symbols used in symbolic models may have little or no semantic meaning. Thus, researchers are interested in developing semantic models for indoor environments.

In [72], Pradhan proposed a relative simple way to represent objects semantically. In their CoolTown project, different places are labeled with URLs. Then semantic information of different places is embedded into their URLs.

A very common way to design semantic models is to use Ontology. In [21], Egenhofer defines the ontology of a vehicle navigation system as the spatial objects of roads, intersections, landmarks, and places that users want to go. The models proposed in [2, 11, 16, 19, 58, 61, 73, 98] all apply ontology to manage the semantic information of the navigation space. Corona and Winter develop an ontology in [16] by collecting concepts from different authors of wayfinding literature. In [2], Anagnostopoulos, etc. propose an ontology-based model, called OntoNav, by combining geometric and semantic information together. The reason why we classify it into the semantic category is because one of its important contributions is that they develop an *Indoor Navigation Ontology (INO)*, which can support both the representation and route planning of a navigation system. Based on the INO, this model is able to obtain a suitable path according to users’ requirements and profiles. The approach proposed in [73], simulate people’s wayfinding behavior by using an agent grounded in the ontology. The wayfinding strategy used in the agent is based on a graph representing the environment. In the graph, nodes represent decision points and edges denote lines of movements. In [58], the author detailed analyzes different architectural objects in indoor space, and organizes the relationships between these objects by using ontology.
Other than ontology, Stoffel, ect. proposed a novel strategy to build a hierarchical graph in [89] to organize the information of different objects in indoor space. Hu, Lee, etc. propose a model containing locations and exits, which are decorated by semantic information ([33]). They maintain the semantic information by organize the locations and exits in a hierarchical structure according to their reachabilities. An advantage of this model is its ability of generating semantically decorated descriptions for paths. Later, Li and Lee extend Hu’s model to further explore the topological relations between different objects in [49, 50]. In [50], Li observes that in a reachability graph, in which nodes are exits and edges represent their reachability information, some polygons bounded by multiple edges represent real areas in indoor space, some polygons do not. According to this observation, she develops an algorithm to detect the real areas in the reachability graph. In [49], Li deduces a lattice from Hu’s model based on the theory of formal concept analysis. The generated lattice can reveal the basic relationships between two entities, such as containment and overlap. Then, the optimal distance represented by the number of exits between two entities can be calculated according to the nearest neighbor relationship on the lattice.

Compared to the symbolic models, semantic models can handle more information for each object in an indoor environment. The semantic information is usually managed by formal structures and maintained by standard rules. Therefore, it is easier to understand and process the information by computers. However, semantic information alone is still not enough for a practical navigation system. In addition, the ability to calculate the shortest path is an important criterion. Without geometric information, models can neither record the areas of different objects nor calculate shortest paths.

2.1.3 Geometric Models

Geometric models use traditional spatial data types for points, lines, and regions to represent geometric objects in the Euclidean space. Information enables one to represent objects by accurate positions so that location information can be retrieved in a
more flexible way. However, it is not enough for a practical navigation system to only use
geometric information because users are used to representing objects by their names or
their semantic information. Thus, most geometric models are hybrid models combining
symbolic or semantic information together with geometric information.

Jiang and Steenkiste [38] represent and manipulate different objects by designing
the so-called Aura location identifier as well as some corresponding operators. An Aura
identifier is composed of both symbolic and geometric information parts describing
an object. By passing an objects Aura location identifier as a parameter to different
functions, the model is able to obtain the corresponding geometric distances and
topological relations. The advantage of this model is that it nicely represents each
object by using the Aura location identifier. However, since the distance it returns is the
Euclidean distance between two objects without considering the obstacles between
them, it is not suitable for navigation.

A time-dependent optimal routing model is proposed in [66] for emergency
evacuation. The path network is built on the basis of the location of sensors, and the
optimal routes are determined after considering environmental information on the
positions of evacuees. Although this model can provide a time-dependent optimal
route for a quick evacuation, the route it provides is highly dependent on the location
of sensors and not on the architectural structure itself. Thus, a poorly settled sensor
network may lead to improper results.

Using grid representation is a common way used in the Robert research area.
There are two general types of methods for the grid-based models. One is try to
decompose the available space into different shapes of cells, such as triangles and
polygons. The union of the generated cells is exactly the available space [46, 100].
the model presented in [46] decompose the space by using different shaped cells.
It first subdivide the space into several triangles by detecting the bottlenecks inside
rooms. Then these triangles are merged into several convex cells in different sizes.
However, this approach is limited to 2D flat planes. These kind of models can precisely represent the space. However, they share a common problem of inefficiency. The other one [3, 4, 82, 83, 96] is to represent the available space by unified shapes (e.g., rectangles). The union of the generated cells may not be exactly the available space, especially for the space on the boundaries. However, since they use simple and unified representative units, they are usually more efficient for the route planning. The model proposed in [4] is one of the most popular grid-based models. In this model, the available space is decomposed into cells marked as obstacle or non-obstacle. Based on this representation, routes can be computed by checking the availability of cells’ movements to their 8 neighbors. This model also support navigation in 3D spaces by filling out the indoor spaces with the obstacle and non-obstacle cubes (as shown in Figure 2-1). The obstacle cubes are further classified into insurmountable and surmountable ones to facilitate the 3D navigation. In [3], a hierarchical model is proposed by merging cells to form topological maps. This model can more efficiently compute the routes when the number of the representative cells is large.

Some models represent cells by using the representative points. Most of them use the center point of a cell as its representative point [23, 54, 55]. Some of them try to determine the representative according to the shape of the cell [52, 53]. Some particular cells (e.g. corridors), are represented by multiple points. models in [34, 76], a connectivity graph, in which nodes denote cells and edges represent exits, is built for the navigation purpose. Some of the models only consider the locations and the connections between exits [33, 42, 49, 50]. The models proposed in [12, 47, 48] considers both the shapes and the exits of the cells. These models all try to find the medial axis of a certain cell by constructing Voronoi diagram. The constructed paths are always on the line whose distances to its two sides are always equal.
2.1.4 3D-based Models

The above mentioned models assume that the floor of the cells are 2D planes. Actually, this assumption is insufficient to represent the entire indoor space. Places with the same $x$ and $y$ coordinate value may locate in different floors. Even in the same floor, cells may have different heights. In addition, it is common to have multiple layers connected by small stairs and slopes in one floor. Thus, some models try to propose 3D models for indoor representations and navigation.

In [47, 48], Lee propose a 3D model to represent the topological relations in indoor space. In this model, Poincaré duality combined with a hierarchical network structure are used to explore the relations between objects. Although this is a 3D model, the 3D features are only used to distinguish different floors. The floors are considered to be flat and the construction of the paths in one floor still focus on 2D data. The same problem happens in the models in [51, 93]. In [56], the authors present a semantic model of interior spaces to facilitate the calculation of the evacuation routes. This model takes into account different features of the interiors, such as the types of the passing (e.g. uni- or bi-directional) and the types of the boundaries (e.g. persistent boundaries like walls and virtual boundaries like openings). By using these features, this model is able to distinguish the accessible parts and non-accessible parts in the indoor space. However, this mode focuses on the surroundings, but ignores the structure of the floor plane. The model proposed in [85, 86] is the only model we have found talking about the structure of the floor plane. As shown in Figure 2-1C, this model defines different types of height to store the layered structure of the floor. Thus, the small stairs and slopes can be captured during the route planning. However, when the author extend this model to support navigation, the 3D structure of the floor is only used to represent discrete and connected spaces, not to construct different kinds of paths.
2.2 Space Syntax Graph

In the late 1970s and early 1980s, Bill Hillier, Julienne Hanson [28] developed a set of theories, called space syntax, which is used to analyze the structure of indoor spaces or small part of urban space. Later, Dalton in [17] and Hillier in [26, 27] further extended this theory. The general idea of the space syntax is to decompose the continuous space into a connected set of discrete subspaces. Each subspace can be measured in terms of its nearby space and accessibility.

According to the general idea, three basic concepts, isovists, axial space and convex space, are developed to describe the connectivity and integration of different components. Convex space is defined as a least number of convex subspaces that can cover the entire space [28]. Proposed by Benedikt [8], isovists analysis is used to explore the volume of visible space from a given point. An isovist is the area that directly visible from a given point. This idea provides a useful way to explore spatial environment and attracted a lot of researchers working on it. Traditional isovist generate the visible space in $360^\circ$ from the given point. In [59], Meilinger et al. proposed a concept of half isovist, which spins a smaller degree, rather than $360^\circ$ to simulate the view of humans. Axial space (also called axial map) is the dominant approach of the space syntax. Its idea is based on the axial lines, which are defined as the longest visibility lines between two spaces. Axial space is composed of the least number of axial lines that are mutually
intersect and cover the whole space. Figure 2-3B is the corresponding axial space of Figure 2-3A.

Although space syntax provide a theoretical way to analyze the spatial configurations, it is not easier to apply this theory in the real world. First, as discussed in [67], the decomposition of sub convex spaces are not fixed. For a given space, there might be multiple ways to decompose the space with the same number of subspaces (as shown in Figure 2-2). Therefore, how to construct a convex space as defined in [28] is non-determined. Second, from the cognitive perspective, one goal of space representation approaches is to retain as much information as possible. However, using axial lines to represent the space will definitely lose some important space configuration information.

Despite the problems that we mentioned above, the theoretical bases of space syntax still holds promise as an important analytical tool in many fields, especially in the area of architectural designs and spatial learning. In recent decades, it shows its usage in many ways, including route choosing, spatial knowledge acquisition, orientation and disorientation [60]. Jiang et al. discussed the possibility of using space syntax within GIS in methodological perspective [35–37]. [68] has demonstrated the benefit of applying axial map to spatial learning and wayfinding strategies. In [69], convex map and axial map are constructed to analyze and forecast the relationships between the spatial patterns of the land and the flows of the pedestrians in Fukuoka City, Japan. Benedikt first introduced approaches to describe spatial spaces by using isovist analysis in [8]. Since the isovist of each point depends on the physical structure, it is very useful to apply the isovist analysis in indoor space. Some of them may share the same Isovist, and some of them have their own isovist. Most if the research on isovists [5, 22, 101] are in the field of architecture to design floor plan and arrange architectural components. In [99], the authors proposed an approach to convert a set of isovists into a graph of
2.3 Approaches to Supporting Range Queries in Outdoor Space

A typical range query is a common database query that retrieves all the records between an upper and a lower boundary. In GIS area, a range queries ia common, navigation-related query, which is used to obtain all qualifying objects within a given range according to the user’s interests. A typical range query in GIS area can be “Provide me all the shoe stores within 50 meters from my current location”. Current study on how to support range queries mainly focus on the outdoor space.

2.3.1 Stationary Range Queries

The range query that asks for interesting objects within a given distance with respect to a static query point is called a stationary range query. The earlier approaches to support stationary range queries [29, 41, 62] use R-Trees [25], R+-Trees [91] and R*-Trees [7] for query processing. These approaches retrieve the qualifying objects based on the Euclidean distances. Therefore these approaches are called the
Euclidean-based approaches. The Euclidean-based approaches are useful when the objects are in a clearing space. However, they are not suitable for the cases that the objects are separated by obstacles. Therefore, the network-based approaches, which take into account the actual reachable distances of the objects, appear to answer the range query based on the real reachable distances. The first network-based approach is proposed in [65] by Papadias and Zhang. This approach, which is called range network expansion (RNE), expands the network from the query point to check all the nodes along the expansion. Because of the nice features of the Voronoi diagram, using Voronoi diagram is another choice of the network-based approaches. The most typical ones are the VN$^3$ model proposed by Kolahdouzan and Shahabi in [39], the PINE model presented by Safa et al. in [78].

2.3.2 Continuous Range Queries

Approaches for stationary range queries assume that the query point is always in a static position. In fact, users are usually in a moving status when they issue a navigation query. The range query that keeps updating results according to the user’s movements is called a continuous range query. The importance of continuous queries is mentioned by Sistla et al. in [84]. In recent years, more and more approaches supporting continuous range queries have appeared. Similar to the stationary range queries, there are two categories of approaches to supporting continuous range queries based on which distance they use. An earlier Euclidean-based approach introduced in [87] performs stationary range queries on several selected sample points respectively. Then the approach called time-parameterized queries proposed in [92] try to answer the range query by incrementally calculating the next result and the objects that might affect it based on the current result. Later on, approaches to supporting continuous range queries try to find the split points which indicate the changes of the qualified objects. The idea of the split points, which is introduced by Tao et al. in [103], is future used by other approaches [40, 77, 90, 102, 106]. To the best of our knowledge, the approach
proposed in [102] is the only one for supporting continuous range queries based on network distances.
CHAPTER 3
SUPPORTING THE SHORTEST PATH ROUTE PLANNING IN INDOOR SPACE

3.1 Problems in Existing Indoor Navigation Models

There are a couple of models that try to convert architectural structure into path networks. Gilliron and Merminod [23] describe a strategy to convert CAD data into a topological model (called Node-Link), which supports the objects like corridors, rooms, ways and paths (Figure 3-1A). Each link is assigned a value of cost corresponding to the time to travel from its start node to its end node. However, this model suffers from two main problems. First, the cost between two nodes must be provided in advance. Second, it lacks the consideration of constraints, such as doors, windows and walls, in an indoor space. Thus, this model cannot lead users to the exact entry of the target cell. More importantly, this model may generate circuitous paths. As shown in Figure 1b, the walls represented by the bolded lines prevent a direct reachability between the rooms. The route generated by the Node-Link model is a circuitous one composed of the center points of cells. To simplify the final paths generated by Node-Link, Lyardet et al. [54, 55] propose a model called CoINS, which eliminates some unnecessary nodes so that the segments can be optimized. For example, Figure 3-1C shows the improved path for the situation shown in Figure 3-1B. Although the CoINS model can optimize the final path, it does not consider accessibility constraints such as doors, windows, and walls.

Werner, Krieg-brückner et al. [43] propose a route graph model which builds abstract routes according to exits, walls, and some other constraints in indoor environments. However, they do not discuss how to build the route graph for an entire indoor space. In addition, they assume route segments have directions. However, in an indoor space, there is no specified direction for places since people can walk in any direction. Models in [12, 47, 48] try to find the medial axis of a certain cell by constructing Voronoi diagram. The constructed paths are always on the line whose distances to its two sides are always equal. As shown in Figure 3-2, Figure 3-2A is the Voronoi diagram
of the simple polygon and Figure 3-2B is the computed medial axis transformed from Figure 3-2A. The generated paths in these models are always on the medial axis. This strategy guarantees enough space for the user. However, it will cause problem when the cell is very spacious.

Lorenz, Ohlbach, and Stoffel [53] develop an improved model which takes some architectural constraints into account when building the route graph. As shown in Figure 3-3A, the model employs some representative points to represent rooms, corridors, and some other objects. Then the calculation of the path is performed among these representative nodes as well as some architectural constraints like doors in Figure 3-3A. The authors extend their model in [88] by considering the shapes of cells. They point out that if the interior of the cell in the final path is a concave region,
several route instructions may be required to show the way to go through the cell, because users may not be able to see the next exit at some locations in the cell. Thus, they decompose the concave regions by detecting the concave vertices as shown in Figure 3-3B. The generation of final descriptions is introduced in [63]. One advantage of this model is that it considers constraints and accessibility in an indoor space so that the final path it generates is a feasible way in real situations. However, it still suffers from some problems. First, the determination of the representative points is not easy. Questions are which location is the center point for an object, or how to determine the number of points and their positions for corridors. Second, the final paths the model generates are not optimal. For example, the solid line in Figure 3-3A is the path from door2 to room106 that the model generates. As we can see, this route is not an optimal one. If room105 is very large, the result will be unacceptable to users. Users will prefer the path specified by the dashed line in Figure 3-3A. A similar problem exists regarding the partitioning of concave regions. Since users cannot see d4 from d3 in Figure 3-3B, the region is composed, and point a is taken as an intermediate location in the path from d3 to d4. In fact, the optimal way from d3 to d4 is from d3 to c, and then from c to d4.
3.2 The Path Construction

Cells such as rooms, corridors, and lobbies are considered as the basic units in an indoor space. However, if we want to provide detailed routing information, it is insufficient to only consider the order of the cells. Other needed information like the location of doors and exits is also very important. A further important aspect are implicit paths in an indoor space as they are commonly used by people. For example, in Figure 3-3A, if the user is in the corridor and intends to go to room105, she will go straight towards door4. The straight line to door4 is an implicit path; it is also the shortest path to room105. In order to obtain the shortest paths in an indoor space, we will explore in this section how to construct the implicit paths for the different cell categories. We will show that and why the geometric shapes and architectural constraints of different kinds of cells affect the construction of the implicit path segments, and then propose an approach to construct the implicit shortest paths for the different cell categories.

There are a number of different kinds of architectural cells in an indoor space. Some of them have similar shapes but serve different purposes, and some of them are totally different with respect to their shapes but play the same role during the routing. For example, rooms with multiple doors can be a part of a passage to a certain destination, while rooms with only one door cannot. According to the geometric and architectural features of the cells, we classify them into four categories: simple cells (Section 3.2.1), complex cells (Section 3.2.2), open cells (Section 3.2.3), and connectors (Section 3.2.4). Each cell has one or more access points that are specific locations where a cell can be entered or left. In this section, we will introduce these four cell categories and show how one can find the implicit path segments in each of them.

3.2.1 Simple Cells

A simple cell is a cell that is enclosed by walls and has only one door. Since it cannot function as a passage, it does not have any inner implicit path segments. It can
only play the role of a start object or a target object, and its entry and exit are at the
location of the door. Figure 3-4A shows a schematic instance of a simple cell. The black
dot represents the only door of the cell. An application example is an office room at the
end of a floor that can be entered and left by a single door. Definition 1 provides a formal
and more abstract definition of simple cells.

**Definition 1.** A simple cell sc is given as a triple \( sc = (id, r, a) \). It represents a cell with a
geometric structure that is enclosed by a boundary (usually walls) and can be accessed
by exactly one access point (usually a door). The component \( id \) is the unique identifier
of this cell. The component \( r \) is a value of the spatial data type region \([81]\) that stores
the areal geometric structure and extent of this cell as a simple, single-component
polygon without holes. The component \( a \) is a value of the spatial data type point \([81]\),
denotes (the location, that is, the xy-coordinates, of) the access point, and lies on the
boundary of \( r \). It is the architectural constraint controlling the accessibility of the cell.

### 3.2.2 Complex Cells

A complex cell is a cell that is enclosed by walls and can be accessed by multiple
doors and/or has a nested structure. It can be considered as either a start object, a
target object, or an intermediate object that contains parts of paths as passages to the
destinations. We distinguish flat complex cells (Section 3.2.2.1) and nested complex
cells (Section 3.2.2.2). Simple cells are special cases of flat complex cells that are
special cases of nested complex cells. Each implicit path segment in a complex cell is
required to be a straight line (that is, a segment) connecting two access points without
intersecting any boundary. Thus, the construction of these implicit path segments highly
depends on the geometric shapes and the locations of the access points of a cell. For
flat complex cells we consider the implicit path segment construction in Section 3.2.2.3.
For nested complex cells we perform this consideration in Section 3.2.2.4. Finally, we
present an algorithm for computing the implicit path segments in flat and nested complex
cells in Section 3.2.2.5.
3.2.2.1 Flat complex cells

The geometry of a flat complex cell is given by a simple polygon, that is, it consists of a single areal component and does not have holes. Further, it has at least two access points on its boundary. Figure 3-4B shows a schematic instance of a flat complex cell. The black dots represent the access points of the cell. Flat complex cells can be often found. An application example is a waiting room in a doctor’s office that can be entered and left through a number of doors. Definition 2 provides a formal definition of flat complex cells.

**Definition 2.** A flat complex cell fcc is given by a tuple \( fcc = (id, r, m, a_1, \ldots, a_m, t, s_1, \ldots, s_t) \) with \( m \geq 2 \) and \( t \geq 1 \). It represents a cell with a geometric structure that is enclosed by a boundary (usually walls) and can be accessed by two or more access points (usually doors). The component \( id \) uniquely identifies this cell. The component \( r \) is a value of the spatial data type region that stores the areal geometric structure and extent of this cell as a simple, single-component polygon without holes. The component \( m \) denotes the number of access points in this cell. The components \( a_1, \ldots, a_m \) are values of the spatial data type point, represent (the locations, that is, the xy-coordinates, of) these access points, and lie on the boundary of \( r \). The component \( t \) keeps the number of implicit path segments in this cell, and \( s_1, \ldots, s_t \) represent these segments.

3.2.2.2 Nested complex cells

Most rooms in an indoor space can be represented by simple cells or flat complex cells, that is, geometrically by simple regions. However, it is also common to find structures like rooms containing other rooms, or corridors with a circular shape.
An application example is an open office environment that in a large office contains separate offices for managers or supervisors. Another example is a laboratory that is subdivided into a collection of adjacent and connected rooms. These rooms themselves can also contain rooms. Further, it can be of interest to model obstacles like desks, chairs, and cabinets in a room since routes cannot pass them. In this section, we will hence discuss more complicated scenarios in which the geometric shape of a cell is modeled by a hierarchy of simple regions. A simple scenario is given by two simple regions where one simple region is located in another simple region (Figure 3-5A). The simplest scenario is given if the outer simple region would have exactly one access point. If a room contains multiple rooms, the structure can be represented by a simple region that contains several disjoint or meeting simple regions (Figure 3-5B). Other structures like rooms that are recursively inside other rooms (Figures 3-5C and 3-5D) lead to a hierarchy of simple regions that is organized with respect to the geometric containment relationship, that is, cells can contain subcells. We call such a hierarchy a *nested complex cell*.

Definition 3 provides its formal definition. It requires a number of concepts that we introduce first. The definition makes use of the topological predicates [20, 80] disjoint, meet, contains, and covers that characterize the relative position between two simple regions and that have a precise and mutually exclusive semantics. Intuitively, the predicate disjoint means that two simple regions do not share any part with each other. The predicate meet implies that two simple regions share a finite collection of boundary points or boundary lines. Boundary lines as one-dimensional objects are required for the
dimensional topological predicate 1-meet. The predicates contains and covers express that a simple region is located within another simple region, without and by touching the boundary respectively. If both regions touch in a line, the dimensional topological predicate 1-covers holds. Further, the operation commonBorder \[81\] computes the shared boundary of two region objects as an object of the spatial data type line. The function getBorderPoints gets a line object and a list of points as operands and returns the set of those points that are located on the line object.

**Definition 3.** A nested complex cell ncc is given by the tuple

\[
ncc = (id, d, k_1 = 1, k_2, \ldots, k_d, c_{1,1}, c_{2,1}, \ldots, c_{d,k_d}, t, s_1, \ldots, s_t)
\]

with \(d \geq 2, k_i \geq 1\) for all \(1 \leq i \leq d\), and \(t \geq 1\). It represents a hierarchical cell structure which consists of a collection of cells \(c_{i,j}\) for all \(1 \leq i \leq d\) and \(1 \leq j \leq k_j\) and in which each cell is enclosed by a boundary (usually walls), can be accessed by one or more access points (usually doors), can have brother cells, and can contain other cells. The component id uniquely identifies ncc. The component d denotes the depth of the cell hierarchy or its number of levels. The depth of the root level is 1. The components \(k_i\) describe the number of cells at the hierarchy level \(i\). The cell \(c_{i,j}\) is the \(j\)th cell at the hierarchy level \(i\) and is represented as a tuple \((id_{i,j}, r_{i,j}, m_{i,j}, (a_{i,j,1}, \ldots, a_{i,j,m_{i,j}}))\). The component id_{i,j} uniquely identifies \(c_{i,j}\). Each component \(r_{i,j}\) is a value of the spatial data type region and stores the areal geometric structure and extent of this cell as a simple, single-component polygon without holes. The component \(m_{i,j}\) denotes the number of access points in this cell, and \(a_{i,j,1}, \ldots, a_{i,j,m_{i,j}}\) represent (the locations, that is, the xy-coordinates, of) these access points located on the boundary of \(r_{i,j}\). The hierarchy level 1 has only one cell which is the root of the hierarchy. All region objects must satisfy the following four conditions:
\[(i) \quad \forall 1 \leq i \leq d \quad \forall 1 \leq j < l \leq k_l : r_{i,j} \text{ disjoint } r_{i,l} \lor r_{i,j} \text{ meet } r_{i,l}
\]
\[(ii) \quad \forall 1 \leq i \leq d \quad \forall 1 \leq j < l \leq k_l : r_{i,j} 1\text{-meet } r_{i,l} \Rightarrow \]
\[\text{getBorderPoints(commonBorder}(r_{i,j}, r_{i,l}), (a_{i,j,1}, \ldots, a_{i,j,m_{ij}})) =
\text{getBorderPoints(commonBorder}(r_{i,j}, r_{i,l}), (a_{i,l,1}, \ldots, a_{i,l,m_{il}}))
\]
\[(iii) \quad \forall 1 \leq i \leq d - 1 \quad \forall 1 \leq l \leq k_{i+1} \exists 1 \leq j \leq k_l \colon r_{i,j} \text{ contains } r_{i+1,l} \lor r_{i,j} \text{ covers } r_{i+1,l}
\]
\[(iv) \quad \forall 1 \leq i \leq d - 1 \quad \forall 1 \leq j \leq k_l \quad \forall 1 \leq l \leq k_{i+1} \colon r_{i,j} 1\text{-covers } r_{i+1,l} \Rightarrow \]
\[\text{getBorderPoints(commonBorder}(r_{i,j}, r_{i+1,l}), (a_{i,j,1}, \ldots, a_{i,j,m_{ij}})) =
\text{getBorderPoints(commonBorder}(r_{i,j}, r_{i+1,l}), (a_{i+1,l,1}, \ldots, a_{i+1,l,m_{i+1,l}}))
\]

The component \(t\) keeps the number of implicit path segments in this cell, and \(s_1, \ldots, s_t\) represent these path segments.

The four conditions regarding the region objects are consistency statements.

Condition (i) avoids an overlapping of the region objects at the same hierarchy level.

Condition (iii) ensures that each inner cell at a hierarchy level \(i + 1\) is enclosed by exactly one single outer cell at the hierarchy level \(i\). Conditions (ii) and (iv) require that the access points located on the common boundary lines of two simple regions are the same for both simple regions. This means, for example, that adjacent rooms with doors should share these doors. The computation of the \(n\) implicit path segments will be described in the following subsections.

Each cell \(c_{i,j}\) is, in principle, a nested complex cell with two hierarchy levels but without path segment information. The outer boundary is given by the simple region \(r_{i,j}\) at the hierarchy level \(i\) and forms the first hierarchy level. The second hierarchy level is formed from the simple regions of those cells at hierarchy level \(i + 1\) that are located inside of \(r_{i,j}\). We will regard such a spatial configuration as a complex region with zero (Section 3.2.2.3), one, or more (Section 3.2.2.4) holes and compute the implicit path segments for it. In case of a complex region without holes, we are dealing with a simple cell if it has exactly one access point, or with a flat complex cell if it has at least two access points. In the former case, no path segment computation has to be performed.
3.2.2.3 Implicit path segments in flat complex cells

The simplest case to construct implicit path segments in flat complex cells is a cell with only two access points that can be connected by a straight line inside the cell. The shortest path from one access point to the other is then the straight line between them (see Figure 3-6A). If a flat complex cell has more than two access points, and all of them can be directly reached from each other inside the cell, we obtain multiple implicit path segments in the cell. The shortest path segments in such a cell are all straight lines connecting any two access points. For example, in Figure 3-6B, we find six implicit shortest path segments in a cell with four access points. The deeper reason for the direct reachability between all access points of this region is that this region is convex, that is, the inner angle between any pair of consecutive boundary segments is less than $180^\circ$. However, in cells with concave shapes, which have at least one pair of consecutive boundary segments with an inner angle of more than $180^\circ$, two access points may not be directly reachable from each other. In Figure 3-8A, the dashed lines show some cases where a straight line connecting two access points is blocked by the boundary (for example, the segment $c - d$).

We now describe our approach to constructing implicit path segments for the case of arbitrary, non-selfintersecting, simple regions (polygons). From Computational Geometry [18] we know that if the interior of a segment connecting two boundary points of a polygon either intersects the boundary or is outside the boundary, this polygon must be a concave polygon. That is, there is at least one vertex whose interior angle

![Figure 3-6](image)

Figure 3-6. Path segments in different types of cells. A) Path segment in a flat complex cell with two access points. B) Path segments in a flat complex cell with multiple access points. C) Path segments in a connector.
Figure 3-7. examples of concave vertices and concave boundaries.

is a reflex angle (degree $> 180^\circ$) on one part of the boundary between the two access points. We call this kind of vertex a concave vertex and the part of the boundary that contains concave vertices a concave boundary. For example, in Figure 3-7A, $x$ is a concave vertex, and the boundary part between $a$ and $b$ containing $x$ is a concave boundary. It is possible to have several concave vertices on a concave boundary. As shown in Figure 3-7B, both $x$ and $y$ are concave vertices. Figure 3-7C shows that not all concave vertices between two points on a concave boundary necessarily contribute to the shortest path segments. An example of a segment that is located completely outside a polygon and is thus not a valid path segment for the polygon is given by $c$ and $d$ in Figure 3-7A.

Our approach to obtaining the shortest path in this kind of situation is to select an appropriate concave vertex on the concave boundary as an intermediate point.

Figure 3-8. The shortest paths in a concave polygon. A) Two access points in a concave region (like $b$ and $c$) cannot reach each other on a straight path segment. B) Potential path segments. C) Final path segments.
to partition the segment into two segments. The partitioning process continues until all generated segments do not intersect the boundary. For example, in Figure 3-8A, the straight segment connecting the access points a and e intersects the boundary. Figure 3-8B shows that this segment is then partitioned into the segments \((a, v_3), (v_3, v_{15}), (v_{15}, v_{14})\), and \((v_{14}, e)\) by the intermediate points \(v_3, v_{14},\) and \(v_{15}\) between them. We observe that \(a - v_3 - v_{15} - v_{14} - e\) is the shortest path between \(a\) and \(e\). Lemma 1 shows that the intermediate points of the shortest path between two access points located on the boundary of a simple region (polygon) are always concave vertices of this boundary.

**Lemma 1.** The shortest path between any two access points in a flat complex cell \(c\) is a polygonal path (that is, a connected series of straight line segments) whose intermediate points are concave vertices of \(c\).

**Proof.** We first show that any such shortest path \(s\) is polygonal. Suppose that \(s\) is not polygonal. Since \(s\) is entirely inside \(c\), and \(c\) is polygonal, there must be a point \(p\) on \(s\) that is in the interior of \(c\), and the part of \(s\) containing \(p\) is not a line segment. Since \(p\) is in the interior of \(c\), there must be a circle of positive radius that is centered at \(p\) and completely inside \(c\) (Figure 3-9A). Since the part of \(s\) containing \(p\) is not a straight line segment, we can always shorten this part by replacing it with a line segment connecting the point where the part enters the circle to the point where it leaves the circle. But this contradicts the shortest path property since any shortest path must be *locally shortest*, that is, any subpath connecting two points \(q\) and \(r\) must be the shortest path between \(q\) and \(r\).

Now we show that the intermediate points of any such shortest path \(s\) are concave vertices on the boundary of \(c\). Any intermediate point \(v\) of \(s\) cannot lie in the interior of \(c\) since we could construct a circle of positive radius that is centered at \(v\) and lies completely in \(c\) so that we could replace the subpath of \(s\) that is inside the circle and turns at \(v\) by a shorter straight line segment. Similarly, any intermediate point \(v\) of \(s\) cannot lie in the interior of a boundary edge. Otherwise, there must be a circle of
positive radius centered at $v$, and half of the circle which contains the line segment parts emanating from $v$ is inside $c$. But we can then replace these line segment parts inside the circle by using a single shorter line segment (Figure 3-9B). This is again a violation of the shortest path property. It is also impossible that an intermediate point $v$ is a convex vertex of $s$ because if we draw a circle of positive radius centered at $v$, some portion of the circle will be inside $c$, and we can find a shorter straight line segment to replace the line segment parts inside the circle (Figure 3-9C). The only possibility left is that the intermediate point $v$ of $s$ is a concave vertex of the boundary of $c$.

3.2.2.4 Implicit path segments in nested complex cells

The hierarchical structure of a nested complex cell seems to indicate that a method for the determination of its implicit path segments is rather complicated. But we will argue now that this determination can be performed \textit{locally} in the hierarchy by considering any two consecutive hierarchy levels (except for the leaf nodes of the
hierarchy) together at a time, determining the implicit path segments locally for such two levels, and collecting all path segments. Figure 3-10A shows a scenario of a nested complex cell with a hierarchy of depth 3. We have two consecutive pairs of hierarchy levels together with the space that they enclose. They are shown in Figure 3-10B for the levels 1 and 2, and in Figure 3-10C for the levels 2 and 3. At level 3 (Figure 3-10D), we cannot find inner cells. Otherwise, the depth of the hierarchy would be 4. The space enclosed is shown in gray color in all three cases. Geometrically, each area filled by gray color has a connected interior and forms a special instance of the spatial data type region for complex regions. This special instance describes a single-component region object with holes. Such an instance is usually called a face [81]. In this sense, Figure 3-10B has one gray face whereas Figures 3-10C and 3-10D have two faces each. Further, the right face of Figure 3-10C and the two faces of Figure 3-10D do not have holes. If a face should have any holes that share a common boundary part, these holes are geometrically merged to a single hole to be in accord with the face definition.

Since the interiors of all faces determined as described above are disjoint, it is sufficient to only compute the implicit path segments for each face locally and individually, and then to collect all path segments found. In a face with holes, segments between access points may intersect not only the outer boundary but also the inner boundaries. When a segment between two access points intersects an inner boundary, some of the vertices on the inner boundary will become intermediate points of the shortest path between the two access points. As Lemma 2 shows, only concave vertices of the outer boundary and the inner boundaries of a face can become intermediate points.

**Lemma 2.** The shortest path between any two access points in a nested complex cell \( c \) of depth 2, whose geometric shape is given by a face, that is, by an outer polygon (simple region) and zero, one, or more inner disjoint or meeting polygons, is a polygonal path whose intermediate points are concave vertices of \( c \).
Figure 3-11. Proof cases for the inner polygon boundaries of a complex cell with a hierarchy of depth 2: A) An intermediate point of the path cannot be a point in the interior of a boundary edge. B) it also cannot be a convex vertex on the boundary. C) It must be a concave vertex of the inner boundary.

Proof. Lemma 1 covers the case that the depth of the cell hierarchy is 1. In this case, there are no inner hole polygons. We have a simple cell if there is exactly one access point on its boundary. Otherwise, we have a flat complex cell $c$ with at least two access points. An intermediate point of the shortest path $s$ between any two access points of $c$ is then always a concave vertex of the boundary of $c$. The same argumentation that we have used in Lemma 1 can also be applied to the inner polygon (hole) boundaries of $c$ if the depth of the cell hierarchy is 2. An intermediate point of $s$ cannot lie in the interior of an edge of any inner boundary (Figure 3-11A) and also not be a convex vertex of any inner boundary due to the radius argument (Figure 3-11B). Hence, if an intermediate point of $s$ is located on an inner boundary, it must be a concave vertex of this inner boundary (Figure 3-11C). Note that the angles that decide whether an end point of an edge of an inner boundary is convex or concave must be taken from the interior side of $c$. 

Figure 3-12. Example of a face with two access points and two possible paths between them.
Since there are two possible directions (clockwise and counterclockwise) to bypass an inner hole boundary, there are potentially two routes that could lead to a shortest path. As shown in Figure 3-12, the dashed line $a - iv1 - iv2 - ov1 - ov2 - b$ and the dotted line $a - iv4 - iv3 - ov2 - b$ are two potential paths from the access point $a$ to the access point $b$. Both sets of points $\{iv1, iv2, ov1, ov2\}$ and $\{iv4, iv3, ov2\}$ could be intermediate points. All intermediate points are concave points.

3.2.2.5 An algorithm for computing all shortest path segments in complex cells

Algorithm 1 shows how to compute all shortest path segments in a nested complex cell by leveraging the insights from Sections 3.2.2.3 and 3.2.2.4. The strategy is to iteratively extract all cell hierarchies from a nested complex cell that have the depth 2 (or 1 at the leaf level) and thus the geometric structure of a face (with holes if the depth is 2 and without holes if the depth is 1). For each such face and its relevant access points, Algorithm 2 locally determines all potential path segments. These are segments that could be later part of the solution. The potential shortest path segments are collected for all faces. Together with all access points of the nested complex cell, they form the input of a well known algorithm to calculate the shortest paths between any two access points.

Algorithm 1 traverses all inner hierarchy levels of the nested complex cell from the root level down to the last but one, deepest level (line 2). All cells of each hierarchy level are traversed (line 3), and for each such cell $c$ we first collect its access points (line 4). The symbols ⟨ and ⟩ enclose a sequence or list of elements. Then we check for $c$ whether it contains or covers any cells at the next deeper level (lines 5 and 6). The Boolean predicates contains and covers are the well known topological predicates used in Definition 3. Those cells that are contained or covered by $c$ are geometrically merged into a complex region object of the spatial data type region (line 7); they form the hole cells of $c$. The operation $\oplus$ denotes the geometric union operation [80, 81]. Further, the access points of each hole cell are added to the access points of $c$ and the already
determined access points of previously considered hole cells (line 7). The operator \( \circ \) denotes the concatenation operator on lists. Next, we add the access points of \( c \) and its holes cells to a global variable \( \text{accessPoints} \) that is supposed to keep all access points of the nested complex cell (line 8). It can happen that two hole cells of \( c \), or \( c \) and one of its hole cells, have a common access point on a shared boundary part. In the first case, the two hole cells geometrically merge to a larger hole cell since their shared boundary part disappears, and their common access point becomes irrelevant at this point. In the second case, the hole cell forms a bay in \( c \), and their common access point becomes irrelevant too. In both cases, both occurrences of the same access point kept for \( c \) must be removed. This is performed by the function \( \text{rpairs} \) (remove pairs) (line 9). For each cell \( c \) and its hole cells, we form a face by applying the geometric difference operation \( \ominus \) \([80, 81]\) to the simple region representing \( c \) and the complex region object representing its hole cells. The resulting face and its access points form the operands of the operation \( \text{DeterminePotentialPathSegs} \) (line 10) that locally determines all potential path segments.

**Algorithm 1**: Computation of the shortest path segments in a nested complex cell

<table>
<thead>
<tr>
<th>Input:</th>
<th>nested complex cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ncc = (id, d, k_1, k_2, \ldots, k_d, c_{1,1}, c_{2,1}, \ldots, c_{2,k_2}, \ldots, c_{d,1}, \ldots, c_{d,k_d}, n, s_1, \ldots, s_n) ) whose construction is incomplete with respect to the shortest path segments ( s_1, \ldots, s_t ) and their number ( t )</td>
<td></td>
</tr>
<tr>
<td>Output:</td>
<td>nested complex cell ( ncc ) including the shortest path segment information</td>
</tr>
</tbody>
</table>

```plaintext
accessPoints ← \( \langle \rangle \); potShortestPathSegs ← \( \langle \rangle \); 
for \( i \leftarrow 1 \) to \( d - 1 \) do 
  for \( j \leftarrow 1 \) to \( k_i \) do 
    \( h_{i,j} \leftarrow \langle \rangle \); \( a_{i,j} \leftarrow \langle a_{i,j,1}, \ldots, a_{i,j,m_i} \rangle \); 
    for \( l \leftarrow 1 \) to \( k_{i+1} \) do 
      if contains \((r_{i,j}, r_{i+1,l})\) or covers \((r_{i,j}, r_{i+1,l})\) then 
        \( h_{i,j} \leftarrow h_{i,j} \ominus r_{i+1,l} \); \( a_{i,j} \leftarrow a_{i,j} \circ \langle a_{i+1,l,1}, \ldots, a_{i+1,l,m_{i+1}} \rangle \); 
    accessPoints ← accessPoints \( \circ \) \( a_{i,j} \); 
    \( a_{i,j} \leftarrow \text{rpairs}(a_{i,j}) \); 
    potShortestPathSegs ← potShortestPathSegs \( \circ \) \( \text{DeterminePotentialShortestPathSegs}(r_{i,j} \ominus h_{i,j}, a_{i,j}) \); 
  potShortestPathSegs ← potShortestPathSegs \( \circ \) \( \text{AllPairsShortestPath}(\text{rdup}(\text{accessPoints}), \text{potShortestPathSegs}) \); 
\( \langle s_1, \ldots, s_t \rangle \leftarrow \text{AllPairsShortestPath}(\text{rdup}(\text{accessPoints}), \text{potShortestPathSegs}) \); 
```
Algorithm 2: (DeterminePotentialShortestPathSegs) Computation of the potential shortest path segments in a nested complex cell represented by a face and its access points.

**Input:** face $F$ (with zero, one, or more holes), list array `accessPoints` of access points on $F$'s boundary

**Output:** list array `potentialShortestPathSegs` of all potential shortest path segments in $F$

```plaintext
intersectionFound ← false; m ← size(accessPoints); potentialShortestPathSegs ← ⟨⟩;

1 for $i$ ← 1 to $m$ − 1 do
2     for $j$ ← $i$ + 1 to $m$ do
3         $p$ ← accessPoints[$i$]; $q$ ← accessPoints[$j$];
4         if not intersects(segment($p$, $q$), boundary($F$)) and in(center(segment($p$, $q$)), $F$) then
5             potentialShortestPathSegs ← potentialShortestPathSegs ◦ ⟨($p$, $q$));
6         else
7             intersectionFound ← true;
8
9 if intersectionFound then
10     concaveVertices ← findConcaveVertices($F$); // list array of concave vertex points
11     $n$ ← size(concaveVertices); // number of concave vertices
12     for $i$ ← 1 to $n$ − 1 do
13         for $j$ ← $i$ + 1 to $n$ do
14             $p$ ← concaveVertices[$i$]; $q$ ← concaveVertices[$j$];
15             if not intersects(segment($p$, $q$), boundary($F$)) and in(center(segment($p$, $q$)), $F$) then
16                 potentialShortestPathSegs ← potentialShortestPathSegs ◦ ⟨($p$, $q$));
17
18 for $i$ ← 1 to $m$ do
19     for $j$ ← 1 to $n$ do
20         $p$ ← accessPoints[$i$]; $q$ ← concaveVertices[$j$];
21         if not intersects(segment($p$, $q$), boundary($F$)) and in(center(segment($p$, $q$)), $F$) then
22             potentialShortestPathSegs ← potentialShortestPathSegs ◦ ⟨($p$, $q$));

22 return potentialShortestPathSegs;
```

of this face and that is presented in Algorithm 2. After the execution of lines 1 to 10, we have ascertained all potential path segments of the nested complex cell. In a last step (line 11), we first remove all duplicates from the list of access points by the function `rdup` (remove duplicates) since in the network to be formed each access point (vertex) is only allowed to appear once. We then take all remaining access points and all potential shortest path segments as input, and apply the algorithm `AllPairsShortestPath` to them.
to compute the shortest paths and thus their segments between any two access points in the network formed by the potential shortest path segments. Well known algorithms to perform this task are the Floyd-Warshall algorithm \cite{13} and Johnson’s algorithm \cite{14}. The result of applying such an algorithm to the scenario in Figure 3-8B is shown in Figure 3-8C. Because only concave vertices can be intermediate points, all shortest paths begin and end with an access point and have 0 or more concave vertices as intermediate points.

It remains to describe the operation \textit{DeterminePotentialPathSegs} whose task it is to determine all potential path segments for a given face and its access points as input. We give this description in Algorithm 2. The algorithm is general in the sense that a face can contain zero, one, or more holes. The overall strategy consists of three steps.

In a first step (lines 1 to 8), we construct all segments between any two access points and check whether they are located inside the face (lines 4 and 5). If this is the case, they are potential shortest path segments (line 6). Otherwise, they either intersect the face boundary (that is, the outer boundary or a hole boundary) or are outside the face (line 8). If no intersection has been found, the algorithm terminates since we have found all potential shortest path segments (line 22).

A check whether a segment is located inside a face is needed at three places in the algorithm, namely in lines 5, 15, and 20. To test whether a segment intersects the face boundary, we use a geometric primitive \textit{intersects} which traverses all boundary segments of the face until it finds one that intersects the segment built for two connection points (access points or concave points) by the constructor \textit{segment}. If we find out that a face does not intersect the segment built for two connection points, it can still be that the segment is outside the face (see Figure 3-7A). We take a variation of the well known plumpline algorithm for point-in-polygon testing (called in here) to check whether the center of this segment (determined by the function \textit{center}) is located inside the face. In summary, if a segment between two connection points does not intersect the
face boundary and the center of the segment is inside the face, then the segment is a potential shortest path segment.

From the second step (lines 9 to 16) on, we have to deal with the fact that there are straight segments that intersect the face boundary or lie outside the face. This means that concave vertices must be part of shortest paths between access points in the face. Since we do not know which concave vertices will later play a role in the shortest path finding process, we determine all of them by employing the function findConcaveVertices (line 10). This function traverses all pairs of consecutive face (outer and hole) boundary segments and checks whether their inner angle is reflex. We construct all segments between any two concave vertices and check whether they are located inside the polygon (lines 14 and 15) since only those are potential shortest path segments (line 16).

In a third step (lines 17 to 21), we create the connections between all access points and all concave vertices. Again we build all segments between them and check whether they are located inside the face. As a final result, we obtain and return all potential shortest path segments in the face (line 22).

Finally, we explore the runtime complexity of the complete algorithm. Note that the algorithm is only applied one time for each nested complex cell since then its all-pairs shortest path graph is stored in its representation in the segments $s_1, \ldots, s_2$. Since Algorithm 2 is applied to each face constructed in Algorithm 1, we assume that in Algorithm 2, $m$ is the maximum number of access points of any face, and $n$ is the maximum number of concave vertices of any face. Further, let $b$ be the maximum number of boundary segments of any face. The predicate intersects requires $O(b)$ time, and the function center can be performed in constant time. Hence, the complexity of finding all potential shortest path segments for any face is $O(b(m^2 + n^2 + mn))$.

In Algorithm 1, we form a face for each cell of the hierarchy levels 1 to $d - 1$. This means we form $\sum_{i=1}^{d-1} k_i$ faces. In order to build a particular face, for each cell of a
hierarchy level \(i\), we traverse all cells of the hierarchy level \(i + 1\) in order to find out a containment relationship between both cells. That is, we have \(\sum_{i=1}^{d-1} k_i k_{i+1}\) containment tests. Each containment test requires \(O((b + b) \log (b + b)) = O(b \log b)\) time for a needed plane sweep algorithm [18]. In total, the time needed for all containment tests is \(O(b \log b \sum_{i=1}^{d-1} k_i k_{i+1})\). From the collected list of access points for each face, we have to remove all pairs of equal access points (function \(rpairs\)) since they are not relevant for the determination of potential shortest path segments. This requires sorting the list and traversing the sorted list to delete equal pairs. This implies \(O(m \log m)\) time for each face and \(O(m \log m \sum_{i=1}^{d-1} k_i)\) time for all faces. Finally, we have to determine the cost of the all-pairs shortest path algorithm (like the Floyd-Warshall) algorithm to compute the shortest paths between all pairs of nodes. In a graph with \(v\) vertices, the worst case performance of this algorithm is \(O(v^3)\). For each face, the number of end points of all potential path segments are restricted by the \(m\) access points and the \(n\) concave vertices. Since we have \(\sum_{i=1}^{d-1} k_i\) faces, the worst time complexity of the all-pairs shortest path algorithm is \(O((m + n)^3 \sum_{i=1}^{d-1} k_i)\). As the overall worst case time complexity we obtain finally:

\[
O(b \log b \sum_{i=1}^{d-1} k_i k_{i+1} + b(m^2 + n^2 + mn) \sum_{i=1}^{d-1} k_i + m \log m \sum_{i=1}^{d-1} k_i + (m + n)^3 \sum_{i=1}^{d-1} k_i)
\]

With \(k = \max\{k_i | 1 \leq i \leq d\}\) and further simplifications, we obtain as the overall worst case time complexity:

\[
O(b \log b \ d \ k^2 (m + n)^3)
\]

Since the determination of the shortest path segments is only performed one time for each nested complex cell, the efficiency of the algorithm is not the major requirement. On the other hand, the algorithmic steps are needed, quite complex, and depend on the five parameters \(b, d, k, m, \) and \(n\).

### 3.2.3 Open Cells

An open cell is a cell for which at least a part of its boundary is not closed by explicit walls or other spatial constraints. Examples of open cells are concourses in airports.
as well as halls and lobbies in buildings. Different open cells can have various open boundary parts with different widths. Even in the same cell, multiple open boundary parts of different widths may exist. These variable widths make it difficult to determine the access points in the open boundary parts since the latter have more the character of “access lines”. Solid boundary parts may have access points like doors. Figure 3-4C shows an example of an open cell; a dashed line represents an open part of the boundary, and a solid line indicates a wall. The formal definition of an open cell is given in Definition 4.

**Definition 4.** An open cell $oc$ is given by a tuple $oc := (id, r, l, m, a_1, \ldots, a_m, t, s_1, \ldots, s_t)$ with $m \geq 0$ and $t \geq 1$. It represents a geometric structure that is partially enclosed by a solid boundary (usually walls) and partially enclosed by an open boundary enabling free access and transit. It must have at least one open boundary part. The component $id$ uniquely identifies this cell. The component $r$ is a value of the spatial data type region that stores the areal structure and extent of this cell as a simple, single-component polygon without holes. The component $l$ is a value of the spatial data type line and represents the open boundary parts of the open cell. That is, the predicate $\text{inside}(l, \text{boundary}(r))$ must hold where the function $\text{boundary}$ determines the border of a region object as a line object and the predicate $\text{inside}$ tests for spatial containment. The component $m$ denotes the number of access points on the solid boundary parts of this cell. The components $a_1, \ldots, a_m$ are values of the spatial data type point, represent (the locations, that is, the $xy$-coordinates, of) these access points, and lie on the solid boundary parts of $r$. That is, $\text{on}(a_i, \text{boundary}(r) \ominus l)$ must hold for all $1 \leq i \leq m$ where the predicate $\text{on}$ checks whether a point is located on a line. The component $t$ keeps the number of implicit path segments in this cell, and $s_1, \ldots, s_t$ represent these segments.

An open cell is different from a complex cell with respect to the open boundary parts. Further, it is difficult to determine the access points on its open boundary parts. When an open cell is the target object in a query, it is reasonable to consider the center
point of the open boundary part as one of the access points of the open cell. However, when an open cell plays the role of a passage, we cannot simply consider the center point because the shortest path can go through any point of the open boundary. To solve this problem, our approach to obtaining the path segments in an open cell consists of three steps. In the first step, we combine all open cells sharing open boundary parts until the combined cell is a complex cell closed by walls. In the second step, we apply the strategy of finding path segments in complex cells, that is, Algorithm 1, to this combined cell to obtain the path segments between the access points on the outer boundary. In the third step, for all open boundary parts, we select the center position of each open boundary part as its access point. Then we construct path segments between these new access points and all other existing access points.

As shown in Figure 3-13A, ac is the shortest path when a user wants to go through these open cells. When an open cell is the target object in a query, the best position to lead users is the center of the open boundary. Thus, in the third step, the center point of each open boundary is selected as an access point. As an example, in Figure 3-13B, d and e are two access points of the regions A and B respectively. When a user standing in the door b of the combined cell wants to go to region B, the best way for her is the segment be.
3.2.4 Connecters

A connector is a cell that connects different levels or floors in a building. A connector can be a stair, an elevator, a paternoster elevator, or another object that can be used to reach different floors. The location where a connector and a floor meet forms an access point in this connector. Figure 3-4D shows an example of a connector. The five dots represent access points and indicate that this connector connects five floors. There are only two directions in a connector: upstairs and downstairs. Once a user knows the number of his current floor and the number of the destination floor, he immediately knows the direction to the destination floor. Thus, we ignore the shape of connectors and assume that every two floors are straightly reachable. Then the path segments in a connector are the predefined segments connecting each pair of access points (see Figure 3-6C). Definition 5 gives the formal definition of a connector.

**Definition 5.** A connector co is given as a tuple \( co := (id, m, a_1, ..., a_m, t, s_1, ..., s_t) \) with \( m \geq 1 \) and \( t = \frac{(m-1)m}{2} \). It represents a geometric structure that connects all floors in a building. The component id uniquely identifies this cell. The component \( m \) denotes the number of access points of this cell. The components \( a_1, ..., a_m \) are values of the spatial data type point, represent (the locations, that is, the xy-coordinates, of) these access points, and form the connecting locations between the connector and the floors. The component \( t \) keeps the number of implicit path segments in this cell, and \( s_1, ..., s_t \) represent these segments. Each segment describes the transition from one floor to another floor.

3.2.5 Accessibility

An important issue of the way finding process is the aspect of accessibility of architectural cells in the indoor space. For example, while an employee in a building may have access to certain office rooms, these rooms might be inaccessible to a customer. Another example is a construction site that prevents people from walking through a
corridor and forces them to bypass it. In our model we control the accessibility of cells by assigning accessibility attributes to both access points and shortest path segments.

The accessibility in an access point or in a shortest path segment is controlled by a time stamp indicating when an access point or a shortest path segment is accessible.

Figure 3-14A shows the architectural map of a small mall together with its access points (doors) and its shortest path segments. The accessibility times are shown in Figure 3-14B for each door and in Figure 3-14C for each shortest path segment. We see that the accessibility of the doors \( d_3 \) and \( d_4 \) as well as the path \( d_3 - d_4 \) is given between 8 am and 5 pm. Only during this period of time, users can enter the music store through the door \( d_3 \) and leave the mall through the door \( d_4 \) by using the path \( d_3 - d_4 \).

The reason why we also assign an accessibility attribute to each path segment is that the accessibility of the interior of a path segment cannot be controlled by its two end points. For example, a path segment may not be accessible because of a construction site while its two end points are accessible. This means that an accessibility attribute only for access points or only for path segments would be insufficient.

The relationship between the accessibility of access points and the accessibility of path segments can be summarized by three observations. First, if an access point is
inaccessible, all its emanating path segments are inaccessible, and vice versa. It does
not make sense that any of its incident path segments is accessible since the access
point can never be reached. Second, if an access point is accessible, there is at least
one emanating path segment that is accessible, and vice versa. In other words, an
accessible access point must have at least one path segment that leads to it. Third,
even if two end points of a path segment are accessible, the path segment can be
inaccessible. This indicates that a path segment can be blocked although its two end
points are accessible due to other paths traversing them.

3.3 The iNav Model: Shortest Path Routing

Navigation is a process that successfully and efficiently leads users from a source
to a destination that users want to reach. Usually, it includes two main requirements.
First, it should be able to find the most appropriate routes to destinations. The most
appropriate routes may be the shortest routes with respect to time, the shortest routes
with respect to distance, the routes without paying fees, or any other routes according
to some user requirements. Second, navigation should be able to provide clear and
accurate descriptions for paths so that users can get to destinations by following the
descriptions. In this section, we will explain how we can obtain the shortest route based
on the implicit path segments constructed in Section 3.2. In Section 3.4 we will introduce
a simple navigation language to generate clear and useful descriptions for routes.

3.3.1 The Direct Path Graph

The most efficient method to calculate paths is the application of the shortest path
algorithm to graphs. In our case, we obtain an appropriate graph for navigation by
assembling the access points, intermediate points, and shortest path segments obtained
from Algorithm 1 into a so-called direct path graph (DPG). Definition 6 specifies such a
graph that we will use for running a shortest path algorithm.

**Definition 6.** A direct path graph $G = (V, E)$ is a graph which reflects all possible
shortest path constructions in a given indoor space scenario. $V$ is a set of access points
Figure 3-15. An example of navigation: A) The indoor space scenario from Figure 3-14A including the current position (cp) of a user. B) The corresponding DPG

and intermediate points, and \( E \subseteq V \times V \) is the set of shortest path segments obtained from Algorithm 1.

In Section 3.2, we have divided the indoor space into several non-overlapping cells and have shown how to construct the implicit path segments according to the shapes of the cells and the locations of the access points. A DPG for an indoor space is composed of the access points, intermediate points, and path segments derived from all cells embedded in this space. The reason why we call it direct path graph is that every edge in the graph represents a straight passage that is fully inside its corresponding cell. Figure 3-15B shows the direct path graph for the indoor space in the Figures 3-14A and 3-15A respectively. The nodes and edges inside each dashed oval belong to the same cell in the indoor space.

A DPG has several nice properties inherited from the shortest path segments. First, a DPG represents the whole infrastructure of the shortest path segments of an indoor space scenario. Therefore, a path from the current location to a target location can be obtained by applying the shortest path algorithm to a DPG. Second, edges in a DPG represent path segments in an indoor space and are straight lines between
access points, or between access points and intermediate points. Therefore, every edge represents the shortest path between any two connected nodes. Third, any two locations between two connected nodes in the graph are visible from each other in the indoor space and can reach each other without encountering an obstacle like a wall. Hence, the next target on a route during navigation can always be seen by a user. Fourth, the length of each edge in a DPG is stored in the attribute length for each path segment and calculated by the Euclidean distance between the segment end points. The length information for the edge labels of the DFG in Figure 3-15B stems from the table in Figure 3-14C.

3.3.2 Navigation Through the Direct Path Graph

Usually, users are interested in finding a path to a certain destination like an office or a store. For example, in Figure 3-15A, they might ask “How can I get to the music store?”. The problem now is that the nodes in a DPG are the access points and intermediate points in the indoor space. This implies that there is no explicit object information about rooms, corridors, stairs, and so on represented in this graph. Thus, a DPG cannot be directly used for answering navigation queries. But in this subsection, we will show how we can nevertheless leverage a DPG to find the desired shortest paths.

A DPG contains all the access points, intermediate points, and path segments for a certain indoor space. Since the current position and the target of a user might not be identical to any of the nodes in the DPG, our first step is to determine a suitable starting node and a suitable target node in a DPG. For representation purposes, we have divided the entire indoor space into several non-overlapping cells, and for each cell we keep its access points explicitly and its intermediate points implicitly as the end points of the shortest path segments. Thus, we locate the surrounding cell regarding the current position of the user and obtain the pertaining access points of this cell. We choose the access point or intermediate point that is nearest to the current position and set it
Figure 3-16. Route planning where the starting node is an intermediate point.

as the starting node for the shortest path algorithm. For example, in Figure 3-15A the current position $cp$ of the user is marked by a triangle. The clothing store is determined as the surrounding cell of the current position. Its access points are $d_8$ and $d_9$; there are no intermediate points. The nearest access point to the current position is $d_9$, which is hence the starting node. A starting node can also be an intermediate point. As shown in Figure 3-16, the intermediate points in this cell are $v_3$, $v_6$, $v_9$, $v_{14}$, and $v_{15}$. For the current position marked by a triangle, the starting node chosen is the intermediate point $v_6$.

Choosing the nearest access point or intermediate point as the starting point does not mean that a user needs to go to this starting point at the beginning. It is only used to represent the user’s current position and determine the access point where the user has to leave the surrounding cell. For our example in Figure 3-15A, we assume that a user wants to go to the shoeB store from his current location marked by the triangle in the clothing store. We know that $d_9$ is the nearest access point, and we find out that $d_9 - d_8 - d_6$ is the shortest path from $d_9$ to the shoeB store. However, the direct path from the current position to $d_8$ is shorter, and hence the user can avoid the path segment $d_9 - d_8$ inside the cell. In case that the user would like to go to the shoeA store, we learn that the shortest path from $d_9$ to the shoeA store is $d_9 - d_7 - d_2$. The access point $d_9$ is the suitable point to enable the user to leave the cell immediately. In a more general situation, if the access point where the user has to leave the cell is not visible from the user’s current position and hence cannot be directly approached on a straight way due to obstacles like walls, we apply the strategy from the Sections 3.2.2.3 and 3.2.2.4.
and compute the shortest path in the interior of the cell from the current position to this access point. Again the intermediate points of this shortest path will be concave vertices of the cell. The first path segment from the current position to the first intermediate point will not be one of the determined shortest path segments. However, the shortest path from the first intermediate point to the access point where the user leaves the cell will consist of determined shortest path segments. From the access point where the user leaves the cell, the shortest path will correspond to the one computed from the original starting node.

The way we determine the target node is different from the decision on the start node. Usually, if we want to know the way to a target cell, we just need to find the shortest way to any of its access points (e.g., doors and some openings) on its boundary. Thus, all access points on the boundary of the target cell are potential target nodes. For example, in Figure 3-15, if our target cell is the music store, then $d_3$, $d_4$ and $d_5$ are the potential target nodes. Since the shortest path algorithm is able to return the shortest paths from the starting node to any other node in a graph [15], we run this algorithm for all potential target nodes and determine that node (access point) as target node with the shortest distance from the starting node.

During the shortest path algorithm, we need to check the accessibility of path segments because some of the edges might not be accessible. For example, in Figure 3-14C, the accessibility of all path segments is controlled by the accessibility of its two end points. If we assume that the current time is 3 pm, the segment $d_7 - d_9$ will not be taken into account for any shortest path computation in this case since the accessibility of $d_7$ is limited to the period from 8 am to 1 pm only.

Some models represent each object by a single node only, as shown in Figure 3-3A. This prevents them from providing shortest paths. As said before, in Figure 3-3A, the path from door2 to room106 that these models will provide is shown by the solid line. This is a circuitous and unrealistic path, and the problem becomes serious when
room105 is very large. Our model overcomes this problem by viewing each object (cell) in general, by taking into account its geometric structure, and by recording all possible path segments in it. Thus, our model can provide a much more appropriate path from door2 to room106, as shown by the dashed line in Figure 3-3A.

3.4 The iNav Model: A Simple Navigation Language

A clear and detailed route description is an important criterion for the quality of a navigation system. Route descriptions provided by GPS based outdoor navigation systems are usually made up of the names, lengths, and turns of roads. While we can also derive length and turn information in an indoor space from the access points and intermediate points, we do not have explicit “roads” in an indoor space but only implicit and invisible path segments. In particular, the implicit path segments we create do not have specific names that we could use in route descriptions. Thus, it is impossible to provide route descriptions that are similar to those GPS systems provide. In this section, we will introduce a simple navigation language that is designed for the iNav system. We do not claim that our navigation language is an appropriate end user language and satisfies all the linguistic and cognitive requirements needed for such a language. But it could be the target language to which an end user language is translated. Section 3.4.1 discusses navigation descriptions in the iNav system and the linguistic means we need for them. Section 3.4.2 explains our method to automatically generate navigation descriptions from a determined shortest path.

3.4.1 Navigation Descriptions

In the iNav model, we use implicit and invisible path segments to describe routes. Due to their invisibility and lack of name, we cannot use these path segments directly for the description of a route. Instead, we use the sequence of connection points (i.e., access points and intermediate points) of the identified shortest path in order to describe the route the user has to walk. This is possible since the connection points physically
Figure 3-17. The navigation descriptions exist and are visible to the users. For example, in Figure 3-14A, the instruction to indicate the path from $d_2$ to $d_7$ is:

\[ \text{walk } x \text{ meters to } d_7 \text{ in corridor;} \]

where the value $x$ is the length of the segment $d_2 - d_7$, and \textit{corridor} is the cell that contains this path segment.

In our model, connection points and path segments have the very important property that any two consecutive connection points that are linked by a path segment are visible from each other and not impeded by obstacles like walls. Thus, users can easily find the next connection point in the instruction. However, if an intermediate point is involved, we cannot use the same instruction as above to describe the path. A feasible way to describe such a path is to use angles and directions. For example, assuming the user comes from room2, the instruction for the path from $a$ to $b$ in Figure 3-17 is:

\[ \text{turn } 25^\circ \text{ to the right, and walk } 20 \text{ meters in corridor;} \]

where $25^\circ$ is the degree of the angle 2, and 20 meters is the length of the segment $a - b$.

From the above two examples, we can learn that different words play different roles in navigation descriptions. For example, the terms \textit{right} and \textit{left} are used to indicate a change of direction, and \textit{corridor} denotes a landmark. According to the roles of different
words, we classify them into four term categories. **Topological terms** like **in** specify the topological relationships between users and objects. **Directional terms** specify the directions users have to take. We distinguish absolute directions, known as cardinal directions, like **east** and **west**, as well as relative directions like **right** and **left**. Cardinal directions are seldom used in an indoor space but much more important in outdoor spaces. We apply relative directions in our navigation descriptions. The relative direction depends on the orientation of the surrounding space and the user’s current position and walking direction. **Landmark terms** point to distinguishable signs or objects that can help users finding locations. For example, the signs of road names are the most important landmarks in an outdoor space. In an indoor space, the door of a room or a room itself can be considered as a distinguishable landmark. Since the landmarks are used to show users the way to their destination, they have to be distinguishable and to be easily found. In our model, the landmarks correspond to the names of cells, and we make sure that every landmark we provide is visible to users. **Metric terms** address quantitative aspects in contrast to the aforementioned term categories that focus on qualitative aspects and only provide a general direction. We use metric terms to provide precise quantitative information so that users are able to obtain more detailed information of the routes. In the previous example, the length of the current path (20 meters) and the degree of an angle (25°) are the metric terms.

We use the scenario shown in Figure 3-18 to explain how we obtain the directional and metric terms of a navigation description. Assume $u - v$ is the current path segment and $v - w$ is the next path segment in the sequence of a path. The angle $\alpha$ is calculated by the law of cosines as $2ab \cos(180° - \alpha) = a^2 + b^2 - c^2$, where $a$, $b$, and $c$ are the lengths of $u - v$, $v - w$, and $u - w$ respectively in Figure 3-18. The calculation of the direction is based on Lemma 3 (without proof).

**Lemma 3.** Assume $u$, $v$, and $w$ are points, $u - v$ is a segment, and we look from $u$ to $v$ (vector view $u \rightarrow v$). The relative direction between $u - v$ and $w$ is defined as follows:
Figure 3-18. The calculation of angles and directions

(i) \( w \) is to the right of \( u - v \) if, and only if,
\[
(v.x - u.x)(w.y - u.y) + (w.x - u.x)(u.y - v.y) < 0
\]

(ii) \( w \) is to the left of \( u - v \) if, and only if,
\[
(v.x - u.x)(w.y - u.y) + (w.x - u.x)(u.y - v.y) > 0
\]

(iii) \( w \) is on \( u - v \) if, and only if,
\[
(v.x - u.x)(w.y - u.y) + (w.x - u.x)(u.y - v.y) = 0
\]

For example, in Figure 3-18, let us assume that \( u - v \) is a path segment in room1, \( v - w \) is a path segment in room2, \( v \) is an explicit access point, \( w \) is an intermediate point, and the coordinates of \( u, v, \) and \( w \) are \((2, 1), (1, 3), \) and \((4, 2)\) respectively.

Then the length of \( u - v \) is \( a = \sqrt{(2 - 1)^2 + (1 - 3)^2} = \sqrt{5} \), the length of \( w - v \) is \( b = \sqrt{(4 - 1)^2 + (2 - 3)^2} = \sqrt{10} \), and the length of \( u - w \) is \( c = \sqrt{(2 - 4)^2 + (1 - 2)^2} = \sqrt{5} \). According to these values, the angle \( \alpha \) can be obtained by applying the formula
\[
\cos(180^\circ - \alpha) = \frac{a^2 + b^2 - c^2}{2ab} = \frac{5 + 10 - 5}{2\sqrt{50}} = \frac{\sqrt{2}}{2}.
\]
Finally, we obtain that \( \alpha \) equals to \( 135^\circ \).

Next, we need to determine the relative direction between the segment \( u - v \) and the point \( w \). According to Lemma 3, we determine the relative direction by comparing the result of \((v.x - u.x)(w.y - u.y) + (w.x - u.x)(u.y - v.y)\) with 0. Since \((1 - 2)(2 - 1) + (4 - 2)(1 - 3) = -5 < 0\), we see that \( w \) is on the right side of the segment (vector) \( u - v \). Therefore, if a user walks from \( u \) to \( v \) and wants to go to \( w \), she needs to turn right by \( 135^\circ \). The final navigation description along the path segments \( u - v \) and \( v - w \) is:

(1) walk 2 meters to \( v \) in room1;

(2) turn \( 135^\circ \) to the right, and walk 3 meters in room2.
3.4.2 Automatic Generation of Navigation Descriptions

In this section, we show how navigation descriptions can be automatically generated by leveraging the four term categories introduced in Section 3.4.1. In Section 3.2, we have distinguished the explicitly known access points and the implicitly generated intermediate points as the segment end points found in shortest paths.

The first node is the user’s current location. Considering that the user can be anywhere in the indoor space, the current location may not be an access point or an intermediate point. If there are only two nodes involved, the second node must be an access point. If there are more than two nodes, the second node can be an intermediate point. This means that if we read nodes one by one, then the first node is neither access point nor intermediate point. The modified figure is attached. In this figure, P from status S is the second node in the generated path. A shortest path is always bounded by two access points indicating the start node and the target node. Intermediate points cannot be end points of a shortest path.

We have introduced how to calculate the directional descriptions and value descriptions. Now we will show how to generate the formal navigation language by using the four descriptions. In Section 3.2, we learn that there are two kinds of nodes, the access points which are the explicit exits of each room, and the intermediate points which are generated during the process of the path construction. The generation of the navigation description depends on the types of the nodes. If the next node is an access point, then it can be considered as a visible landmark, otherwise, more detailed information should be provided in order to get to the next node. Figure 3-19 shows the status transformation diagram of the navigation language:

**input data**: a sequence of nodes indicating the generated shortest route. \( w \) is the current node that is read from the sequence.
Figure 3-19. The status transformation of the language generation and directions

**S**: $S$ is the start status of the language generation. In this status, the system read the first three nodes from the input data, and check the type of the third node $w$. If $w$ is an access point, go to the status $P$, else go to the status $U$.

**P**: In this status, the current node is an access point. The corresponding instruction generated in this status is based on the current node. Then read the next node, check its type, and go to the corresponding status.

**U**: In this status, the current node is an intermediate point. The corresponding instruction generated in this status is based on the current node. Then read the next node, check its type, and go to the corresponding status.

**E**: The end of the process.

In the status $P$, the current node is an access point, which can be used as a visible landmark. Assuming the previous node is $p_v$ and the current node is $p_w$, then the instruction generated in this status is:

*Walk “length” to $p_w$ in “room$_x$”*

“length” is a value description indicating the distance between $p_v$ and $p_w$. It is calculated by applying the formula $\sqrt{(p_{vx} - p_{wx})^2 + (p_{vy} - p_{wy})^2}$. $p_w$ is a landmark and *in* is a topological description indicating the relation between this path segment and *“room$_x$”*. 
Table 3-1. The path chain from room106 to room103

<table>
<thead>
<tr>
<th>start node</th>
<th>next node</th>
<th>length</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>current location</td>
<td>$d_9$</td>
<td>10</td>
<td>room106</td>
</tr>
<tr>
<td>$d_9$</td>
<td>$d_7$</td>
<td>14</td>
<td>room104</td>
</tr>
<tr>
<td>$d_7$</td>
<td>$d_3$</td>
<td>13</td>
<td>corridor</td>
</tr>
</tbody>
</table>

In the status $U$, the current node is an intermediate point, which is not a distinguishable landmark in indoor space. Thus, the navigation instruction should contain more detailed information to help users to reach that position. Assuming the previous two nodes are $p_u, p_v$ and the current node is $p_w$, the instruction generated in this status is:

*Rotate “degree” to the “direction”, and walk “length” in “room$_x$”*

“Degree” and “length” are value descriptions. The “degree” indicates the degree of the angle that the user should take, and the “distance” provide the distances between $p_v$ and $p_w$. “direction” is a directional description which tells users to turn right or left, and $in$ is a topological description indicating the relation between this path segment and “room$_x$”.

For example, Table 3-1 shows the shortest path from the location marked by the triangle in the clothing store to the music store in Figure 3-14A. The final description generated for this path is

(1) walk 10 meters to $d_9$ in room106;
(2) walk 14 meters to $d_7$ in room104;
(3) walk 13 meters to $d_3$ in corridor.

The information of path segments is implicitly indicated in each instruction. For example, the path segment indicated by the first instruction is from the current location to $d_9$, and the path segment shown by the second instruction is from $d_9$ to $d_7$.

Figure 3-17 gives an example that some intermediate points are involved. Assuming the degree of angle 1, 2 and 3 are 50°, 25° and 30° respectively, the description of the path from room2 to room9 is:
(1) rotate 50° to the right, walk 30 meters in the lobby;
(2) rotate 25° to the right, walk 20 meters in the corridor;
(3) walk 10 meters to $d_{14}$ in the corridor.
CHAPTER 4
SUPPORTING RANGE QUERIES IN INDOOR SPACE

4.1 Approaches to Supporting Range Queries in Outdoor Space

To the best of our knowledge, current studies mainly focus on range queries in outdoor spaces based on the Euclidean or network distances, but rarely any approach is proposed for indoor spaces.

4.1.1 Stationary Range Queries

A stationary range query asks for interesting objects within a given distance with respect to a static query point. Approaches supporting stationary range queries can be subdivided into two categories: Euclidean-based approaches and network-based approaches. Euclidean-based approaches check the distances between objects by their relative positions in the space while network-based approaches consider their actual reachable distances.

4.1.1.1 Euclidean-based approaches

The most typical Euclidean-based approaches proposed in [29, 41, 62] use R-Trees [25], R+-Trees [91] and R*-Trees [7] for query processing. Figure 4-1 shows an example of R-tree with the capacity of 3 entries per node. Neighboring points (e.g. a, b in Figure 4-1) are gathered in to one node and represented by a minimum bounding rectangle (E1). Then nodes are recursively clustered until the root node is formed. The process of a range query starts from checking the root, to each node overlapped with the range circle whose center is the query point q and the radius is the required distance, so that all qualifying objects within this circle can be obtained.

Due to its simplicity and efficiency, R-tree becomes a popular index structure for spatial databases. However, the major disadvantage of R-tree is that objects are evaluated based on their Euclidean distances, not the actual reachable distances. However, in indoor space, objects are separated by walls, which made it impossible
to evaluate the distances between objects by their Euclidean distances. Thus, the Euclidean-based approaches are not suitable for indoor space.

### 4.1.1.2 Network-based approaches

More practical solutions to support range queries are based on spatial networks. In [65], Papadias and Zhang propose a network-based approach, called range network expansion (RNE), to support stationary range queries. It first selects all segments whose distances to the query point are less than the given range distance. Then for each selected segment, check the corresponding nodes on the R-tree. For example, in Figure 4-2, assuming segments \((n_1, n_2), (n_1, n_3), (n_3, n_4),\) and \((n_4, n_5)\) are the segments inside the given range. Then the gray nodes in the R-tree are the visited nodes, and the final objects of the range query are \(b, d\) and \(f\).

Using Voronoi diagram is another choice of the network-based approaches. In [64], Okabe et al. first introduce the approach to convert the network into a Voronoi
diagram. This idea is then used by Kolahdouzan and Shahabi in their Voronoi-based Network Nearest Neighbor ($\text{VN}^3$) model proposed in [39]. In $\text{VN}^3$, the entire network is decomposed into several smaller parts, each of which is represented by a first-order Voronoi diagram, called NVP. Then the intra and inter distances between the cells of the Voronoi diagram are pre-computed for the distance evaluation purpose. The PINE model presented by Safa et al. in [78] uses a first order Voronoi diagram to partition the network into several sub-networks and computes their inter distances. Then a method similar to the RNE approach is applied to compute the intra distances of these sub-networks. In [78], the authors mention that PINE requires less disk access time and less CPU time than $\text{VN}^3$.

The network-based approaches provide the ability of finding the exact network distances between objects. Thus, they are superior to the Euclidean-based approaches when considering their practical utilization. Moreover, These approaches can support not only the range queries, but also some other common queries like the $K$ Nearest Neighbor (KNN) queries and Closest Pair queries. Generally, the RNE approach has better performance than the Voronoi-based approaches for range queries. However, if the object distribution is dense, the Voronoi-based approaches are usually better, because RNE have to retrieve more data from the database to obtain the final objects.

### 4.1.2 Continuous Range Queries

Approaches for stationary range queries assume that the query point is always in a static position. In fact, users are often in a moving status when they issue a navigation query. The importance of continuous queries is mentioned by Sistla et al. in [84]. They propose a data model called Moving Objects Spatio-Temporal (MOST) for presenting moving objects in database systems, which focuses on the syntax and semantics of the query processing. In recent years, more and more approaches to supporting continuous range queries have appeared which also can be classified into Euclidean-based approaches and network-based approaches.
4.1.2.1 Euclidean-based Approaches

An earlier Euclidean-based approach introduced in [87] first selects several sample points on the path, and then performs stationary range queries on each selected sample point. This approach suffers from a performance and accuracy problem: the more sample points we select, the more accurate but poorer the performance will be. The less sample points we take, the better but less accurate the performance will be.

In order to overcome the performance and accuracy problem, later approaches extend the concept of time-parameterized queries proposed in [92] to better support continuous queries. The time-parameterized approach incrementally calculates the next result and the objects that might affect it based on the current result. The result of Time-parameterized query is in the form of $<R, T, C>$, where $R$ is the current result, $T$ is the period time when $R$ is valid, and $C$ is the set of objects that might affect $R$ after the time of $T$. The time-parameterized approach can support the continuous range query but the cost of its incremental calculation is large. In [103] Tao et al. propose an idea of finding split points which indicate the changes of the qualified objects, to support continuous nearest neighbor queries. As shown in Figure 4-3, $a$ and $b$ are two interesting objects on the way from the start $S$ to the end $E$. The split point, which is determined by the position of $a$ and $b$, indicates that the nearest object exchanges from $a$ to $b$. The approach proposed in [102] (shown in Figure 4-4) first determines all possibly qualified objects by drawing a region according to the given range (indicated by solid lines). Then, the split points ($s_1, s_2, \ldots, s_7$) are determined by the intersections between the path and the dashed circles. The distances from the center point of each circle to its related split points are the given range.

4.1.2.2 Network-based Approaches

To the best of our knowledge, the approach proposed in [102] is the only one for supporting continuous range queries based on network distances. This approach is motivated by the idea of continuous K-nearest neighbor queries proposed in [40]
As shown in Figure 4-5, this approach first selects a segment that contains no intersection node (e.g. $AB$). Then it incrementally checks the nodes in nearby segments within a given range. The split points are determined by checking the condition $Sp[A, x] > dis[A, B]$, where $x$ is an interesting object, $dis[A, B]$ is the distance between $A$ and $B$, and $Sp[A, x]$ is computed by the formula $\text{given range} - dis[A, x]$.

For example, assuming the given range distance is 10 meters, we can learn that $Sp[A, o1] = 10 - 3 = 7$. Therefore, the split point $s_{o1}$ on the segment $AB$ indicates the validity of the object $o1$.

In Euclidean-based approaches, the objects are evaluated based on their Euclidean distances and not on the actually reachable distances. Considering the walls separating the indoor space, it is impossible to apply Euclidean-based approaches to indoor space. The network-based approaches provide the ability of finding the exact network distances. Thus, they are superior to the Euclidean-based approaches when considering their practical utilization. However, none of them can be directly applied to indoor spaces for accurate computation of range queries. In this paper, we will propose two approaches to supporting continuous range queries in indoor spaces respectively.
4.2 Supporting Stationary Range Queries

In Chapter 3, we have demonstrated how to construct the network structure for indoor space. The construction of the network has two important features that can nicely support range queries:

- All cells in indoor spaces are represented by at least one node in a DPG.
- For any node in the network, we can find the shortest distances between this node and all other nodes.

The first feature makes sure that no cell will be missing in the graph, and the second feature ensures that it is possible to find all the cells that are reachable within a given distance. In this section, we propose the Indoor Range Network Expansion (IRNE) algorithm to support stationary range queries, and the Indoor Range Region Division (IRRD) algorithm to support continuous range queries in indoor space. These two algorithms utilize the idea of the Incremental Network Expansion proposed in [65], and extends it to fit the indoor environment.

4.2.1 Incremental Network Expansion

The Incremental Network Expansion (INE) approach presented in [65] is a network-based approach used for the Nearest Neighbor Search. Its basic idea is to expand the network segments from the query point to obtain the qualifying objects. As shown in Figure 4-6, assuming a user wants to find the nearest object of interest (denoted by black rectangles) from q, INE will first check the objects in the segment...
\[ n_1 n_2 \]. Then, the segment \( n_1 n_7 \) emanating from \( n_1 \) will be checked, followed by the segments \( n_2 n_3 \) and \( n_2 n_4 \) emanating from \( n_2 \). The expansion process will continue until the nearest qualifying object is found. Finally, \( p_5 \) is returned as the final result.

### 4.2.2 Indoor Range Network Expansion

Once we have built the DPG for the entire indoor space, we can process stationary spatial graph range queries by applying an extended version of the INE algorithm to the graph. The underlying graph problem is a variation of the single-source shortest path problem by Dijkstra with a number of improvements summarized, for example, in [94], in which we have to find the shortest paths from a source vertex to all other vertices in the graph. However, in our case, we are only interested in those shortest paths that satisfy a particular distance condition and further thematic conditions. In other words, our goal is to prune the search graph as part of the DFG as much as possible. An example is the stationary range query “find all shoe stores within 10 meters”. Here “10 meters” is the distance or range condition, and “shoe store” is a thematic condition. Since a users current position \( cp \), which has the role of a query point and source vertex, is not restricted to the locations on the segments of the DPG, it is difficult to determine the starting segment. In order to overcome the problem, the *Indoor Range Network Expansion (IRNE)* algorithm (shown in Algorithm 3) takes two steps to compute stationary spatial graph range queries.
In the first step, if the query point \( cp \) is not on any segment of the DFG \( G \), we temporarily compute the shortest paths from \( cp \) to all access points that belong to the cell \( c \) containing \( cp \). In this way, we connect \( cp \) to \( G \) and obtain a temporarily new graph \( G' \). We leverage the operation \text{ExtendDFG} \ (\text{line} \ 1) \) to accomplish this. It deploys concepts that are similar to those in Algorithm 1. That is, if direct segments from \( cp \) to the access points of \( c \) are not possible since they would intersect the boundary of \( c \), we have to identify the concave points that are needed as intermediate points on the shortest paths.

In the second step, we perform the actual spatial range search by leveraging a modification of Dijkstra’s algorithm. Our extended DFG \( G' \) is a weighted, undirected

\begin{algorithm}
\caption{Indoor Range Network Expansion}
\begin{algorithmic}[1]
\Input (i) DPG \( G \); (ii) point \( cp \) denoting the current position of the user; (iii) cell \( c \) containing \( cp \); (iv) predicate \( p \) that is applied to a node of \( G \) and checks a thematic condition; (v) network distance \( r \) representing the search range
\Output set \( O \) of objects of interest; array \( \text{pred} \) that enables the determination of the shortest paths from \( cp \) to any object of interest in reverse order
\begin{algorithmic}
\State \( G' = \text{ExtendDPG}(G,c,cp) \); // \( G' = (V',E') \);
\State \( Q \leftarrow \{cp\} \); \( \text{ndist}[cp] \leftarrow 0 \); \( \text{pred}[cp] \leftarrow \emptyset \);
\ForAll {\( v \in V' - \{cp\} \)} \State \( \text{ndist}[v] \leftarrow \infty \); \( \text{pred}[v] \leftarrow \bot \);
\EndForAll
\ForAll {\( e \in E' \)} \State \( \text{visited}[(v,u)] \leftarrow \text{false} \);
\EndForAll
\While {\( Q \neq \emptyset \)} \State Find a vertex \( v \in Q \) such that \( \text{ndist}[v] = \min \{ \text{ndist}[u] \mid u \in Q \} \);
\State \( Q \leftarrow Q - \{v\} \);
\ForAll {\( u \in \text{Adj}[v] \)} \State if not visited\((v,u)\) then \State \( \text{visited}[(v,u)] \leftarrow \text{true} \);
\EndIf
\If {\( \text{ndist}[u] > \text{ndist}[v] + \text{weight}[(v,u)] \)} \State \( \text{ndist}[u] \leftarrow \text{ndist}[v] + \text{weight}[(v,u)] \);
\State \( \text{pred}[u] \leftarrow v \);
\EndIf
\If {\( \text{ndist}[u] \leq r \)} \State \( Q \leftarrow Q \cup \{u\} \);
\EndIf
\If {\( \text{ap}(u) \land p(u) \)} \State \( O \leftarrow O \cup \{u\} \);
\EndIf
\EndWhile
\end{algorithmic}
\end{algorithm}
graph with nonnegative edge weights. In line 2 we initialize a set $Q$, which has the role of a priority queue, with the current position $cp$. Further, we make use of two arrays $ndist$ and $pred$. The array $ndist$ stores the network distance between $cp$ and all qualifying objects of interest. We set the distance between $cp$ and itself to 0. The array $pred$ allows us to construct the shortest path between $cp$ and all objects of interest in reverse order. We set the predecessor of $cp$ to undefined and initialize the set $O$ of objects of interest with the empty set. In lines 3 and 4 we initialize the network distances from $cp$ to all other nodes in $G'$ with the value $\infty$ and set the predecessor of all these nodes to undefined. Then we mark each edge of $G'$ as unvisited (line 5).

The priority queue $Q$ keeps all those nodes that already belong to the set $O$ of objects of interest and from which we can hope to find further objects of interest. The goal is to find the shortest path to these result nodes to increase the chance to find other objects of interest. Hence, the same node can appear several times in $Q$. The reason is that if we find a shorter path from $cp$ to a node $v$, this can have positive impact on adjacent nodes of $v$ that could now qualify. The algorithm terminates as soon as $Q$ is empty (line 6). From $Q$ we always take the node $v$ with the smallest network distance from $cp$ (line 7) and remove it from $Q$ (line 8). For each adjacent node $u$ of $v$ we check whether the edge $(v, u)$ is unvisited (line 10). If $(v, u)$ has already been visited before, we ignore $u$ since going from $u$ to $v$ and back to $u$ cannot contribute to finding a shortest path. If the edge $(v, u)$ has not been visited before, we mark it as visited (line 11). Then we check whether we are able to lower the current network distance from $cp$ to $u$ to a new minimum when we traverse $(v, u)$ (line 12). As long as the network distance from $cp$ to $u$ is greater than the given range $r$, $u$ is not part of the solution. But if the network distance is within the search range (for example, within 10 meters) (line 13), $u$, which can be an access point or intermediate point, is inserted into $Q$ (line 14) to later continue the search from here for potentially other qualifying nodes, and the predecessor of $u$ is set to $v$ (line 15). Whether $u$ is added to the set $O$ of objects of interest depends on two
further conditions (line 16). First, \( u \) must be an access point and not an intermediate point. This is determined by the predicate \( ap \). Second, additional thematic conditions (for example, the target must be a shoe store) must hold. We have represented these thematic conditions by a given predicate \( p \).

For the first step, the time complexity of constructing path segments from the current position to the access points of the cell \( c \) is \( O((m + n)^3) \) where \( n \) is the number of access points and \( m \) is the number of concave vertices in the current cell. In fact, the values of \( m \) and \( n \) can be assumed to be relatively small in practise and will especially be much smaller than \( |V| \). For the second step, in practise, we can expect that only a small part of \( G' \) will be traversed. But depending on the size of the range \( r \), in the worst case, the complete graph \( G' \) has to be explored. This brings us then back to the single-source shortest path problem. As is known from the literature, the total running time of Dijkstra's algorithm is \( O(|E| + |V| \log |V|) \) if we store the vertices of the priority queue \( Q \) in a Fibonacci heap [28], and \( O(|E| \log |V|) \) if we store these vertices in a regular binary heap [29]. In both cases, \( ndist[v] \) is the key of a node \( v \).

As an example of the IRNE algorithm, we look back to Figure 4-7A that shows an indoor floor plan with the path segments of its DFG and a current position \( cp \) represented by the triangle in the clothing store. Figure 18b shows the explicit and extended DFG \( G' \) after the first step of Algorithm 3 with all network distances between adjacent nodes as edge weights. The temporary path segments from \( cp \) to all access points in the clothing store are shown by dashed lines. For reasons of simplicity, this DFG does not contain intermediate points. We further assume that all the access points are accessible. For the query “find all the shoe stores within 10 meters”, in the second step of the IRNE algorithm, \( Q \) first contains the node \( cp \), which is then removed from \( Q \). The adjacent nodes of \( cp \) are \( d_8 \) and \( d_9 \) with \( ndist[d_8] = 5 \) and \( ndist[d_9] = 3 \), and the edges from \( cp \) to the two nodes are marked as visited. Both nodes are inserted into \( Q \) since their network distance from \( cp \) is less than or equal to \( r = 10 \). However, both
nodes do not belong to the set $O$ of objects of interest since they are not doors of shoe stores. The node with the smallest network distance in $Q$ is $d_9$, which we remove from $Q$. We check all adjacent neighbors of $d_9$ which are $cp$, $d_8$, and $d_7$. The edge between $d_9$ and $cp$ has already been visited and is thus ignored. The edge between $d_9$ and $d_8$ is marked as visited but leads to a total cost of 11 from $cp$ to $d_8$ which is larger than $r$ and larger than the current $ndist[d_8]$. The edge from $d_9$ to $d_7$ is marked as visited. We obtain $ndist[d_7] = ndist[d_9] + weight[(d_9, d_7)] = 3 + 6 = 9 \leq 10$. Hence, $d_7$ is inserted into $Q$. But $d_7$ is not inserted into $O$ since it is not the door of a shoe store. The next node in $Q$ with the smallest network distance from $cp$ is $d_8$ which is removed from $Q$. Its adjacent nodes are $cp$, $d_9$, $d_7$, $d_2$, $d_3$, and $d_6$. The edges $(d_8, cp)$ and $(d_8, d_9)$ have already been visited. The other three edges are marked as visited. For the node $d_7$ we maintain the old value $ndist[d_7] = 9$ and insert $d_7$ into $Q$ where it already is. For the nodes $d_2$ and $d_3$ we obtain $ndist[d_2] = 20$ and $ndist[d_3] = 17$ which all exceed the threshold value 10. For the node $d_6$ we obtain $ndist[d_6] = 7 \leq 10$. Therefore, $d_6$ is inserted into $Q$. Since this node is an access point ($ap(d_6) = true$) of a shoe store ($p(d_6) = true$) named shoeB, $d_6$ is inserted into $O$. The next node in $Q$ with the smallest network distance from $cp$ is $d_6$ which is removed from $Q$. Its adjacent nodes are $d_8$, $d_7$, $d_2$, $d_3$, and $d_5$. The edge $(d_6, d_8)$ has already been visited. The other edges are marked as visited. For the node $d_7$ we maintain the old value $ndist[d_7] = 9$ and insert $d_7$ into $Q$ where it already is. For the nodes $d_2$ and $d_3$ we get $ndist[d_2] = 19$ and $ndist[d_3] = 14$ which both exceed the threshold value 10. For the node $d_5$ we get $ndist[d_5] = 10$; hence, we insert $d_5$ into $Q$. The node $d_5$ is also an access point to a shoe store named shoeB so that it is inserted into $O$. Next, $d_7$ is taken from $Q$. As one can see easily, all edges emanating from $d_7$ have either already been visited, or the shortest path from $cp$ to them exceeds the threshold 10. The same holds for the node $d_5$ that is removed from $Q$. Finally, $Q$ is empty, and the solution is $O = \{d_5, d_6\}$ leading to the shoe store shoeB.
4.3 Supporting Continuous Range Queries

Traditionally, approaches of supporting continuous range queries for outdoor spaces try to find the split points on the road at which the qualifying objects will change. When the query point moves within the interval between two split points, the result of the range query remains the same, and when the query point passes over one split point, the set of the qualifying objects changes accordingly. Since we don’t have explicit paths in indoor spaces, the approach of finding split points cannot be applied to cell-based indoor structures. The worst way to answer continuous range queries in indoor spaces is to apply the $IRNE$ algorithm for each position that the user moves. However, this way will consume a lot of energy due to CPU computation, and not feasible for a real world product. In this section, we introduce our approach, called $Indoor$ $Range$ $Division$ ($IRD$) for continuous range queries that can dramatically reduce the computation time and quickly answer continuous range queries.

In indoor spaces, when the user goes from one room to another, she is actually passing through multiple exits. We observed that, although the positions of the query points are changing dynamically, the positions of the exits and the objects are static, and the distances between the exits and the objects are fixed. Once we obtain the qualifying objects that are reachable from one exit, we can monitor the distance between the query point and the exit to determine the qualifying objects that can be reached from the query point. For example, assuming cell $c$ has one exit $a$, and the qualifying object that can
Figure 4-8. Examples of finding split points. A) A cell with one accesspoint. B) A cell with multiple accesspoints.

be reached from a is $O_b$. We observed that from any point $p$ in $c$, if the $\text{dist}(a, p)$ is less than the value of $\text{range} - \text{dist}(a, O_b)$, we can get to $O_b$ through $a$ within the given range. Here, the value of $\text{range}$ and $\text{dist}(a, O_b)$ are fixed. To find the qualifying objects for any point in $c$ whose position is changing dynamically, we only need to monitor the distances from $p$ to each exit in $c$.

In order to obtain qualifying objects based on the distances between the query point and the exits, one of our tasks is to find the relationships between the distance (i.e. the distance between the query point and the exits) and the qualifying objects that are reachable from the exits. As shown in Figure 4-8A, the rectangle represents a cell $c$ with one accesspoint $a$. Assuming the given range is 10, we can learn that $O_1$ and $O_2$ are two qualifying objects that can be reached from $a$. It is clear that from any position $p$ inside $c$, whose $\text{dist}(a, p)$ is less than 4 and greater than 2, the user can get to $O_1$ within 10, and if the $\text{dist}(a, p)$ is less than 2, both $O_1$ and $O_2$ are reachable from $p$. Here $< 0, 2 >$ and $< 2, 4 >$ are two intervals indicating the relationships between $\text{dist}(a, p)$ and the qualifying objects. If $\text{dist}(a, p)$ is between 0 and 2, the qualifying objects for $p$ are $O_1$ and $O_2$, and if $\text{dist}(a, p)$ is between 2 and 4, only $O_1$ is the qualifying object from $p$. We can represent the relationships between the distances and the qualifying objects by using the interval representation that is defined in Definition 7. The result for
Figure 4-8A can be represented as $< 0, 2 > \{ O_1, O_2 \}; < 2, 4 > \{ O_1 \}$. If the cell has more than one accesspoint, each accesspoint will have its own interval representation. In Figure 4-8B, there are two accesspoints $a$ and $b$ in the cell. The interval representations for $a$ and $b$ are shown in Table 4.3.

**Definition 7.** Assuming $r$ is the given range, and the list of the qualifying objects that can be reached from accesspoint $a$ is represented as:

$$< r_1, O_1 > < r_2, O_2 > \ldots < r_i, O_i > < r_{i+1}, O_{i+1} > \ldots < r_n, O_n >, r_i < r_{i+1}$$

where $O_i$ is a qualifying object and $r_i$ is the distance between $a$ and $O_i$.

The interval representation for $a$ is:

$$< 0, r - r_n > \{ O_1, \ldots, O_i, \ldots, O_n \}, \ldots, < r - r_{i+1}, r - r_i > \{ O_1, O_2, \ldots, O_i \}, \ldots, < r - r_2, r - r_1 > \{ O_1 \}$$

Our approach to support continuous range queries contains two phases. In the first phase, when the user enters a cell, we generate the interval representations for all accesspoints in the cell. The first phase is done once for all when the user enters the cell. In the second phase, we keep tracking the distances between the query point and the accesspoints. The final qualifying objects are obtained by looking up the interval representations generated in the first step.

Algorithm 4 shows our approach to compute the interval representations for all accesspoints in one cell. There are four steps in Algorithm 4. The first step is to construct the extended DPG by adding the shortest paths from the query point to all the exits in the cell (line 1). In the second step, we compute the qualifying objects that can be reached from each exit by using the algorithm for answering stationary range queries (line 3-4). The complexity of this step is discussed in Section 4.2. In the third step, the list of the objects are sorted according to their distances to the exit (line 5). The nearest object from the exit will be the first element in the sorted list. Finally, the interval representation for each accesspoint is generated at the fourth step (line 7-14). We omit the detailed explanation here since this step is simply a computation based on
Algorithm 4: Indoor Range Division

Input: (i) DPG G; (ii) point cp denoting the current position of the user; (iii) cell c containing cp; (iv) predicate p that is applied to a node of G and checks a thematic condition; (v) network distance r representing the search range

Output: set O of objects of interest;

1. \( G' = \text{ExtendDPG}(G, c, cp); \)
2. \( \text{resultSet}[ap] = \emptyset; \)
3. \( \text{finalList}[ap] = \emptyset; \)
4. foreach accesspoint ap in c do
   5. \( \text{result}[ap] = \text{Indoor Range Network Expansion}(G', cp, c, p, r); \)
   6. \( \text{sort}(	ext{resultSet}[ap]); \)
7. foreach accesspoint ap in c do
   8. \( \text{dist}_\text{prev} = r; \)
   9. \( \text{dist} \)
   10. for i = 0 to k do
        11. \( \text{finalList}[ap][i].\text{min} = r - \text{dist}_\text{prev}; \)
        12. \( \text{finalList}[ap][i].\text{max} = r - \text{resultSet}[ap][i].\text{dist}; \)
        13. for j = 0 to i do
            14. \( \text{finalList}[ap][j].\text{obj}.\text{add}(	ext{resultSet}[ap][i].\text{O}); \)
       15. \( \text{dist}_\text{prev} = \text{resultSet}[ap][i].\text{dist}; \)

Definition 7. The complexity to compute the interval representation is \( O(nm^2) \), where \( n \) is the number of accesspoints and \( m \) is the average number of qualifying objects that can be reached from each accesspoint. In fact, the values of \( m \) and \( n \) can be assumed to be relatively small in practise and will especially be much smaller than the number of nodes in the graph.

If the user moves from one position to another inside the same cell, we only need to compute the distance from the query point to each exit and find the corresponding objects by looking up the interval representations. If the cell has more than one accesspoint, the final qualifying objects will be generated by combining all the qualifying objects obtained from each interval representation. For example, assume the user is currently in cell \( c \) shown in Figure 4-8B. Table 4.3 shows the interval representation of Figure 4-8 by using Algorithm 4. When the user moves to position \( p \) in \( c \), we can compute the distance from \( p \) to \( a \) and \( b \) (i.e. \( \text{dist}(a, p) \) and \( \text{dist}(b, p) \)). Assuming the given range is 10, and \( \text{dist}(a, p) \) is 3 and \( \text{dist}(b, p) \) is 2, by looking up the interval
representations in Table 4.3, we can learn that the qualifying objects can be computed by \( \{O_1\} \cup \{O_3\} \), which is \( \{O_1, O_3\} \).

Because we only compute the distances between query points and exits, the CPU time for answering the continuous range queries are dramatically reduced. In later Chapters, we will show the performance of the IRD algorithm by comparing the CPU time with the IRNE algorithm.

### Table 4-1. Interval representation for accesspoints in Figure 4-8B

<table>
<thead>
<tr>
<th>accesspoints</th>
<th>interval representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( &lt; 0, 2 &gt; {O_1, O_2}; &lt; 2, 4 &gt; {O_1} )</td>
</tr>
<tr>
<td>( b )</td>
<td>( &lt; 0, 1 &gt; {O_3, O_4}; &lt; 1, 4 &gt; {O_3} )</td>
</tr>
</tbody>
</table>
For indoor route planning models the most important users are pedestrians. However, besides the pedestrians, there are other potential users, such as people in wheelchairs, small indoor autos, robots, and carts carrying products. In previous literature, several indoor navigation models have been proposed to support route planning in indoor spaces, but usually only for pedestrians. There have been a few models which are also designed for people in wheelchairs. However, to the best of our knowledge, none of them are able to provide a universal model that can support route planning for all kinds of users.

The models designed for pedestrians approximate the position of users by points. This approximation is not suitable for other kinds of users, as for these users, other information needs to be studied to check accessibility, and this is complicated than point-based models. Another problem is that most of the existing models ignore the obstacles that can be found within indoor spaces. There are many potential obstacles for indoor spaces, such as tables, chairs, decorative plants, and steps that are used to connect layers in the same floor. Those obstacles will affect the amount of indoor spaces that is accessible, especially when the user has a large size. In addition, most of the existing models are 2D-based models. Without the data of the third dimension, these models are not able to handle the structure of the entire space. As a consequence, they cannot check the accessibility from different dimensions.

We have proposed a LEGO model [104, 105] that can provide feasible routes for different types of users for indoor spaces. Our solution consists of a two-phase method that includes a representation phase followed by an accessibility checking phase. In the first phase, a LEGO-based representation model is proposed to represent the 3D structure of the indoor space. The entire space is approximated by several LEGO cubes with three main categories: plane_cubes representing planes, stair_cubes representing
stairs and obstacle_cubes representing obstacles. By using these cubes the entire indoor space can be represented in detail. In the second phase, the LEGO cubes are merged into blocks by using a novel strategy to obtain the maximum accessible widths, heights and lengths in different locations. These values are then used in order to evaluate the accessibility of different users. Finally, a LEGO graph recording all the accessibility information is built to apply efficient route search algorithms. The generated route consists of a sequence of blocks that are accessible by the user.

5.1 3D Representation of the Indoor Space

An indoor space is composed of different kinds of cells. Each cell unit is restricted by a lot of architectural constraints such as floors, walls and exits. The structure of a cell could be very complicated if the components of the cell have complex structures. For example, a pyramid ceiling may affect the accessibility of certain places inside the cell, and small stairs on the floor are only suitable for pedestrians. In this section, we will discuss our approach to representing the 3D structure of the indoor space.

5.1.1 LEGO-Based Approximation

Our representation model is inspired by LEGO's toy bricks. By using different shapes of the toy bricks, we are able to build various models of cell spaces. Thus, our approach to model 3D spaces in cells is to approximate the entire space by cubes. For simplicity, we only use cubes, called LEGO_cubes, with the same basic area as our representative units. Each LEGO_cube has its own height and type according to the object it represents. A LEGO_cube is non-dividable, and it can only represent one object or a part of an object. The 2D projection of all the cubes in one cell is a grid with equal sized squares. Figure 5-1 shows an example of a cube in a cube-shaped cell and two cubes with different heights in a pyramid-shaped cell. As shown in Figure 5-1B, the height of each cube depends on the distance between the floor and the ceiling in the corresponding area, which the cube locates. The size of the square controls the granularity of the representation. The basic area is the minimum unit for representing the
A B C D

Figure 5-1. Examples of cubes. A) A cube in a cell with regular shape. B) Cubes in a pyramid shaped cell. C) The 2D projection of a floor with rectangular shape. D) The 2D projection of a floor with triangular shape.

2D structure of the cells. The smaller the square is, the more the details of the indoor space can be represented.

Figure 5-1A is the simplest example of representing the indoor space by using LEGO_cubes. However, in most cases, the structures in cells are not so simple. Irregularly shaped floors and ceilings, obstacles, and the complicated shapes of cells will increase the difficulty of the representation. In our model, we classify different objects into three main categories: planes, stairs and obstacles.

The formal definition of the LEGO_cube is given in Definition 8.

**Definition 8.** A LEGO_cube $LC := (id, type, slope, w, h)$ is a basic 3D unit of representing objects in indoor space. $id$ is the unique identifier of the LEGO_cube. $type$ records the type of the LEGO_cube. The value of the $type$ can be obstacle_cube, stair_cube or plane_cube. $slope$ is the slope of a plane_cube. If the $type$ of the LEGO_cube is obstacle_cube or stair_cube, the slope value is always 1. $w$ is the width of the basal area of the LEGO_cube. $h$ is the height of the LEGO_cube.

5.1.1.1 The approximation of planes

When a floor and a ceiling are flat, and there is no obstacle between them, it is easy to fill out the available 3D space between them by using LEGO_cubes. As shown in Figures 5-1C and D, the 2D projection of the representation is a grid structure. Figure 5-1C is an example of a cell with a rectangular shape. When the boundary of
the cell is not horizontal or vertical the boundary is represented by multiple LEGO_cubes connected by the corners of the cubes' basic areas (as shown in Figure 5-1D). The resolution of the approximation depends on the size of the basic area. If we reduce the size of the basic area, the space can be represented more precisely.

When a floor or a ceiling is not flat, the available space can be approximated by using multiple LEGO_cubes with different heights. The vertical location of a cell's basic/top area depends on the average height of the corresponding area in the floor/ceiling. Sloping floors or ceilings are represented by a set of gradually ascending/descending LEGO_cubes. Figure 5-2A shows the longitudinal section of the representation of a sloping floor. Figure 5-2B shows a way to determine the vertical location of the basic area of such a LEGO_cube. Assuming the vertical locations of the end points of the sloping floor are 0m and 0.2m, then the vertical location of the basic area of the representative LEGO_cube is 0.1m, which is the average vertical location of the two end points. The approximation for a sloping ceiling is the same as a sloping floor. As shown in Figure 5-2C, the top areas of the LEGO_cubes are gradually descending according to the shape of the ceiling.

Sometimes, a plane may slope too much for some users to use. This plane is actually an obstacle that is unavailable for those users. In our model, we use a threshold for the slopes to control the availability of the planes. If the sloping of a plane
exceeds the threshold, this plane is considered as an obstacle and is represented by obstacle_cubes.

5.1.1.2 The approximation of stairs

Similar to the approximation of the sloping planes, a stair is represented by a set of ascending or descending LEGO_cubes. The width of the LEGO_cube cannot exceed the width of the corresponding steps. Thus, each step is represented by one or more LEGO_cubes. The vertical locations of the basic areas are determined by the locations of the corresponding steps. Figures 5-2D and 5-2E shows an example of the longitudinal section and the 3D diagram of the approximation.

5.1.1.3 The approximation of obstacles

Obstacles refer to the objects whose occupied areas are not available for users. They can be walls, tables, chairs, and other objects. When an obstacle lays on a floor, no matter how high it is, this area is considered as inaccessible for users in wheelchairs. However, it may be accessible for pedestrians. Usually, different obstacles have different shapes. However, their representations can be classified into three types.

The first type is the obstacles that are too high to be passed over. Typical examples are the walls, furniture such as tables, and some decorations such as potted flowers. Among these obstacles, some of them reach the ceiling and some of them still have free space above them. Since the free space above these obstacles cannot be used as passages, it is actually unavailable for users. Thus, the obstacles of this type are represented by LEGO_cubes whose top areas reach the ceilings. This means the space from the floor to the ceiling in this location is unavailable.

The second type is the obstacles that pedestrians can pass over. In our model, these obstacles are considered as curbs. They are accessible for pedestrians, but unavailable for users in wheelchairs to pass. These curbs are considered as small stairs and represented by stair_cubes.
The third type is obstacles in the air. The spaces below them are available for users. This type of obstacle will limit the height of the available space below them. The space from the floor to the ceiling in this area will be represented by two different LEGO_cubes. The bottom one represents the available space and the upper one represents the obstacle. The top area of the bottom cube will be the basic area of the upper cube.

Sometimes, different parts of an obstacle may belong to different types. For the obstacles with combined shapes, we first divide them into multiple parts according to the classification of the shapes, and then approximate them by using different strategies.

5.2 The Accessibility of Different Types of User

The goal in our paper is to provide feasible routes for all different users. The accessibility is affected by multiple things such as the walls, the exits and the obstacles inside rooms. We will introduce how we check the accessibility of the user's width, height, and length.

5.2.1 The Maximum Widths

In order to check the accessibility of the width, we have to find the maximum available width in all places. The maximum width in different places can be restricted by different obstacles. For example, Figure 5-3A is the 2D projection of a cell represented by LEGO_cubes. The white, black and grey cubes represent the available space, obstacles and stairs respectively. From the figure, we can learn that the maximum accessible horizontal width in the location of the cube a, b and c is the same. This maximum width can be obtained by merging the plane cubes in a horizontal direction until we meet an obstacle cube.

5.2.1.1 Blocks with the maximum widths

The cube-based representation introduced in Section 5.1 divides the entire indoor space into multiple LEGO_cubes. Each cube has its own unique type indicating whether it is part of a plane, a stair, or an obstacle. Usually, an object is represented by several connected LEGO_cubes. Thus, if a LEGO_cube is available for the user (e.g. a part
of a plane), its neighbor cubes may also be available for this user. By merging similar LEGO_cubes, we are able to find the maximum available width for a particular area.

As shown in Figure 5-3A, by merging nearby cubes, the maximum rectangle-shaped block containing the cube $a$ is the rectangle with the corner cubes of 1, 2, 3 and 4. In reality, the width of a block depends on the direction of the movements. If you move in a vertical direction, the width of this block contains 8 cubes, and if you move along the horizontal direction the width of the block becomes 3 cubes. The maximum blocks extended from one LEGO_cube may be different when used to merge directions. For example, the maximum block containing cube $c$ merged along the horizontal direction is the rectangle $(1, 2, 3, 4)$; while along the vertical direction, the maximum block is the rectangle $(1, 5, 6, 7)$. Therefore, in our model, there are usually two blocks extended from the same cube.

Before we introduce our merging strategy, we will first discuss several conditions involved in the merging process. Firstly, only LEGO_cubes of the same type can be merged together. Different types of LEGO_cubes represent different kinds of objects throughout the indoor space. It is impossible for us to walk from a plane into an obstacle, and for some users, stair cubes are not accessible to them. Secondly, although the heights of the LEGO_cubes reflect the height of the available space inside the cells, the accessible height of a cell is actually controlled by the height of the exits. Assuming the default height of one cell is the maximum height among all the exits of this cell, then
the LEGO_cubes higher than the default height can be accessed without restriction. Therefore, the LEGO_cubes higher than the default height can be merged together. For the LEGO_cubes lower than the default height, only the cubes with the same height can be merged together. Thirdly, the number of LEGO_cubes used to represent the cells will affect the efficiency of the merging process. If there are a lot of LEGO_cubes, the process may take a long time. We also observed that blocks merged from different cubes may be the same. For example, in Figure 5-3A, the block merged from the cube \( a \) and from the cube \( b \) is the same block. Thus, in order to avoid duplicates and improve the merging process, we can reduce the number of starting cubes. As shown in Figure 5-3A, from any randomly selected cube, the maximum width we can obtain always depends on the boundary cubes, which are next to other types of cubes. Thus, we only choose the boundary LEGO_cubes to be the starting cubes in order to perform the merging.

Our merging process in the horizontal direction starts from a boundary cube. The cubes that are horizontally extended from this starting cube are merged one by one to obtain the maximum width. As shown in Figure 5-3B, the cubes which are horizontally next to cube 1 are merged one by one until we reach the boundary. This step produces a temporary block with the maximum width obtained from the starting cube (e.g., the rectangle \((1, 2, 3)\)). Then, all the cubes directly above the temporary block are checked to see if they have the same type and satisfy the height condition. If so, they will be merged into the temporary block. For example, the cubes 4, 5, 6 that are directly above the rectangle \((1, 2, 3)\) can be merged into the rectangle \((1, 2, 3)\). If one of the cubes does not satisfy this condition, the cube extension in this direction will stop. Therefore, the cubes above \((4, 5, 6)\) cannot be merged. The cubes under \((1, 2, 3)\) are merged by using the same strategy. The final block generated from cube 1 is the rectangle \((4, 6, 18, 16)\). If we apply the algorithm along the vertical direction, the rectangle \((10, 16)\) will
be the corresponding block extended from cube 1 that contains the maximum vertical width.

The result of the merging process is a set of blocks. There are three relationships between any two blocks. The first relationship is disjointed. For example, in Figure 5-3B, the block (10, 16) vertically extended from cube 1 and the block (12, 18) vertically extended from cube 3 are disjointed. The second relation is adjacent. Two adjacent blocks share a part of the boundary (e.g. the block (19, 20) horizontally extended from cube 19, and the block (21, 22, 23, 24) horizontally extended from cube 22 are adjacent). Two blocks can also be overlapped. In Figure 5-3B, the block (10, 16) and the block (4, 6, 18, 16) are overlapping. However, one block cannot be fully inside another block. This is because if one block is fully inside another block, it can also be extended to form the outer block by merging with nearby cubes. It is also impossible to have two blocks with relationships similar to Figure 5-4D, because the boundary \(a\) of the block \(B\) in Figure 5-4D can be extended to meet the boundary \(b\) of the block \(A\).

### 5.2.1.2 The improvement of the merging

Although this merging strategy can generate blocks with the maximum widths in the horizontal and vertical directions in cells, it may produce unnecessary blocks in some situations. For example, the irregular obstacle separating two available spaces in Figure 5-3C leads to irregular boundary cubes. According to the merging strategy, the block extended from the boundary cubes will be a set of adjacent blocks shown by the bold lines.

However, compared to the slashed area, the small cubes near the boundary do not usually contribute to the available space. Therefore, for simplicity and efficiency, when the boundary between an obstacle and an available space has a zigzag shape, this boundary can be simplified into a straight line before performing the merging.
5.2.1.3 The algorithm

Algorithm 5 shows how we merge the cubes to generate blocks with the maximum width in a horizontal direction in one room. The input is a matrix, $cubes[n][m]$, that records all the cubes in one room. From the bottom left cube, $cubes[0][0]$, we check the cubes on its right side and find the first cube, $c_1$, with a PLANE type (see line 4-5). From $c_1$, we repeatedly add the cubes with PLANE type on the right side until we hit a cube with non-PLANE type (see line 6-10).

At this point, the maximum horizontal width of $c_1$ is found. In line 11-20, we check the types of the cubes moving upwards. From the line of the cubes that have been found in Algorithm 5, line 6-10, we check the whole line of the cubes above it. If they are PLANE cubes, we add the whole line into the block. Similarly, line 21-30 is used to check the cubes going downwards.

5.2.1.4 Connectors between blocks

Once we generate the blocks, the routes from the starting place to the users’ destination can be considered as the block-by-block paths. The connections between two blocks (called connectors) control the maximum accessible width and height between one block and the next. There are two types of connector, one is used to connect two blocks with the same type, and the other is used to connect two blocks which are of different types (e.g., the connectors between planes and stairs).

**Connectors Between Two Blocks with the Same Type**

According to Section 5.2.1.1, there are three relationships between two blocks, disjointed, adjacent and overlapping. When the two blocks are adjacent or overlap, users can move from one block to the other if their connector is wide enough.

When two blocks are adjacent, the connector between them is the intersected boundary. For example, the line $a$ is the connector between block $A$ and $B$ in Figure 5-4A.
Algorithm 5: CubeMerge

**Input**: Matrix `cubes[n][m]`

**Output**: list array `blocks`

1. \( ki\_down \leftarrow 0; \) \( ki\_up \leftarrow 0; \) \( kj\_down \leftarrow 0; \) \( kj\_up \leftarrow 0; \)
2. for \( i \leftarrow 1 \) to \( n \) do
3.     for \( j \leftarrow 1 \) to \( m \) do
4.         while `cubes[j][i].getType()` !\(=\) PLANE do
5.             \( i++; \)
6.         if \( i \neq m \) then
7.             \( ki\_down = i; \)
8.             while \((i < m) \) and \( (cubes[j][i].getType()) == \) PLANE) do
9.                 \( i++; \)
10.            \( ki\_up = i - 1; \)
11.        boolean \( flag = \) TRUE;
12.        for \( jj \leftarrow j \) to \( 0 \) do
13.            for \( ii \leftarrow ki\_down \) to \( ki\_up \) do
14.                if `cubes[j][ii].getType()` !\(=\) PLANE then
15.                    \( flag = FALSE; \)
16.            if \( flag \) == FALSE then
17.                \( kj\_down = jj + 1; \)
18.            else
19.                if \( jj \) == \( 0 \) then
20.                    \( kj\_down = jj; \)
21.        \( flag = TRUE; \)
22.    for \( jj \leftarrow j \) to \( n \) do
23.        for \( ii \leftarrow ki\_down \) to \( ki\_up \) do
24.            if `cubes[j][ii].getType()` !\(=\) PLANE then
25.                \( flag = FALSE; \)
26.        if \( flag \) == FALSE then
27.            \( kj\_up = jj - 1; \)
28.        else
29.            if \( jj \) == \( (n - 1) \) then
30.                \( kj\_up = jj; \)
31.            \( ki\_down = cubes[0][0].getX() + cubes[0][0].getLength() * \) \( ki\_down; \)
32.            \( ki\_up = cubes[0][0].getX() + cubes[0][0].getLength() * (ki\_up + 1); \)
33.            \( kj\_down = cubes[0][0].getY() + cubes[0][0].getLength() * \) \( kj\_down; \)
34.            \( kj\_up = cubes[0][0].getY() + cubes[0][0].getLength() * (kj\_up + 1); \)
35.            Rectangle \( b = \) new Rectangle ( \( ki\_down, ki\_up, kj\_down, kj\_up \) );
36.            `blocks.add(b);`
When two blocks are overlapping, their boundaries will always be crisscrossed as shown in Figure 5-4B and C. The connector between two overlapping blocks is the diagonal of the intersected rectangle. As shown in Figure 5-4B and C, the line $a$ is the maximum connector between block $A$ and $B$.

**Connectors Between Available Spaces with Different Types**

The connectors between two blocks with the same type can reflect the accessible widths between two areas. However, when trying to find the maximum accessible widths between two different types of available areas (e.g., planes and stairs), the blocks generated by applying the merging strategy may produce incorrect maximum connectors, especially when the boundaries are not horizontal or vertical.

Figure 5-5 shows some examples of boundaries between a plane and a stair. The white part is the available plane and the grey part represents the area of the stair. The irregular lines and the black cubes denote the accessible and inaccessible boundaries between the plane and the stair respectively. The rectangular blocks bounded by bold lines in the plane are the blocks generated by applying the merging strategy on the boundary cubes. The maximum accessible widths between the plane and the stair for Figures 5-5A, b and c are denoted by the dashed lines. From these figures we can see that none of the blocks can capture the maximum accessible width between the plane and the stair. Therefore, we are unable to obtain the maximum accessible width between these two areas.
Figure 5-5. The connectors with the maximum width between a plane and a stair cannot be found by using the basic merging Algorithm.

These incorrect results are caused by the merging process. The merging process will stop when a cube with another type is met. If the boundary between a plane and a stair is not horizontal or vertical, this process will not be able to generate a block containing the entire boundary between the two available spaces. In order to capture the maximum accessible width between two different available areas, we have to generate blocks containing as much of the boundary as possible.

Our approach to solving this problem is to generate blocks by merging plane_cubes and stair_cubes together. This means that if a stair_cube (plane_cube) is met with merging plane_cubes (stair_cubes), this stair_cube (plane_cube) is also merged into the block. As shown in Figure 5-6A, B is the block created by merging both the plane_cubes and the stair_cubes. The block A denoted by the dashed lines is the minimum bounding box of the accessible boundary. The connector between A and B is the diagonal c, which reflects the maximum accessible width between the plane and the stair.

This approach also works when the boundary is affected by obstacles. As shown in Figure 5-6B, the rectangle (1, 2, 3, 4), (1, 8, 6, 5) and (8, 7) demonstrates the blocks produced by applying the new approach. The connectors a, b, and c correctly reflect the maximum accessible widths in different directions in this scenario.

The definition of a connector is given in Definition 9.

**Definition 9.** A connector \( C = (loc_1, loc_2, height) \) is a list of connected cubes representing the maximum accessible width and height when moving from one block to the other.
Figure 5-6. Computing connectors between a plane and a stair. A) Without the effect of obstacles. B) With the effect of an obstacle

\( \text{loc}_1 \) and \( \text{loc}_2 \) are the center points of the two end cubes in this connector. \( \text{height} \) is the minimum height of the two blocks sharing this connector.

5.2.2 The Maximum Heights

The accessible height is the second condition we have to check. Although the heights of LEGO cubes can reflect the heights of the available space inside cells, the accessible height of a cell is actually controlled by the heights of the exits. Assuming that the default height of one cell is the maximum height among all its exits, any area higher than the default height can be accessed without restriction. Therefore, in our merging process, the LEGO cubes higher than the default height will be merged together. For the LEGO cubes lower than the default height, only the cubes of the same height can be merged together.

This ensures that all the cubes in one block either have the same height or are higher than the default height. In the case of the former, the height of the cubes is the height of the block, and in the latter case, the default height becomes the height of the block. For each pair of adjacent or overlapping blocks, the maximum accessible height is the minimum height of the two blocks.

5.2.3 The Maximum Lengths

The process of checking accessible lengths is very complicated. Turns, obstacles, and user’s sizes all have an impact on the maximum accessible length (examples are shown in Figures 5-7A, B and Figure 5-8A). In our model, we propose an approach to
provide users with a feasible way to their destinations. The generated route may not be optimal, but it is guaranteed to be feasible.

There are several reasons why we cannot provide the optimal routes. Firstly, the shapes of the users may be different. It is hard to check the availability for every part of the object in all places. For example, in Figure 5-7C, the rectangle cannot pass through these obstacles. However, in Figure 5-7D, although object A shares the same rectangular shape, it is able to go through the path. Even if we can find a way to check the accessibility for every part, it is inefficient and unpractical to do so. Secondly, in a real world scenario, users may prefer to use more comfortable ways rather than the optimal one. For example, it will not be a realistic route if users have to make several attempts to find the right angle to make a turn. Thus, it is better to provide a route with enough space.

As we know, the movements of different users’ can be vary. However, if we deconstruct the movements into small steps, they can be classified into two main categories: going straight and making turns. To check the accessibility of the length for a straight path, we only need to compare the length of the object with the length of the straight path. However, checking the accessibility for turns is difficult.

In our paper, we will first introduce an approach that can check the accessible lengths for all general cases for an indoor space. Then we will refine the approach for special cases.
5.2.3.1 The approach for general cases

In reality, no matter how sophisticated the final path is, it is always composed of segments and turns. Therefore, instead of considering the whole environment, we will focus on checking the accessibility of one turn for the object.

Let’s start from a simple case. Figure 5-8A shows a typical corridor turn. One important observation is that if the minimum bounding circle of the object can be contained in the turning corner (the grey area), this object is able to make the turn. This corner concept can be applied to our LEGO representation model. In Section 5.2.1, we discussed ways in which to find the maximum accessible widths by merging cubes into larger blocks. Any two blocks may be disjointed, adjacent or overlapping (as shown in Figure 5-4). Users may only need to make turns when they are going from one block to another through the corresponding connector.

**Lemma 4.** Given an object $O$ and a corner area that can be represented by a rectangle $A$, the object can access this corner if the minimum bounding circle of $O$ can be contained in $A$.

**Proof.** As shown in Figure 5-8A, the polygon $A$ represents a corner area and $O$ represents the object. $d$ is the diameter of the minimum bounding circle of $O$. It is clear that if the minimum bounding circle can be contained in $A$, the diameter of the circle $d$ must be less than the width and length of the corner area $A$. Therefore, the object $O$ can access the corner $A$. □

The scenario for overlapping blocks (as shown in Figure 5-4C) is similar to the typical corridor turns. If the minimum bounding circle of the object can be contained in the overlap area, this object is able to make the turn into this connector. However, for the corner in the typical corridor turn, the shape and the size is restricted by the walls, while for the overlapped area of two blocks, its boundary may not be restricted by obstacles. One possible solution is to extend its boundary to find the maximum corner areas.
As shown in Figure 5-8C, blocks $A$ and $B$ are generated according to the layout of the obstacles. In this scenario, we can extend the boundary of the overlapping area to form a larger accessible space, as indicated by the dashed lines in Figure 5-8C. In fact, this extension process is unnecessary; because the merging process of the LEGO model guarantees that any block with the maximum accessible space will be generated. Therefore, any possible overlap areas can be captured by our model. For example, the area indicated by the dashed boundary in Figure 5-8C is the overlap area of the block $C$ and $D$. We can find the maximum alternative corner area of $A$ and $B$ by looking for the largest corner area that contains the current one.

Figure 5-4B shows another kind of overlap relationship between two blocks. For this kind of scenario, according to the locations of other connectors, users may or may not have to make turns. Taking Figure 5-8E as an example, $A$ and $B$ are two adjacent blocks, and $c$ is the connector between $A$ and $B$. Assuming $a$ and $b$ are two connectors connecting $A$ and other blocks, if one user goes from $a$ to $c$, she can go straight to $B$. However, if she goes from $b$ to $c$, then she will probably need to take a turn. Our approach to handle this scenario is to find the maximum overlap area for the two blocks. If the minimum bounding circle of the object can be contained in the maximum overlap area, the object can go through the two blocks without any problem. For example, in Figure 5-8E the grey part is the overlap area between $A$ and $B$. According to the location of the obstacles, this area can be extended to the area indicated by the dashed lines. In figure 5-8F, since the minimum bounding circle of the object $O$ can be contained in this extended overlap area, $O$ can successfully go from $A$ to $B$.

The situation of the two adjacent blocks shown in Figure 5-4A is similar to the overlapping blocks in Figure 5-4B. As shown in Figure 5-8G, $A$ and $B$ are two adjacent blocks, and $c$ is the connector between $A$ and $B$. Assuming $a$ and $b$ are two connectors connecting $A$ with other blocks, if one user goes from $a$ to $c$, she can go straight to $B$. However, if she goes from $b$ to $c$, then she will probably need to take a turn. To simplify
Figure 5-8. Demonstrations of checking accessibility in different scenarios

the two cases, we have observed that if we extend the connector $c$ to form a larger block (as shown in Figure 5-8H), the scenario becomes the same as the overlapping blocks we discussed before. Therefore, we can apply the same strategy to check the accessibility of the adjacent blocks.

As an extension of the minimum bounding circle approach, the maximum inner circle of the object can also be used to check the non-accessibility of the turns. If the maximum inner circle of one object cannot be contained in the corner area, this object definitely cannot access this corner.

**Lemma 5.** Given an object $O$ and a corner area that can be represented by a rectangle $A$, the object cannot access this corner if the maximum inner circle of $O$ cannot be contained in $A$.

**Proof.** As shown in Figure 5-8B, the polygon $A$ represents a corner area, $O$ represents the object and $C$ is the maximum inner circle. It is clear that if the smallest object whose maximum inner circle is $C$, cannot access the corner $A$, all the other objects that have this maximum inner circle cannot access $A$.

The smallest object whose maximum inner circle is $C$ is actually the circle $C$. Since $C$ cannot be contained in the corner area $A$, it definitely cannot access $A$. Therefore, any
objects whose maximum inner circle are the same as or greater than $C$ cannot access $A$.  

5.2.3.2 The approach for special cases

This minimum bounding circle approach makes sure that the provided routes are feasible for users. However, this approach is not precise enough. For some particular scenarios, we are able to refine the accessibility check.

For indoor spaces, corridor corners are one of the most common areas in which users may have problems in successfully passing through. For the traditional $90^\circ$ corners, we have developed an approach to refine the accessibility checking. As shown in Figure 5-9A, the rectangle $(a, b, c, d)$ is the minimum bounding box of an object (user). The segment $(o, a)$ is parallel to one side of the corner, and $(o, a)$ has the same width of $w$. We have noticed that if the length of $(o, b)$ equals or is less than $v$, then this rectangle can make the turn. Point $b$ has the same situation. In Figure 5-9B, the length of $(o, b)$ is equal to $v$. If the length of $(o, a)$ is equal or is less than $w$ then this rectangle can make the turn.

Therefore, our approach contains two steps to check the accessibility of a $90^\circ$ corner. Firstly, the minimum bounding rectangle (MBR) of the user is constructed (e.g. $(a, b, c, d)$ in Figure 5-9A). Secondly, assuming the boundaries $(a, b)$ and $(c, d)$ are longer than $(a, d)$ and $(b, c)$, we find a point $o$ on the boundary $(c, d)$ so that the length of $(a, o)$ is equal to the width of one side of the corner (e.g., $w$ in Figure 5-9A). If the length of $(b, o)$ equals or less than the width of the other side of the corner (e.g., $v$ in Figure 5-9A), then the user can successfully make the turn.

We can perform the second step in another way, in which we try to find a point $Q$ so that the length of $(b, Q)$ is equal to the width of one side of the corner. If the length of $(a, Q)$ equals or less than the other side of the corner, the user can make the turn. Otherwise, this corner is not feasible for the user.
Figure 5-9. Refinement the approach for evaluating the maximum length for 90° corners.

Figure 5-10. The LEGO graph. A) A floor plane with obstacles and stairs. B) The graph reflecting the connectivity of the blocks. C) The corresponding LEGO graph

5.3 The LEGO Graph

Most of the existing path searching algorithms (e.g., the shortest path search and the A* algorithm) are graph-based algorithms. In this section, we will discuss how to build a graph to support route searching algorithms.

As discussed in previous sections, the indoor space is approximated by LEGO cubes, which are further merged to form larger blocks. Users can walk block by block
in order to reach their target. The accessible widths, heights and lengths are restricted by these blocks and the connectors between them. In order to support the accessibility checks, this information must be stored in the graph. One solution is to build a graph in which nodes denote blocks and edges represent connectors.

Figure 5-10B is such a graph and consists of all the blocks and connectors for the scenario shown in Figure 5-10A. One big problem of this graph is that the distance of each path is stored in nodes instead of edges. Therefore, it is difficult to apply the shortest path algorithms.

In reality, the process of walking block by block is the same as the process of walking connector by connector. A better solution is to build a graph in which the nodes denote all the connectors, and edges represent their distances. In our model, this kind of graph is called a LEGO graph. Definition 10 is the formal definition of the LEGO graph.

**Definition 10.** A LEGO graph \( LG = (V, E) \) is a graph which reflects all possible paths with different accessible widths, heights and lengths in a given indoor space scenario. \( V \) is a set of connectors with the information of the supportable lengths \(<L>\). \( E \) is a set of implicit paths in the format of \(<N(D,W,H,T)>\), where \( N \) is the name of the edge, \( D \) is the distance between two connected nodes, \( W \), and \( H \) are the maximum accessible width and heights. \( T \) is the type of the edges, which can be plane, obstacle or stair.

The values attached to each edge are determined as follows:

- **D**: The length of an edge in a LEGO graph is the distance between the center points of the two end nodes.
- **W**: The accessible width of an edge depends on the maximum widths of the two end nodes. It will be set to be the minimum width of the two nodes.
- **H**: As discussed in previous sections, our generated blocks are always rectangles, and the connectors are either on the boundary or inside the block. Thus, the path between two connectors is always inside the corresponding block. The accessible height of an edge is the height of the block.
- **T**: Since each edge is inside one block, there is only one type for each edge. For example, if the cubes in one block are all plane, the path is plane.
• $L$: The accessible length is maintained in nodes, which is the diameter of the maximum circle introduced in Section 5.2.3. The reason we don’t check the length in edges is because if the minimum bounding circle of the user can be contained in the extended connecting area, there must be enough space for the user’s length to fit.
CHAPTER 6
THE IMPLEMENTATION

This section contains two parts: the evaluation of the efficiency for the 2D model which supports the shortest paths route planning and the demonstration of our implemented iNav system. The evaluation part shows the elapsed times that are spent for constructing all path segments in different scenarios, and the average efficiency for computing a route between two randomly selected locations. The demonstration part presents our implementation of the iNav system, including the 2D and the 3D system. The 2D system supports the shortest path search and range queries, and the 3D system provides the feasible routes for different types of users.

In reality, the navigation system includes three parts: the determination of the user’s current location, the process of the navigation queries, and the provision of understandable results. Based on the assumption that the user’s current location can be obtained with the use of other equipment, our system focuses on processing the queries and visualizing the results. The capturing of the user’s current locations are simulated by clicking the mouse to record the position.

6.1 The Evaluation

In previous sections, we have introduced our solution for supporting the shortest route planning in an indoor space. This solution is composed of two separate phases: 1) data loading and path-segment construction, 2) route planning. The goal of our experiments is to evaluate the performance of the whole solution. Especially, the experiments are designed to answer two questions: 1) Is it practical to construct all the path segments in a building with a complicated structure? 2) Is it efficient to discover the routes in a large-scale graph that is generated in the first phase?

**Experiment Setup:** Our experiments are conducted on a laptop with Genuine Intel(R) CPU, 1.3GHZ and 4.0GB RAM. The operating system is Microsoft Windows 7, and the program runs on Java 2 Standard Edition SDK v1.6.0. To evaluate the
performance of our solution, we measure the time cost during three procedures: 1) loading the data into databases, 2) constructing path segments, 3) and planning the routes.

**Data Set:** We collect the floor maps from Shands at UF [1], which is a national recognized hospital. This hospital resides in a 7-floor building with complex topology. To evaluate the efficiency of our algorithm, we choose the most two complicated floor maps as our data set. Since the door information is missing, we randomly choose door location in each room. Figure 6-7A illustrates the map of floor 1. In order to simulate the environments containing multiple floors, all the maps of the floor \( n \geq 2 \) are the same, as shown in Figure 6-7B. All these maps are manually extracted and represented in XML as the data sets. Figure 6-7C shows the statistics information of the generated data sets. When the number of floors is 100, the number of cells (i.e., rooms) and access points reach 2695 and 7972, respectively.

**Experiment 1** evaluates the performance during the first phase (data loading and path-segment construction). In our solution, this phase is a pre-processing procedure for generating a Direct Path Graph, whose nodes represent access points and edges.
Figure 6-2. Evaluation of the performance for route planning. A) The time spent for loading data sets. B) The time spent for path construction.

Experiment 2 examines the efficiency of route planning between a source point and a target object based on the generated Direct Path graph. One goal of the second
phase is to find a shortest path from a source point $S$ (i.e., a particular point on a map) to a target object $T$ (e.g., a room). The histogram shown in Figure 6-3 demonstrates the time spent in route planning for 10000 samples. Each sample corresponds to a node pair $\langle S, T \rangle$, which are randomly selected from a 100-floor map. After pruning the nodes and edges whose corresponding floor is different from the one of either $S$ or $T$, our solution can generate the expected path in 65.36 milliseconds in average. The maximum execution time is 167 milliseconds and 94.55% route planning takes less than 70 milliseconds. The experiments in a real-world scenario show the potential of our solution.

**Experiment 3** evaluates the efficiency of our proposed algorithm for stationary range queries, which aims at finding the shortest paths for all the qualifying objects within a specific distance range from the source point. Figure 6-4 illustrates the time spent on the range queries with different range values. For each value, the time cost is an average of 100 range-query execution time, where each source point is randomly selected. We observe that the time cost linearly increases from 55.92 milliseconds for the range of 100 meters to 237.27 milliseconds for the range of 1000 meters. These results clearly show our solution is practical and efficient in a real-world application.

**Experiment 4** examines the performance of the algorithm for computing continuous range queries. In our experiment, 100 query points are randomly selected within a cell.
Figure 6-5. Evaluation comparison between IRNE and IRD. A) Number of dynamic query points is 100. B) Number of dynamic query points is 1000

Then, for the range values from 100 meters to 1000 meters, we compute the qualifying objects for all the query points by applying the algorithm for computing stationary range queries (IRNE) and continuous range queries (IRD). The results shown in Figure 6-5A indicates that the IRD algorithm is more efficient than IRNE. The advantage of the algorithm IRD will be more obvious when the user is passing through a larger cell. In the next experiment, we repeat the same tests but use a larger number of query points to indicate that the user is passing through a large cell. Figure 6-5B illustrates that, with the larger number of query points, the time spent for IRD is almost the same as the previous experiment (shown in Figure 6-5A), while the time for IRNE is dramatically increased.

### 6.2 The Demonstration

There are two systems implemented. One is used to support the shortest route planning and range queries in a 2D space, and the other is used to provide route planning for different types of user in a 3D space. The two systems share the same architecture as shown in Figure 6-6.

#### 6.2.1 System Architecture

The architecture consists of five components: User Interface, Data Processor, Query Processor, Result Processor, and Storage/Retrieval Manager. User Interface provides a tool for users to upload the formatted data and check the visualized paths.
Figure 6-6. The architecture of the iNav system

for the correct directions. *Data Processor* analyzes the data and creates implicit path segments. The storage and retrieval of the data are managed by the *Storage/Retrieval Manager* component. Once the database is ready, *Query Processor* will process the user's query, and *Result Processor* will generate the descriptions and the visualized results for the user.

### 6.2.1.1 The data processor

Data Processor is used to analyze the input data, create the implicit path segments, and store all the information into the database through the Storage/Retrieval Manager. The input data should be an XML file which specifies the structure of the building.

The XSD file which specifies the format of the XML data for the 2D system is shown in Appendix A.

There are four types of objects used to describe the indoor environment: *room*, *corridor*, *stair* and *elevator*. Rooms and corridors can be simple cells, complex cells,
or open cells, depending on their structures. Stairs and elevators are two types of connectors. An example of a simple cell is shown as follows:

```xml
<room>
  <name>502</name>
  <parent>0</parent>
  <child>0</child>
  <tag>office</tag>
  <floor>5</floor>
  <door>
    <name>502</name>
    <coordinateX>25</coordinateX>
    <coordinateY>29</coordinateY>
    <availableslot>8:00-17:00</availableslot>
  </door>
  <shape>
    <wall>
      <startX>25</startX>
      <startY>0</startY>
      <endX>40</endX>
      <endY>0</endY>
    </wall>
    <wall>
      <startX>40</startX>
      <startY>0</startY>
      <endX>40</endX>
      <endY>30</endY>
    </wall>
    <wall>
      <startX>40</startX>
      <startY>0</startY>
      <endX>40</endX>
      <endY>30</endY>
    </wall>
    <wall>
      <startX>40</startX>
      <startY>0</startY>
      <endX>40</endX>
      <endY>30</endY>
    </wall>
  </shape>
</room>
```
The tag *door* specifies the location and the available time of an access point in the cell, and the tag *shape* records the boundary of the cell. If the cell is a complex one, there will be multiple doors associated with that cell in the file. If the shape of the cell is a region with holes (not another cell having access points), the *innerboundary* tag should be used to describe the shapes of the holes. The tag *parent* and *child* are used to indicate the containment relationships between different cells. After the file is uploaded to the system, *Data Analyzer* will analyze the file, and *Path Creator* will create the implicit path segments.

Once the data is ready, Storage/Retrieval Manager will store the structure and the created path segments into the database. In our implementation, the PostGreSQL database is used to store all the data, and the postGIS, which adds support for geographic objects to the PostgreSQL, is used to handle the spatial features of the data. Thus, access points, path segments and shapes of cells are stored as the types of *POINT, LINE* and *POLYGON* in the database. After the operation of the data processor users are able to view the map of the building through the User Interface. Figure 6-7 is the user interface of the iNav system. The center of the interface shows the map of
the building. If the building has multiple floors, each floor will be shown in a separate tab page. Users are able to view the maps of different floors by switching the tabs. As shown in Figure 6-7, the 5th floor is composed of one complex cell (room 502), several simple cells, and a corridor with a hole (the center region without a name in it).

The input data for the 3D system is more complex than that used for the 2D system. Our system is based on cube representation. However, it is impractical for users to provide the cube-based structure of data to the system. There are several approaches which can be used to store 3D structures. One such approach is to represent 3D objects by recording their boundaries (e.g. Doubly-Connected Edge List (DCEL) [18], Quad-Edge [24], and Triangular Irregular Network (TIN) [97]). The other option is to deconstruct 3D objects into small parts and organize the small parts using hierarchical structures, such as the Octree [57], and the Constructive Solid Geometry (CSG) tree. In our model, the boundary-based approach is used to record the structure of the indoor space.

Figure 6-7. The user interface
According to the most representative structure of the indoor space, each cell is deconstructed into five main components: floor areas, ceiling areas, walls, doors and obstacles. Obstacles are further broken down into top areas, bottom areas and surrounding faces. All the areas and faces can be represented by a set of polygons. Since all objects follow a standard structure, they can be represented by a formalized XML format. Appendix B shows an example of the input XML data that represents the structure of the cell in Figure 5-1A.

6.2.1.2 The query processor

The 2D iNav system supports two varieties of query: routing queries and range queries. Users can initiate routing queries through the User Interface by indicating their nearest access points and their targets (as shown in the right side of Figure 6-7). They can also post range queries by indicating their nearest access points, along with their preferred ranges and interests (as shown in the dialog form of Figure 6-7). The iNav system will retrieve all the implicit path segments from the database and construct the direct path graph.

The posted queries will be processed by applying the Dijkstra’s algorithm or the IRNE algorithm proposed in Chapter 4. In the 2D iNav system, if users want to post continuous queries they can show their movements by clicking the mouse on the map. The result produced by the Query Processor is a sequence of nodes indicating the routes.

The 3D system supports route planning for arbitrary-shaped users. Similar to the 2D system, users can initiate their queries by indicating their nearest exits, their targets, widths, heights and lengths. The 3D system will retrieve all the connectors and build the LEGO graph. Dijkstra’s algorithm will be applied to generate the final path which is composed of a sequence of connectors between blocks.

In order to make results more understandable, the query results will be further processed by the Result Processor.
6.2.1.3 The result processor

The Result Processor in the 2D iNav system is used to visualize the results produced by the Query Processor and produce the descriptions of the routes as discussed in Section 3.4. It first analyzes the results obtained from the Query Processor and retrieves all the necessary information related to the results from the database. Then it determines the locations of the nodes and draws the path according to the sequence of the nodes. Figure 6-8 shows the visualized route from room 502 to room 515, in which the red nodes are the involved access points and intermediate points. The Result Processor also generates the description of this route and displays the description at the bottom of the interface. By using the visualized map and the description of the route, the user is able to get to the destination in the shortest distance.

In our 3D model the paths are composed of blocks and connectors that are fit for the users’ size. In order to show the paths more clearly, the result visualization in our 3D system contains two parts: showing the lines connecting the different connectors in sequence, and highlighting the blocks involved in the path. As shown in Figure 6-13, the path from d22 to room6 is indicated by the lines from d22 to d3, and from d3 to d6. Besides the lines, there are some blue areas indicating the blocks involved in the generated path.

6.2.2 Experiments

6.2.2.1 Experiments of route planing in 2D indoor spaces

Example 1. Find the shortest path from room 502 to room 615.

Figure 6-9 is the visualized result of the route. Figure 6-9A shows the path on the 5th floor and Figure 6-9B shows the path on the 6th floor. Figure 6-9C displays the changes to the route according to the user’s current location, as indicated by clicking the mouse (the green node in Figure 6-9C).

We obtain the description of the route as follows:

Start from door 502_1 to door 615
Example 2. *Find paths from outside to nested cells.*
Figure 6-10. The results of Example 2. A) The generated route from room 1 to room 6. B) The generated route from room 1 to room 4. C) The generated route from room 1 to room 7.

In Figure 6-10, room 2 is a nested cell containing room 3, room 5 and room 6. Inside room 3, there is another cell, room 4. Figures 6-10A, b and c are three examples of route planning in buildings with nested cells. The first one is to test the paths to the first level nested cells (the cells whose parent cell is not nested in other cells). The result of this case is shown in Figure 6-10A. The second one is to test the paths to deeper level nested cells (the cells whose parent is also nested in other cells). The result of the test is shown in Figure 6-10B. The third test, which is illustrated by Figure 6-10C, is to test paths that are from the cells in one side of the nested cell to the cells in the other side of the same cell, going through the entire nested cell.

**Example 3.** *Find paths from nested cells to outside cells.*

Similar to Example 2, we use three cases to test the paths from nested cells to cells outside. The first one is to test paths from deeper level nested cells (cells in nested cells) to the first level nested cells (shown in Figure 6-11A). The second example, illustrated in Figure 6-11B, is to test the paths from deeper level nested cells to outer cells. The third example tests the paths from the first level nested cells to the outer cells. Figure 6-11C shows the result of this case.

**6.2.2.2 Experiments of range query in 2D indoor spaces**

**Example 4.** *Find all the offices within 30 meters of the door of room517.*
Figure 6-11. The results of Example 3. A) The generated route from room 4 to room 2. B) The generated route from room 4 to room 7. C) The generated route from room 6 to room 7.

Figure 6-12 is used to test the results of range queries and continuous range queries. Assuming all the rooms are marked as offices, Figure 6-12A shows the paths to all offices within 30 meters of the door of the room504. The blue nodes are the qualifying offices, and the red lines are the paths from the start node to the corresponding targets. From Figure 6-12B we can learn that there are 7 qualifying offices in the 5th floor and 3 more in the 6th floor. The red node between 517 and 518 is the connector of the 5th floor and the 6th floor. The result changes with the movements of the user. As shown in Figure 6-12C, when the user goes to the location indicated by the red node, the offices within 30 meters of the red node change accordingly.

6.2.2.3 Experiments of route planning for arbitrary-shaped users

**Example 5.** Check the accessibility of doors.

The goal of Experiment 5 is to check the accessibility of doors. In Figure 6-13, $d6$ is 50 inches, and all other doors are 100 inches. One user is in the position of $d22$ and wants to go into room6. If the user's width is less than 50 inches, she can go into room6 by entering door $d6$ (as shown in Figure 6-13A). However, if the user's width is greater than 50 inches, she has to go into room5 first (as shown in Figure 6-13B).

**Example 6.** Check the accessibility between blocks.
Figure 6-12. The results of Example 4. A) The offices within 30 meters from the room 517 on the 5th floor. B) The offices within 30 meters from the room 517 on the 6th floor. C) The qualified offices changes according to the position of the mouse clicking.

Figure 6-13. The results of Example 5. A) The width of the user is 30 inches. B) The width of the user is 80 inches.

The goal of Experiment 6 is to check the correctness of the generated paths when there are obstacles blocking the way. As shown in Figure 6-14, there are two obstacles in room3. The longer one is 50 inches from the three sides of the walls. Figure 6-14a and Figure 6-14b shows two different paths from d22 to room5. The path in Figure 6-14A is for the users whose widths or lengths are less than 50 inches, and the path in Figure 6-14B is for the users whose widths or lengths are greater than 50 inches. The red circles are either doors or the center points of the involved connectors.
Figure 6-14. The result of Example 6. A) The length of the user is 30 inches. B) The length of the user is 80 inches.

Figure 6-14c and Figure 6-14d illustrate the paths when more blocks are involved. In Figure 6-14A and B, there are only two blocks involved in the generated path in room3; while in Figure 6-14C, the number of involved blocks becomes 4. The generated path from d22 to room4 are shown in Figure 6-14C, when the user’s width and length are less than 50 inches, and the path for users wider or longer than 50 inches is shown in Figure 6-14D.
CHAPTER 7
CONCLUSIONS

This dissertation presents a spatial model for supporting route planning in indoor spaces. My major research contributions include the following three aspects. First, the iNav model (described in Chapter 3) provides an effective solution to the discovery of the shortest path between the source point and the target object. Second, my proposed range-driven routing approach (discussed in Chapter 4) is able to efficiently determine all the qualifying objects within a specific walking distance. Third, based on the LEGO model (presented in Chapter 5), the generation of a feasible route can be fully automated from the source point to the target object, when the size of a moving object cannot be approximated to a point.

The presented solution makes a significant contribution to the research of design of indoor navigation systems, especially for addressing the three fundamental route planning issues (i.e., distance-driven, range-driven and size-driven route planning). Although indoor navigation is our major application scenario, our proposed solution can be applied in the other related systems, such as, shopping guide systems, recommendation systems for designing architectural structures, and emergency evacuation systems. In the industrial community, much effort has been devoted to designing and implementing outdoor route planning systems (e.g., in GPS navigation systems). With increasing requests for indoor navigation, we believe the indoor route planning is indispensable in the next generation of navigation systems.

The following interesting directions are suggested for future research topics:

**Semantics-based route planning** is to integrate semantic information into the geometric-based route planning. In a real-world scenario, we observed there exist various semantic resources, such as user profiles, road marks, and iconic objects. Through effective reasoning on these resources, indoor route-planning systems can further improve the accuracy of route recommendation.
**Automation of map interpretation** is a critical step in the productization of our proposed indoor route planning prototype. As major sources of indoor structure, building maps are normally drawn in some widely-used industrial software, such as *AutoCAD* and *ProE*. Thus, it is highly desirable to automate map interpretation for resolving such an information-transformation issue.
<xml version="1.0"?><xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com" elementFormDefault="qualified">
  <xs:element name="building">
    <xs:complexType mixed="true">
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="room"/>
        <xs:element ref="corridor"/>
        <xs:element ref="stair"/>
        <xs:element ref="elevator"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="room">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="parent" type="xs:string"/>
        <xs:element name="child" type="xs:string"/>
        <xs:element name="tag" type="xs:string"/>
        <xs:element name="floor" type="xs:int"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" ref="door"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="innerboundary">
          <xs:complexType>
            <xs:sequence>
              <xs:element maxOccurs="unbounded" ref="wall"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
<xs:element name="corridor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="tag" type="xs:string"/>
            <xs:element name="floor" type="xs:int"/>
            <xs:element maxOccurs="unbounded" minOccurs="0" ref="door"/>
            <xs:element maxOccurs="unbounded" minOccurs="0" name="innerboundary">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element maxOccurs="unbounded" ref="wall"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:element name="shape">
    <xs:complexType>
        <xs:sequence>
            <xs:element maxOccurs="unbounded" ref="wall"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:sequence>
  <xs:element maxOccurs="unbounded" ref="wall"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="stair">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="tag" type="xs:string"/>
      <xs:element maxOccurs="unbounded" ref="door"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="elevator">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="tag" type="xs:string"/>
      <xs:element maxOccurs="unbounded" ref="door"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="door">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="tag" type="xs:string"/>
      <xs:element maxOccurs="unbounded" ref="door"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="name" type="xs:string"/>
<xs:element name="coordinateX" type="xs:double"/>
<xs:element name="coordinateY" type="xs:double"/>
<xs:element name="floor" minOccurs="0" type="xs:int"/>
<xs:element name="availableslot" maxOccurs="unbounded" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="wall">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="startX" type="xs:double"/>
      <xs:element name="startY" type="xs:double"/>
      <xs:element name="endX" type="xs:double"/>
      <xs:element name="endY" type="xs:double"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
APPENDIX B
AN EXAMPLE OF THE FORMATTED DATA FOR THE 3D SYSTEM

<room>
  <name>room1</name>
  <tag>office</tag>
  <level>1</level>
  <door>
    <name>d1</name>
    <level>1</level>
    <bottom>(500,200,0) (500,300,0)</bottom>
    <top>(500,200,200) (500,300,200)</top>
    <availableslot>8:00-17:00</availableslot>
  </door>
</room>

<floor>
  <area>(0,0,0) (500,0,0) (500,500,0) (0,500,0)</area>
</floor>

<ceiling>
  <area>(0,0,200) (500,0,200) (500,500,200) (0,500,200)</area>
</ceiling>

<face>
  <area>(0,0,0) (0,0,200) (0,500,200) (0,500,0)</area>
  <area>(0,0,0) (0,200) (500,0,200) (500,0,0)</area>
  <area>(500,0,0) (500,200) (500,500,200) (500,500,0)</area>
  <area>(500,500,0) (500,500,200) (0,500,200) (0,500,200)</area>
</face>

<obstacle>
  <name>Table</name>
  <bottom>
    <area>(650,50,0) (750,50,0) (750,450,0) (650,450,0)</area>
  </bottom>
</obstacle>
REFERENCES


BIOGRAPHICAL SKETCH

Wenjie Yuan has received her M.S. and B.S. from the Nanjing University of Science & Engineering in China in 2006 and 2004. She started her study in the University of Florida in 2006. Her research focuses on spatial modeling of indoor navigation systems, especially on the design of data structures for database storage and navigation strategies for way finding.