Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Offline Navigation for Pedestrians on Android Devices

Maksym Zinnatov

Course of Study:	Informatik (B.Sc.)
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Prof. Dr. Stefan Funke
Commenced:	February 14, 2019
Completed:	July 14, 2019

Abstract

Applications for street navigation are widely used and leverage the massive publicly available map data sources. On the other hand, applications for pedestrian navigation, especially navigation within buildings, are few and far between.

This paper describes the development of a native Android app for pedestrian navigation, including the preparation and representation of navigation data in the form of a graph, design of the graphical user interface, and selected implementation details.

Based on floor plan images and building on the graph data generation tool developed in the course of a previous research project, the navigation data for buildings in the Vaihingen campus of the University of Stuttgart were modelled as an abstract graph. The application allows the user to specify the start and goal points, and finds an optimal route between them. The performance and scalability potential of the app was assessed in a benchmarking process.

Contents

1 Introduction

1.1 Pedestrian Navigation and Its Challenges

Over the last two decades, navigation applications have become extremely widespread. This mainly applies to street navigation, where, especially since the introduction of GPS handsets[12] and incorporation of positioning chips in every smartphone, users are now able to plan a route between locations in nearly every populated area of the globe. On the other hand, applications for pedestrian navigation, especially navigation within buildings, are few and far between. In addition, the coverage of these applications is also limited to a relatively small amount of buildings located mostly in large cities or university towns.

There are several reasons for this discrepancy, the primary factor being the vast difference in the amount of map data available: While street navigation applications can draw on a wealth of publicly available data sources, which have been built up over decades (e.g. commercial applications such as Google Maps, or community-built open data projects such as OpenStreetMap), indoor navigation applications have no comparable basis of building data to draw on. Both Google Maps and the OpenStreetMap, for example, provide indoor data only for a tiny subset of popular buildings. Therefore, to make indoor navigation practicable on a larger scale, it is necessary to first reduce this data deficit. For the purposes of this paper, the necessary data were acquired based on 2D floor plans using a tool developed in the course of a research project [17].

Another challenge in indoor navigation is accurate positioning. Whereas outdoor positioning is highly successful with global navigation satellite systems (GNSS), and almost every commercially available smartphone has GNSS capabilities, satellite navigation cannot be used effectively within buildings due to weak signal reception and signal attenuation through walls and floors [9]. Instead, there is no single sensor that can provide acceptable accuracy for navigation indoors in all cases [10]. Other means and sensors need to be used for indoor positioning, such as WLAN fingerprints, Bluetooth beacons, RFID tags, inertial measurements, or visual markers. [11].

This paper attempts to demonstrate an example of a smartphone application for pedestrian navigation, including the tools for collection and processing of data required for its operation in an easy to use and extensible format. The data for this demonstration are based on the floor plans of buildings in the Vaihingen campus of the University of Stuttgart.

1.2 Technologies Used

The main programming language used in the development of the pedestrian navigation application for this paper was Java (JDK 8), with Android Studio v2.2.2 used as the IDE. The Java part of the project does not use any external dependencies other than those provided with the Android Studio by default. Some implementation details are discussed in chapter 4.

The editor, which was used to prepare the navigation data as discussed in section 2.4, was developed using JavaScript. JavaScript was selected for this purpose because it enables rapid development of user interfaces and provides a plethora of additional frameworks and libraries. One primary drawback of using a JavaScript-based editor inside a web browser was its limited ability to work with the local file system, which required a number of workarounds.

Python programming language (version 3.6) was utilized for some supporting scripts, such as cropping the building floor plans for the editor, combining smaller graphs together into a single file (as described in section 2.5), adding information on room occupants, as well as for some visualization tasks to aid the editing.

2 Data Preparation

Parallel to the development of the Android application, an important task was to prepare the data for input. Since the navigation problem at hand boils down to graph traversal, an efficient way to represent the graph data was required.

2.1 Representing Graph Data

In practice, there are several popular data structures used to represent graphs. Among the most common are adjacency lists, adjacency matrices, and incidence matrices.[4]

An **adjacency list** stores vertices as records or objects, with a list of connected vertices stored for each vertex. Conveniently, this approach allows saving additional data within the vertex and edge objects/records.

An **adjacency matrix** is a 2D matrix representing the presence or cost of an edge between two vertices as a number in the matrix, where the row represents the source vertex and the column represents the target vertex.

An **incidence matrix** is a 2D Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether a vertex in a row is incident to an edge in a column.

The product requirements do not specify any manipulation of vertices and edges during runtime (i.e. adding or removing vertices or edges), therefore, the only concern was the efficient storage and retrieval of the entire graph. Moreover, the graph is expected to be quite sparse, with the branching factor around 2. This makes an adjacency list, with $O(|V| + |E|)$ time and space complexity of storage, the preferred option.

2.2 Graph Data Format

The graph data is stored in a variant of the FMI graph data format used at the Institute for Formal Methods in Computer Science at the University of Stuttgart (German: FMI)[7]. An FMI file is an example of an adjacency list in the form of a space-delimited, human-readable text containing all of the properties of a single graph. Each file consists of:

- a blank line
- number of nodes
- number of edges

- a line for each individual node, containing:
    0. localID
    1. globalID
    2. x-coordinate
    3. y-coordinate
    4. elevation
- a line for each individual edge (the actual **adjacency list** part):
    0. sourceID
    1. targetID
    2. cost
    3. type
    4. max-speed
- a blank line

An excerpt from an FMI file:

```
...
529                                     # of nodes
1254                                    # of edges
0 100000 263 183 -1             0th node
1 100001 363 133 -1             1st node
...
528 100528 496 406 3    last node
0 176 1 2 3                             0th edge
0 289 1 2 3                             1st edge
...
```

In the current implementation, the localID field is in fact the line number. For a particular node, it may change as other nodes are added or removed. The globalID field, on the other hand, is persistent and uniquely identifies each node. The mapping between the local and global IDs is established when a graph file is read from the disk or saved.

This format was used in the course of the Programming Project[15], and its efficient representation in Java is discussed in section **??**. However, this format does not include any additional data on the nodes, which may be important for navigation, such as the presence of doors, stairs, and elevators, wheelchair accessibility, or points of interest.

Therefore, for the current project, the FMI format was extended by adding a string (in quotes) as the last (index 5) item in the line representing each node.

If necessary, it may contain data in a widespread information exchange format, such as JSON or XML; however, for the sake of simplicity, a simple `key=value;` form was selected.

Examples of tags:

```
indoor=room // the node is a room
indoor=area // the node is in an open area without walls
name=-1.010 // the node is room number -1.010
```

An example of a wheelchair-accessible restroom node with room number -1.010:

```
270 270 675 445 -1 indoor=room;facilities=toilets;wheelchair=yes;name=-1.010;
```

Adding these additional properties greatly improves the usefulness of the graph, as it enables the user to perform a search for the nearest restroom or nearest elevator, or to find out if an area is wheelchair accessible. This information can then be used in any application which can read text files. As with XML, additional tags can be created by the user, allowing the FMIX to be adapted to the particular use case.

2.3 Annotations Used

Since indoor navigation systems are generally developed in isolation for each venue, such as a shopping mall, an office building, or a university campus, there appear to be no widely used standards for encoding useful information in navigable graphs. In order to have at least some potential for consistency with other projects, the key/value pairs used in the application were partly inspired by vocabulary proposed by the contributors of the OpenStreetMap's Simple Indoor Tagging schema.[14]

Some tags used in the application are:

- indoor=room;
- indoor=corridor;
- stairs=yes;
- wheelchair=no;

Doors are an important object in indoor navigation. Often, finding a way to a certain location (e.g. an office, a conference room, or a lecture hall) means finding a way to a door leading to it.

The data produced for the application include door types, e.g.:

- door=hinged;
- door=sliding;
- door=revolving;

According to OpenStreetMap Wiki [13],

> "This is relevant for wheelchair users, walking disabled persons or parents with strollers to choose a door model they prefer."

Incorporating other door features, such as type of doorknob, is also possible; however, acquiring such data on the scale of the university campus would require more field research than appears feasible and thus falls outside of the scope of this paper.

Labeling points of interest is another important aspect of any navigation app. Besides the labels contained in the name tag (e.g. `name=PC Pool`, or `name=S-Bahn`), the data includes tags such as facilities (e.g. `facilities=toilets`) and person indicating the primary occupant of an office (e.g. `person=Mustermann, Max, PhD;`). The allocation of offices can be dynamic and for this reason is updated at the build stage (see section 2.5); it can be at least partly automated using a web scrapper.

2.4 Editor (developed in the course of the Research Project)

This section is largely based on the Research Project (Acquisition and Modelling of Building Plans for Indoor Navigation)[17].

The centerpiece of the graph creation system is the Editor GUI (figure 2.1). It is written in a combination of html and JavaScript, primarily using the JointJS[8] diagramming library, which itself requires the Backbone.js, Lodash and jQuery libraries. In its current form, the Editor GUI requires an internet connection at startup, in order to allow the above-mentioned libraries to be fetched. After the initial fetch, the internet connection is no longer needed.

At startup, the Editor first fetches the necessary libraries, then loads the previously cropped images into the browser, using the building information located in the "data.js"file (see Chapter 3). The user can then either load and edit an existing graph from an FMI(X) file or else start a new graph by clicking anywhere on the building image and creating a first node.

The building images are displayed one level at a time and can be selected using a drop-down menu. Selecting a level also displays the nodes and edges which were created on that level. A special *multi-level mode* allows the user to see not only the nodes on the current level, but also those on levels adjacent to it. This functionality is useful when connecting nodes between levels at stairs or elevators.

In general, the user interface follows a paradigm in which a single-click creates a graph component (node, edge) and a double-click deletes the component. Nodes, for example, are created by single-clicking on an empty space in the image. Once created, nodes can be moved by clicking on and dragging them. An edge between two nodes is created by single-clicking on an existing node, then single-clicking on the node it should be connected to. Deleting a node or edge is accomplished by double-clicking on it. When a node is deleted, all edges connected to it are also deleted.

Graph nodes can be assigned different types at the time of creation via a text box or at a later point in time using a special *Select* mode. Additionally, vertical chains of *stairs* or *elevator* nodes can be created using the *Repeat and join* function; this creates a column of identical *stairs* or *elevator* nodes, one on every level, then links these nodes automatically with edges, enabling navigation between levels.

The above-mentioned *Select* mode, which disables the creation of edges, has two functions. It allows the user to change the properties of a node by single-clicking it and typing or editing the properties in a text box, and it is also used to select the source node when conducting a shortest-path search inside the graph.

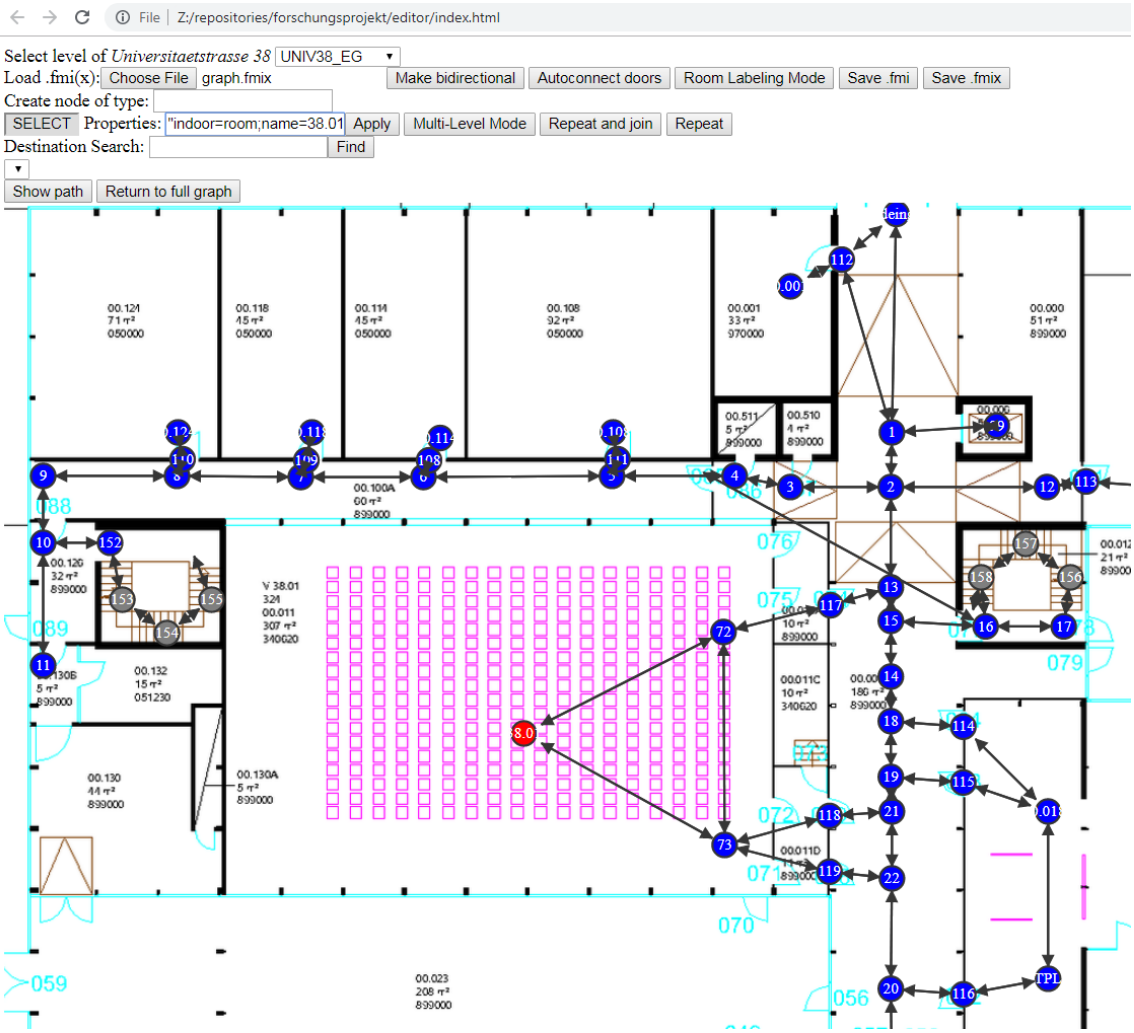

Figure 2.1: Editor GUI

2.5 Connections between Building Graphs and the Campus Graph

As described in the editor chapter, the navigation data comes in logical blocks referred to as buildings. This is a logical block convenient to manage in the editor.

Of course, there does not have to be a 1:1 mapping between physical buildings in the real world and the "buildings" referred to in this context: should the application be used for areas other than the Vaihingen campus of the University of Stuttgart, an actual building may be split into several such blocks. Several buildings may be contained within a single building graph. Even the outdoor section is considered a "building" for application purposes.

However, navigation needs to be performed across all the available graph data. A path will often start at one building (or outside) and end at another; sometimes, the shortest way for a pedestrian would be to go through a building, where such possibility is available.

From the editing point of view, graphs of different buildings are connected by adding an `interface=x;` tag for a given node. As mentioned in section 2.2, each node has a unique "global" ID; these identifiers are used to connect the nodes.

For instance, if node 6 needs to be connected to node 2000001, the developer selects one of these nodes in the editor and labels it with an interface tag, indicating the ID of the other node, e.g.: `interface=20000001;`. Since no real-time management of nodes and edges is required, the actual connections are established at the build stage. A Python script (contained in the `tools/build_global_graph.py` file) invoked during the build process combines all buildings found in the `src/main/assets/maps` directory into a file named `graph.fmix`, checking the data for integrity in the process. When an interface tag is encountered, the script creates a bidirectional zero-length edge between the two nodes uniquely identified by their IDs.

3 Android Native Navigation App

3.1 Input Data Formats

The application requires data in the following formats:

- image files used as floor plans have to be bitmaps in .png, .jpg, or .gif format. [1]
- .fmix graph data files
- text files with additional information, such as `people.csv` used to fill in office allocation data by the Python build script.

3.2 GUI Overview

The GUI consists of the following main components, as illustrated in figure 3.1:

- **The map/floor plan**, covering the entire application screen. It can be manipulated as a typical map application: drag with one finger to pan the view, pinch to zoom in or out. The floor plan can be supplemented by pins that mark the starting location (stick figure), target location (flag), and transitions between buildings (door). Door-shaped pins can be tapped to switch to a different building.
- **Search field**. This can be used to set the starting or target location based on the tags assigned to the nodes, e.g. room numbers, names of the people, or facilities. After a prefix of at least three characters typed in by the user is found in the index, the user is offered a list of potential matches, each entry containing the name of the building, room number (if available), and the tag that matched the description. By default, the selected list item is set as the goal of the route.
- **Toolbar**:
    1. “advance"button. This moves the user position (starting location) along the path to the current target.
    2. “start"button. This switches the interface to the “starting location” mode, so that tapping the map or running a search sets the starting point of the route.
    3. “goal” button. This switches the interface to the “set target” mode, so that tapping the map sets the end point of the route.
    4. “center” button. This centers the viewport on the starting location.
    5. “up” and “down” buttons. These switch the view to a different floor.

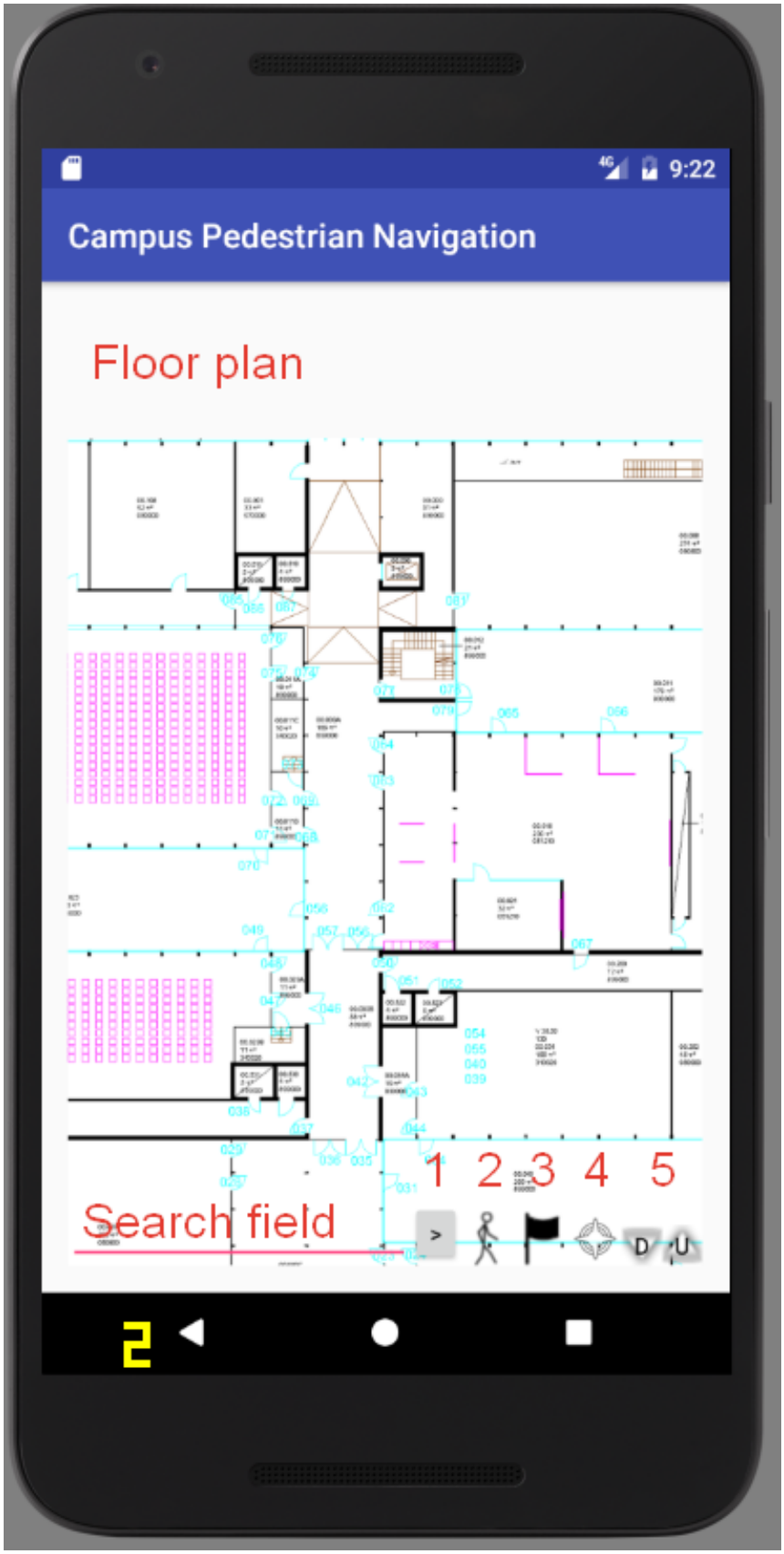

Figure 3.1: User interface

Figure 3.2: starting location set

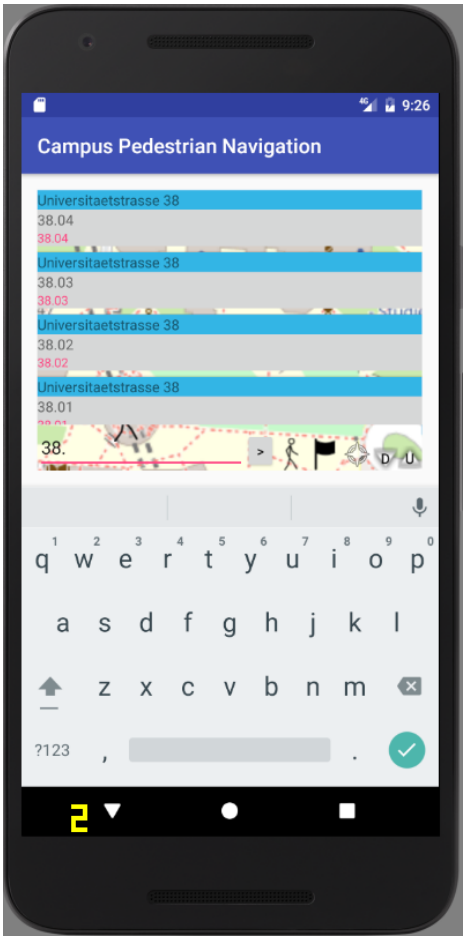

Figure 3.3: searching for rooms beginning with "38."

3.3 User Manual

Typically, the first step is selecting one's starting point. There are several possibilities to do that:

- To select the starting point on the current map screen: tap the "start" button, then tap a location on the map.
- To select the starting point based on search results: tap the "start" button, enter a query into the search field, then tap one of the results in the list.
- To set the starting location by reading an NFC tag: if your device has the Near Field Communication (NFC)[2] capability and a physical NFC tag is available at your location, hold your device next to it (typically at 4 cm or closer), and the application will attempt to process it. If the hex value of the tag is found in the navigation data, the starting location will be set to the corresponding point.

Figure 3.4: the outdoor part of the route, including a *door* icon

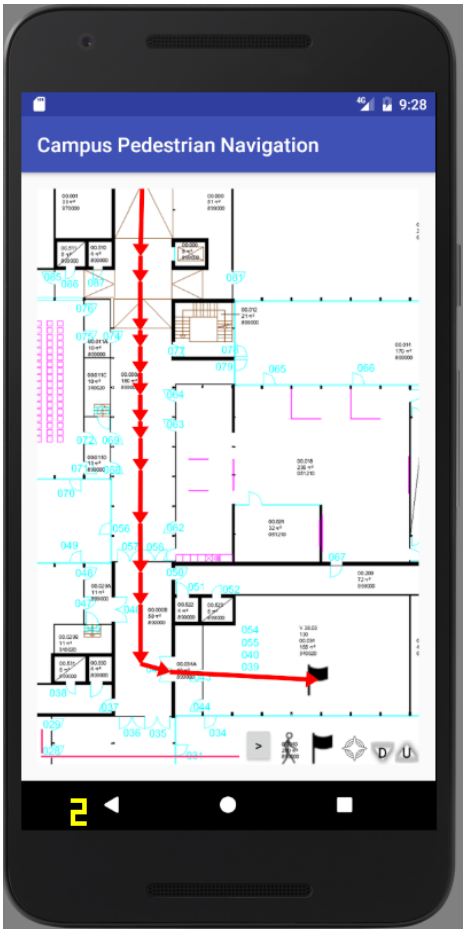

Figure 3.5: the indoor part of the route, including the *goal* icon

After the starting location has been selected by one of the ways described above, a stick-figure-shaped icon will appear on the corresponding map location, and the viewport will be centered on it (fig. 3.2).

The next step is to select the destination. Again, there are two ways to accomplish this:

- To select the destination on the current map screen: tap the “goal” button, then tap a location on the map.
- To select the destination based on search results: enter a query into the search field, then tap one of the results in the list (fig. 3.3).

If the selected destination is accessible from the starting location, the shortest path to it will be plotted on the map in red arrows. If the destination is located in a different building (or if the path goes through a building), the path will be interrupted by door-shaped icons (fig. 3.4). Tapping such an icon will switch the map view to the floor plan of the desired building.

The goal will be marked with a flag-shaped icon (fig. 3.5).

Table 3.1: Performance and scalability tests

Graph	Nodes	Edges	Read time	
			device, ms	emulator, ms
Actual navigation data (early version of the campus map)	4 646	8 989	96	17
Same graph, additionally populated with random nodes and edges (expected maximum number of nodes for the current project)	30 000	47 230	264	99
Same graph, additionally populated with random nodes and edges	70 000	107 620	534	222
Same graph, additionally populated with random nodes and edges (extreme example)	105 000	160 857	826	465

To explore the path, use the pan and/or zoom features of the map screen. Enter the buildings by tapping door icons, or advance on the path step by step by pressing the “advance” button.

At any time, you can set a new starting location or destination.

3.4 Performance and Scalability

Performance and UI responsiveness are important factors that determine the overall user experience. One of the goals for development of this application was to keep the loading and response times low.

An efficient implementation of reading and storing the graph data is discussed in chapter 4. Since the pedestrian navigation app was intended to deal with relatively small graphs (tens of thousands of nodes, at most), some of the loading efficiency was traded for quick retrieval of text data (such as room number tags) from indexes represented internally as Java HashMaps. To assess the overall performance of the app, accounting for potential further expansion of data, several performance tests were conducted. The tests were performed on a *Huawei P20 lite* device (Android build 9.1.0.132, 4.0 GiB RAM), and, additionally, on an Android emulator (run on an Intel i5-3570k@3.40GHz, 16.0 GiB RAM)

Table 3.1 and figure 3.6 describe the data sets used for the performance testing and the results obtained.

As expected, the loading time is roughly linear in the number of nodes. The total time required to read the graph data remains under 1 second even for the most extreme example, and is under 300 ms for the number of nodes actually expected to be used by the application.

The time required to perform a Dijkstra search over the entire graphs is very low (tens of milliseconds).

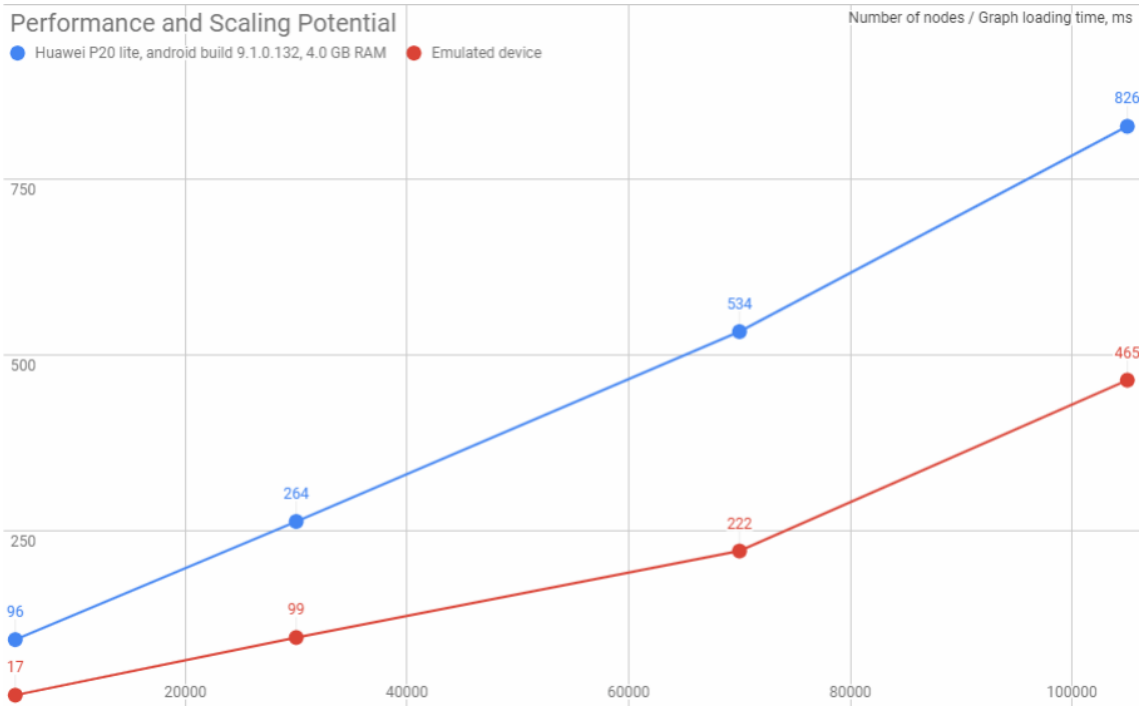

Figure 3.6: Performance and scalability

4 Selected Implementation Details

4.1 Application Structure

The application is ready for distribution as an Android package (.apk file). Internally, it follows the typical structure of an Android Studio project [3]. Within the module folder, the src/main subfolder contains, among others, the following parts:

- `AndroidManifest.xml`: the xml file describing the components of the application;
- `java/` Java source code of the application;
- `res/` Resources used by the application, such as pin icons (referred to as drawables).
- `assets/` Android uses this folder for resources structured in a file system. Leveraging this ability to structure such resources in a hierarchical fashion, the application uses this folder as the source of all map data, grouped on a per-building basis. Each subfolder in the `assets/maps/campus` directory contains the following information on one building:
  - `data.json` contains the information about the building and each of its levels. For the entire building, it stores the scale (`pixelsPerMeter`), the building id (used as a prefix in the global numbering of graph nodes), and the building description. For each level, it contains the level description (e.g. "Ground Floor"), the level number used for the ordering of levels (the ground floor being level 0 and underground floors having negative numbers), and the relative path to the level image file.
  - `graph.fmix` contains the graph information, as described in section 3.1
  - image files, referred to in `data.json`, are used as the floor plans, one image per level.

Unit tests (used to test and demonstrate the functionality of the major parts of the program, such as reading in the graph file or performing a Dijkstra search), together with the test data, are located in the `src/test` subfolder.

4.2 Efficient Representation and Reading of Graph Data

This paper partly builds on the results achieved as a part of the Programming Project ([15]), which involved developing a web-based route planner for the road network of Germany. One of the challenges was the efficient reading of graph data presented in the .fmi format (as described in 3.1) and representing it in memory. Considering the size of the data (the text file of 2.71 GiB contained 25 115 477 vertices and 50 790 030 edges), it had the potential to become a major performance bottleneck.

The object-oriented approach, in which the nodes would be represented as Java objects and edges as references to them, would have entailed a prohibitively large memory and performance overhead.

The solution was to use arrays of primitive types (`int[]` and `float[]`). Several arrays were used to store node information (id, latitude, longitude); one array contained edge information (for an edge number `i`, the array items with indexes `i*3`, `i*3+1`, and `i*3+2` contained the source node number, target node number, and edge weight, respectively); and the offset array was used to keep track of the edges starting from a given node, e.g. in the following example,

index	value
0	0
1	3
2	3
3	6

the 0^{th} node is the source of edges 0,1,2; the 1^{st} node has no outgoing edges; the 2^{nd} node is the source of edges 3,4,5; and the edges 6 and above (if such edges exist) start at node 3.

This representation is demonstrated in the `Graph` class of the current project.

The collections used in the Java standard library operate on objects and not primitive types (e.g. `Integer` instead of `int`). Since the Dijkstra algorithm involves a priority queue, the approach described here necessitated a priority queue class that works with primitive types (see the `BetterPriorityQueue` class). Its implementation is similar to an ArrayList in that the underlying structure is an array (more precisely, two arrays, one for keys and one for values of the priority queue elements). When the number of items in the queue approaches the size of the array, the `resize()` method is called. Resizing the array in this context actually means creating a new array of a larger size and copying the data from the old array to the new one. This procedure is efficiently implemented in the standard library as the `java.util.Arrays.copyOf()` method.

4.3 Model-View-Presenter Pattern

Architecturally, the application loosely follows the Model-View-Presenter Pattern. This is a popular extension of the Model–View–Controller pattern[6]:

- the Model is responsible for storing and managing the data;
- the View presents the data to the user; and
- the Presenter acts as the intermediary between the Model and the View, querying the Model to retrieve or modify the data, and instructing the View to update the presentation.

Figure **??**, showing several most important classes, illustrates the implementation of this philosophy in the project.

Here, the presentation logic, contained in the PanZoomView class discussed in detail in 4.4, is decoupled from the data storage and retrieval logic encapsulated in the `Graph` class. The `NavigationController` class acts as the Presenter: it processes the user input received from the View, decides how the input affects presentation, querying the `Graph` class for navigation data if necessary, and instructs the View to update the presentation accordingly.

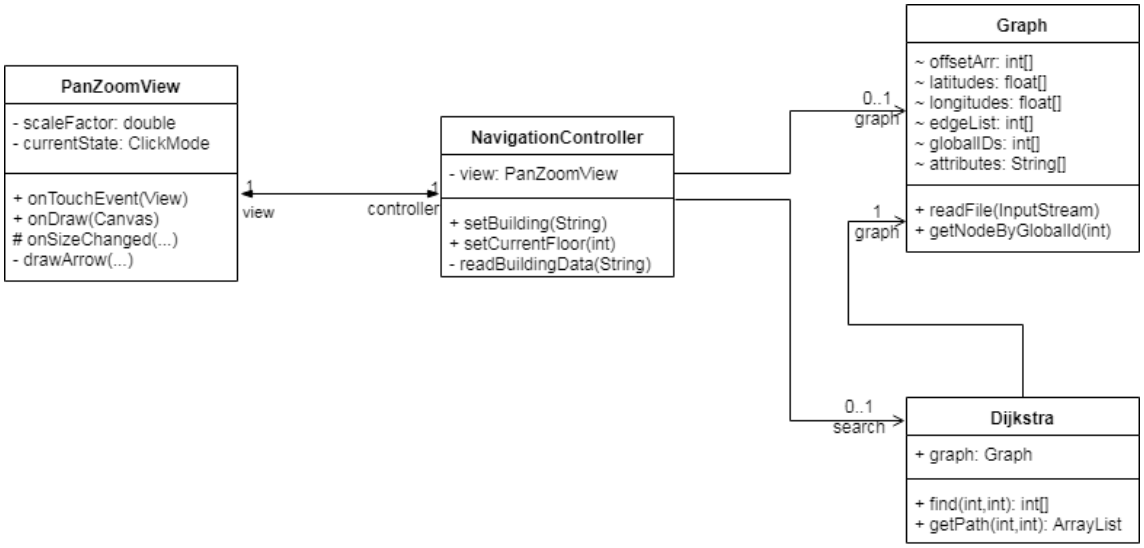

Figure 4.1: Model–View–Presenter architecture

For example, when the View receives a click event intended to set the navigation target:

- it passes the coordinates to the Presenter's `setTarget()` method;
- the Presenter queries the Model to retrieve the node nearest to the given coordinates and plan the shortest routes to such node;
- the Presenter's `updateArrows()` and `updatePins()` methods instruct the View to plot the new path and mark the target on the canvas.

The View stores the presentation state (the current viewport and the zoom factor), whereas most of the application state (the current path, current building, etc.) is maintained by the Presenter.

4.4 Android Canvas and the Custom View

The main part of the application's user interface is presenting the user with a building plan and drawing a route on top. The application implements this functionality using the Android Canvas (`android.graphics.Canvas`) class within a custom view.

The Canvas API provides an efficient way to present custom drawable elements [16]. In this case, those included the background image (building floor layout), the path (arrows), and pins (images marking the source and the target, as well as the transition nodes between buildings and levels).

An important feature of the canvas is the ability to apply transformations to the coordinate system (such as translation, rotation, and scaling), as well as saving and restoring the intermediate states. Translation and scaling help handle the user's gestures to change the image scale and to move the viewport:

```
canvas.translate(corner.x,corner.y);
canvas.scale(scaleFactor,scaleFactor);
```

after such transformations, all the drawable bitmaps are placed on the canvas, e.g.:

```
canvas.drawBitmap(floorPlanBitmap, 0,0, null);
```

with translation and scaling applied.

The ability to save the coordinate system before the transformations and restore it afterwards is useful in drawing custom geometry. The following code fragment is used in the View's `drawArrow()` method (the result is shown in figure 4.2):

```
canvas.save();
canvas.rotate(angle,from.x,from.y);
canvas.translate(from.x,from.y);

Path path = new Path();
path.moveTo(0, 0);
path.lineTo(0, -ARROW_WIDTH);
path.lineTo(length - ARROWHEAD_LENGTH, -ARROW_WIDTH);
path.lineTo(length - ARROWHEAD_LENGTH, -ARROWHEAD_WIDTH);
path.lineTo(length, 0);
path.lineTo(length - ARROWHEAD_LENGTH, ARROWHEAD_WIDTH);
path.lineTo(length - ARROWHEAD_LENGTH, ARROW_WIDTH);
path.lineTo(0, ARROW_WIDTH);
path.close();

canvas.drawPath(path, paint);
canvas.restore();
```

Drawing a custom path is easy to implement due to a fixed reference frame. Without the transformation, the code draws a horizontal arrow facing to the right. The rotate and translate methods transform the reference frame, so that no changes need to be made to the path coordinates.

Since the reference frame is saved before drawing an arrow and restored after that, multiple arrows can be drawn back-to-back by repeatedly calling the same method.

PanZoomView is a custom class that inherits from `android.view.View`. Its purpose is to present the canvas filled with the drawable elements described earlier in this section, as well as to provide the intuitive pan and zoom functions based on the user gestures.

To implement this functionality, the custom view class overrides the following methods of its parent, `View`: `onTouchEvent(MotionEvent event)` handles the user actions, such as (referred to by the constants defined in the android.view.MotionEvent class):

- `ACTION_DOWN` and `ACTION_UP` detect a touch and release by a single finger, respectively; depending on the current UI state, these events either activate/deactivate the panning motion or signal the placement of source/target pins.
- `ACTION_POINTER_DOWN` and `ACTION_POINTER_DOWN` represent the same for multiple fingers;
- `ACTION_MOVE` is used to perform the panning.

Additionally, the `onTouchEvent` method passes the event to an instance of android.view.ScaleGestureDetector class, which handles any "pinch" gestures interpreted as zooming in or out. It changes the scaleFactor variable used in the `onDraw\lstinline` method.

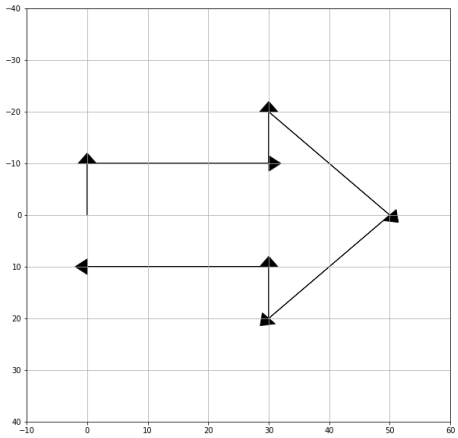

Figure 4.2: arrow in a standard reference frame

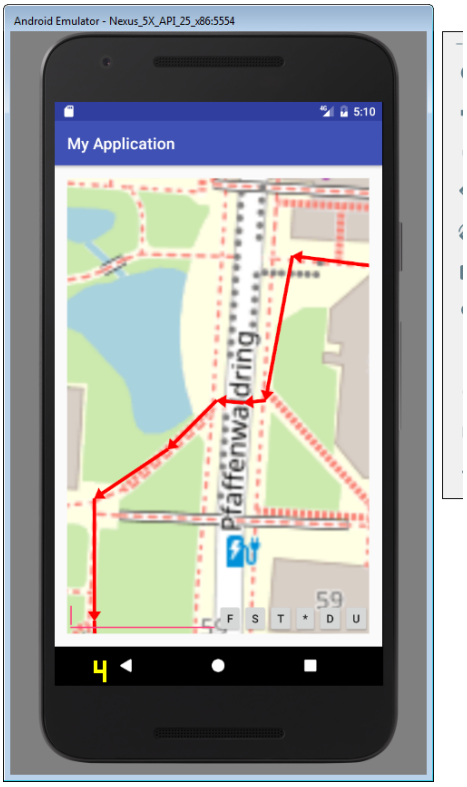

Figure 4.3: multiple arrows drawn using the same code with canvas rotation and translation

`onDraw(Canvas canvas)` contains the drawing logic (drawing the background bitmap, arrows, and pins). It is responsible for updating the reference frame using the `Canvas.translate` and `Canvas.scale` methods. This method is called whenever the view's `invalidate()` method is called, i.e. the view needs to be redrawn. The pan and zoom actions, for example, call `invalidate()` after the viewport has changed.

4.5 Pathfinding (Dijkstra Algorithm)

An important part of any navigation application is finding the shortest route from a source to a destination. This project implements it using Dijkstra's algorithm, a well-known algorithm for finding the shortest paths between nodes in a graph [5].

The implementation of this algorithm is contained in the Dijkstra class, which is instantiated for an instance of the Graph class. As soon as the source node is known, the `find()` method can be called to generate the shortest distance tree from the given node to all reachable nodes.

To do this, the algorithm:

1. Initiates the arrays to store the distance to the ith node (defaulting to `Integer.MAX_VALUE`) and the previous node on the path to it from the source, a boolean array to track the nodes that have already been visited, and a priority queue that only contains the start node at the beginning.

2. While the queue is not empty:
    - remove the nearest node *next* from the queue and mark it as visited.
    - for each neighbor of the *next* node: if the distance to such neighbor via the *next* node (`newDistance`) is less than the previously calculated distance to such neighbor, update the *distance* and *previous* arrays with this information; insert the neighbor to the queue with a priority equal to newDistance.

```
init(start);
while (!queue.isEmpty()) {

   int next = queue.pop();
   if (visited[next]) {
      continue;
   }
   visited[next]=true;

   if (next == goal) {
      valid = goal;
      return distance;
   }

   int ubound;

/* implementation details skipped in the listing: nodes with indexes between getOffset(next) and
     ubound represent the neighbors of node next */

   for (int edgeListIndex = getOffset(next);
               edgeListIndex < ubound;
               edgeListIndex = edgeListIndex + 1) {
      int neighbor = getTarget(edgeListIndex);
      int newDistance = distance[next] + getWeight(edgeListIndex);
      if (newDistance < distance[neighbor] && newDistance >= 0) {
         distance[neighbor] = newDistance;
         previous[neighbor] = next;
         edgeToPrevious[neighbor] = edgeListIndex;
         queue.insert(neighbor, newDistance);
      }
   }
}
```

Implemented with a binary heap (see `BetterPriorityQueue` in section 4.2), it has a time complexity of $O((|E| + |V|) \log |V|)$, where $|E|$ is the number of edges and $|V|$ the number of nodes. In our case, the graph is quite sparse, with $|E| \approx 2 \times |V|$; therefore, it is safe to assume the performance of the pathfinding step to be loglinear in the number of nodes.

5 Summary and Outlook

5.1 Summary

In this paper, we attempted to address the problem of pedestrian navigation, focusing on indoor navigation, by developing a native Android application that can be used offline and would use a simple and extensible format of navigation data. The application enables the user to perform searches and wayfinding across multiple buildings of the University of Stuttgart in Vaihingen, and demonstrates the sufficiency of available floor plan data to provide navigation capabilities after manual processing with the tools described above.

5.2 Outlook

At its current state, the application provides a relatively limited set of features. However, the extensibility of the data format allows adding functionality to adapt it for various purposes. Tagging the nodes with accessibility information could prove immensely helpful for users with mobility problems. Other tags can be introduced to accommodate any existing navigation infrastructure: the current version demonstrates basic NFC reading capabilities, whereas other options for indoor positioning may include reading QR codes or checking Wi-Fi fingerprints. Drawing a path on a floor plan appears to be a task-oriented implementation, which may not be the most convenient option for an end-user. An opportunity for improving the user experience would be incorporating visual navigation cues.

Bibliography

[1] *Android Developers Guide: Drawable resources*. URL: https://developer.android.com/guide/topics/resources/drawable-resource (cit. on p. 15).

[2] *Android Developers Guide: Near field communication overview*. URL: https://developer.android.com/guide/topics/connectivity/nfc (cit. on p. 17).

[3] *Android Studio User Guide: Projects overview*. URL: https://developer.android.com/studio/projects (cit. on p. 21).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms (Second ed.)* 2001. Chap. Section 22.1: Representations of graphs, pp. 527–531. ISBN: 0-262-03293-7 (cit. on p. 9).

[5] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390 (cit. on p. 25).

[6] M. Fowler. *GUI Architectures*. 2006. URL: https://www.martinfowler.com/eaaDev/uiArchs.html#Model-view-presentermvp (cit. on p. 22).

[7] *Institut für Formale Methoden der Informatik (homepage)*. 2019. URL: http://www.fmi.uni-stuttgart.de/ (cit. on p. 9).

[8] *JointJS: Visualize and interact with diagrams and graphs*. URL: https://www.jointjs.com/opensource (cit. on p. 12).

[9] K. Kaemarungsi. "Design of Indoor Positioning Systems Based on Location Fingerprinting Technique". PhD thesis. University of Pittsburgh, 2005 (cit. on p. 7).

[10] H. A. Karimi, ed. *Indoor Wayfinding and Navigation*. 2015. ISBN: 9781482230840 (cit. on p. 7).

[11] P. Krishnamurthy. "Technologies for Positioning in Indoor Areas". In: 2015 (cit. on p. 7).

[12] *Mobile Navigation Services Taking Off In North America And Europe*. 2008. URL: http://www.gpsdaily.com/reports/Mobile_Navigation_Services_Taking_Off_In_North_America_And_Europe_999.html (cit. on p. 7).

[13] OpenStreetMap contributors. *Proposed Features: Door*. URL: https://wiki.openstreetmap.org/wiki/Proposed_features/door (cit. on p. 11).

[14] OpenStreetMap contributors. *Simple Indoor Tagging*. URL: https://wiki.openstreetmap.org/wiki/Simple_Indoor_Tagging (cit. on p. 11).

[15] *Programmierprojekt: Ein webbasierter Routenplaner*. 2017. URL: http://www.fmi.uni-stuttgart.de/alg/teaching/w17/ppMoGa/ (cit. on pp. 10, 21).

[16] P. Ryan Harter. "[droidcon SF] droidcon SF 2017: Canvas Drawing for Fun and Profit – retrieved from https://www.youtube.com/watch?v=H05mF0qrBVA". In: 2017 (cit. on p. 23).

[17] D. Schmitter, M. Zinnatov. “Bachelor-Forschungsprojekt-INF: Acquisition and Modelling of Building Plans for Indoor Navigation”. 2019 (cit. on pp. 7, 12).

All links were last followed on July 3, 2019.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature