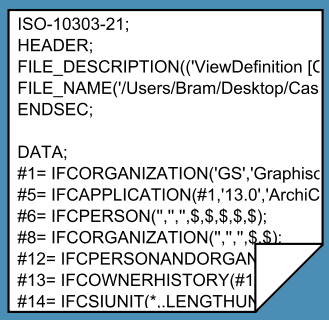
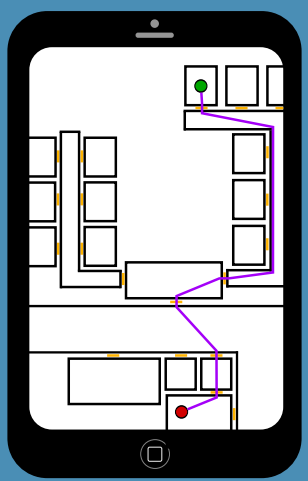
The Journey from IFC Files to Indoor Navigation



IFC FILE · EXTRACTION · INTERPRETATION · INDOOR SPACE MODEL · DECOMPOSITION · ROUTING · INTRA-PARTITION DISTANCES · INDOOR NAVIGATION



Master Thesis
Software Engineering
Department of Computer Science
Aalborg University

Group sw105f13
Christian de Haas
Mikkel Boysen
Supervisor: Hua Lu

Summary

This project continues the work done in [7], where an indoor space model was defined and a prototype system, to extract and interpret indoor data from IFC files, was developed. The indoor space model defined in [7] captures connectivity and accessibility, including doors, elevators, stairs, and rooms in all shapes. Based on this model, extraction and interpretation processes were implemented in the prototype, which would make it possible to autonomously interpret relationships between indoor elements. The prototype was developed with a UI to view and edit the extracted indoor data, such that the data could be used to create the indoor space model of the building represented in the IFC file from which the data had been extracted.

The topic for this project is to bridge the gap between IFC files and indoor navigation, using the indoor space model and prototype developed in the preceding project. Through analysis of the prototype developed in [7], several shortcomings were found, which led to the improvements and extensions made in this project. For the data extraction and interpretation part, this includes a correct implementation for extracting building elements, and a generic mapping approach that is not limited to handle partitions with specific shapes. The prototype database is updated to utilise spatial data types, queries, and indices. The UI has been improved and extended with a functionality to connect partitions to the outdoor area in order to obtain a complete topology of the indoor space. Furthermore, a functionality to insert access points between partitions has also been added to the UI. Several techniques, to improve data management, has been added, including decomposition of partitions, which has been implemented by refining an existing approach to facilitate application on real world data.

To prove that indoor navigation can be made possible, through use of the data extracted by the prototype, an app with routing functionality has been developed. The routing algorithm used to implement this functionality, makes use of intra-partition distances between access points. In order to calculate these distances, a novel solution is proposed. Through experimental evaluation, this solution is proved to provide better results than calculating these distances by applying Dijkstra's algorithm.

Aalborg University
Department of Computer Science

Title:
A Journey from IFC Files
to Indoor Navigation

Topic:
Database Technology

Semester:
Software Engineering
10^{th} semester
Feb. 1^{st}, 2012 - June 7^{th}, 2013

Group:
sw105f13

Members:
Mikkel Boysen
Christian de Haas

Supervisor:
Hua Lu

Copies: 4

Report - pages: 63

Appendices: 5

Total pages: 74

Abstract:

This report documents a project based on developing a prototype as a solution to extract and manage indoor data from IFC files and demonstrate the use of it in a navigation app.
The project continues the work done in [7], in terms of both improvements and extensions. Completion of the prototype back-end is done by altering the extraction process to remove found errors, enhancing the mapping procedure to provide significantly better results, harness the advantages of using a spatial database, and adding useful features to the back-end UI. Additionally, a partition decomposition algorithm is refined and implemented to avoid dead space in the R-tree used by a spatial index. A front-end has been developed as an app with indoor navigation features. This includes the development of a routing functionality and a novel method to calculate intra-partition distances, which is evaluated through experimental tests.

Preface

This report is the documentation of a Master Thesis project, made by group sw105f13, at the Department of Computer Science, Aalborg University. The project started at the 1st of February 2013, and ended at the 7th of June 2013.
Conventions used in the report are as follows:

- Citations are indicated by square brackets.
- Abbreviations appear in their extended form in the first use and short in the following appearances. Some abbreviations are deemed so conventional that presenting them in their extended form is unnecessary.
- In order to improve the readability of the report, pronouns are used irrespectively of gender. For instance, "he" refers to he/she.
- The terms "computation" and "calculation" are used interchangeably.
- The term "line segment" is used for a line that is bounded by to distinct points and has the length equal to the Euclidean distance between those points.
- The term "polyline" is used for a continuous line composed of one or more line segments.
- References to the "prototype system" or "prototype" refers to the whole system, including the back-end and front-end.

Appendices are attached at the end of the report and the digital version of the report will be available for other students on the AAU digital library: `http://projekter.aau.dk/projekter/`

Acknowledgements are given to the following people:

- Hua Lu for supervising the group during the project.

Signatures

Mikkel Boysen	Christian de Haas

Contents

Chapter 1

Introduction

The need for indoor navigation systems is increasingly being observed and emphasized [9], [2], [4]. Navigation inside indoor spaces is helpful for people that are not familiar with the structure of the building or are in a hurry to get from one place to another, e.g., from check-in to a specific gate at an airport. Such an indoor navigation system requires an appropriate model that represents the indoor space topology. On the other hand, indoor spaces are described in architecture industry formats like the Industry Foundation Classes (IFC) where the geometric representation for doors and rooms is the focus but indoor topology is only implicit or even incomplete.

This report presents a prototype system that is intended to bridge the gap between the widely used industry standard IFC and practical indoor navigation systems. The prototype extends previous work, documented in [7]. This includes improvements to existing features, new designs and techniques for spatial data management, and navigation features that utilise the indoor space data extracted by the prototype.

The report includes the following chapters: A project analysis to recapitulate the extend of the work done previously on the prototype and to explore and elaborate possible modifications and extensions. A project definition to state the problem defined by this project, including delimitations. As a solution to this problem, the prototype is explained in detail, with emphasis on the modifications and extensions made in this project. The implementation and evaluation of an indoor routing algorithm is presented. Finally, the development of an app, which utilises the indoor data and the routing algorithm, is presented as the link between indoor navigation and the data extracted by the prototype.

Chapter 2

Project Analysis

This chapter contains an analysis of the project domain, and begins with a brief recapitulation of the work done in [7]. Based on this, possible modifications and extensions to the prototype system are described.

2.1 Previous Work

In the project analysis of [7, p. 15-16], the idea for an indoor navigation solution is created, by researching existing navigation solutions created for outdoor spaces, to form a basis for the project development. Described briefly, the idea is to develop a prototype to extract data from digital representations of buildings, creating indoor space models to be used for navigational purposes. The back-end developed in [7] concerns extraction of data, from IFC files, needed to create an indoor space model of the building represented by the file. The indoor space model for this purpose is also proposed in [7]. Delimitations were made to strengthen the focus of the project, discarding 3D support, restricting the supported building representation file format to IFC, and choosing Java as the programming language for the prototype system.

The indoor space model in [7, p. 24-28] includes partitions, access points, and connectors. This makes it possible to represent connectivity between partitions on the same floor and different floors. The model also defines directional accessibility rules, which represents accessibility. Furthermore, it supports calculation of distances between access points inside partitions, which in turn can be used for shortest-path routing.

The development of the back-end in [7] required a thorough analysis of the IFC file structure, in order to extract building elements correctly. Implementation of topological mappings, e.g. between partitions and access points, was found necessary, since such relationships are not explicitly available in the IFC file.

The design used to create the database was purely relational, and was

based on the hierarchical structure of IFC files and the indoor space model.

An evaluation of the extraction and interpretation process was performed to highlight possible problems with performance, number of entities extracted, different representations of IFC elements, and mapping of access points to partitions. Only the mapping process showed significant flaws, and sources of error were pointed out.

2.2 Possible Modifications & Extensions

Based on the work of [7], the general idea of an indoor navigation system still holds for this project. It can, however, be narrowed down to create a more concrete problem statement. To do this, the possible development directions for this project are reviewed in this section.

As concluded through the evaluation of the back-end system in [7, p. 57-58], modifications can be made to increase the accuracy of the mapping functionality. Furthermore, several parts in other areas of the prototype can be improved or extended to enhance the overall quality.

In this section, the most relevant of the possible modifications and extensions are revised shortly, in order to provide an overview and a basis for choosing among them.

2.2.1 Mapping

The implementation of the mapping functionality is limited to handle access points on partition edges parallel with the x- and y-axis. Furthermore, mappings of more than two partitions to the same access points can occur with the implementation made in [7]. An improvement to this functionality is to make it support edges with any direction and to ensure that no more than two partitions can be mapped to an access point.

2.2.2 Enhanced Data Extraction

During the evaluation of the back-end, several sources of error were found in the data extraction and interpretation process. These include inverted partitions, partitions with odd directions, and flaws in the IFC file. To improve the data extraction, the first two of the sources of error should be eliminated. The last source of error, concerning flaws in the IFC, is outside the scope of what is handled by the back-end and should be eliminated in the creation or maintenance of the IFC file.

2.2.3 Spatial Database

The data extracted through the back-end system is stored in a PostgreSQL relational database. This makes it possible to maintain a hierarchical struc-

ture of indoor entities and to maintain topological relationships. However, spatial data is also extracted, such as polygons to represent the shape of a partition, and such data can be stored in a more efficient way by using a spatial database. To support spatial data, the PostGIS spatial extension can be applied to the PostgreSQL database. This will also give access to various spatial queries, which can improve performance of the interpretation process, as mentioned in [7, p. 62], as well as future implementations.

2.2.4 Decomposition of Partitions

If the spatial extension PostGIS is used, spatial indices are created using an R-tree, which utilises the minimum bounding rectangles (MBRs) of the geometric shapes stored in the database. The shape of a partition varies and while some might be almost rectangular, others can be short in one dimension and long in the other, or even be concave. As such, a high amount of dead space can occur in the MBR of a tree node in the R-tree, degrading the query performance of the tree.

Reducing the amount of dead space can be considered a general improvement to any system using a spatial database. As proposed by [5], this can be handled by decomposing partitions with irregular regions into sub-partitions with smaller, regular regions. However, the decomposition algorithm proposed in [5] has several shortcomings, which means that solutions to these must be found in order to implement decomposition of partitions.

2.2.5 Indoor Distance Support

Indoor distances can be utilised in a routing functionality, using shortest-route calculation, and with an indoor navigation app, this feature is evident to be implemented. Although the support for indoor distances is included in the indoor space model, it is only described briefly in [7, p. 26-28]. A thorough and detailed description of how indoor distances are computed is necessary if indoor routing is to be implemented.

2.2.6 Indoor Navigation App

As mentioned in [7, p. 29-47], the development of a smartphone app, as a front-end for the prototype, requires that all the data needed by such an app is provided by the back-end. As such, this app would also be a valuable extension to the prototype system in that it would demonstrate the use of the data generated by the back-end. Together with back-end part of the prototype it would bridge the gap between IFC files and indoor navigation, while visualising a contextual use of the data.

2.2.7 Positioning & Tracking

If an indoor navigation app is developed, it would be desirable to be able to position an object in the indoor space and possibly track it in case it moves over time. Several solutions to this have been proposed, which are described in [7, p. 19-20]. These solutions are based on different technologies such as WiFi, Bluetooth, RFID, or a hybrid of these.

2.2.8 Back-end UI

The back-end user interface (UI) makes it possible for a user to use the data management features available, in order to view and edit the extracted data. However, as stated in the delimitation in [7, p. 22], development of the UI was limited to provide a proof of concept, rather than a detailed, fully developed UI.

In order to increase the usability of the back-end UI, various improvements could be made to the graphical part of the UI, as well as redesigning the user workflow for each feature. As stated in [7, p. 63], issues were encountered when designing some of the back-end UI features which enables the user to edit the extracted data. A thorough analysis of these issues could result in solutions, that, together with the previously mentioned general improvements to the UI, would create a basis for implementing these features.

2.2.9 3D Support

As mentioned in [7, p. 17-20], modelling the indoor space in 3D can be useful in some cases. It would be easier to manage and represent obstacles and elevated areas in the indoor space if the prototype system is extended to support 3D. 3D support is especially useful if the system to be developed is meant for disabled people as their movement patterns are much dependent on obstacles and elevated areas.

2.2.10 Object-Relational Mapping

As described in [7, p. 62-63], implementation of an object-relational mapping (ORM) framework would enhance the prototype system in terms of extensibility and persistency. Future work that includes changes made directly to the database or database transactions, would be easier to perform.

Chapter 3

Project Definition

Based on the overview of possible modifications and extensions to the project prototype, stated in Section 2.2, the project definition is specified and described in this chapter. As a result of the analysis of the work done in the preceding project, possible improvements were found in different areas of the prototype. Regarding the extraction and interpretation process, some building elements are not extracted correctly and a more complete implementation of the mapping between building elements can be made. Furthermore, the back-end UI has room for improvement in terms of both functionality and usability. To empirically show the usefulness of the data generated by the back-end of the prototype, an app can be developed to use the data by providing navigation features. The importance of such an addition to the prototype is evident, in that it displays the purpose of the data extraction process.

3.1 Problem Definition

As previously mentioned, this project is a continuation of [7]. A recapitulation of the work done previously to this project has been made and based on this, the problem definition is:

To finish the development of the prototype system, such that it produces the data necessary to generate an indoor space model of a given building representation. Furthermore, the prototype system should incorporate a front-end that utilises this data through indoor navigation features.

3.2 Project Delimitation

To tighten the focus of this project, a project delimitation is made. The modifications and extensions described in 2.2 are revised and those that do

not directly contribute to solve the problem of the project, or are considered outside the project scope, are excluded.

Although an implementation of an ORM framework would increase the quality of the code in various ways, it is considered to be too far from the project scope. Such an implementation would require significant work to be done, in terms of changes to the database communication wrapper, and is therefore left for future improvements, in order to concentrate the workload of this project on the implementations necessary to solve the problem.

Making the prototype support 3D representation of an indoor space would be an interesting feature, as it would give a different, and possibly better, user experience, especially in regard to the navigation app, compared to a 2D representation. It is, however, not prioritised to be within the scope of this project as it would not add significant value in terms of new functionality over a 2D representation in the prototype system.

Indoor positioning and tracking is considered an interesting direction for this project. However, the effort required to implement such a feature is considered to be enough to create another project solely based on this feature. Additionally, numerous research papers exist in this field, and to contribute with a novel solution would require thorough analysis and research.

Elevators and stairs in IFC files are, as mentioned in [7, p. 32-33], defined for representation and not for capturing the relationship between the floors they connect. Through an analysis of several different IFC files, no general representation has been found for either stairs nor elevators. This hinders the implementation of these building elements in the extraction process. An alternative solution, to capture this information, is to enable users to add it through the back-end UI. However, this solution is expected to be a comprehensive task if it is to include a reasonable level of usability. The development of such a solution is not within the project scope, and as a result, stairs and elevators, and thereby connectors in the indoor space model, have been excluded in the following parts of the development in this project: The data extraction process of the back-end because of the aforementioned reasons and the navigation functionalities of the front-end because it uses, and is tested with, the extracted data.

3.3 Contributions

The premises for this project is the work done in [7], which can be summarised to the following:

- An indoor space model, capturing indoor topology and accessibility
- A back-end system to extract and interpret indoor space data from IFC files, based on the indoor space model
- A back-end UI to view and manage the extracted data

To solve the problem stated in Section 3.1, this project aims to contribute with the following:

- Improvements to existing back-end features
  - The mapping functionality (Section 2.2.1)
  - Data extraction and interpretation (Section 2.2.2)
  - Utilisation of spatial data types and queries (Section 2.2.3)
  - The data management UI (Section 2.2.8)
- New back-end features
  - Decomposition of partitions to avoid irregular or imbalanced shapes that could degrade query performance using spatial indices (Section 2.2.4)
  - Calculation of indoor distances to support shortest-path routing in the navigation app (Section 2.2.5)
- Development of a navigation app that utilises the extracted indoor data (Section 2.2.6)

3.4 Architectural overview

An overview of the basic architecture of the prototype is illustrated in Figure 3.1. Communication between the back-end and front-end parts of the prototype system is to be handled through a client-server architecture, which is described in Section 7.1.

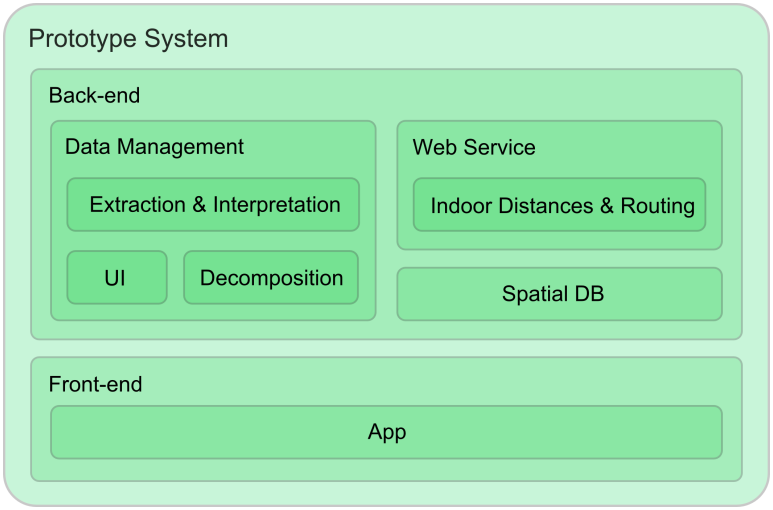


Figure 3.1: *An overview of the prototype system architecture.*

Chapter 4

Decomposition

As described in Section 2.2.4, decomposition of irregular or imbalanced partitions is done to reduce the amount of dead space in the MBRs of the polygons representing the partitions, since dead space can degrade the query performance in the R-tree index of the spatial database.

This chapter describes the decomposition algorithm proposed in [5] and its shortcomings. Solutions are proposed to these and a refined decomposition algorithm is given. In addition, two requisite data cleansing steps are introduced. Finally, a description of how data persistence is assured during decomposition is given.

Partition decomposition affects several parts of the prototype system, and the description of it is therefore given first to avoid reference and readability issues.

4.1 Original Decomposition Algorithm

The implementation of decomposition is based on the work of [5], which presents the pseudo algorithm for decomposition seen in Algorithm 3 in Appendix B. To decompose concave partitions, turning points are used. A turning point in a partition is a point that creates an internal angle greater than 180 degrees. When a concave partition is decomposed, it is split by a line segment drawn perpendicular to the longer dimension of the partition through the turning point closest to the middle of the partition. This is done recursively until no more turning points exist. Convex partitions are decomposed if the lengths of their dimensions are imbalanced, i.e. the ratio between the width and height exceeds a predefined threshold, T_{shape}. Here, the partition is split by a line segment drawn perpendicular to the longer dimension through the middle point on that dimension. This is also done recursively until the partition is balanced.

Through analysis and testing, a number of shortcomings has been found in the original decomposition algorithm. This has led to a redesign of the

algorithm before its implementation in the prototype. The modifications are described in the following sections.

4.2 Recalculation of Turning Points

In Algorithm 3, the set of turning points is given as a parameter in the function. When decomposing using turning points, the set of turning points is updated by removing the turning point on which the split was made. This way of managing turning points can cause problems, which is illustrated by the examples in Figure 4.1. The first problem is shown in Figure 4.1a. In this example, the partition is decomposed into the sub-partitions A and B through turning point tp_1. When sub-partition A is examined to find out if it should be decomposed or not, the set of turning points will contain turning point tp_2 which is incorrect as tp_2 is not a turning point of sub-partition A.

Another problem is shown in Figure 4.1b. In this example, the partition is also decomposed into sub-partitions A and B through turning point tp_1. Here, the problem appears after the decomposition, where tp_2 no longer is a turning point, as it has been eliminated as a result of the partition split. However, according to the algorithm, only tp_1 is removed as it was the turning point which the split was made through. To avoid these problems, the set of turning points needs to be recalculated for each decomposition that involves turning points.

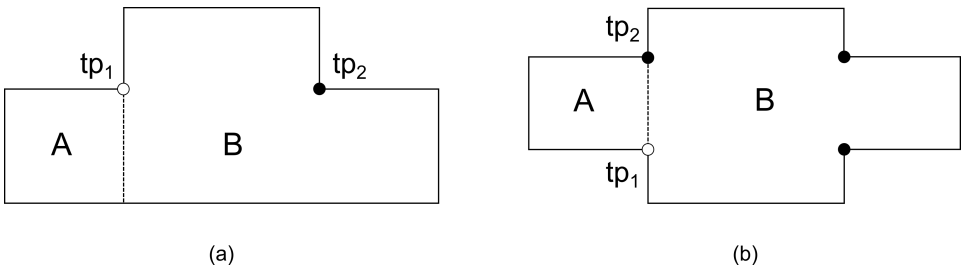


Figure 4.1: *Two examples of partitions where recalculation of turning points is necessary.*

4.3 Dead Space Threshold

After testing the decomposition algorithm with real world data, special scenarios, where decomposition could be considered inappropriate, were discovered. Two of these examples can seen in Figure 4.2. In Figure 4.2a, the decomposition creates many small sub-partitions. In Figure 4.2b, the decomposition creates a very imbalanced partition, i.e. a partition with significantly longer width than height. Further decomposition of this sub-partition involves multiple dimensional splits, again resulting in many small

sub-partitions. The amount of small sub-partitions created scales inversely proportional with the percentage of dead space in the MBR, which, in practice, can cause memory problems. Considering the small amount of dead space removed and the large amount of small sub-partitions created, decomposition on these kinds of partitions is considered unnecessary. Therefore, a threshold, $T_{deadspace}$, is introduced to ensure that partitions are not decomposed unless they have a minimum predefined percentage of dead space in their MBRs.

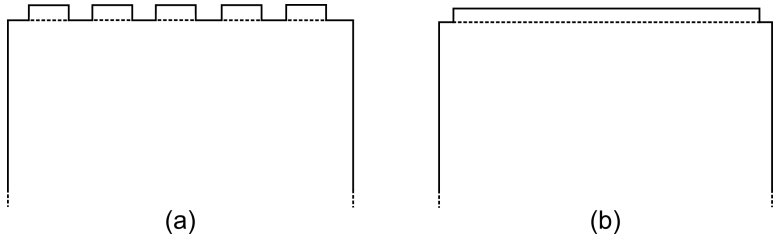


Figure 4.2: *Two examples of partitions where decomposition is inappropriate.*

4.4 Best Split on Turning Point

In the original algorithm, splitting the partition using turning points is performed perpendicular to the longer dimension. However, this will not always result in a split where the sub-partitions created are most balanced, i.e. the ratio between the width and height is closest to equal. Sometimes, a split made perpendicular to the shorter dimension can result in a more balanced split than a split made perpendicular to the longer dimension.

An example of this can be seen in Figure 4.3. The two scenarios shown in the figure are identical apart from how the split is performed. As it can be seen, the height of the partition exceeds the width and the split should, according to the original algorithm, be performed perpendicular to the height, as illustrated in Figure 4.3a. This, however, creates a large and a small sub-partition, where the small partition is imbalanced. If the split is performed perpendicular to the width of the partition, as seen in Figure 4.3b, more balanced and evenly sized sub-partitions are created. As such, the solution to this problem is to find and use the split which creates the most balanced sub-partitions.

4.5 Alignment on Dimension Split

Decomposition of imbalanced convex partitions works as intended in the original algorithm. However, when introducing the dead space threshold, described in Section 4.3, there is a possibility that the partition still contains turning points and therefore is concave. This can cause problems when

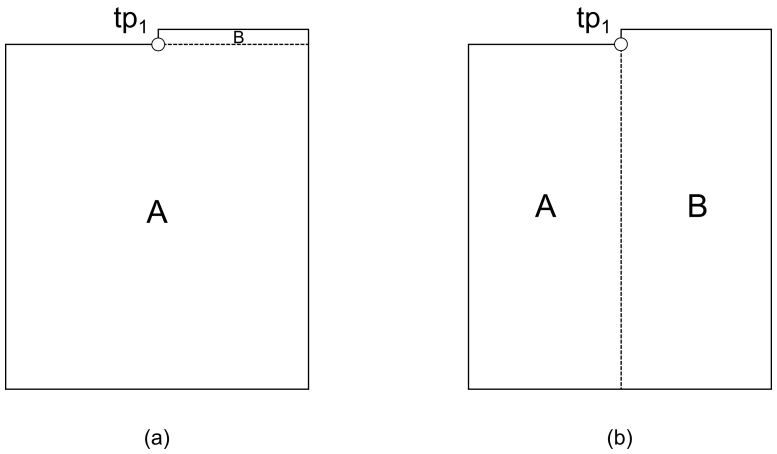


Figure 4.3: *An example of a partition split performed on either dimensions.*

making the dimensional split, i.e. the split perpendicular to the longer dimension through the middle point on that dimension. An example of this is shown in Figure 4.4, where the partition is decomposed into two sub-partitions A and B. By examining the partition before it is decomposed, it can be seen that it does contain a turning point. However, assuming that the percentage of dead space does not exceed the threshold, the partition is not decomposed using turning points. After the dimensional split, the percentage of dead space in sub-partition A may exceed the threshold, allowing for decomposition using the turning point. Using the best split solution, described in Section 4.4, would create a small sub-partition in the upper right corner of sub-partition A. This small sub-partition becomes more imbalanced the closer the dimensional splitting line is to the turning point, possibly resulting in multiple dimensional splits, i.e. the same problem as described in Section 4.3.

The solution to this problem is to search the partition for turning points when making the dimensional split. If any turning point in the partition is within the distance of a predefined threshold, T_{align}, the splitting line is aligned with the turning point, eliminating the aforementioned problem.

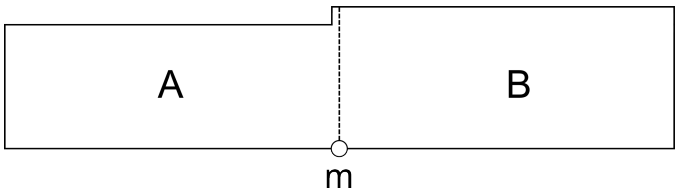


Figure 4.4: *An example of a dimensional split that results in an imbalanced sub-partition.*

4.6 Refined Decomposition Algorithm

The solutions described in Section 4.2 - 4.5 have been used to refine the original decomposition algorithm, and the result can be seen in Algorithm 1. The highlighted line numbers in the algorithm mark the parts of the algorithm which have been in refined in this project.

The refined algorithm takes as input the partition region r and three threshold values: T_{shape}, $T_{deadspace}$, and T_{align}. First, in Line 2, the set of turning points is found for r. This is an improvement to the original algorithm, as explained in Section 4.2. Next, in Line 3, a check is made to see if r is concave and the amount of dead space in r exceeds $T_{deadspace}$, as described in Section 4.3. If that is the case, r is decomposed using turning points in Line 4-9. Here, the turning point t closest to the middle of r is used to create two splitting lines. The splitting line that results in the best split, described in Section 4.4, is used to divide r into two or more regions, $\{r_i\}$. The decomposition algorithm is then run recursively on each of those regions.

If the check in Line 3 is not valid, another check is performed to determine if r is imbalanced, as seen in Line 12. If that is the case, r is decomposed using dimensional split, which can be seen in Line 13-22. Here, the middle point m on r's longer dimension is used to create a splitting line s. Then s is aligned with any nearby turning point using the threshold T_{align}, as described in Section 4.5. Finally, r is divided into two or more regions, $\{r_i\}$, and the decomposition algorithm is run recursively on each of those regions.

Algorithm 1 Refined Decomposition

1: **function** REFINED DECOMPOSE(region r, threshold T_{shape}, $T_{deadspace}$, T_{align})
2: find set of turning points P for r;
3: **if** r is concave and deadspace(r) $> T_{deadspace}$ **then**
4: select a *turning point* $t \in P$ on r's boundary, such that t is closer to the middle of r;
5: create two splitting lines, one for both dimensions, to find the best split;
6: use the best splitting line to divide r into two or more regions: $\{r_i\}$;
7: **for** each r_i in $\{r_i\}$ **do**
8: `Decompose`(r_i, T_{shape}, $T_{deadspace}$, T_{align});
9: **end for**
10: **else**
11: let $R(r)$ be the **MBR** of r;
12: **if** $\frac{min(len(R(r)_1, len(R(r)_2)}{max(len(R(r)_1, len(R(r)_2))} < T_{shape}$ **then**
13: find the middle point m on r's longer dimension d;
14: create a splitting line s perpendicular to d through m;
15: **if** a *turning point* $t \in P$ is within T_{align} of s **then**
16: align s to cut through t;
17: **end if**
18: use s to divide r into two or more regions: $\{r_i\}$;
19: **for** each r_i in $\{r_i\}$ **do**
20: `Decompose`(r_i, T_{shape}, $T_{deadspace}$, T_{align});
21: **end for**
22: **end if**
23: **end if**
24: **end function**

4.7 Data Cleansing

Besides the modification made to the decomposition algorithm, two data cleaning steps have been implemented to run before the decomposition. Real data testing resolved in irregularities with some polygon representations of partitions, which led to inappropriate decompositions. By performing the data cleansing, the geometric representation becomes less precise. However, this is a reasonable trade-off when generating data for the indoor space model. Furthermore, if such precision is required, e.g. for a graphical representation of the indoor space, the original geometric data can be stored separately.

4.7.1 Proximity Points

The first data cleansing step is removal of proximity points, i.e. points that are in proximity to each other. The reason for introducing this cleansing step originates from experience with partitions that contain small irregularities. Whether these irregularities are intentional or design flaws is unknown, but for the purpose of indoor navigation, they are considered inappropriate to include in the data, as they do not alter the shape of a partition significantly nor serve any purpose to any navigation features. Apart from being unnecessary in general, the data makes decomposition of such partitions complicated and often results in inappropriate decompositions.

An example of a partition that contains proximity points can be seen in Figure 4.5. The process of removing these points is as follows: Go through each point of the polygon and remove a point if it is within a predefined distance of the previous point.

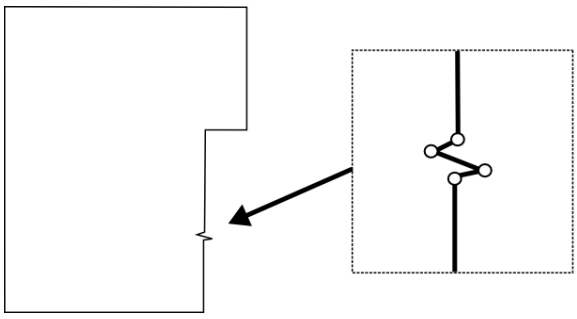


Figure 4.5: *An example of a partition that contains proximity points.*

4.7.2 Grid alignment

The second data cleansing step is grid alignment. The reason for implementing this step originates from experience with partitions, where some points have almost identical x- or y-values. An example of such a partition can be

seen in Figure 4.6, where turning points tp_1 and tp_2 have almost identical y-values. However, when drawing the splitting line from tp_1, the problem described in Section 4.3 and 4.5 occurs, eventually resulting in many small sub-partitions. The solution to this problem is to align points of a partition with each other if they exists within a predefined threshold to each other in one dimension.

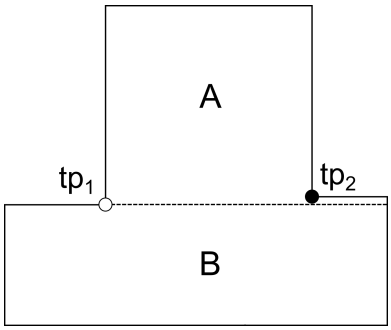


Figure 4.6: *An example of a partition where grid alignment is necessary.*

4.8 Decomposition & Data Persistence

Subsequent to decomposing a partition, each sub-partition created is stored and mapped to the original partition. Additionally, every relationship between any access point and the original partition must be used to create similar relationships for the sub-partitions created to ensure persistent data. This is done by reconnecting each access point, connected to the original partition, to the sub-partition closest to the access point.

Furthermore, the connectivity between sub-partitions must be captured. This is done by creating *virtual* access points between any two sub-partitions that touch each other, i.e. have intersecting walls. The geometrical representation of a *virtual* access point is a line segment with the length of the mutual part of the intersecting walls. An example of creating virtual access points after a decomposition is shown in Figure 4.7, where the two dashed line segments represent the virtual access points created between sub-partition A and B, and B and C.

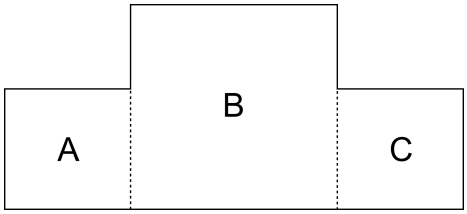


Figure 4.7: *An example of sub-partitions connected by virtual access points.*

Chapter 5

Back-end Improvements

This chapter describes the enhancements made to the existing features in the back-end of the prototype system. This includes improvements to the database, the data extraction and interpretation, and the data management and representation UI.

5.1 Database

In this project, several modifications have been made to the database. These include the requirements set by the implementation of partition decomposition, a redesign to represent the relationship between partitions and access points in the indoor space model, and a spatial extension to utilise spatial data types and queries. These changes have led to a new ER-diagram, which is illustrated in Figure 5.1. The database and ER-diagram changes are explained in the following sections.

5.1.1 Decomposed Partitions

The implementation of partition decomposition, described in Chapter 4, requires the ability to store sub-partitions including their relationship with their original partition. This is done by representing sub-partitions as *Partition* entities and introducing a link table to represent the one-to-many relationship between an original partition and any number of sub-partitions.

Another solution could be to add a self-referencing foreign key to the *Partition* entity, which would represent the relationship to an original partition. However, original partitions do not have this relationship, thus resulting in NULL values in the foreign key column for each existing original partition. This is considered bad practice, which is why the link table solution is preferred, as seen in the renewed ER-diagram in Figure 5.1

Furthermore, the *type* column has been added to the *AccessPoint* entity, due to the introduction of *virtual* access points, as mentioned in Section 4.8.

Assigning types to access points is done to distinguish different types of access points.

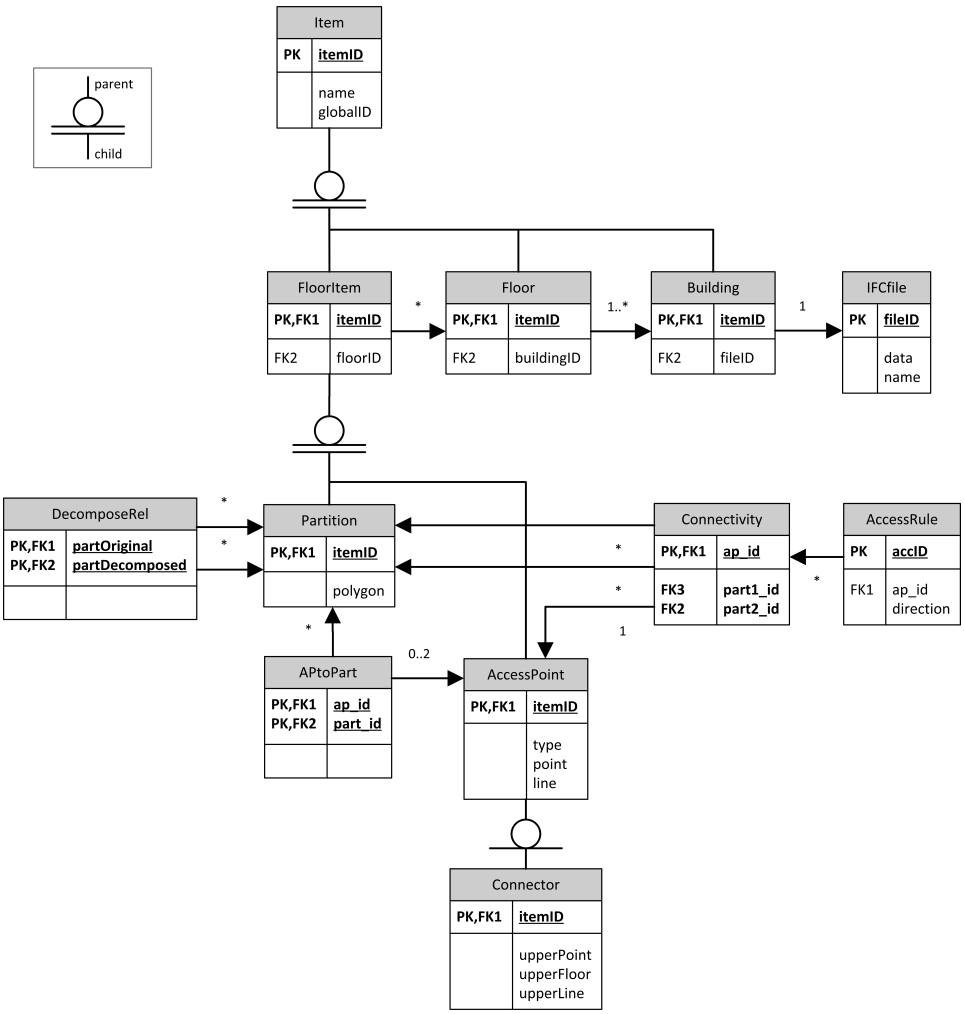


Figure 5.1: *The renewed ER-diagram for the prototype database.*

5.1.2 Indoor Space Model Representation

As mentioned in Section 3.2, extraction of stairs and elevators is not a part of the project scope. However, in order to support future incorporation of stairs and elevators, connectors are still a part the database design. The representation of connectors in the database is renewed by using a *Connector* entity instead of the *ConnectorPart* entity used in [7]. This is done to create a solution that directly represents connectors from the indoor space model. The *Connector* entity inherits from the *AccessPoint* entity and extends it with three attributes: *upperFloor*, *upperPoint*, and *upperLine*. Thus, as in-

tended, the many-to-many relationship between access points and partitions is also inherited by the *Connector* entity.

A review of the *AP-Edge* entity solution from [7], which was used to represent edges in the indoor space model, revealed a problem: The entity does not capture directionality as intended, which is essential to represent an edge. In order to capture directionality, both partitions connected by a given access point must be represented in the same entity. If this is the case, the foreign key columns can be used to reflect which direction a row represents, e.g. a *from* and a *to* foreign key referencing the *Partition* entity.

This solution creates a row for each edge in the model, which can be considered redundant in terms of connectivity, and since only accessibility rules require indication of directionality, another design is considered: Introducing a boolean attribute to the *AccessRule* entity which indicates which direction the rule is to be applied on, e.g. true for *part1* to *part2* and false for *part2* to *part1*. This solution requires only one row for each access point in the *Connectivity* entity.

The *Connectivity* entity could be considered a replacement for the link table *APtoPart*. However, during the process of back-end data management, it is expected that some access points are mapped to one or zero partitions due to missing data, incorrect data, or outdoor connectivity, which would result in NULL values in the columns of the *Connectivity* table. For this purpose, the link table, *APtoPart*, is maintained in the design. The columns in the *Connectivity* entity are given NOT NULL constraints, and a trigger is implemented to insert a row in the *Connectivity* table if two rows, with the same access point reference, exist in the *APtoPart* table. The SQL code for the trigger can be seen in Appendix D. This solution separates the entities used for the back-end data management and the indoor navigation features using the indoor space model, but maintains data persistence.

5.1.3 Spatial Data Types

The spatial extension, PostGIS, is applied to the PostgreSQL database to enable spatial capabilities. Through the use of PostGIS, the spatial data types, *point*, *line*, and *polygon*, can be utilised. This allows for a redesign of the original ER-diagram, presented in [7], in terms of representing geometrical information. As a result, the location of access points and connectors is represented by the spatial data type, *point*, as seen in the ER-diagram. Additionally, as described in Section 5.2.3, access points are represented by a line which is also a spatial data type. Partitions are represented by the *polygon* data type. These changes eliminate the use of the *Coordinate* and *Polyline* entities from the original ER-diagram, since the geometric information for partitions, access points, and connectors is now a part of their database entities.

In addition to simplifying the database design, the usage of spatial data

types makes it possible to apply spatial indices on the entities using them. In PostGIS, R-Trees are used as spatial index structures, which use the MBRs of the geometric features. As mentioned in Chapter 4, this is the background for implementing partition decomposition, in order to ensure that the search speed of queries that uses spatial indices is not degraded by significant amounts of dead space in MBRs.

5.2 Data Extraction & Interpretation

This section describes the improvements made to the data extraction and interpretation. This includes removal of intermediate points, correction of the errors described in Section 2.2.2, and the mapping procedure.

5.2.1 Intermediate points

As described in [7, p.50], the purpose of removing intermediate points is to avoid false positives when mapping access points to partitions. However, the implementation from [7] is limited to remove intermediate points on edges parallel to either the x or y-axis. A solution to this limitation, allowing removal of intermediate points on every edge, has been implemented in this project. Checking whether a point is an intermediate point is done by creating a line through the previous and following point to see if they intersect. If they do, the point must be intermediate, and can therefore be removed. This implementation is facilitated through the use of a spatial database, as it uses the spatial query *ST_Intersects*.

5.2.2 Error Corrections

Several sources of error, resulting in wrong data interpretation, are described in [7, p. 58-61]. These sources are divided into errors in the extraction and interpretation implementation and errors in the making of the IFC file. The first type encompasses inverted partitions and partitions represented by an incorrect direction. These errors have been removed by adjusting implementation in this project.

5.2.2.1 Inverted Partitions

This error is caused by a formerly unknown variable in the extraction of partitions, that affects calculations with the offset in the y-dimension. Even though substantial analysis of this problem has resulted in a solution, that makes the extraction correct, no practical reason for the introduction of such a variable has been found.

5.2.2.2 Incorrect Directions

The extraction of partitions in [7] is limited to handle directions aligned to the x- and y-axis. However, other directions are possible in partition representations, which require the implementation of a more general and complete solution. To handle all directions, the implementation has been altered to use linear transformation [3] to rotate the coordinates according to the direction.

5.2.3 Mapping of Indoor Elements

The procedure of mapping access points to partitions, described in [7, p. 33-37], is based on a threshold value to adjust the size of the area in which to search for partitions that are to be mapped to a given access point. However, as concluded in an evaluation of this procedure [7, p. 53-61], the threshold value must be individually set for each access point to avoid incorrect or insufficient mapping. Furthermore, this mapping procedure is limited to involve partitions with walls parallel to the x- or y-axis. Instead of developing a method to include individual threshold settings, a generic mapping approach is proposed in order to achieve better mapping results.

A fundamental difference, in the data representation of access points used in this project, is that the geometric shape of an access point, represented by a line segment, is extracted from the IFC file. It is assumed, that an access point is part of a wall, and that any partitions connected to an access point must intersect with a line segment drawn perpendicular to the line segment representing the access point. This assumption is used to create a generic mapping procedure which can handle partitions that are not parallel to the x- or y-axis. The mapping process implemented in this project can be described by two steps.

The first step is a coarse filtering step to quickly prune the search space to a small amount of candidate partitions while keeping the computational cost low. This is done by creating a line perpendicular to the line representing the access point with a predefined range. The spatial query $ST\_Intersects$ is then performed on this line and each partition on the same floor as the access point, where non-intersecting partitions are removed as candidates. This filtering step can be seen in Figure 5.2a where the partitions A, C, E and G are eliminated as candidates as they do not intersect with the created line.

The second and final step is only run if more than two partitions are found in the first step. In this step, all the walls in the remaining candidate partitions are inspected and if a wall intersects with the line introduced in the first step, the partition the wall belongs to is stored along with the shortest distance from the wall to the access point. The two partitions containing the walls with the shortest distances to the access point are assumed to

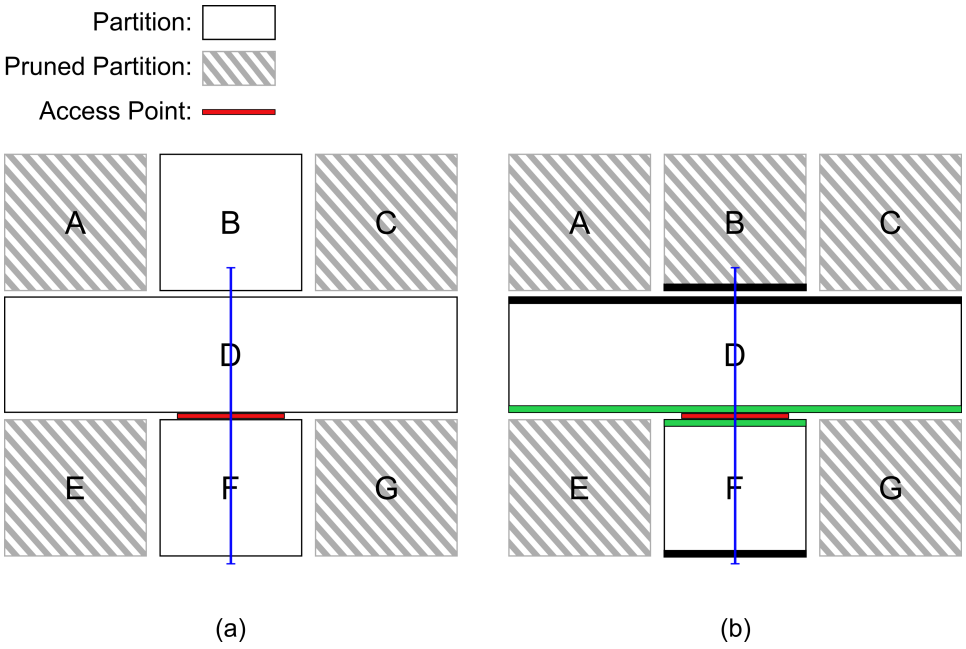


Figure 5.2: *An illustration of the two steps performed during the mapping of indoor elements.*

be the partitions connected by the access point and are thus mapped to it. This step is shown in Figure 5.2b, where the walls intersecting with the line created in step two are highlighted. The two walls highlighted in green represent the two walls that have the shortest distance to the access point. As the two walls highlighted in green belong to partitions D and F, those are the partitions mapped to the access point.

5.2.3.1 Re-evaluation of Mapping

An evaluation of the proposed mapping procedure is made to compare it with the procedure used in [7] in terms of mapping results. The evaluation is performed on the same IFC files as used in [7] and the results for both procedures are shown in Table 5.1. As it can be seen, no access points are being mapped to more than two partitions with the proposed procedure. The reason for this, is that the final step in the mapping procedure only selects the best two of the candidate partitions.

The error corrections described in Section 5.2.2 alter the data used to perform the proposed mapping compared to data used in [7]. It is assumed that the error corrections affect the mapping results positively, but only for a few access points in some IFC files.

The mapping results are significantly better for every file with the procedure proposed in this project. However, some data irregularities still exist

File name	Procedure (threshold)	Partitions mapped			
		0	1	2	>2
"AC11"	old (100)	5	72	0	0
	old (200)	5	70	2	0
	old (400)	0	5	72	0
	new (n/a)	**0**	**1**	**76**	**0**
"Office_A"	old (100)	49	53	0	0
	old (200)	20	41	40	1
	old (400)	19	38	44	1
	new (n/a)	**0**	**15**	**87**	**0**
"Nem-FZK"	old (100)	2	3	0	0
	old (200)	2	3	0	0
	old (400)	1	1	3	0
	new (n/a)	**0**	**2**	**3**	**0**
"Cassio"	old (100)	64	364	5	0
	old (200)	44	112	276	1
	old (400)	36	95	300	2
	new (n/a)	**12**	**86**	**335**	**0**
"Clinic"	old (100)	121	127	1	0
	old (200)	54	98	92	5
	old (400)	47	98	95	9
	new (n/a)	**2**	**21**	**226**	**0**
"Dds"	old (100)	2	72	32	0
	old (200)	2	22	82	0
	old (400)	2	11	89	4
	new (n/a)	**0**	**8**	**98**	**0**
"HITOS"	old (100)	48	135	15	0
	old (200)	24	75	98	1
	old (400)	21	74	102	1
	new (n/a)	**1**	**16**	**181**	**0**
"Statsbygg"	old (100)	31	65	28	0
	old (200)	20	23	80	1
	old (400)	13	41	69	1
	new (n/a)	**0**	**1**	**123**	**0**

Table 5.1: *The test results of mapping access points to partitions. The test results from [7] are included, indicated by the procedure "old", for easy comparison with the test results from the procedure proposed in this project, indicated by the procedure "new".*

in the extracted data which affect the mapping results.

Each file has one or more access points that are intended to connect a partition with the outdoor space, but these access points are only mapped to one partition, since outdoor mapping is performed through the back-end UI.

For *Office_A* and *Clinic*, bathroom stall doors are represented as access points, but these access points are only mapped to the bathroom partition, since the stall is not represented as an individual partition. These access points could be considered useless in terms of indoor navigation and could be removed through the back-end UI.

Some windows in the IFC files *Cassio* and *Clinic* are represented as access points, and are therefore extracted as such. These kinds of access points can lead to mapping with only one partition, but should be removed through the back-end UI either way.

The *Cassio* file contains various missing partitions, which causes many access points to be mapped to only one or zero partitions, as seen in the results.

As mentioned in [7, p. 60-61], *HITOS* and *Statsbygg* include partitions within partitions, which is not supported by the mapping procedures. However, after reviewing the mapping results graphically, only a few incorrect mappings occur because of this.

5.3 Data Management UI

The main contribution to the back-end UI in this project is two new features: Outdoor connectivity and connecting partitions. Besides these features, various small changes and improvements have been made throughout the continued development of the back-end UI. Although the back-end UI has been updated in this project, it still exists as a proof of concept in terms of data management and representation.

5.3.1 Outdoor Connectivity

In [7, p. 55], an analysis of the evaluation results shows that some access points are only mapped to one partition because of outdoor connectivity. Such mapping violates the indoor space model, as all access points must be mapped to exactly two partitions. To solve this, an outdoor partition is introduced, which can be used to capture the outdoor connectivity. Since it is not possible to determine if an access point with only one mapped partition represents outdoor connectivity or not, the option of assigning outdoor connectivity to access points is given to the user. This feature is implemented, such that if a user selects an access point with only one mapped partition, the option of connecting that access point to the outdoor

partition is present. An example of this feature can be seen in Figure C.2 in Appendix C.

5.3.2 Connecting Partitions

As part of the development of the prototype, many different IFC files has been reviewed, and it has been discovered that the occurrence of partitions without any nearby access points is frequent. While some files have one or two partitions of this kind, other have many. An example of this can be seen in Figure C.1 in Appendix C. Whether leaving the partitions inaccessible is intentional or if access points are simply missing in the IFC is unknown. However, since there is a possibility that access points are missing, the feature of inserting access points through the back-end UI is considered important, and is therefore implemented in this project.

The feature allows a user to select a partition and view any nearby partitions that can be connected to the selected partition. An example of this is shown in Figure C.3 in Appendix C. The list of possible connections is created using a spatial range query on the partitions on the same floor. The user can then select a partition from the list and click the *Connect Partition* button to create an access point between the two partitions. The created access point will then connect the two partitions, as shown in Figure C.4 in Appendix C. The position and orientation of the created access point is computed through various geometrical calculations and comparisons using the position and shape of the two partitions. This procedure will not be described in greater detail in this report.

Chapter 6

Indoor Distances & Routing

To demonstrate a use of the data extracted through the back-end, by adding a routing functionality to the navigation app, a routing algorithm is implemented in this project.

As explained in [7, p. 26], the indoor space model needs to capture indoor distances in order to support efficient routing. Since indoor spaces do not contain explicit paths, as it is possible to move freely around, the shortest distances between access points within every partition are used to generate the shortest route. In the work done in [7], access points and partitions are stored with coordinates representing their location in a 2D Cartesian coordinate system. However, the calculation of distances is not implemented but made possible by storing these coordinates. This chapter describes the algorithm used for routing and how calculation of intra-partition distances between access points is implemented.

6.1 Routing Algorithm

The routing algorithm implemented for this project is defined in [4] and can be seen in Appendix E. It is based on door-to-door (access point to access point) distances and is able to calculate the shortest route between two positions in the indoor space. This is done by using an exploratory approach to find possible shortest routes between two points and updating the result if a shorter route is found as the different paths are traversed. As the algorithm runs, the currently shortest indoor distance from a source door to a destination door is stored and used to avoid unnecessary computations if more than one source or destination door exist. Furthermore, the door-to-door distances calculated at a given time are reused to limit the search. Accessibility, in terms of directionality, is also utilised to prune the search space.

6.2 Intra-Partition Distances

The calculation of intra-partition distances between access points is not the focus of [4], and is therefore not described. The shortest distance between two access points can be assumed to be the Euclidean distance between them. However, as described in [7, p. 26], this is not always correct, as the path that represents this distance can be obstructed by obstacles or walls in concave shaped partitions.

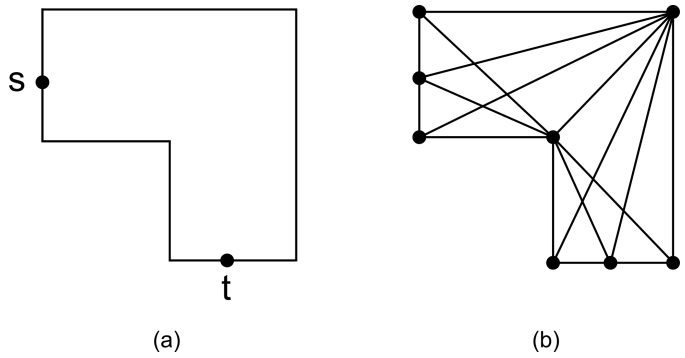


Figure 6.1: *An example of creating a graph using a partition and the start and end point.*

A straightforward approach to calculate intra-partition distances is to use Dijkstra's graph search algorithm [1, p. 658-662], which resembles the approach proposed in [10]. This is done by creating a graph using all the vertices in the partition polygon, and the start and end point as graph vertices. Edges are applied between graph vertices if a line segment can be drawn between them without it intersecting the polygon or being outside it. If an edge is applied, the weight is set to be the length of it, i.e. the Euclidean distance between the two vertices. An example of creating such a graph is illustrated in Figure 6.1, where Figure 6.1a represents the partition with start and end points, and Figure 6.1b represents the graph created. When Dijkstra's algorithm is run on the graph, the shortest distance between the start and end point is computed. This approach computes the shortest intra-partition distance between two access points, but the computation can be exhaustive if the partition polygon has many vertices. The following sections describe an attempt to create a more efficient approach to compute these distances, starting with a review of the iNav approach [9].

6.2.1 iNav Approach

In [9], a proposal for calculating distances in concave shaped partitions is given. Here, the boundary of a partition, created by the intersection with the line segment, is defined as the concave boundary. A concave boundary can have one or more concave vertices. The approach to calculate the shortest path is to choose a concave vertex on the concave boundary as an intermediate point. This point is used to create two new line segments. This process is repeated for each line segment until no intersection with the partition walls occurs.

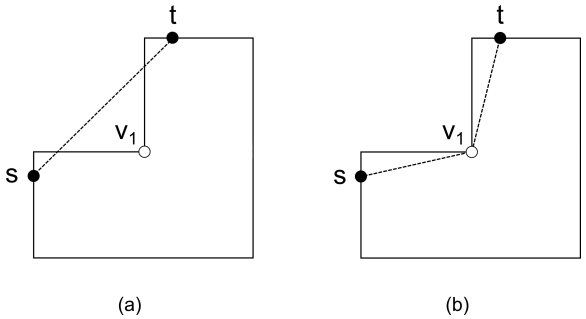


Figure 6.2: *An example of how the iNav approach calculates the intra-partition distance between access points s and t.*

An example of this can be seen in Figure 6.2 where the shortest distance between the access points s and t is desired. In Figure 6.2a, the line segment between s and t is illustrated. It is clear that this direct path is not a possible path to walk, as it intersects two walls. In this case, the concave boundary is the boundary between the intersecting points which includes the concave vertex v_1. In Figure 6.2b, the iNav approach is applied, where v_1 is chosen as the intermediate point and thus the shortest distance from s to t is obtained.

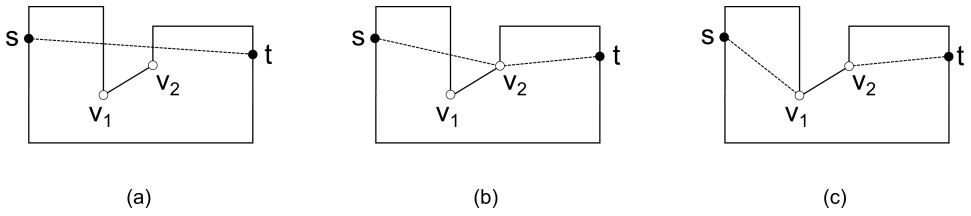


Figure 6.3: *An example where the iNav approach is insufficient and calculates an incorrect shortest distance between access points s and t.*

However, this approach is naive as the intermediate point is arbitrarily chosen, which can cause problems when more than one concave vertex is present. An example of this is shown in Figure 6.3 where the distance obtained will vary based on the order in which the concave vertices are chosen as intermediate points. In Figure 6.3a, the line segment between s and t

creates a concave boundary with two concave vertices, v_1 and v_2. In Figure 6.3b, v_2 is chosen as the intermediate point and the line segments, from s to v_2 and from v_2 to t, are created. Here, the first line segment still intersects with the wall and is therefore split using the only concave vertex on its concave boundary, v_1. In Figure 6.3c, the assumed shortest distance, from s to t, is obtained as the path $s - v_1 - v_2 - t$ using the iNav approach. However, this is not correct, as the shortest path is $s - v_1 - t$. In order to calculate distances in concave partitions correctly, a new solution, called *Walk the Line*, is proposed in this project.

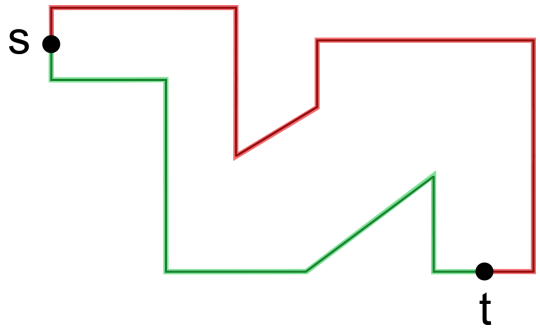


Figure 6.4: *An example of the initial split made on the boundary of the partition in Walk the Line.*

6.2.2 Walk the Line

The procedure of Walk the Line is defined in Algorithm 2. The algorithm take as input a partition p, a start point s, and an end point t. As part of the initialisation, two polylines are created from the boundary of p, using the start and end point, as seen in Line 3. An example of this procedure is depicted in Figure 6.4, where the two highlighted polylines, which constitute the boundary of the partition, are created using points s and t as splitting points.

The algorithm returns a set of points, R_{path}, which constitute the shortest path from s to t. After adding s to R_{path} and initialising a temporal point *temp* to be s in Line 4-5, a **while**-loop is run from Line 6-23, which encompasses the code to find the shortest path.

This part of the algorithm is explained using the example shown in Figure 6.5. In Figure 6.5a, the line segment from s to t is represented by a dashed line, which corresponds to Line 7. This line intersects with the boundary of the partition, and an intermediate point must be found, which is done in Line 11-18.

Here, the two polylines created in the initialisation are used and since the

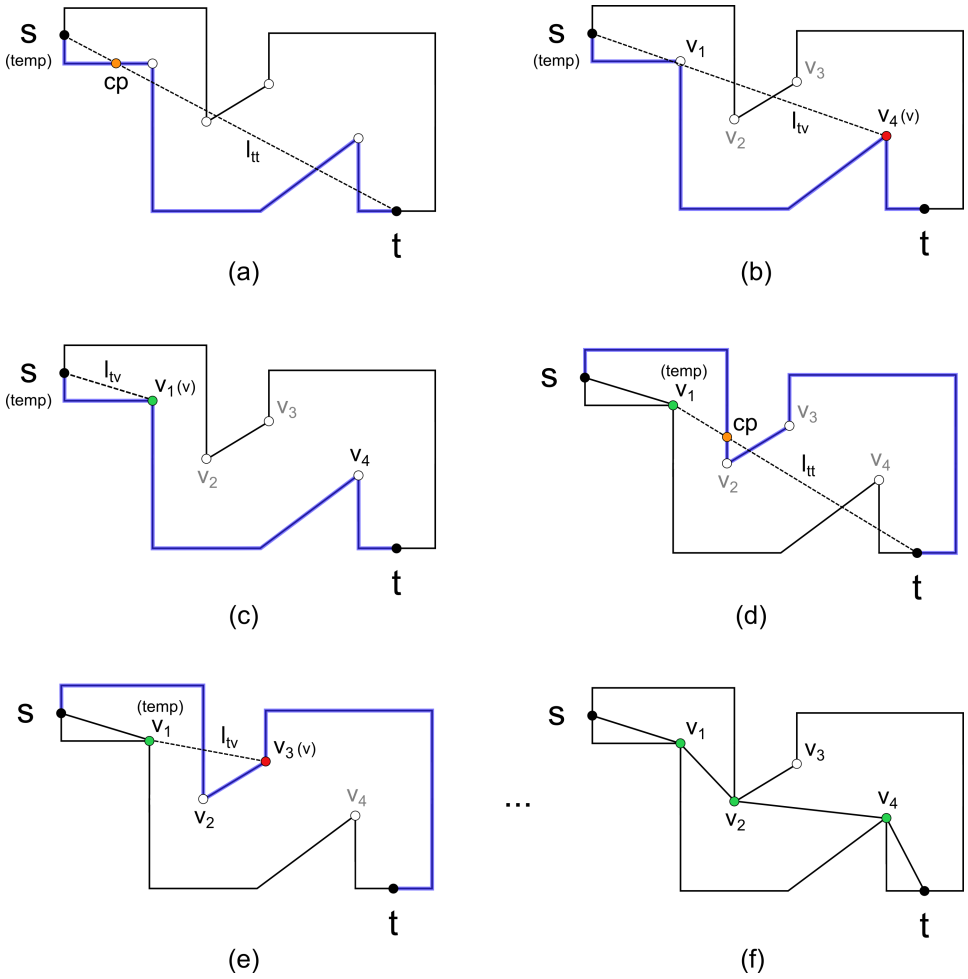


Figure 6.5: *An example demonstrating the procedure of how to calculate shortest intra-partition distances using Walk the Line.*

intersecting point closest[1] to s, the point cp, lies on the polyline highlighted in blue, the intermediate point is chosen from the concave vertices on this polyline by "walking the line". This is done by iterating through the concave vertices, starting with the one encountered first when traversing the polyline from t to $temp$, in this case v_4. For each concave vertex, the line segment from $temp$ to the vertex is drawn, and the first vertex, where the line segment does not intersect, and is within, the boundary of the partition, is chosen as the intermediate point. This process is shown in Figure 6.5b and 6.5c. First v_4 is tested, but the line drawn intersects with the boundary of the partition, so the next concave vertex, v_1, is tested and it passes as an intermediate point since the line segment drawn does not intersect with the boundary of the partition. As such, v_1 is added to R_{path} and chosen as the new $temp$, as seen in Line 14-15.

Figure 6.5d illustrates the start of a new **while**-loop iteration, and the procedure shown in Figure 6.5a is repeated with v_1 as $temp$. In this iteration, cp lies on the other polyline created in the initialisation, which is then used. The **for**-loop in Line 11-18 is repeated once again, as shown in Figure 6.5e, where the first concave vertex encountered on pl, v_3, is tested. The process is continued and repeated until t is reached, as shown in Figure 6.5f.

Algorithm 2 Walk the Line

```
 1: function WALK THE LINE(partition p, point s, point t)
 2:     set of points R_path; point temp;
 3:     split the boundary of p into polylines pl_1 and pl_2 using s and t;
 4:     add s to R_path;
 5:     temp ← s;
 6:     while temp != t do
 7:         draw a line segment l_tt from temp to t;
 8:         if l_tt intersects with the boundary of p then
 9:             set point cp to the intersecting point closest[1] to temp;
10:             set polyline pl to pl_1 or pl_2, depending on which one contains cp;
11:             for each concave vertex v on ls, starting with the one closest to t on pl do
12:                 draw a line segment l_tv from temp to v;
13:                 if l_tv does not intersect and is within the boundary of p then
14:                     add v to R_path;
15:                     temp ← v;
16:                     break
17:                 end if
18:             end for
19:         else
20:             temp ← t;
21:             add t to R_path;
22:         end if
23:     end while
24:     return R_path;
25: end function
```

[1] The point which has the shortest Euclidean distance to another given point.

6.2.2.1 Limitations

Through extensive testing, a single scenario has been found where the shortest intra-partition distance cannot be found using the Walk the Line algorithm. This scenario is illustrated in Figure 6.6. The scenario occurs when the line segment, drawn in Line 7 in Algorithm 2, lies outside the polygon. This should be checked for in the **if**-statement in Line 8, in the same way as it is done in Line 13. However, even though this check is made, the algorithm is not able to handle this scenario using the procedure in Line 8-17, since no *cp* exists, making it impossible to determine *pl* in Line 10.

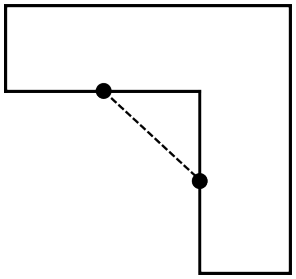


Figure 6.6: *A scenario where the Walk the Line algorithm cannot determine the shortest route, since the line segment drawn between the two points lies outside the polygon.*

Another limitation to the Walk the Line algorithm is the assumption that the start and end points are positioned as access points, i.e. on the boundary of the partition. However, when computing the shortest route, other types of intra-partition distances are used, e.g. the shortest distance between the start point and an access point inside the partition where the start point is located. As such, the algorithm is limited to run for intra-partition distances between access points.

6.3 Final Solution

As described, two limitations exist when using the Walk the Line algorithm. However, the algorithm can still be applied to the scenarios where these limitations do not influence the computation. As such, a solution is proposed which includes both the Dijkstra's algorithm and the Walk the Line approach, by using Dijkstra's algorithm in the scenarios where Walk the Line comes short.

It is assumed that Walk the Line is a more efficient algorithm than Dijkstra's in the scenarios where it can be applied. In order to verify this assumption, several tests have been performed. The test configurations, results and analysis are described in Chapter 8.

Chapter 7

Navigation App

The development of an app, capable of providing navigational services, is done to create a bridge between the data extracted by the back-end and indoor navigation, by visualising a contextual use of it. Additionally, the routing algorithm is used to provide a routing feature through the app interface. This chapter describes the architectural choice, design and functionality of the app.

7.1 Architecture

The architecture used to develop the app is a client-server architecture. The client (app) sends requests to the server (back-end) to retrieve data. However, the architecture details can vary, and to determine them is a matter of choosing what should be handled on the client side and on the server side. In order to do this, two contrary architectures, shown in Figure 7.1 and 7.2, are compared to find pros and cons. The parts of the back-end that do not have any influence on these specific architectures have been omitted from the figures. An overview of pros and cons in the two architectures can be seen in Table 7.1.

The architecture in Figure 7.1 is mainly based on the ability to use the app when a connection to the server is not available, by storing indoor data

	Pros	Cons
Figure 7.1	- Offline client usage	- Spatial database on device - Storage space on device
Figure 7.2	- No client storage - No client route computation - Global cache	- Frequent communication

Table 7.1: *The pros and cons for the two app architectures shown in Figure 7.1 and 7.2.*

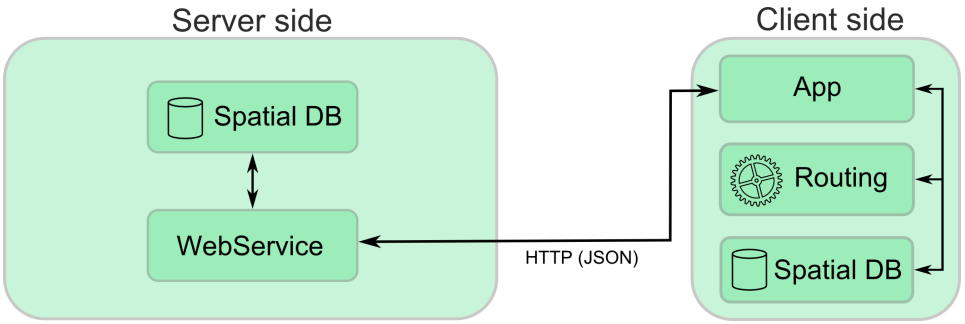


Figure 7.1: *An architecture where most tasks are handled on the client side.*

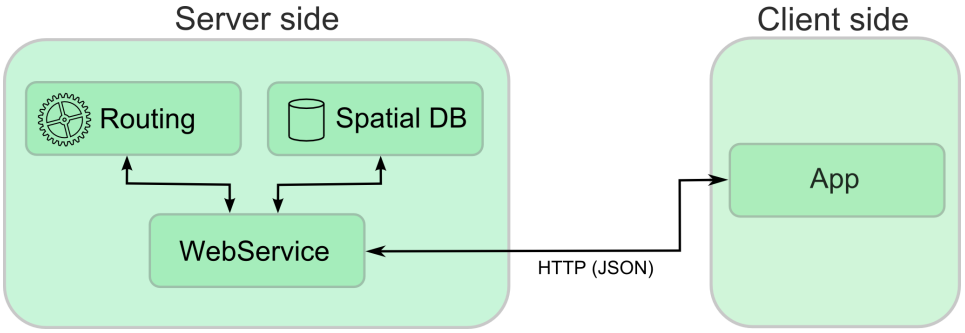


Figure 7.2: *An architecture where most tasks are handled on the server side.*

and running the routing algorithm on the client side. However, this solution requires that a spatial database is available on the client side device, which is uncommon. Furthermore, a certain amount of storage space on the device is required to store all the indoor data needed. In the case that a user wants to store data for an entire city, it could exceed a reasonable amount of storage space required.

The focus of the architecture depicted in Figure 7.2 is to create a lightweight app, by handling both storage and route calculation on the server side. This ensures that no significant amount of storage space is required on the client side, since all data needed by the client is requested from the server. However, it can be assumed that during normal use, frequent client-server communication is needed. It is also assumed that the computation power on the server side is better that on the client side, which makes this architecture favourable in situations where the data transfer rate between the client and the server is insignificantly low. Additionally, a global intra-partition distance cache can be obtained by calculating routes on the server-side.

Considering the high availability of internet access, achieved through mobile telephony communication (3G or 4G) and WiFi hotspots, and that the app is mainly developed for visualisation purposes, the architecture in Figure 7.2 is used in development of the app.

7.2 Workflow

The workflow design of the app is based on simplicity, by including only the minimum amount of actions possible to represent the data extracted via the back-end and to enable use of the routing functionality.

The workflow is illustrated using an activity diagram, which is shown in Figure 7.3. The first action taken, after opening the app, is browsing a list of available indoor building maps. The user is able to refresh the list of maps, re-requesting the list of maps from the web service on the server side. After a map is chosen, a floor must be also be chosen before a graphical representation can be displayed. During the display of a graphical floor map representation, several actions can be taken. In order to navigate the floor map, either zooming or panning can be performed. A start and an end point, for a route request, can be chosen, as well as clearing already chosen points. A route can be requested if both a start and an end point has been chosen. By doing so, a route calculation request is send to the server side and the route is returned and displayed. Finally, the user can either go back to browse the list of indoor building maps, or choose another floor on the map currently chosen.

7.3 User Interface

In order to give an impression of how the app looks and feels, several screen shots from it are provided in Figure 7.4. As described in the workflow in Section 7.2, the user is able to retrieve a list of the stored building maps, choose a building and a floor, and finally get a graphical representation of it.

Figure 7.4 contains five screen shots which represents the steps to be taken in order to retrieve a route, from one point to another, on a given floor. In Figure 7.4a, a graphical overview of the floor selected by the user, is given. In Figure 7.4b, the user has panned and zoomed to a desired location on the indoor map and chosen a start point, marked by a green dot, for the route that he wants to be calculated. To choose a start or end point, the user has to make a long touch at the desired location of the point, which triggers a dialogue similar to the one shown in Figure 7.4c. In Figure 7.4c, the user has made a long touch and has to choose if he wants to choose the selected location as an end point, replace the already chosen start point, or clear the points such that none are chosen.

In Figure 7.4d, it appears that the user chose the selected location as an end point, which is represented by the red dot. At this point the user has chosen a start and end point and only needs to request the route from the server. In Figure 7.4e, the route has been requested and returned, and is represented by the magenta coloured line.

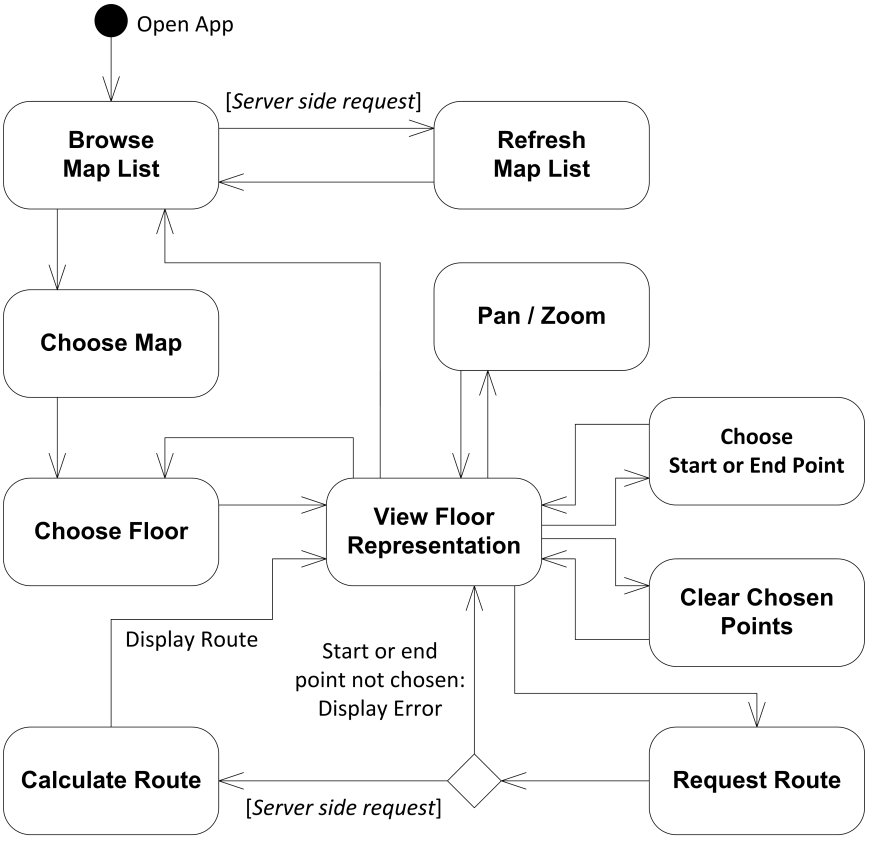


Figure 7.3: *The workflow of the app, represented as an activity diagram.*

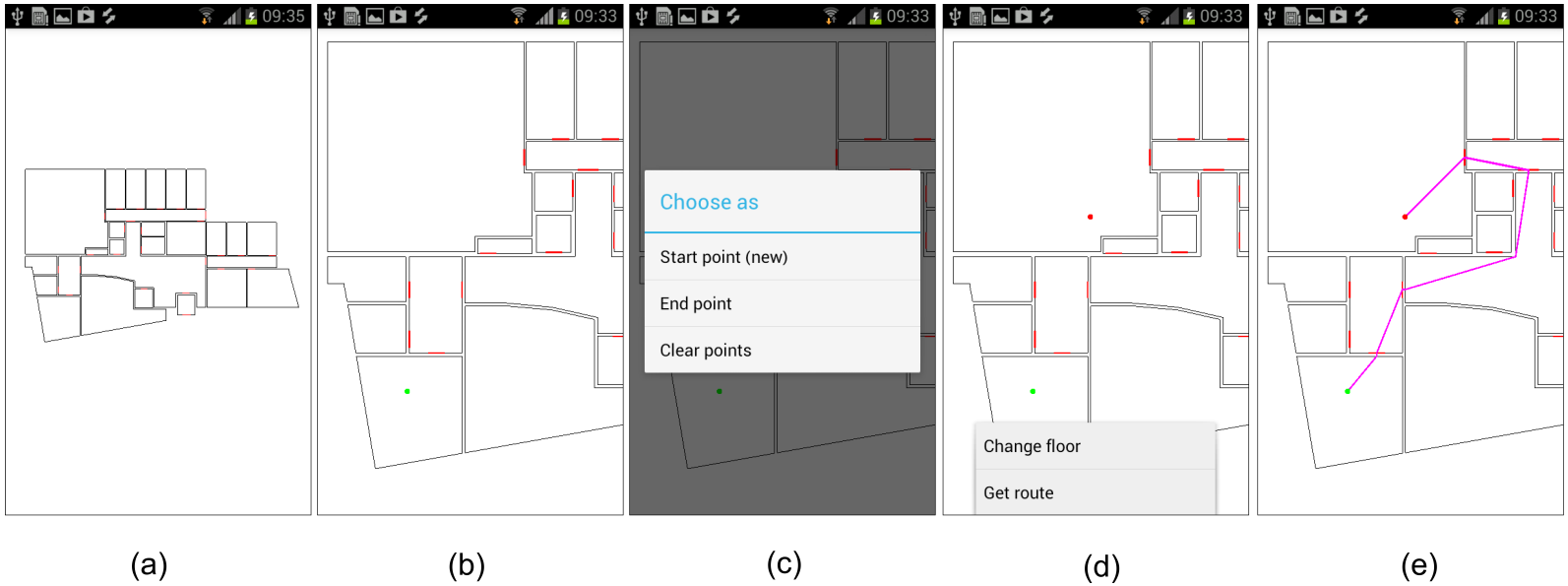


Figure 7.4: *Screen shots of the app UI, demonstrating the use of the routing functionality.*

Chapter 8

Experimental Evaluation

Several tests have been performed using the routing algorithm implemented in this project. All tests are run on the back-end of the prototype (server side), due to the architectural choice explained in Section 7.1. The tests have been performed to analyse the effect of the following:

- Computing intra-partition distances using the solution described in Section 6.3 (WTL) vs. using only the Dijkstra's algorithm approach (OD)
- Storing access point to access point distances in a cache and reusing them in subsequent route computations
- Decomposing partitions before calculating routes

The first test is performed to verify the assumption that WTL is faster than OD when computing intra-partition distances during the routing algorithm.

The second test is performed to show the influence of using a cache for storing intra-partition distances between access points. The cache is implemented such that when an intra-partition distance is requested by the routing algorithm, a lookup is performed on the cache to see if that distance already have been calculated, and the distance is returned if so. If not, the distance is calculated, stored in the cache, and then returned. The cache is maintained over multiple routing requests, which means that the distances stored in the cache can be used by future route calculations.

The third test is performed to show the influence of decomposing partitions before routes are computed. However, during decomposition, virtual access points are created to connect sub-partitions. The intra-partition distance computations use the centre of an access point, which results in inaccuracies in terms of computing the shortest route, as illustrated in Figure 8.1.

This chapter describes the different test configurations, the results and the analysis of them.

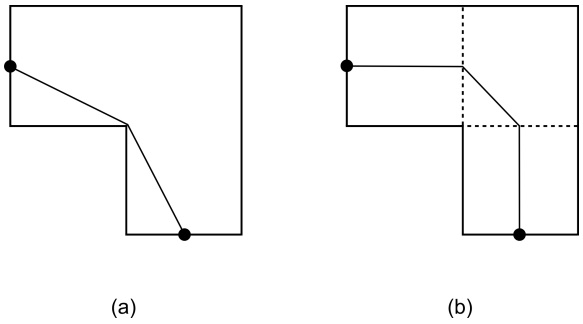


Figure 8.1: *An example of a route calculated before and after the partition is decomposed. After decomposition, the calculated route is no longer the shortest possible, because it must go through the centre of each virtual access point.*

8.1 Test Configuration

The computer used for all the tests has the following configuration: An Intel Core 2 Duo P8600 @ 2.4 GHz processor, with 2.25 GB RAM, running Windows 7. In order to generate multiple random routes, a function to find a random start and end point is implemented. This function does not allow the start and end points to be in the same partition. Furthermore, the following configurations are set for the different tests:

WTL vs. OD This test is run with four different IFC files, which differ in number of access points, partitions, and concave partitions. The route algorithm is run for different files because the computation time of intra-partition distance computations vary with the shape of the partition polygon and the placement of access points. For each file, the routing algorithm is run twice using the same 50 randomly generated start and end points; once for WTL and once for OD.

Using a cache (WTL vs. OD) This test is performed by running each algorithm, starting with an empty cache, for 50 randomly generated routes. This process is repeated 50 times to calculate the average computation time for each algorithm after a specific number of routes. Four IFC files are also used for this test.

Decomposing Partitions For both of the aforementioned tests, another test is performed after decomposing partitions in the extracted data. In the routing algorithm, the WTL solution is used for these tests. This method is referred to as WTLD.

An overview of the IFC files used in the tests is shown in Table 8.1.

File name	Floor name	Decomposed	#Partitions	#Access points
"Cassio"	"Ground Floor"	No	170	192
		Yes	241	266
"Clinic"	"First Floor"	No	155	173
		Yes	211	231
"Office_A"	"Level 1"	No	60	68
		Yes	93	105
"Dds"	"3. etasje"	No	37	42
		Yes	52	57

Table 8.1: *A list of the files and specific floors that have been used for testing. The number of partitions and access points contained on each floor is included.*

8.2 Results & Analysis

The results from the first test, WTL vs. OD, are illustrated using a bar chart for each IFC file, shown in Figure 8.2, 8.3, 8.4, and 8.5. The bar charts show the computation time using OD in percentage of the computation time using WTL for the same route, e.g. the computation time using OD for the first route in Figure 8.2 is equal to 200% of the time used to calculate the route using WTL. It can be seen, in each of the charts, that route calculation using OD requires longer computation time than with WTL. However, the computation time greatly varies from route to route, for which the reason can be assumed to be a varying number of concave partitions included in the routes.

In the results obtained using the *Clinic* file, OD is significantly slower than WTL, up to 23 times, compared to the test results obtained using the other files. Through a review of the indoor elements contained in the extracted data from the *Clinic* file, it is found that the file contains a relatively high amount of concave partitions, which in most cases are connected to many access points. This means that in many cases, during the route calculation, an intra-partition distance in a concave partition has to be calculated, due to the high amount of connections between access points inside the partition. Furthermore, many partitions, with a high amount of vertices in their polygon representations, exist in the *Clinic* file. This can increase the computation time additionally, due to Dijkstra's algorithm using a min-priority queue implemented with a binary min-heap having a worst case running time of $O((|E| + |V|) * log|V|)$ [1, p. 661-662], [6].

As mentioned, the difference in computation time between using OD and using WTL varies significantly depending on the route. To provide an overview of the average computation time for one route, for each of the different IFC files, a bar chart is shown in Figure 8.6. For the file, *Clinic*, the average computation time using OD is 114.4s. However, the bar chart is cut off at a computation time of 25s, as it is clear that the average computation time for OD is significantly longer than for WTL in all of the tested files.

Another important note is that the average computation time using

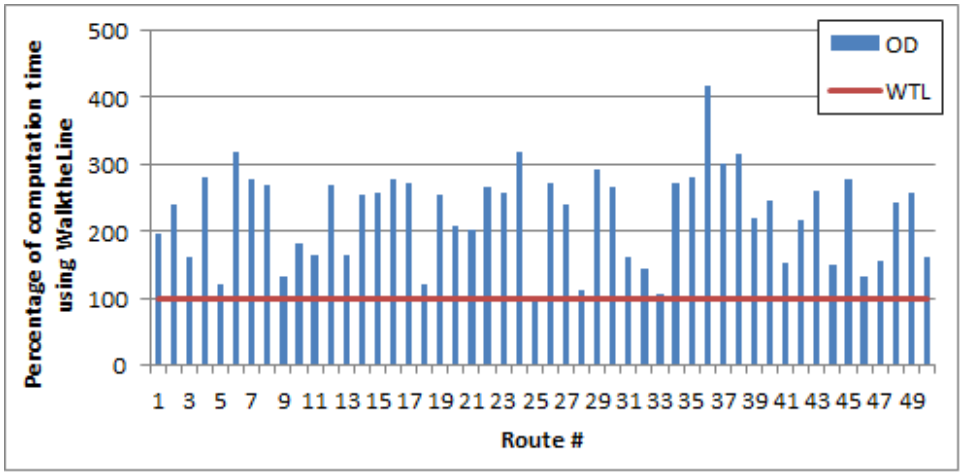


Figure 8.2: *The results for the "Office_A" file, i.e. how OD performs compared to WTL when the same 50 routes are run for both algorithms.*

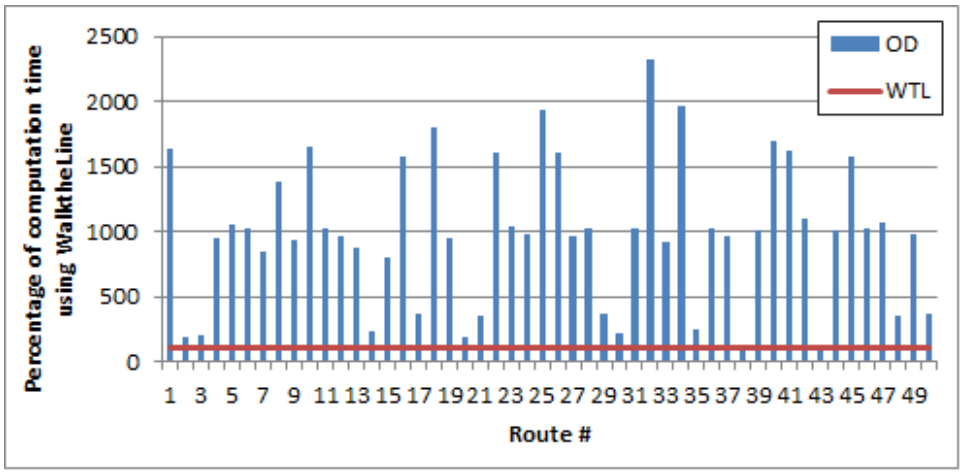


Figure 8.3: *The results for the "Clinic" file, i.e. how OD performs compared to WTL when the same 50 routes are run for both algorithms.*

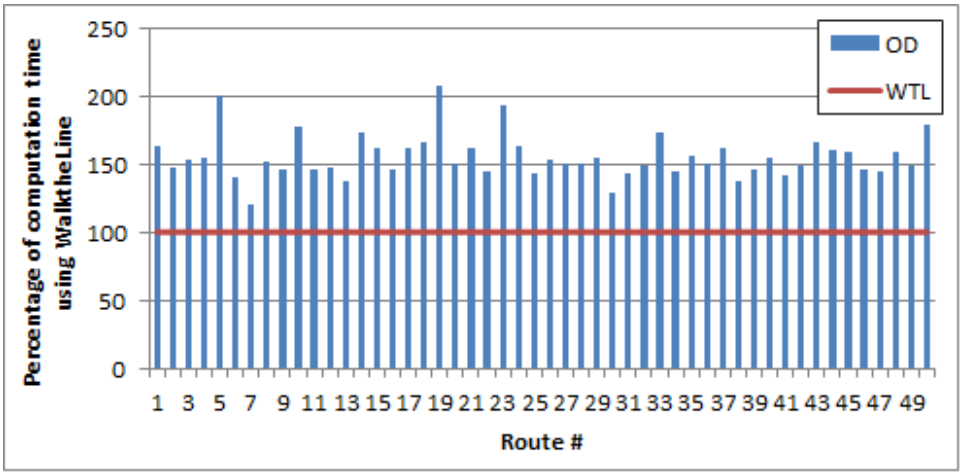


Figure 8.4: *The results for the "Cassio" file, i.e. how OD performs compared to WTL when the same 50 routes are run for both algorithms.*

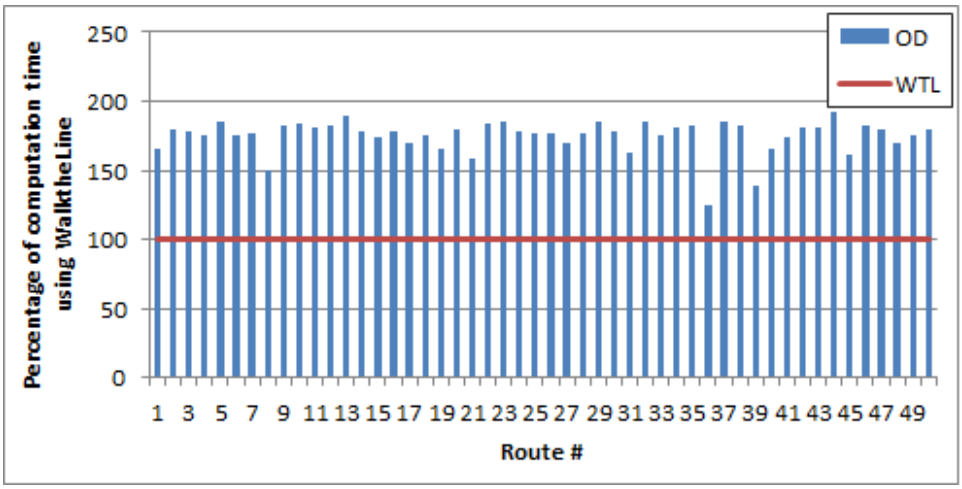


Figure 8.5: *The results for the "Dds" file, i.e. how OD performs compared to WTL when the same 50 routes are run for both algorithms.*

WTLD outperforms WTL and OD. This is an expected result, as the number of concave partitions is reduced substantially after the decomposition. Unfortunately this algorithm does not provide the shortest route because it uses centre of virtual access points. An example of this is shown in Figure 8.1.

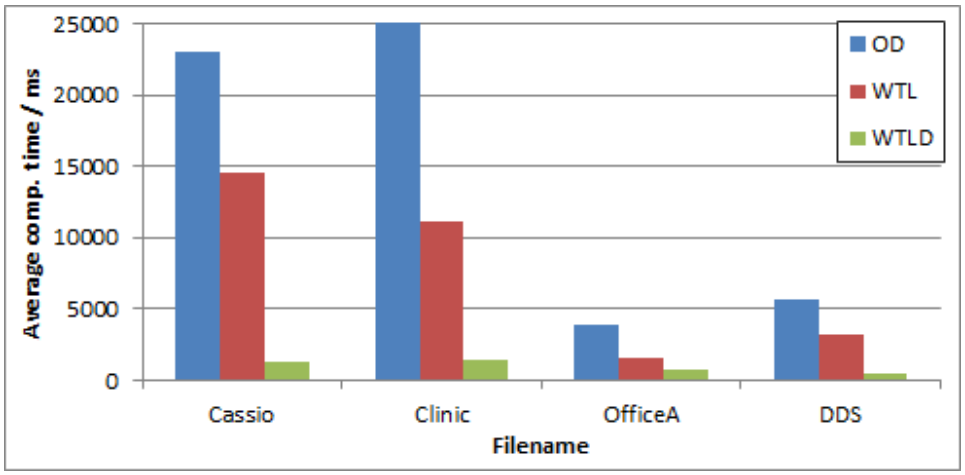


Figure 8.6: *The average computation time for one route for each of the four files tested. The average is computation time is shown both for OD, WTL, and WTLD.*

The results of the cache tests can be seen in the line charts in Figure 8.7, 8.8, 8.9, and 8.10. For the first couple of routes, in each of the charts, the obtained result is that WTLD is faster than WTL, which is faster than OD. This is an expected result as the cache is very small at this point and the results reflect the base performance of each algorithm, as already analysed.

An observation, that can be made from these test results, is the number of routes to be run before the cache is used almost exclusively, i.e. when almost all of the intra-partition distances are stored in the cache. As it can be seen in the four charts, the average computation time for each of the algorithms converges as the size of the cache increases and a conclusion is that WTL is faster than OD until only the cache is used, for all tested files. However, it is assumed that the number of routes to be run, before the cache is used exclusively, increases with the number of possible routes that can be taken in the indoor space. It can be seen in the results, that with the two files, *Clinic* and *Dds*, WTL and OD seem to converge after 18 and 8 routes have been calculated, respectively. Analysis of the files show that *Clinic* contains a significantly larger amount of possible routes than *Dds*, which supports the assumption.

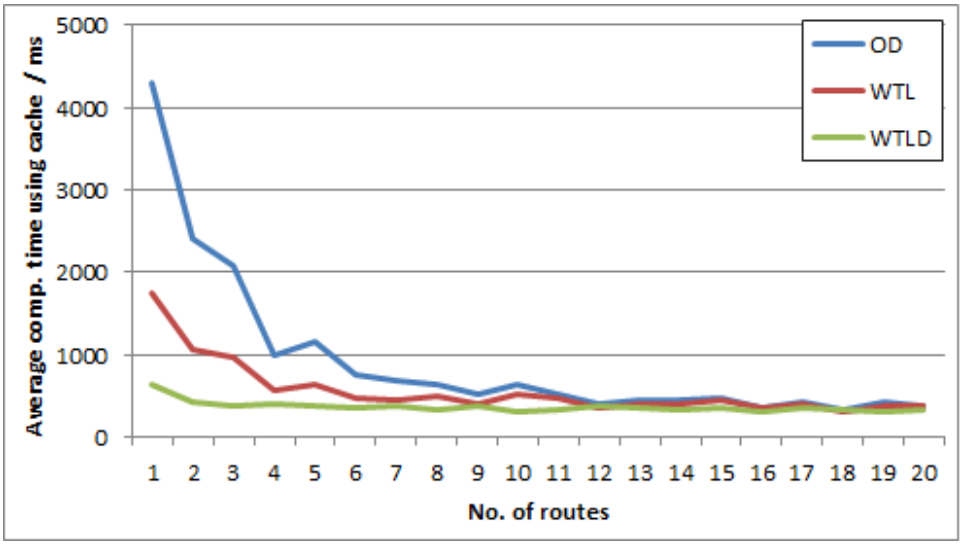


Figure 8.7: *The average computation time for OD, WTL, and WTLD, when a cache is used. These results are obtained from testing the "Office_A" file.*

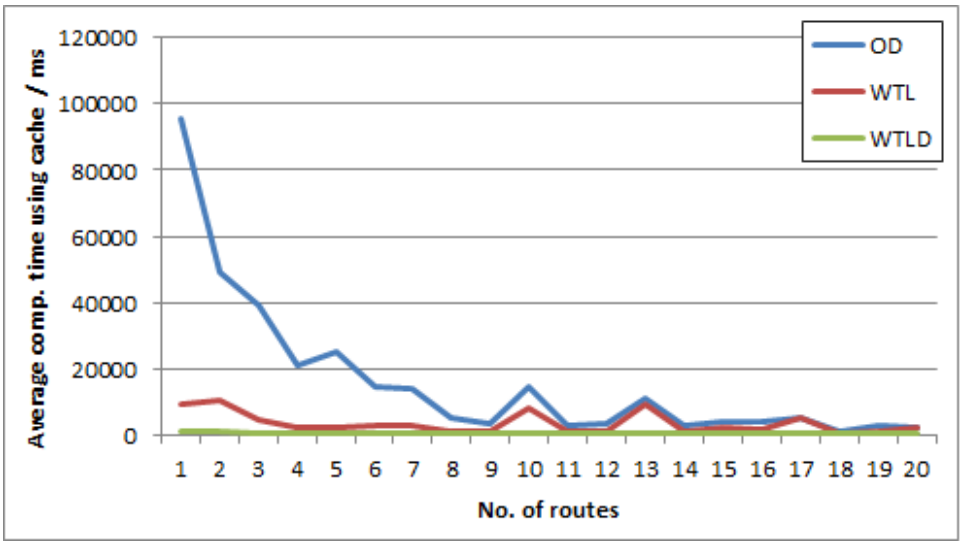


Figure 8.8: *The average computation time for OD, WTL, and WTLD, when a cache is used. These results are obtained from testing the "Clinic" file.*

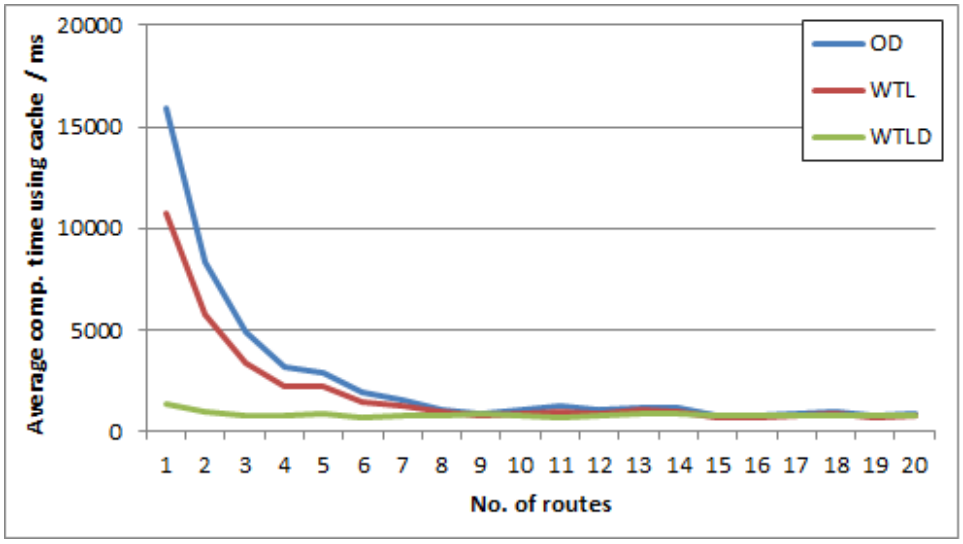


Figure 8.9: *The average computation time for OD, WTL, and WTLD, when a cache is used. These results are obtained from testing the "Cassio" file.*

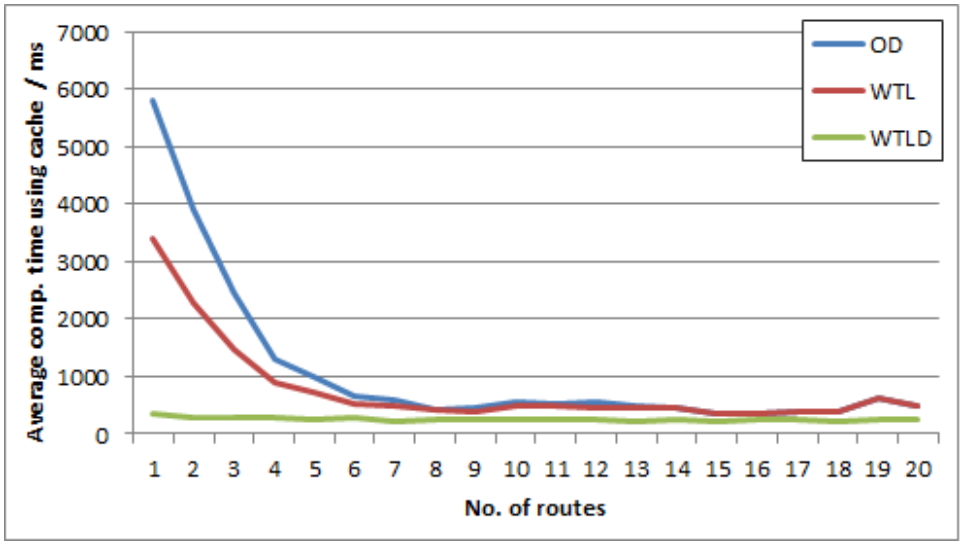


Figure 8.10: *The average computation time for OD, WTL, and WTLD, when a cache is used. These results are obtained from testing the "Dds" file.*

8.3 Findings

The findings are that it is preferable to use WTL over OD both with and without a cache. When using a cache the effect of using one approach over another decreases as the cache gets populated with intra-partition distances. Using WTLD gives a great performance compared to calculating routes with non-decomposed partitions, but the effect of this also decreases over time by using a cache. In the case that no cache is available, the choice of using WTL over WTLD is a trade-off between performance and getting the actual shortest route.

Chapter 9

Conclusion

As stated in the project definition in Chapter 3, the problem of this project is to improve the prototype developed in [7], such that the data produced by the back-end can be used for indoor navigation purposes. Additionally, the problem involves the development of a front-end with navigation features to demonstrate a use of the extracted data. As a solution to this problem, the following contributions have been made in this project:

- Decomposition of partitions
- Improved back-end extraction and interpretation, database, and data management UI
- Intra-partition distance calculation
- Indoor navigation app with routing functionality

Due to the use of a spatial database, and thereby spatial indices, effort has been put into the implementation of partition decomposition, since irregular and imbalanced partitions can degrade spatial query performance. The implementation is based on the work of [5], and includes solutions to several shortcomings found in the original decomposition algorithm. Partition decomposition is considered an important contribution, because it improves an existing decomposition algorithm to handle real world data, and because it is a general improvement to how spatial data is handled in this project.

Several enhancements are made to the back-end, including a redesign of the database, an improved mapping approach, and development of two key features in the back-end UI: Outdoor connectivity and connecting partitions.

In order to implement a routing algorithm, that can utilise the indoor data captured, a method to compute intra-partition distances has been developed. This method includes a specifically developed geometry-based solution, called Walk the Line, to be used in scenarios where it is applicable,

and applies Dijkstra's algorithm as a secondary solution. The method is evaluated and the test results show that the proposed method is better than using Dijkstra's algorithm exclusively in all aspects.

The routing functionality, as well as the indoor data captured, is visualised through an app, which allows users to view different floor maps, from buildings which have been extracted from IFC files using the back-end UI.

In conclusion, the prototype system developed in this project bridges the gap between IFC files and indoor navigation, by extracting and transforming data via the back-end, which can be applied for routing and navigation features in an app.

Chapter 10

Future Work

This chapter describes different valuable additions that can be made to the project in the future.

10.1 Connectors

As mentioned in Section 3.2, stairs and elevators are not extracted from IFC files, and are therefore not included in some parts of the development in this project.

However, as part of future work, a solution to include stair and elevator information could be developed. As mentioned, this could be done by enabling users to include this information through the back-end UI. This would require the possibility of displaying multiple floors, possibly using a 3D representation, to create an overview of the position of a staircase or an elevator and the partitions, and thereby floors, it connects. Such a solution would require extensive UI development and possibly include several usability aspects.

10.2 Positioning

As mentioned in Section 2.2.7, positioning and tracking is a valuable addition to an indoor navigation system. An interesting approach, regarding future work, would be to apply a positioning feature to the app, by enabling the ability to locate the current position of the device when the app is run. This would also enable the user to select his current position as the starting point for a route.

10.3 IFC File Parser

The parser for IFC files developed by Open IFC Tools [8], which is used in this project, is limited to parsing entire files, storing all the parsed data in

memory at run time. This is not a durable solution for two reasons: Because most IFC files contain more information than needed in the extraction process, and because some files contain so much detail that it is not possible to store the parsed data in memory, which means that these files cannot be used.

Future work could involve the development of a parser for IFC files, that would allow parsing of specific building elements by request. This would be a better solution, since the response time of the extraction process would be reduced significantly, and the level of detail contained in the IFC file would not be of influence.

Chapter 11

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[2] Christian S. Jensen, Hua Lu, and Bin Yang. Indoor-a new data management frontier. *IEEE Data Eng. Bull.*, 33(2):12–17, 2010.

[3] David C. Lay. *Linear Algebra and Its Applications*. Pearson Addison Wesley, 2006.

[4] Hua Lu, Xin Cao, and Christian S. Jensen. A foundation for efficient indoor distance-aware query processing. *28th IEEE International Conference on Data Engineering (ICDE)*, 12:438–449, 2012.

[5] Hua Lu, Xike Xie, and Torben Bach Pedersen. Efficient distance-aware query evaluation on indoor moving objects. *29th IEEE International Conference on Data Engineering (ICDE)*, pages 434–446, 2013.

[6] Oracle. java.util class priorityqueue¡e¿. `http://docs.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html`, 2011. Found: 29/5 - 2013.

[7] Aiste Pilvinyte, Christian de Haas, and Mikkel Boysen. Indoor space data: A prototype for extraction and interpretation. *n/a*, 2012.

[8] Open IFC Tools. Open ifc java toolbox v1.0.1. `http://www.openifctools.com/Open_IFC_Tools/login.php`, 2010. Found: 30/5 - 2013.

[9] Wenjie Yuan and Markus Schneider. inav: An indoor navigation model supporting length-dependent optimal routing. *In 13th AGILE Int. Conf. on Geographic Information Science*, 2010.

[10] Jun Zhang, Dimitris Papadias, Kyriakos Mouratidis, and Manli Zhu. Spatial queries in the presence of obstacles. *ACM Transactions on Database Systems (TODS) Volume 36 Issue 2, Article No. 9*, 2011.

Appendix A

CD-ROM

The CD-ROM contains the following:

- A Java Project @ \Application\
- An Android Application Project @ \Application\
- A Dynamic Web Project @ \Application\
- IFC files @ \IFC files\
  - AC11-Institute-Var-2-IFC.ifc (2.770 MB)
  - Cassiopeia.ifc (10.816 MB)
  - Clinic_A_20110906_optimized.ifc (12.889 MB)
  - Dds_BardNa.ifc (42.729 MB)
  - HITOS_070308 Elevator.ifc (62.594 MB)
  - Nem-FZK-Haus-2x3.ifc (10.157 MB)
  - Office_A_20110811_optimized.ifc (4.004 MB)
  - Statsbygg-HIBO-ARK-20080410.ifc (66.951 MB)
- A PDF file @ \Report\

The Java project contains the source code for the back-end of the prototype except from the web service.
The Android Application Project contains the source code for the navigation app.
The Dynamic Web Project contains the source code for the web service.

The IFC files are the ones used for the mapping re-evaluation made in Section 5.2.3.1 and the experimental evaluation made in Chapter 8.

The PDF file is a digital version of the report.

Appendix B

Decomposition

Algorithm 3 Original Decomposition

function ORIGINAL DECOMPOSE(Region r, a set of turning points P, threshold T_{shape})
  if r is concave **then**
    let $R(r)$ be the **MBR** of r;
    select a *turning point* $t \in P$ on r's boundary, such that t is closer to the middle of r;
    draw a splitting line perpendicular to the longer dimension d to divide r into two or more regions: $\{r_i\}$;
    for each r_i in $\{r_i\}$ **do**
      Decompose(r_i, $P - \{t\}$, T_{shape});
    end for
  else
    if $\frac{len(R(r)_1)}{len(R(r)_2)} > T_{shape}$ or $\frac{len(R(r)_1)}{len(R(r)_2)} < T_{shape}$ **then**
      find the middle point m on r's longer dimension d;
      draw a splitting line perpendicular to d to divide r into two regions: r_1 and r_2;
      Decompose(r_1, P, T_{shape});
      Decompose(r_2, P, T_{shape});
    end if
  end if
end function

Appendix C

Back-end UI

Figure C.1: *An example from the IFC file "HITOS", which includes many partitions without any nearby access points.*

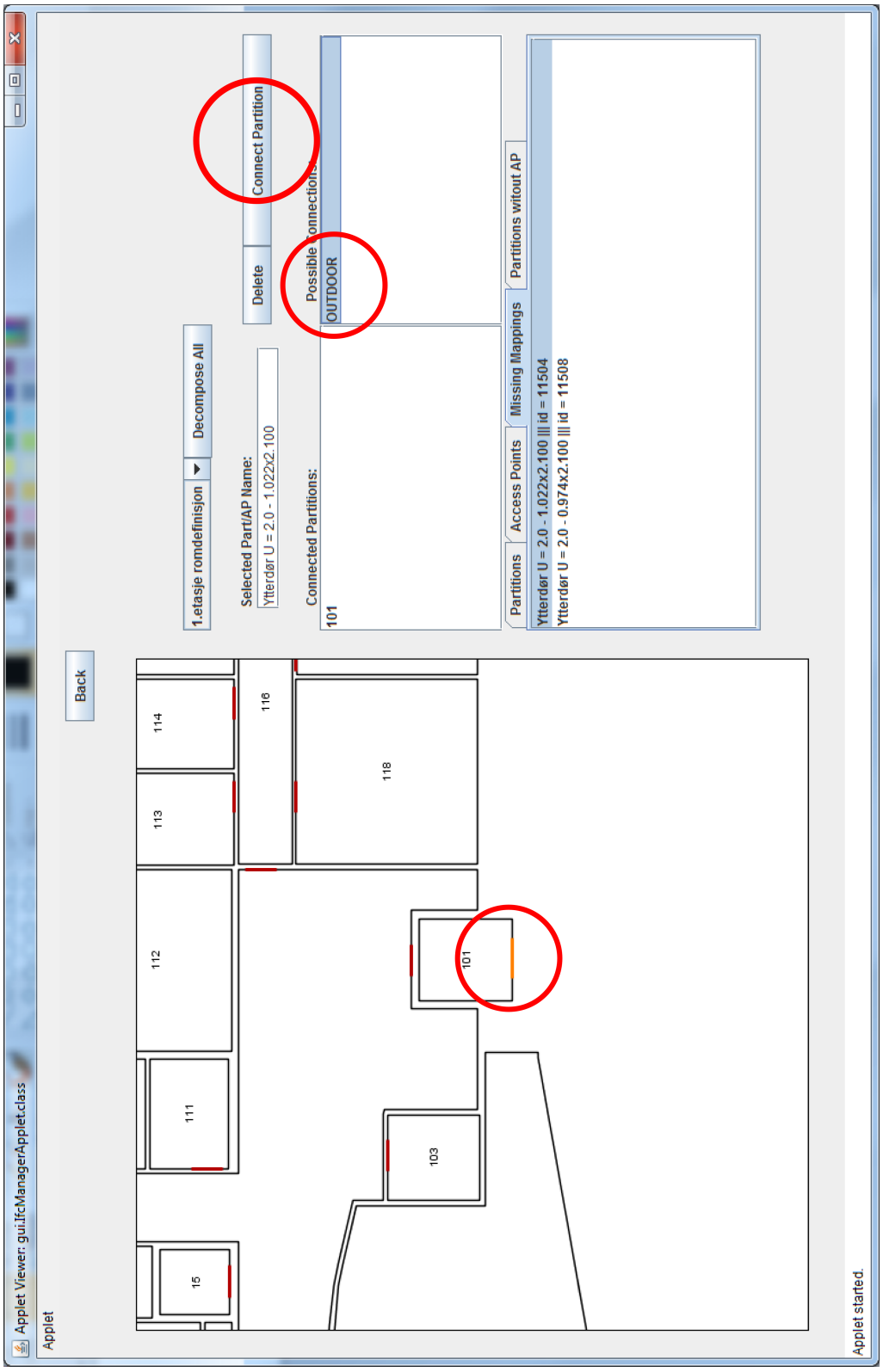


Figure C.2: *An example of how to connect an access point, which has only one partition mapped, to the outdoor partition via the back-end UI.*

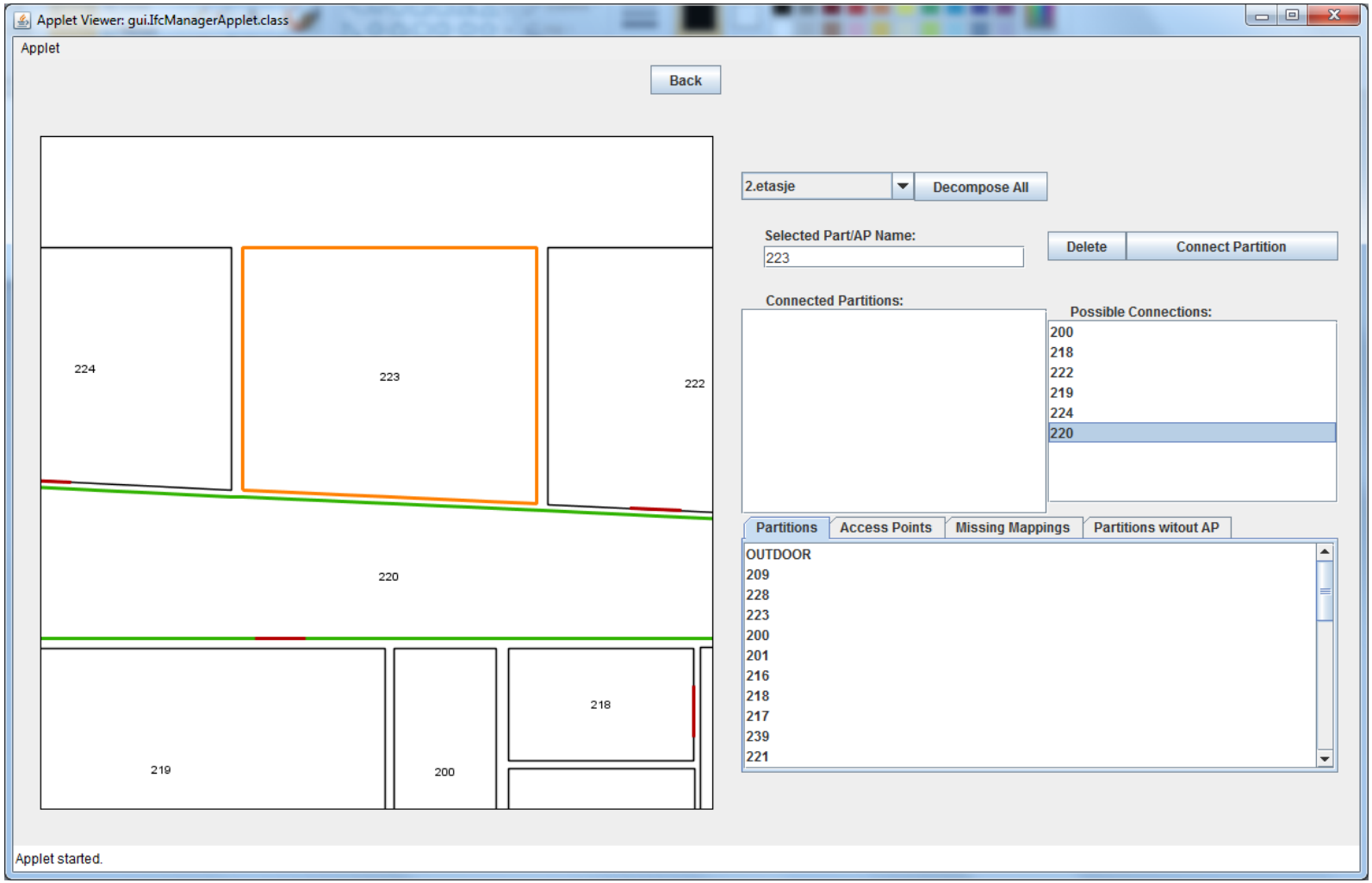


Figure C.3: *An example of how to connect two partitions by inserting a new access point via the back-end UI. The selected partition is highlighted in orange and the partition highlighted in green is the one selected from the list of partitions that are possible to connect to.*

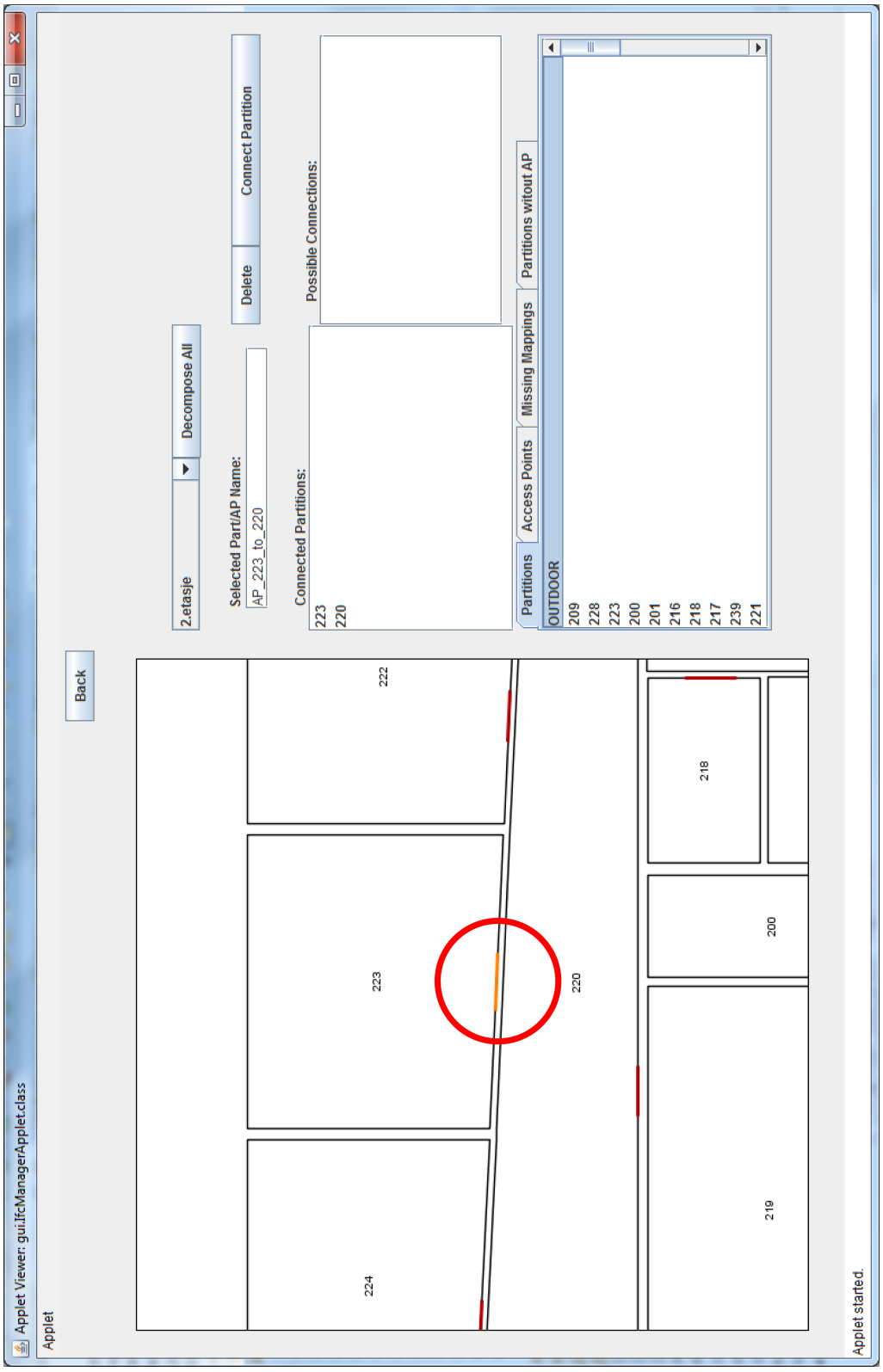


Figure C.4: *This shows the access point created from connecting the two selected partitions in Figure C.3.*

Appendix D

Connectivity Trigger

```
1 CREATE TRIGGER trig_a2p_connectivity
2   BEFORE INSERT
3   ON aptopart
4   FOR EACH ROW
5   EXECUTE PROCEDURE ins_connectivity();
```

Listing D.1: *This is the trigger which inserts a row into the Connectivity table.*

```
1 CREATE FUNCTION ins_connectivity()
2  RETURNS trigger AS
3  BEGIN IF tg_op = 'INSERT' THEN
4  IF EXISTS (SELECT part_id AS partid FROM aptopart WHERE ap_id =
       NEW.ap_id ) THEN
5  INSERT INTO connectivity(ap_id, part1_id, part2_id) SELECT NEW.
       ap_id, part_id, NEW.part_id FROM aptopart WHERE ap_id = NEW.
       ap_id;
6  END IF;
7  RETURN new;
8  END IF;
9  END
```

Listing D.2: *This is the trigger function used in the trigger defined in Listing D.1.*

Appendix E

Routing Algorithm

Algorithm 4 **pt2ptDistance3**(source position p_s, destination position p_t)

1: $v_s \leftarrow$ getHostPartition(p_s)
2: $v_t \leftarrow$ getHostPartition(p_t)
3: $doors_s \leftarrow P2D_{\sqsubset}(v_s)$
4: $doors_t \leftarrow P2D_{\sqsupset}(v_t)$
5: **for** each door $d_s \in doors_s$ **do**
6: $np \leftarrow$ the partition in $D2P_{\sqsubset}(d_s) \setminus \{v_s\}$
7: **if** $P2D_{\sqsubset}(np) = \{d_s\}$ **and** $np \neq v_t$ **then**
8: remove d_s from $doors_s$
9: **end if**
10: **for** each door $d_t \in doors_t$ **do**
11: $dists[d_s][d_t] \leftarrow \infty$
12: **end for**
13: **end for**
14: $dist_m \leftarrow \infty$
15: **for** each door $d_s \in doors_s$ **do**
16: $doors \leftarrow \emptyset$
17: **for** each door $d_t \in doors_t$ **do**
18: **if** $dists[d_s][d_t] = \infty$ **and** $dist_V(p_s, d_s) + dist_V(p_t, d_t) < dist_m$ **then**
19: add d_t to $doors$
20: **end if**
21: **end for**
22: initialize a min-heap H
23: **for** each door $d_i \in \Sigma_{door}$ **do**
24: **if** $d_i \neq d_s$ **then**
25: $dist[d_i] \leftarrow \infty$
26: **else**
27: $dist[d_i] \leftarrow 0$
28: **end if**
29: enheap($H, \langle d_i, dist[d_i] \rangle$)
30: $prev[d_i] \leftarrow$ null
31: **end for**

```
32:     while H is not empty do
33:         ⟨d_i, dist[d_i]⟩ ← deheap(H)
34:         if d_i ∈ doors then
35:             doors ← doors \ {d_i}
36:             if dist_m > dist_V(p_s, d_s) + dist[d_i] + dist_V(p_t, d_i) then
37:                 dist_m ← dist_V(p_s, d_s) + dist[d_i] + dist_V(p_t, d_i)
38:             end if
39:             (v, d_j) ← prev[d_i]
40:             while d_j ≠ d_s do
41:                 if d_j ∈ doors_s and d_j > d_s then
42:                     dists[d_j][d_i] ← dist[d_i] − dist[d_j]
43:                     if dist_m > dist_V(p_s, d_j) + dists[d_j][d_i] + dist_V(p_t, d_i) then
44:                         dist_m ← dist_V(p_s, d_j) + dists[d_j][d_i] + dist_V(p_t, d_i)
45:                     end if
46:                 end if
47:                 (v, d_j) ← prev[d_j]
48:             end while
49:             if doors = ∅ then
50:                 break
51:             end if
52:         else if d_i ∈ doors_s and d_i < d_s then
53:             for each door d_j ∈ doors do
54:                 dists[d_s][d_j] ← dist[d_i] + dists[d_i][d_j]
55:                 if dist_m > dist_V(p_s, d_s) + dists[d_s][d_j] + dist_V(p_t, d_j) then
56:                     dist_m ← dist_V(p_s, d_s) + dists[d_s][d_j] + dist_V(p_t, d_j)
57:                 end if
58:             end for
59:             break
60:         end if
61:         mark door d_i as visited
62:         parts ← D2P_⊏(d_i)
63:         for each partition v ∈ parts do
64:             for each unvisited door d_j ∈ P2D(v) do
65:                 if d_j ∈ P2D_⊏(v) then
66:                     if dist[d_i] + G_dist.f_d2d(v, d_i, d_j) < dist[d_j] then
67:                         dist[d_j] ← dist[d_i] + G_dist.f_d2d(v, d_i, d_j)
68:                         prev[d_j] ← (v, d_i)
69:                     end if
70:                 end if
71:             end for
72:         end for
73:     end while
74: end for
75: return dist_m
```