NTNU – Trondheim
Norwegian University of
Science and Technology

Automatic Generation of Walkable Paths in Buildings to Support Indoor Wayfinding

Ole Mikkel Sjølie

Master of Science in Informatics
Submission date: April 2013
Supervisor: John Krogstie, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

We have created a prototype system that generates walkable paths in a building using AutoCAD floor plans. Our system is designed to work with the CampusGuide, a commercial indoor navigation system. Path-generation is done once for each floor plan, creating a network of walkable paths between all points of interest. The A* algorithm we have implemented takes roughly fifteen seconds to build paths for a building floor, which is much quicker than what a human can do manually. The error rate is fairly low, but still an issue for commercial use.

Sammendrag

Vi har laget et prototypesystem som genererer gangbare veier gjennom en bygning ved hjelp av AutoCAD-plantegninger. Systemet vårt er utviklet for bruk med CampusGuiden, et kommersielt system for innendørs navigering. Sti-generering blir gjort én gang per plantegning og skaper et nettverk av gangbare veier mellom utvalgte steder. A*-algoritmen vi har implementert bruker omtrent femten sekunder på å konstruere veier per etasje, mye raskere enn det et menneske kan gjøre manuelt. Feilraten er relativt lav, men fortsatt et problem for kommersiell bruk.

Preface

This is the documentation of my Master's Thesis in Informatics at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The thesis has been written for Wireless Trondheim.

The help from my supervisors has made this project possible. I wish to thank my supervisor at NTNU, professor John Krogstie, for his support and direction throughout this project. I would also like to thank my co-supervisors at Wireless Trondheim, Gunnar Rangøy and Åsmund Tokheim, for the support and guidance received from them as well.

I am also grateful to Aksel W. W. Eide for suggestions and reviewing.

Ole Mikkel Sjølie

Contents

Contents **vi**

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**
1.1 Motivation 1
1.2 Research Questions 2
1.3 Report Description 2

2 Background **3**
2.1 Indoor Navigation 3
2.2 Relevant algorithms 8

3 Research method **9**
3.1 Design Science 9

4 Requirements **13**
4.1 Functional requirements 14
4.2 Non-functional requirements 14

5 Implementation **17**
5.1 Tools used 18
5.2 Creating a maze 19
5.3 Shortest Path 23

6 Evaluation **29**
6.1 Functional Requirements 29
6.2 Non-functional Requirements 35
6.3 Output results 39

7 Discussion **47**

8 Conclusion and future work **49**
8.1 Conclusion 49
8.2 Future work 50

Bibliography **51**

A Line generation pseudo-code **53**

B Contents of ZIP-attachment **55**

List of Figures

2.1 Indoor navigation at EPFL 4
2.2 Google Maps indoors at Copenhagen Airport 5
2.3 Screenshot from the CampusGuide at NTNU 6
2.4 Screenshot from QuantumGIS showing paths 7

5.1 Use case 18
5.2 Doors method 1 21
5.3 Doors method 2 21
5.4 Square algorithm 22
5.5 The possible neighbours of a square. 23
5.6 Wall costs 25
5.7 Paths all-to-all 26
5.8 Reduced amount of nodes needed 27
5.9 Lines generation algorithm 28

6.1 Connecting a new path to already existing paths 31
6.2 Door improperly placed 32
6.3 Possible corrections we would have done 33
6.4 Lack of connections 34
6.5 The graphical user-interface for visualization 36
6.6 Close-up of the critical section from building 17. 42
6.7 Example of all paths in building 15. 43
6.8 Example of all paths in building 33. 44
6.9 Example of all paths in building 2. 45

List of Tables

6.1 Performance comparison 37
6.2 Output results 40
6.2 Output results 41

Chapter 1

Introduction

1.1 Motivation

In the last few years indoor navigation has become more widespread. Services like the CampusGuide [1][2][3] and Google Maps Indoors [4], launched in 2011, allow their users to get indoor directions on a map via a web browser. While previously a paper map could be used, showing all destinations in a building, one can now get exact paths directly to the room being searched for instead. This makes it possible to make maps that are more relevant for each individual user, by for example showing a line from the users current position to their goal.

All the routes in a building need to be created somehow. Creating walkable paths manually is a time-consuming process and with indoor navigation becoming more widespread, there could potentially be much time saved by automating the work. There are several open-source tools to do the work manually, but as there seems to be no definitive standard being used when creating floor plans, it's difficult to automate for all types of floor plans. The aim of this thesis is to make a prototype system that automatically creates paths through floor plans based on the information found in The Norwegian University of Science and Technology's (NTNU) floor plans. These floor plans are stored in an AutoCAD DXF file format [5]. We will attempt to solve it in a way that allows it to be easily adapted to work with floor plans from other sources as well.

1.2 Research Questions

1. Is it possible to create a system that can automatically find walkable paths through a building purely from information taken from an AutoCAD floor plan?
2. If it is possible, how does it compare to a human doing the work manually?

1.3 Report Description

The following is an outline of the remaining chapters in the thesis.

Chapter 2: Background This chapter describes systems related to our work.

Chapter 3: Research Method Here we will talk about the method we used to reach the final system.

Chapter 4: Requirements Lists all the requirements that the system has to fulfil.

Chapter 5: Implementation This chapter goes into details of how the system that was made.

Chapter 6: Evaluation The system will be evaluated up against the requirements set in Chapter 4.

Chapter 7: Discussion We then discuss if we have answered any of our research questions.

Chapter 8: Conclusion and Future Work We finish the report with our conclusion and talk about possible improvements to our system.

Chapter 2

Background

This chapter will be describing systems involving indoor navigation and how far the technology has come.

2.1 Indoor Navigation

More powerful mobile devices, along with more people having access to these devices, has given indoor navigation a significant boost the last years. This allows more people to swap out the regular paper map with mobile applications providing the same or better use. These mobile devices often include technology like WiFi, Bluetooth and cameras, all of which can find the user's position indoors. Having easier access to indoor navigation makes these systems more likely to spread to more buildings in the future, therefore we consider it highly relevant for our system.

A mobile application can provide advantages over paper maps for example by giving each user more personalized directions. If a user needs directions that are navigable with a wheelchair, a mobile application can be designed to only give appropriate paths. The same goes for users not wanting to use elevators, and the paths generated could make sure to only use doors that are currently open for use. If parts of a building is unavailable due to construction work, the paths leading there could be re-routed without the need to make a new map. In addition to this, the users will not have to search to find the physical paper map, but can instead make use of the mobile application directly on their phone, computer or tablet.

One of these indoor navigation systems can be found at École polytechnique fédérale de Lausanne (EPFL), an university in Switzerland. EPFL has a website titled the EPFL Orientation Tool [6], allowing people to search for directions on the EPFL campus from one room to another. An example of a path can be seen in Figure 2.1. The very straight lines in the path make us believe that there is some sort of automation that has made these paths. We have tried to contact EPFL about their system, but have received no response.

Figure 2.1: Indoor navigation at EPFL showing directions from room *BC 124* to *INR 130*.

Another relevant service is Google Maps, which was launched in the beginning of 2005. This service started out with outdoor maps, but recently, in November 2011, added the possibility of indoor maps [4]. Currently the indoor service has over 10000 floor plans available, in places like airports, railway stations, museums, universities and more [7]. Even though Google provides outdoor navigation, our tests on their indoor service did not show any directions indoors. An example from Copenhagen Airport can be seen in Figure 2.2. We see it as likely they are going to expand their directions to be working indoors at some point.

The next indoor navigation system we will talk about is the CampusGuide [1], made public in 2011, created by Wireless Trondheim which initiated this project. The key features of The CampusGuide is as following:

1. **An interactive map**, through a HTML5 website, showing all buildings on the Gløshaugen campus. By clicking on these buildings, a user can view any

Figure 2.2: Google Maps indoors at Copenhagen Airport

of the buildings floor plans. A website also enables users to easily access the service on huge amounts of today's technical devices.

2. **User positioning**, utilizing GPS and WiFi to position users on a map both indoors and outdoors. This lets users that are not familiar with the campus get an easier overview, in addition to this, it provides a good starting point when looking for directions.

3. **Indoor and outdoor directions**, which allows a user to search for over 13000 rooms in over 60 buildings, spanning over 350.000 square metres of space. Users can also find directions to the nearest study area, computer lab, parking place, toilets and more.

4. **Shareable links**, which lets users share directions to a location over e-mail, instant messaging and similar.

An example of the CampusGuide can be seen in Figure 2.3. The feature that is the most interesting to us, is the indoor navigation. If we would want to expand the

system to be available at other locations, just displaying the floor plans for some buildings would not take very long to set up, as building architects have already created the necessary floor plans. But once you also want indoor navigation, possibly around in hundreds, or thousands of rooms, it starts being very time-consuming as much more manual work to create walkable paths is needed. The 13000 rooms that are searchable on Gløshaugen, have all been manually mapped into a network of connected points. This mapping was done in QuantumGIS (QGIS), an open-source geographic information system. In QGIS we placed points inside every room or at important locations for all buildings, and connected these points to create a network of paths that represents where people can walk. An example showing these networks can be seen in Figure 2.4.

Figure 2.3: Screenshot from the CampusGuide at NTNU, showing directions through a building at Gløshaugen.

If a room contains a point that is connected to this path network, users can get directions to that room. Therefore it is important to include paths to as many rooms as possible. The CampusGuide uses a shortest-path algorithm on the network of

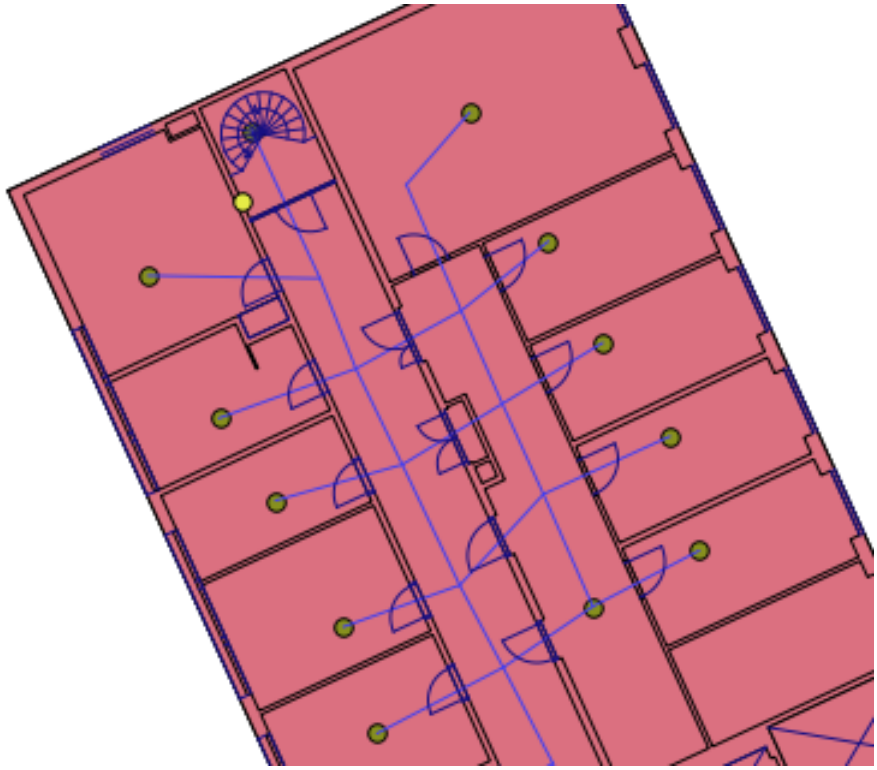


Figure 2.4: Screenshot from QuantumGIS showing paths. Green circles represent points, while the blue lines represent connections between them.

paths to get directions from any room included in this network, to any of the other rooms.

All the previous systems have been based on technology visually showing floor plans or directions on a screen, but there are also systems whose goal is to lead robots around in buildings or through mazes [8]. These robots often automatically navigate through the use of equipped sensors, for example cameras being able to spot nearby objects and measure the distance to objects. These systems can be used purely to navigate robots around in a building, or even to help vision impaired users [9]. Another notable application of these algorithms to solve mazes, has been to calculate the size of circuit boards by making sure all components fit on the boards. We think several of the methods used for robots will be relevant in a system that automates paths through a floor plan.

One significant difference is that robots often do all their work in real-time, while the system we create will pre-process all the information that later will be used for routing. A system for vision impaired users or robots also has a higher validity requirement. For example imagine if a blind user or a robot is being lead into walls or unknowingly down stairs, it will have bigger ramifications than unimpaired users trying to navigate a room. Either way, lower validity will lead to more displeasure from the users, and possibly more manual work to correct the mistakes.

2.2 Relevant algorithms

Before making our system, we wanted to discover other areas that could be using relevant methods that we could apply to our system. We knew that a multitude of games and mazes would have a similar need for path finding. Games often have game units navigating through buildings or maps, similar to what we aim to do in floor plans.

In the first-person shooter game called F.E.A.R., Jeff Orkin and Monolith Productions have explained [10] that they use an algorithm called A* [11] (pronounced A-Star) for choosing the actions enemy units should use, and also to plan the paths units will walk. A* is a best-first search that implements heuristics to reach its goal faster and can be used for a wide amount of purposes.

A* is also already implemented in the CampusGuide to find the shortest-paths between points, which works well and quickly calculates paths, therefore we found it logical to continue looking into this algorithm for our system.

Chapter 3

Research method

This study is made up by two important pieces. The first is the prototype system being developed. The second is the evaluation of how this prototype system performs.

3.1 Design Science

To create this system we followed the seven design-science research guidelines explained by Hevner et. al [12]. They are as following:

Guideline 1: Design as an Artifact *Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.*

The artifact in our case is the prototype system that will create walkable paths through NTNUs floor plans. It will do this based on the information that have been imported from the AutoCAD DXF files provided by NTNU.

Guideline 2: Problem Relevance *The objective of design-science research is to develop technology-based solutions to important and relevant business problems.*

The prototype will aim to solve the time-consuming process of manually creating paths. As indoor navigation has received much attention in recent years, much due to the rapid growth of high performance smart-phones packed with technology, indoor navigation will likely be expanded to many new locations. Therefore we hope to create a system that can save a good deal of time for these new locations.

Guideline 3: Design Evaluation *The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.*

A set of functional and non-functional requirements have been created to evaluate the developed artifact. A few of the key points that will be evaluated is the *completeness* of the system, if all relevant paths are generated. Another is the *validity*, to make sure we do not create paths that should not be there. The *performance* of the system will be evaluated, to make sure the speed the artifact operates at is fast enough, and the last point is to make sure the *output* is appropriate to be used for indoor navigation. The design evaluation methods that will be used includes doing controlled experiments on the existing floor plans at NTNU, in addition to running functional testing, to locate failures and defects.

Guideline 4: Research Contributions *Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.*

The contributions from this research will be the prototype artifact. In addition to this, the methods used will be thoroughly described and should provide a suitable foundation for further research.

Guideline 5: Research Rigor *Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.*

We will follow the research methods described here to create the artifact. The evaluation will be done continuously against our requirements. Every time a path is generated, we plan to compare it to the requirements set based on visualization.

Guideline 6: Design as a Search Process *The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.*

Similar research will be used as a starting point, and we will try extract relevant information and apply it to the artifact we create. In addition to this, experimentation will be performed if no relevant research can be found.

Guideline 7: Communication of Research *Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.*

The information presented will be technology-oriented when we feel it is important to explain the methods used more in detail, particularly in the implementation chapter, but will when possible be explained in a way that should be understandable from a management-oriented perspective audience as well.

To begin with, we documented what the prototype system should include. This gave us a starting point of requirements, which we broke up into smaller tasks. By breaking the project into parts, we could focus more on each small task, make sure each of them work as intended, and then step-by-step include completed tasks into the prototype system. Before starting, we did a state-of-the-art research, checking if anyone had made anything similar that we could use for our tasks. That way we could avoid reinventing the wheel and try to improve methods that have already been constructed.

Several of the features of the prototype system were tested in specially made test environments before being imported to the full system. For example, our shortest-path algorithm was tested on an actual generated maze instead of being tested on floor plans, this way we could verify that the algorithm worked as it was supposed to, without worrying if other parts were wrong. When a task was working as intended on its own, it would be ready to be imported to the prototype system.

Tasks were completed based on the requirements we set up. The highest priority tasks were looked at first, but all requirements were kept in mind so we would have a greater chance of satisfying them all. The first goal for completing a task was just to make it work, and not necessarily doing it the fastest or best way possible, but rather spend some iterations on improving requirements we felt needed it after we had a more complete system.

Chapter 4

Requirements

To reach our goal of making a prototype system that automatically finds walkable paths through a building floor plan, we had made certain requirements. Previously Wireless Trondheim had manually created this path data per floor plan using QuantumGIS (QGIS), which was a very time-consuming process. We have together with Wireless Trondheim set up some functional and non-functional requirements to properly automate the process. The requirement ID's will be referenced throughout this report.

4.1 Functional requirements

The functional requirements of the system are tasks that the system is meant to do.

ID	Description	Priority
FR1	The system will take a building id and floor as input. These ids will be pointing to buildings and floors in a PostgreSQL database.	High
FR2	The system will acquire floor plan information from a PostgreSQL database. The information in the database comes from NTNUs AutoCAD DXF files, imported by Wireless Trondheim's import script.	High
FR3	Previously created paths will connect to the new paths generated.	High
FR4	The system will output a set of paths visiting points of interest (POI) in a building. Every room will have a POI, and it is expected that these will be visited if nothing is blocking the way to them. The paths should be valid, that is, not go through walls or blocked passages to reach these POIs. The fewer points in a paths created to reach a POI, the better, as it allows shortest-path algorithms to run faster on the output.	High
FR5	The system will allow the user to accept the output before exporting it to the database.	High

4.2 Non-functional requirements

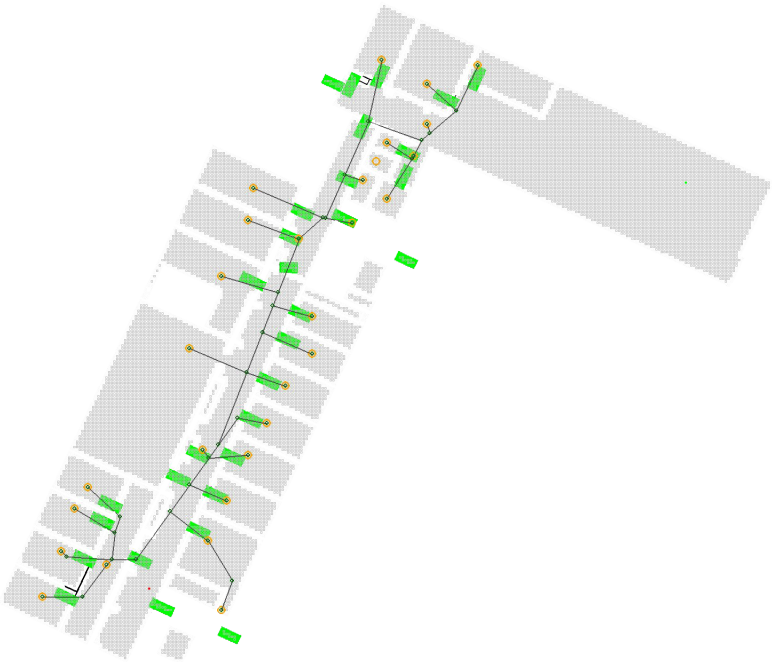
These non-functional requirements describes how the system should be when solving the functional requirements.

ID	Description	Priority
NFR1	The system shall be able to run on a normal desktop computer.	High
NFR2	The system shall be using a graphical user-interface for visualization.	Medium
NFR3	The performance of the system shall be faster than a user doing the same task. This is a tricky requirement as there are so many different types of buildings, some may take five minutes for a user to complete, while others take hours. The goal for this requirement is that the system should always do the job faster than a trained user at doing this job, or at least provide a basis for the user to complete the rest of the job manually.	High
NFR4	The system shall support Windows, Mac and Ubuntu.	Low
NFR5	The system shall require minimal user-input.	High
NFR6	The system shall be easily adapted to work with floor plans from sources such as Building Information Modelling (BIM) and plain images of floor plans.	Low
NFR7	The system shall output data that can be used in the CampusGuide without taking any further actions.	High

Chapter 5

Implementation

In this chapter we will talk about how we made and navigated a maze based on information from a floor plan.



Brief introduction

The reason why we want to convert a floor plan into a maze is that there already exists several techniques of solving mazes. Sutherland [8] describes a method for solving mazes by a computer. The mazes Sutherland solved, so called arbitrary-wall mazes, are comparable to what floor plans look like today.

Figure 5.1 illustrates the general use case of our system.

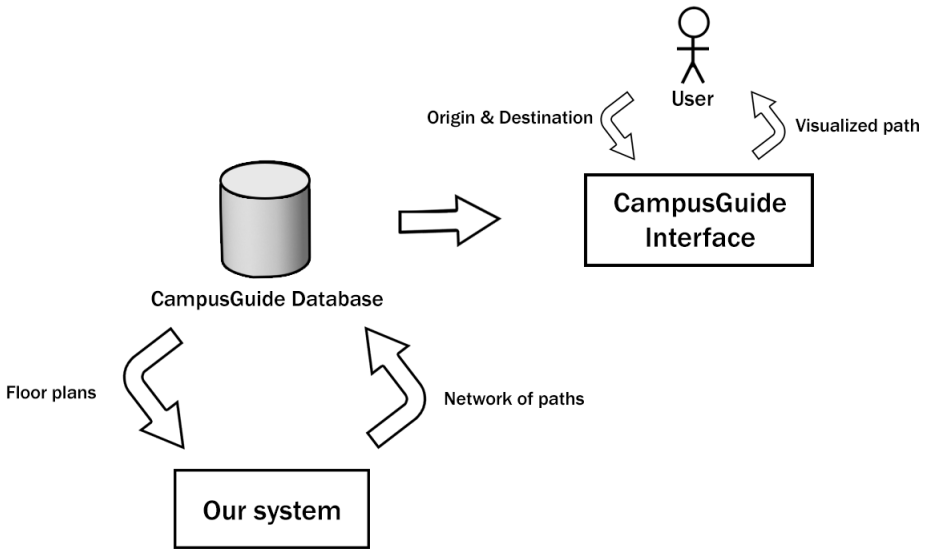


Figure 5.1: Our system will use floor plan information from the database as input, and output a network of paths back to the database. These paths will be used by the CampusGuide to generate visualized paths for users requesting a path from a start position to a destination.

5.1 Tools used

The prototype system that we will be describing in this chapter has been built with Python 2.7. In addition to the standard Python, a few Python libraries have been used, namely pygame [13] and Psycopg [14]. PostgreSQL [15] is the database that stores the floor plan information we have access to. Our installation of PostgreSQL has an extension called PostGIS [16] that adds support for geographic content in SQL queries, allowing us to store and fetch geometries like lines and polygons. PostGIS

also has a wide array of useful geographic functions that we will use to manipulate the floor plan data.

Pygame is a library normally used to make games, though in this system it is used to visualize what is going on with our algorithm. To include scrollbars for our graphical user-interface, we use a library for pygame called ezscroll [17]. The library Psycopg allows Python to access our PostgreSQL database.

5.2 Creating a maze

Rooms and doors were seen as the most important parts of the floor plan to make any progress converting it into a maze. Rooms alone could allow us to navigate only inside that particular room, which could be useful for a conference hall or similar rooms not separated by doors. Once we want to connect separate rooms together, we have a bigger problem. We could connect rooms based on parts of rooms which are nearby each other, though this could lead to going through walls instead of going through doors. With the help of knowing where a door is located in a room, we would also be able to navigate between rooms.

Our plan was to make an image which included the information needed to make the maze, and then navigate the maze based on the information we would be able to extract from this image. The reason behind using an image as the initial storage, is that it would allow us to use the algorithm on images of floor plans as well. It would likely be less useful on an image of a floor plan, than of one made from the information found in an AutoCAD DXF file, but at least our system would not be completely limited to DXF files in the future. Though the system algorithms would run on images of floor plans, it was not a priority to make these work more than as a proof of concept.

The AutoCAD DXF file format, which floor plans have been imported from, is a vector graphics format where elements are stored in layers often separated by name. In a DXF file, every room and door is usually made available as separate line segments or polygons. These elements have been imported into our PostgreSQL database system for ease of use. To further import these polygons into our image, we had to convert every coordinate in the polygons from a map coordinate reference system, to local coordinates. The local coordinates start from 0,0 in the upper left corner of our image. This was done by going through our polygon coordinates and

finding the minimum longitude and latitude value, and we call them offsets. By subtracting these offsets from every longitude and latitude in our polygons, they are displayed in our image with local coordinates. It is important to note that the polygons will not be displayed as they would on a map. A longitude and latitude coordinate at (0,0) centres at the equator, while the local coordinate systems (0,0) would go to the upper left corner. This will not matter other than visually, since upon exporting back to the database the offsets will be added to every coordinate.

Creating the initial image

With this local coordinate system, we used pygame to draw the room polygons onto an image as white entities on a black background and display the image our application window. This makes it easy to separate rooms from the background. By checking the pixel colour at any point (X,Y) of the window, a value other than black (0) means something is drawn there.

Adding doors

To connect the room polygons we tried two methods with varying success. A big challenge with the doors was to figure out which direction a door is facing. For example if a door would be positioned close to a corner of a room, it would be difficult to tell which wall it passes through from looking at the floor plan. Since we at first saw no obvious way of solving it, the first method we used was to create a radius of walkable space around every door. This let us connect room polygons, though it had its flaws. One flaw shows up in the corner case. With this radius method rooms would sometimes mistakenly be connected with other rooms nearby that actually had no door connection. This can be seen from the bottom left case in Figure 5.2.

A possible solution in some cases would be to reduce the radius expanded around doors, though then a new flaw would sometimes appear, namely that other rooms that should be connected, would no longer be connected due to the smaller radius of the doors. There was no easy solution to this for a while. In some cases one could think being able to have too many connections would be more beneficial than missing out on some rooms, but this could lead to more work having to fix the paths manually later.

After some discussions with Wireless Trondheim, we managed to find a much better door solution. The new solution took use of Postgis functions. Since a door

object often consists of two line segments and a curve, we found that we could look for the line segment being the furthest away from a parallel wall. This line segment will in most cases point in the same direction as the door. The line segment could then be used to draw a line with a buffer added around it, to make it appear as a door. If a door object had only one line segment and a curve, we assumed the line segment was pointing in the doors direction, as if the door was fully opened. A sliding door, which consists of only one line segment and no curve, could have a door generated from it by drawing a door in the direction of the normal of the line segment. The area of the doors created were to be considered walkable to be able to connect room polygons. An example of the second method to find doors can be seen in Figure 5.3. This second method provided way better accuracy.

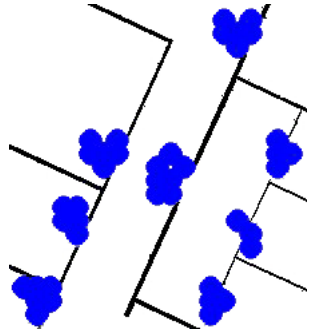


Figure 5.2: Door method 1: Connecting rooms with a radius around doors, shown in blue.

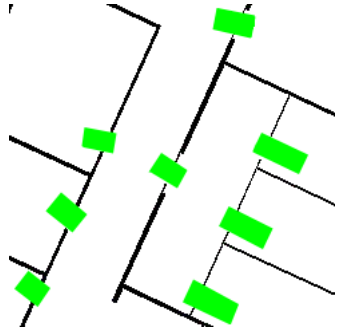


Figure 5.3: Door method 2: Connecting rooms with a PostGIS method of door placement. Doors shown in green.

Squares - the nodes in our graph

Sutherland mentioned in his article that one could break up the available areas into small squares, where every square is connected to its neighbours. He mentioned that although it is an easy way of representing the data, it requires a large amount of computing. With today's computers, this large amount of computing should no longer be a hindrance. To make these squares we exploited the image we had previously made. By going through every pixel in the image, the position of a square of size *width* and *height* will be valid if the area of the square includes no black pixels. An important note here is that we loaded the image into a buffer before fetching pixel colours. Before we used the buffer, pygame would fetch a new copy of the image every time we requested a pixel colour, increasing loading times by a ten-fold. The size of the squares is important. Larger squares would lead to less data to traverse, as there would be less squares in total to fill the floor plans and therefore fewer neighbours. Using smaller sized squares would be more accurate, since there will be a greater amount of positions to navigate to. Larger squares could lead to more narrow parts of floor plans not being filled, as the squares would be too large to fit. Starting with smaller squares, so that the majority of the floor plan area is covered, seems safer. An example of how the squares are placed on a sample image can be seen in Figure 5.4.

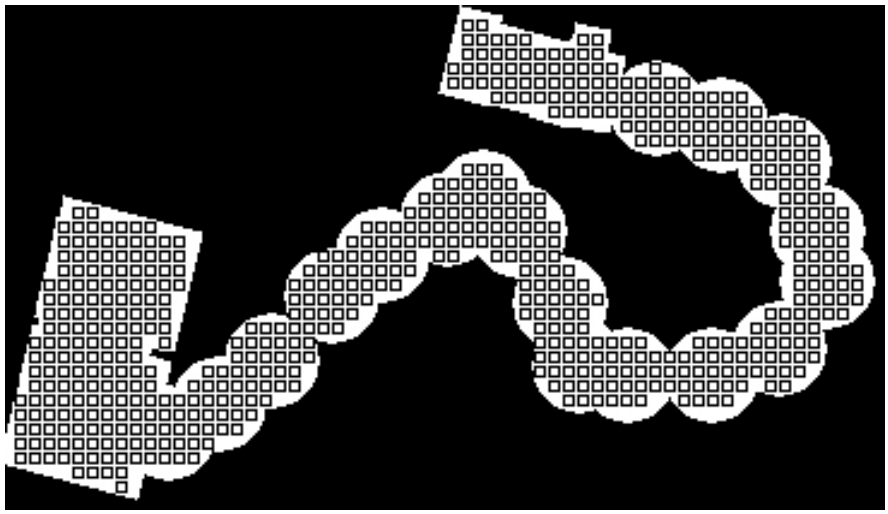


Figure 5.4: Shows how the algorithm places the squares on a sample image.

5.3 Shortest Path

A* Search Algorithm

The search algorithm chosen for finding the shortest path between a pair of points was A* [11]. A* is in widespread use in game development, for example for moving objects in a game grid. A* avoids higher cost squares on the way, if possible, by implementing a heuristics function. If the heuristic never over-estimates the distance to the goal from a certain point, A* will always find an optimal path if one exists. In the case where the heuristic over-estimates, the path found may not be the shortest, but the running time will often be faster. For our purposes, finding the actual shortest path is not critical. However, underestimating the cost avoids the risk of convoluted detours. Therefore we chose to underestimate the cost, even though it lead to slightly increased calculation time. The requirements for A* to run is a set of nodes, these nodes must have a cost and a list of neighbours. To initiate the algorithm, there must also be a start and goal node. In our case a node is equivalent to a square, the cost is the distance and the neighbours are the 8-connected squares found around a square as seen in Figure 5.5.

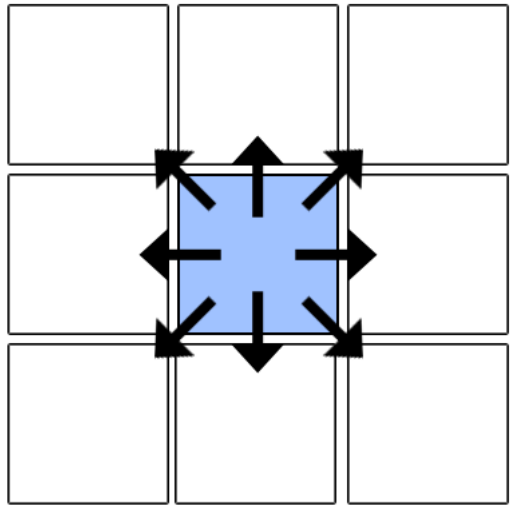


Figure 5.5: The possible neighbours of a square.

A start and goal square was naturally required to make a path, so this also fit into the algorithm requirements. All in all A* was thought to be suitable for solving the maze. According to Russel and Norvig [18], there is no other optimal algorithm

that guarantees that less nodes are visited. Though it has its negative aspects too. A* is memory hungry, as all nodes are stored in memory during execution and it can also use a lot of processing power.

The heuristic function we have implemented for use in A* has been designed to prefer straight paths by adding a slight cost to go diagonally. This cost lets the algorithm avoid going zig-zag instead of going straight. Without the slight cost for diagonal moves, going one step north, then north-east, then north-west would have the same cost as just moving north three times in a row, which creates slightly messier paths. The actual function can be seen below, the max term gives the amount of moves in total, while the min term gives an added cost for the amount of diagonal moves. The relative costs of diagonal and orthogonal moves are approximately the same as for Euclidean distance. Euclidean distance would have been a natural solution, but due to using a discrete grid of squares, this method is simpler and avoids using square roots.

function HEURISTICCOST(x1, y1, x2, y2)
  return $3 * max(abs(x2 - x1), abs(y2 - y1)) + min(abs(x2 - x1), abs(y2 - y1))$
end function

Wall costs

When being navigated around in a building, it was beneficial to have paths moving slightly away from the walls so they were easier to spot instead of moving tightly alongside them, therefore we implemented additional costs for squares with fewer neighbours or squares close to other squares with few neighbours. This was done by subtracting the amount of neighbours of a square from the maximum amount of neighbours possible to have, then multiply the output with a configurable wall cost value. All squares were then iterated over a few times and every square chose the highest wall cost of its neighbours, subtracted by a small amount to let it decrease over distance. Colouring the squares based on their wall cost gives an output that can be seen in Figure 5.6, the higher wall cost the darker red.

Generating Paths

After A* and related algorithms and functions had been set up, it was time to run the algorithm on our graph of squares. Goal nodes could easily be calculated from room polygons, by taking a centre point inside them, for example with ST_Centroid from PostGIS. For the sake of comparison between the manual work and the automatic

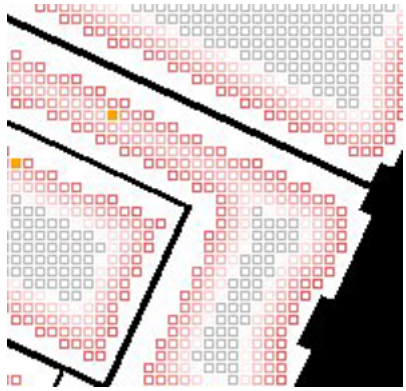


Figure 5.6: Wall costs: Squares with higher cost leads to darker red. Grey squares have no wall cost.

work we used already existing points of interests as goals. At first we would create paths between all goal nodes to all other goal nodes, that way we would create all possible paths between the rooms on the floor plan. An example of this can be seen in Figure 5.7. This was slower than necessary and it turned out it was easy to improve the speed while still maintaining sufficient quality.

Running A* from every goal to four other random goal locations gave very similar results while spending a much shorter amount of time on computing the paths. While these paths were routing well, they used a vast amount of squares in total. Representing every square in the database would lead to the shortest path search for the whole building having to search through more nodes than it should need to. For a small building this would not have a big impact, but for a whole city it would make a huge difference. The first method we utilized to reduce the amount of nodes these paths would generate was to decrease the path cost of already used paths. This lead to new paths more often using squares already existing in previously calculated paths. There was still too many squares used for each path, a bigger change had to be done. The next method tested was to check for paths with straight lines with three or more squares in a row, with no neighbouring squares to other paths. This method let us remove all nodes between the start and end points of the new line generated, as shown in Figure 5.8. A problem was that it didn't handle paths with short zig-zag patterns or paths with many neighbouring nodes from other paths.

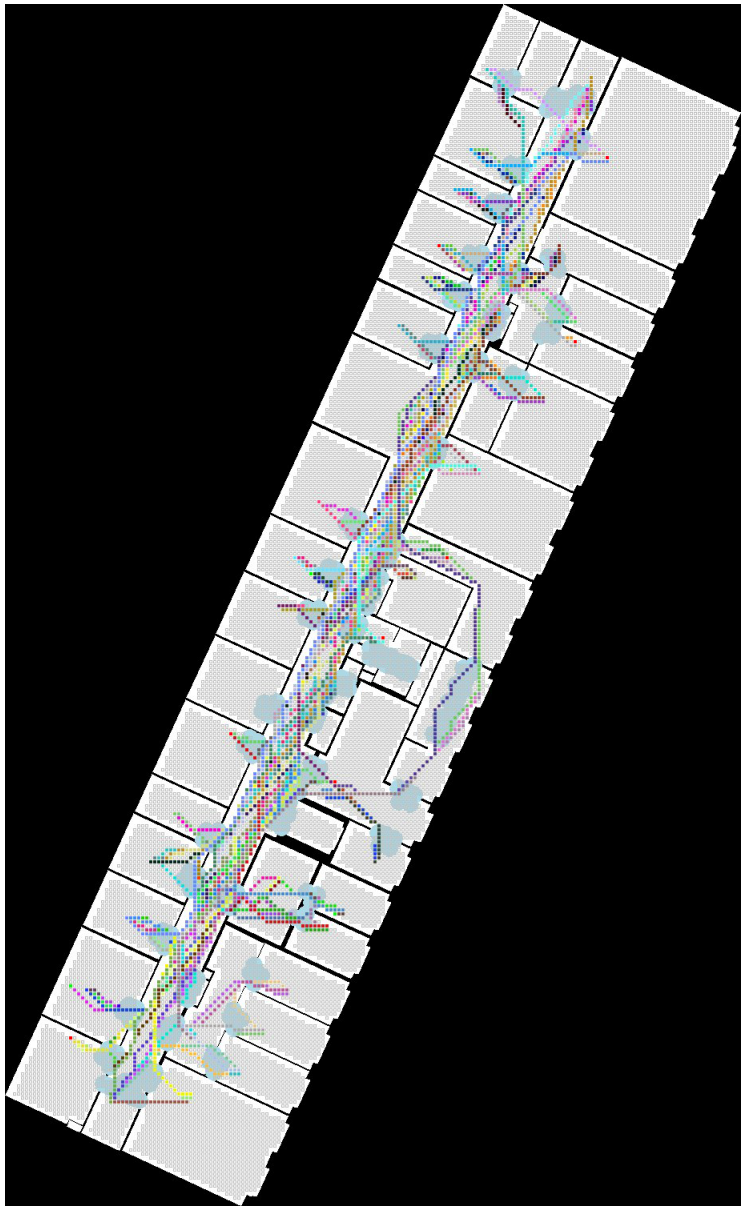


Figure 5.7: An example of all to all paths. Separate colours represent separate paths, though many paths overlap each other. The worse door positioning method was still in use here.

We thought there would be a possibility to improve this further by using what A* returns as a guide rather than putting any of it in the database. Previously all our

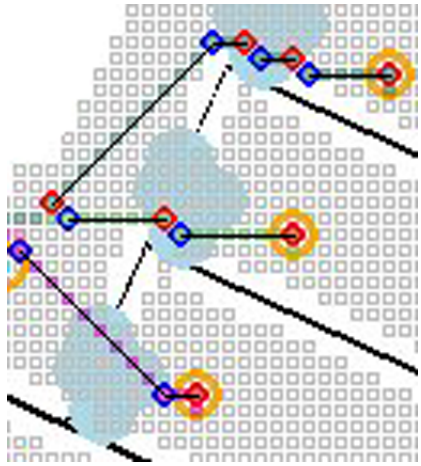


Figure 5.8: Example of reducing the amount of nodes needed by extracting straight lines in paths. Black lines connect the red and blue circles, allowing us to ignore all squares along the line. In this small sample over 30 nodes were removed.

paths would start from a random destination and go towards the goal. There could be benefits by starting at the goal and from there look for paths to hook onto. The focus should be on creating paths with few points when possible.

When starting at the goal square, we looked through the path generated by A* in reverse. By looking in reverse and starting with the goal square, we could later make shorter paths to already generated lines and often avoid going through the whole A* result. For every square in the reverse path we would test if a line could be made from the goal square to the square in the A* path. This was done with the help of Bresenham's line plotting algorithm [19]. Bresenham's algorithm finds every point between two points, which allowed us to test if a path crossed a wall. If a path passed no walls, it was considered valid. We iterated through the reverse path and found the first square where the path was invalid from the goal to the given square. Then we would create a line from the goal to the square before the invalid one, and use this square as a new starting point. From this new starting point the algorithm continues to scan through the rest of the path until it hits another square creating an invalid line, it creates a line to the previous square and the algorithm repeats itself until a line hits the destination square. While this method creates very good paths from a goal to a destination, it does not connect lines to each other. A visualization of the algorithm ran from one point to another can be seen in Figure 5.9.

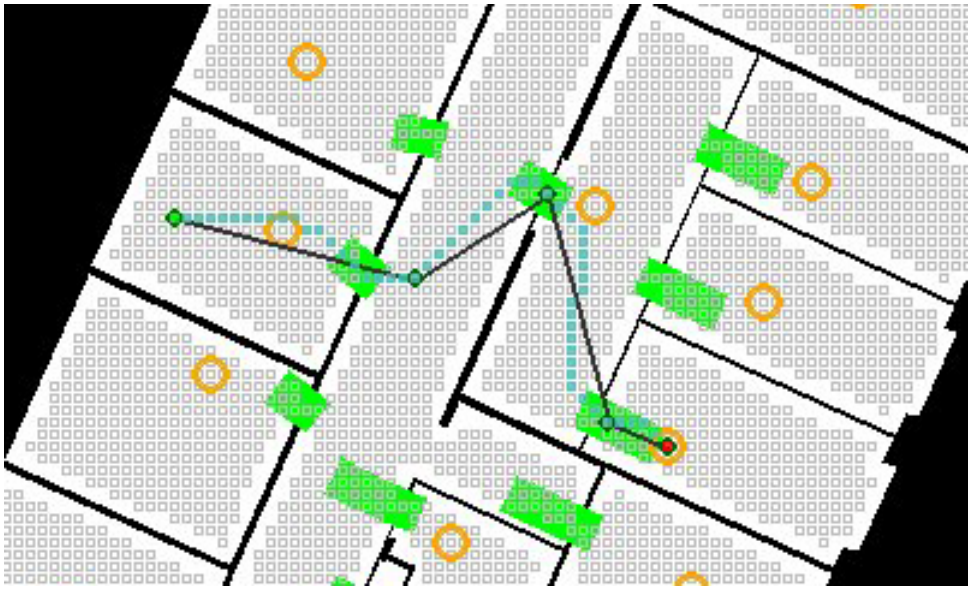


Figure 5.9: Lines generation algorithm. The A* path can be seen as the teal squares, starting from the green circle on the left, to the red circle on the right. The line generation starts from the red circle and backtracks guided by A* back to the green circle.

To make connections between the lines an additional step was added while scanning for invalid paths. In this step we iterated through all points on every line. We checked if a valid path existed between any of the points and the current square in the path. If valid paths were found, a connection would be made to the closest valid point on the line by splitting it at that point and then terminating the algorithm. Any recently added lines, were made sure not to be targets for the current path, otherwise paths would just connect to their own lines. We had decent end results with this algorithm, albeit it was slow when there was many long lines. To fix this we adjusted the algorithm to only check every 25th point on every line, as a pixel perfect closest point was not necessary. Pseudo code of the line generation algorithm can be seen in Appendix A, Algorithm A.1.

The full source code for the system can be found in the attachment described in Appendix B.

Chapter 6

Evaluation

In this chapter we will evaluate the prototype system we have made, up against the requirements for this system. The requirements we are evaluating against can be seen in Chapter 4.

6.1 Functional Requirements

FR1

ID	Description	Priority
FR1	The system will take a building id and floor as input. These ids will be pointing to buildings and floors in a PostgreSQL database.	High

The system solves FR1, only a building id and floor number is required to begin generating paths. The system is easily adapted to use the id of any buildings on NTNU Gløshaugen, either through modification of the source code, or by using command line arguments. It requires less input than what would be necessary in QGIS to load the same specific floor, as QGIS requires the user to manually write filters with SQL.

FR2

ID	Description	Priority
FR2	The system will acquire floor plan information from a PostgreSQL database. The information in the database comes from NTNUs AutoCAD DXF files, imported by Wireless Trondheim's import script.	High

This requirement was solved by using *Psycopg* to connect and fetch polygons, doors and points of interest from the PostgreSQL database. This prototype system does not involve the import script, as we felt it was better to separate the process of importing the floor plans to the database and automatically generating paths. It is important to note that if the database structure changes, the relevant SQL in this system also has to be updated accordingly. Most of the system does not use SQL to run, so there are only the import and export parts of the system that need to be updated.

FR3

ID	Description	Priority
FR3	Previously created paths will connect to the new paths generated.	High

FR3 was solved by giving existing points and paths their database ids, that way we could easily keep track of new points and also keep the existing points in the database. By checking if POIs had existing paths connected to them, we could also exclude generating new paths to them in our code. This was tested by importing previously existing data from the database and checking if we could connect paths to it, an example of this can be seen in Figure 6.1.

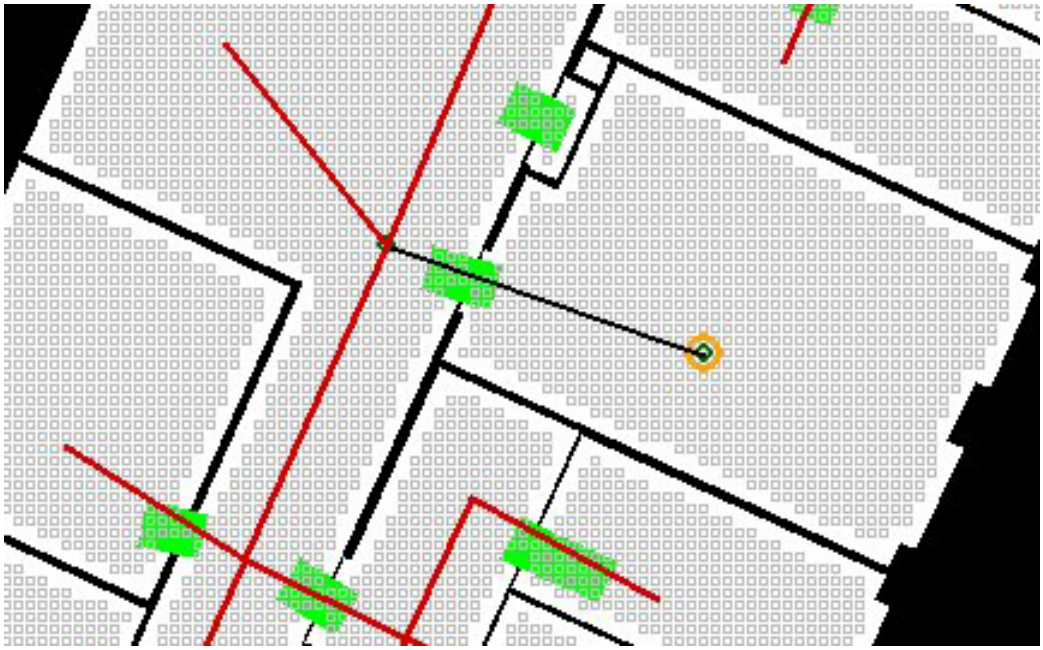


Figure 6.1: Connecting a new path to already existing paths. Lines coloured red are already existing paths, the black line is a new generated path.

FR4

ID	Description	Priority
FR4	The system will output a set of paths visiting points of interest (POI) in a building. Every room will have a POI, and it is expected that these will be visited if nothing is blocking the way to them. The paths should be valid, that is, not go through walls or blocked passages to reach these POIs. The fewer points in a paths created to reach a POI, the better, as it allows shortest-path algorithms to run faster on the output.	High

To satisfy FR4s requirement for paths to be valid, we had to connect rooms with doors to make sure paths did not go through walls.

Doors This is why it was important to us to make a good method of importing doors. The final door method gave correct results in most cases, but it does happen that some doors are improperly placed. When a door was improperly placed it usually leads to paths not being generated to rooms as can be seen in Figure 6.2. In bad cases this could lead to whole areas of a floor being unexplored. This has not yet happened in our testing, but if it would ever be a problem, one could add a door manually to the spot where it happens, and the rest of the algorithm would be able to continue as expected.

The amount of errors varies from floor plan to floor plan, but when we look at Figure 6.7, the result has two errors. A close-up of the same Figure can be seen in Figure 6.2. In Figure 6.9, however, there are no errors and all points of interest are reached despite of the bigger size of the building.

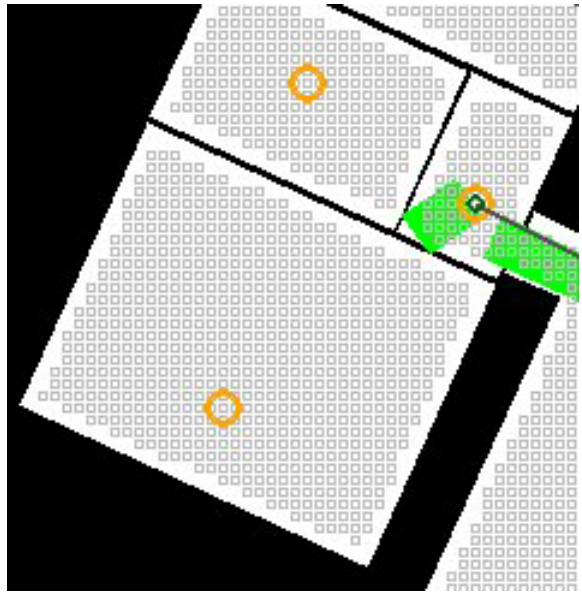


Figure 6.2: Door improperly placed, leading to paths not being created to the two orange circles.

Paths The paths were for the most part being correctly generated, even for buildings we had never done any testing on during the creation of the algorithms. In the requirement it was stated that the amount of points in a paths created should be few, so that it does not slow down a shortest-path algorithm being run on the output from this prototype system. We think we have met this requirement as the system in many

cases makes exactly the same choices as we would have made if we manually created the paths ourselves. The algorithm makes few nodes in total, but there is likely still some room for improvement, as we have not had enough time to experiment with minimizing it more.

The method we use does have its flaws too. When looking at Figure 6.3, the red circles mark two spots where it would make sense to cut down on the amount of lines. On the left side we have marked parts of the line we would want to remove in a yellow box, then we would connect the orange point of interest in the red circle just to the right of it to the remaining part of the line. The same would be done for the red circle to the right in the figure. We would cut away the shortest one of the two nearly parallel lines and connect the orange POI to the remaining line.

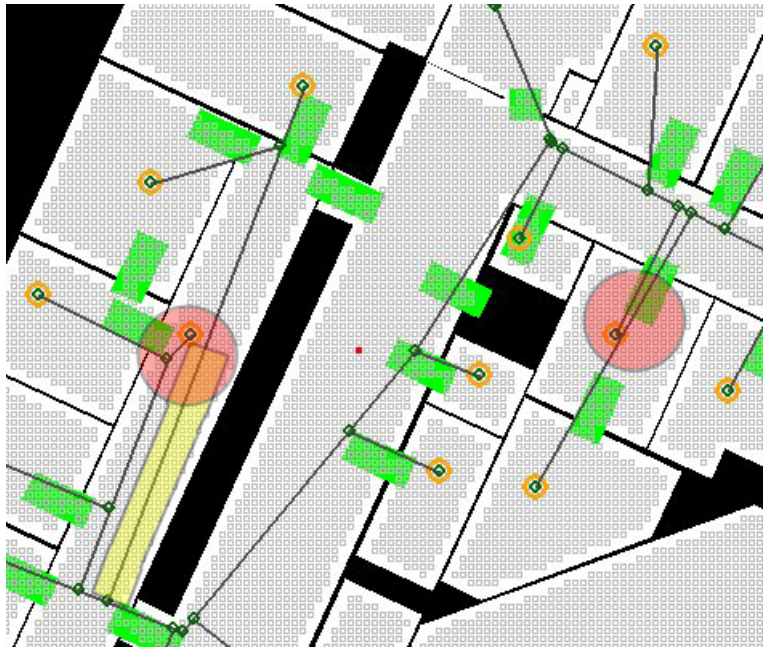


Figure 6.3: Possible corrections we would have done marked with the red circles.

Some places which would be obvious to connect for us, were not connected by the algorithm. An example of this can be seen in Figure 6.4. The illustrated red dotted line is not required to make paths around, but it could give shorter and more relevant paths for people navigating the building. Therefore after paths for a building has been generated, it would be smart to inspect the created paths. If any additional

paths should be created, then manually add them. There is likely a possibility of connecting these paths automatically, though we have not had time to explore this.

Figure 6.4: Lack of connections: while manually creating paths, we would have added a line where we have drawn the red dotted line now.

FR5

ID	Description	Priority
FR5	The system will allow the user to accept the output before exporting it to the database.	High

FR5 is solved by letting the user press a button in the graphical user-interface to insert the data into the database. This way the user can decide to not use the output, for example just to visualize what sort of output one would get in a certain building.

6.2 Non-functional Requirements

NFR1

ID	Description	Priority
NFR1	The system shall be able to run on a normal desktop computer.	High

NFR1 was solved by developing the system to not require excessive amounts of resources. On our development computer with 16 GB of RAM, with an Intel Core i7-2600K Processor, we never surpassed 450 MB RAM usage, even for the largest floor plans on NTNU. Most computers at the time of writing will have much more than 450 MB RAM available and should not be a limitation for the system. Other resources used should not be a limitation, though a slower CPU will lead to slower run-times.

NFR2

ID	Description	Priority
NFR2	The system shall be using a graphical user-interface for visualization.	Medium

Even though the graphical user-interface (GUI) in NFR2 had a priority of medium, we felt it was important to have a GUI to visualize the algorithms running so that we could spot potential problems faster, without having to import the data directly into the database. Therefore this was actually prioritized to get done as soon as possible. Pygame provided tools to both visualize our work, and later use what was visualized to reach our desired output. The GUI was changed several times throughout the project. At first the window size would be as big as the floor plan in question, this could mean a window size of 4000x5000 pixels in some cases, and was difficult to display on a normal computer screen or work with, so later on we decided to add scrollbars to allow for a window size that would fit all screens. The current GUI can be seen in Figure 6.5.

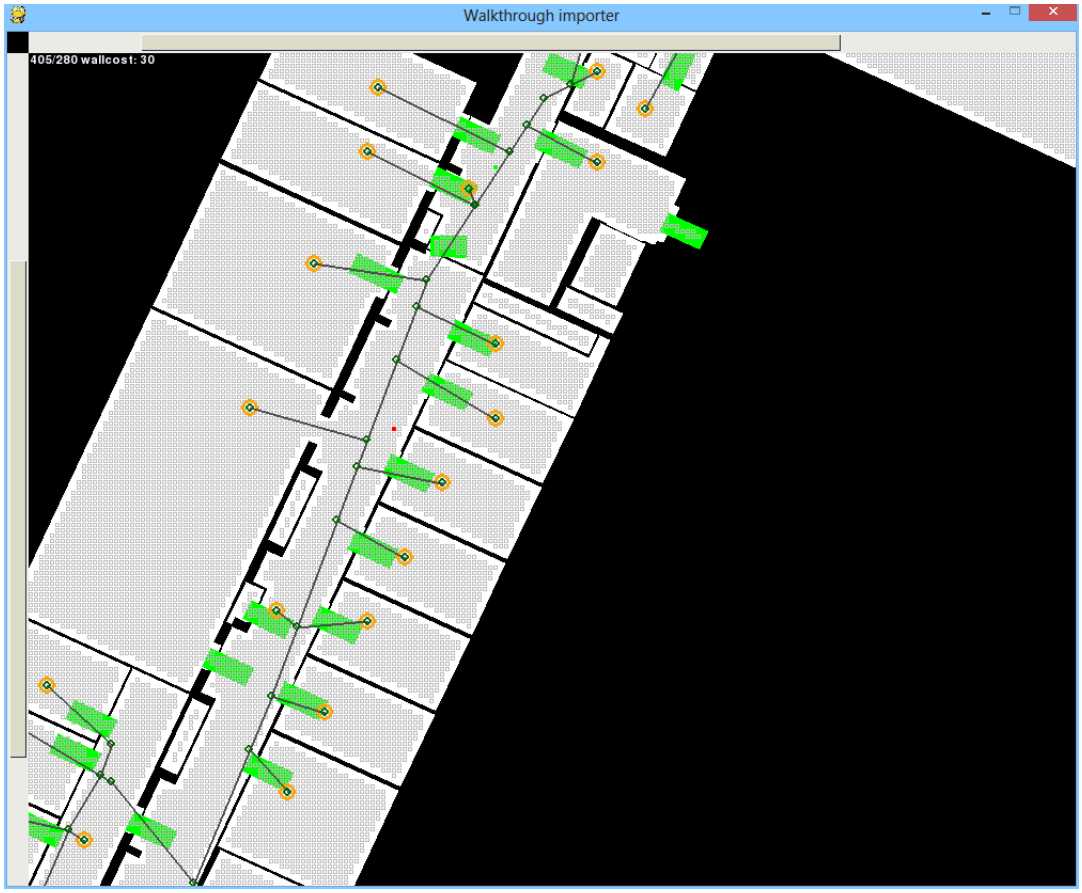


Figure 6.5: The graphical user-interface for visualization.

NFR3

ID	Description	Priority
NFR3	The performance of the system shall be faster than a user doing the same task. This is a tricky requirement as there are so many different types of buildings, some may take five minutes for a user to complete, while others take hours. The goal for this requirement is that the system should always do the job faster than a trained user at doing this job, or at least provide a basis for the user to complete the rest of the job manually.	High

Figure	**Building ID, (floor)**	**Squares**	**Loading (sec)**	**Path generation (sec)**	**Total time (sec)**	**By hand**
Figure 6.7	15, (2)	31713	7.3	8.1	15.4	7 minutes
Figure 6.8	33, (1)	18289	5	6	11	4 minutes
Figure 6.9	2, (1)	79287	20	55	75	20 minutes

Table 6.1: Performance comparison of the system versus doing it by hand. The *loading time* was the time it took to load all information and create all squares. The *path generation* was the time it took generate all paths. The *by hand* time is how long it took to manually map the floor plan.

We will show the time taken to generate paths for various floor plans, the time is measured through python and should be accurate based on our test computer with 16 GB RAM and an Intel Core i7-2600K Processor. Table 6.1 shows the time expenditures of our algorithm on a number of different cases. For the sake of comparison, we have also manually created the paths for these test cases. Based on the numbers seen in 6.1, even if something goes slightly wrong, one would still have plenty of time to fix any errors, or even run the whole process again, compared to doing it all by hand. Note, however, that the comparison is less favourable for larger test cases. By inspection, the time to do it by hand is roughly linearly proportional to the number of points, whereas the algorithm is quadratic. This seems to suggest that our program will not run feasibly on very large, complex floor plans. A larger amount of output results can be seen later in Table 6.2, though these do not include any numbers on doing the process by hand.

Our prototype only runs on a single core to calculate the paths, so several processes can be run at the same time on computers with multi-core CPUs. This would further increase the throughput, by for example completing paths for all floors of a building at the same time.

NFR4

ID	**Description**	**Priority**
NFR4	The system shall support Windows, Mac and Ubuntu.	Low

The prototype system has been programmed in python, as python works on all these platforms. Due to this requirement's low priority, it was not something that was tested often during the development, but we made sure to only use libraries that should work on all mentioned platforms. Microsoft Windows 8 was the operating system being developed on, and the system has been tested on Mac and Windows and all features are working as intended.

NFR5

ID	Description	Priority
NFR5	The system shall require minimal user-input.	High

The system only requires to have a building id and floor value specified, which can be done either through command-line arguments or by editing variables in the source code. However, several additional options are modifiable in the source code, for example the size of squares in the floor plans, which is a feature only intended to be modified by power-users.

NFR6

ID	Description	Priority
NFR6	The system shall be easily adapted to work with floor plans from sources such as Building Information Modelling (BIM) or plain images of floor plans.	Low

The workload of customizing the system to work with other sources will depend on how much metadata the floor plans contain. Our system only requires a collection of rooms, doors and destinations to visit to create walkable paths for a building. From the BIM handbook [20], it seems that BIM holds the necessary information about rooms and doors, while the destinations can be extracted from the centre of each room. Based on this, we estimate that our system can be adapted to support BIMs without much work, though we have not had the time to look further into this.

A worse scenario would be floor plans only as images. In this case, we would have to use computer vision to recognize doors to be able to connect rooms. Our method to create a maze as an image would already allow us generate paths from an origin to a destination, as long as there are no walls or doors blocking the way. Parsing data from the floor plans would likely be a substantial task. Even though floor plans often

have similarities, it is also likely that there will be differences between them, making it difficult to support all types of floor plans. If, however, doors and destinations can be parsed from the floor plan, then the remaining path-generation can be done by our system. To finish we would have to manually position the floor plan at a longitude and latitude to make the paths useful for an indoor navigation system utilizing a map.

NFR7

ID	Description	Priority
NFR7	The system shall output data that can be used in the CampusGuide without taking any further actions.	High

Since the output of the system is imported directly to the database, it allows generated paths to instantly be taken into use by the CampusGuide. That said, the paths are not optimal as was seen when evaluating FR5. A weakness with our algorithm is that it does not create lines in parallel with walls. If the algorithm would try to create parallel paths, we would possibly have nicer looking paths. When users manually create paths in buildings, they would try to make the paths look good in addition to connecting rooms, in our algorithm the look of the paths is something that could be improved on to avoid having to take further actions.

6.3 Output results

Table 6.2 is filled with numbers taken from running the system on the first floor of all buildings on the NTNU Gløshaugen campus. The table shows the amount of rooms, POIs, paths, errors and time taken to generate the paths for each building. The amount of errors comes from each POI that the system did not manage to create paths to. The amount of rooms represents the amount of room polygons loaded, though these are not always large enough rooms to have a point of interest. The high amount of errors for building ID *17* is due to half of the building getting cut off from the rest of the paths and a great amount of the POIs are unreachable. In Figure 6.6 the problem area is marked as red. The reason for the building becoming cut off is that two neighbouring room polygons have a slight gap in-between them, which the algorithm interprets as a wall.

Table 6.2: This table shows the amount of rooms, POIs, paths, errors and time taken to generate the paths for each building on NTNU Gløshaugen campus.

Building ID	**Floor**	**Rooms**	**POIs**	**Paths**	**Errors**	**Generation time (sec)**
1	1	138	26	52	0	36.35
2	1	179	68	145	0	99.19
15	1	58	21	42	0	19.70
17	1	111	47	36	26	33.30
18	1	63	30	51	4	24.15
19	1	40	20	38	0	10.16
20	1	23	13	19	0	5.34
24	1	90	52	94	4	96.37
25	1	43	27	53	0	14.84
26	1	53	30	59	0	22.58
27	1	42	24	46	2	18.03
28	1	58	29	42	3	11.28
29	1	65	45	84	0	49.34
30	1	152	110	207	5	256.71
31	1	78	49	103	1	28.05
33	1	43	29	56	1	12.63
34	1	100	27	46	3	14.55
35	1	85	32	65	0	18.51
36	1	86	41	72	2	23.26
37	1	81	44	78	2	22.35
38	1	47	38	62	1	23.01
39	1	60	44	82	0	22.35
41	1	68	44	76	0	21.44
43	1	166	58	104	0	81.76
44	1	94	23	42	3	30.91

Table 6.2: This table shows the amount of rooms, POIs, paths, errors and time taken to generate the paths for each building on NTNU Gløshaugen campus.

Building ID	**Floor**	**Rooms**	**POIs**	**Paths**	**Errors**	**Generation time (sec)**
45	1	76	49	95	0	24.98
46	1	20	7	7	2	6.54
52	1	23	14	25	0	4.22
53	1	35	35	62	0	11.28
54	1	63	42	71	0	14.95
55	1	84	34	68	0	18.29
56	1	16	4	7	0	2.62
57	1	83	60	113	3	41.26
58	1	226	179	322	12	492.85
61	1	11	7	12	0	2.14
62	1	19	15	27	0	3.41
64	1	45	23	43	0	11.75
65	1	19	17	30	1	5.63
66	1	16	12	19	3	8.08
67	1	402	137	269	3	414.52
68	1	51	34	63	0	12.89
70	1	53	35	64	2	16.12
76	1	32	21	41	0	14.49

Screenshots

A couple of complete screenshots of all automatically generated paths in some floor plans can be seen in Figures 6.9, 6.8 and 6.7. More screenshots can be found in the output section which is described in Appendix B.

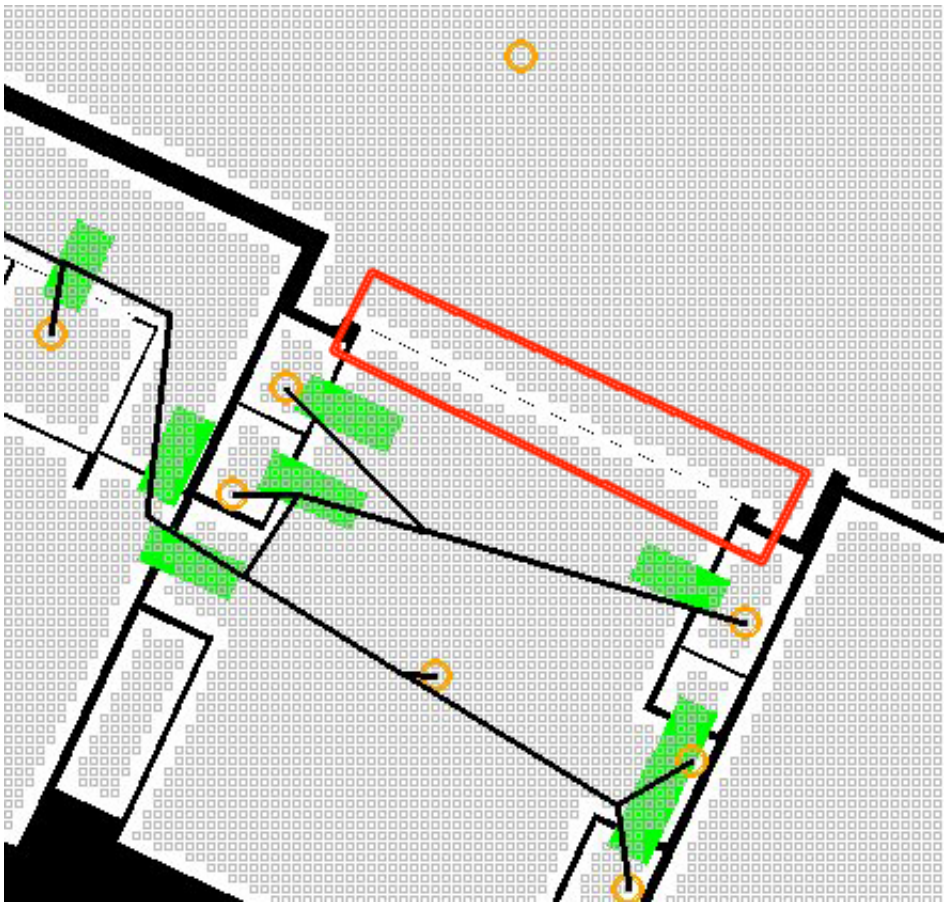


Figure 6.6: A close-up of the critical section from building 17. Parts of the building are unreachable. The problem area is marked with red lines and occurs due to two room polygons having a gap in-between them.

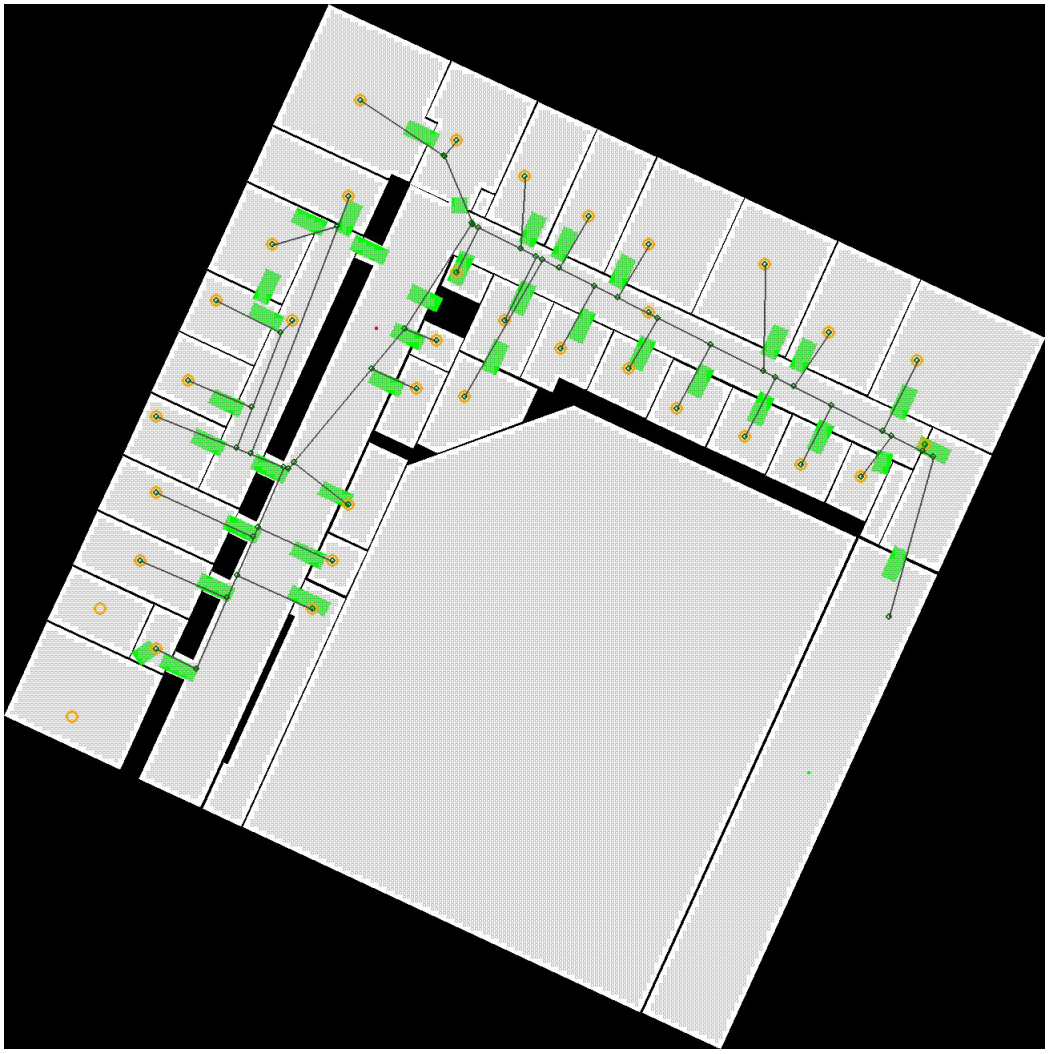


Figure 6.7: All paths generated on the first floor in building 15.

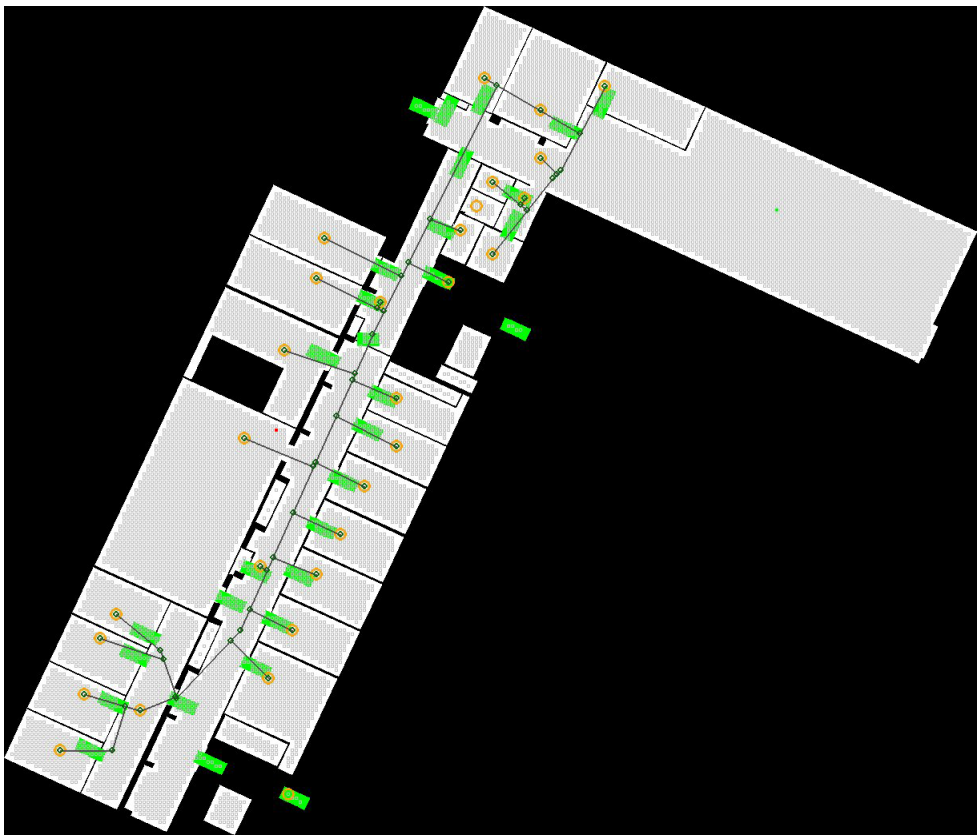


Figure 6.8: All paths generated on the first floor in building 33.

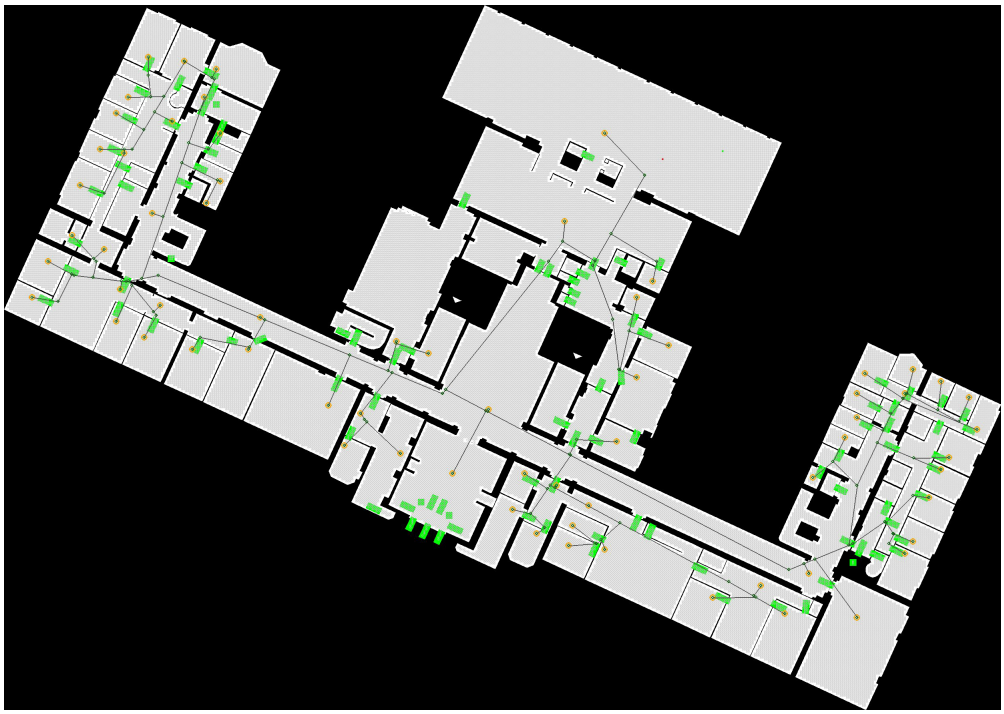


Figure 6.9: All paths generated on the first floor in building 2.

Chapter 7

Discussion

In this chapter we will discuss whether we answer our research questions.

1. Is it possible to create a system that can automatically find walkable paths through a building purely from information taken from an AutoCAD floor plan?
2. If it is possible, how does it compare to a human doing the work manually?

Is it possible to create a system that can automatically find walkable paths through a building purely from information taken from an AutoCAD floor plan?

Based on our evaluation, we have shown that it is possible to create a system that automatically navigates and creates paths through a building only based on information from an AutoCAD DXF floor plan, though it heavily relies on how the information is stored. As different building architects often have different ways of creating their floor plans in AutoCAD, there will often be a need for some manual work while importing the data. For example, rooms are gathered from the floor plan files based on layers named as *ROM*. In another building, these layers might be named something completely different. If this is the case, it would not be very difficult to just modify the name when importing the rooms.

One could imagine worse cases, where all rooms are connected as one polygon, and walls are drawn onto the floor plan. This could lead to bigger problems with our

implementation, though it could, for example, be solved by importing the walls into our picture as black lines.

In essence it is important that the information found in the floor plans can be easily extracted. The two most important being the room polygons and doors. Without this information, the result will be worse. As all buildings do not follow the same standards, one will in many cases have to modify the import method to suit a new building. In our project we did not have this problem with importing data, as we were told to work on data imported with an already working import script for the NTNU floor plans.

If it is possible, how does it compare to a human doing the work manually?

In the evaluation we showed how we could map a building floor in roughly fifteen seconds. There is however some randomness involved in the creation of paths. In some buildings the paths may not give as good results as doing it manually, much to do with the order that paths are created in, since a new point of interest tries to connect to the closest line. If the closest line is in a bad spot, the quality of the output decreases.

We think the paths generated do provide a solid starting point, and with the combination of automation and doing some finishing touches manually, there is a large amount of time that can be saved for every floor plan.

Chapter 8

Conclusion and future work

This chapter concludes the studies by presenting and evaluating the work done. We will finish the chapter by discussing future work that we think can improve the system.

8.1 Conclusion

We have created a system for automatically generating the walkable paths of a building floor. This will allow the system to give a path to move between any two given points on the floor. Our system uses information parsed from the floor plans of the building in question as input. We have worked specifically with floor plans in AutoCAD at NTNU. The system could in theory be used for any set of floor plans, but it would take some work to adapt it to the specific formatting standards used in different environments. We have tailored the output specifically to be used with the CampusGuide, a commercial application for helping the user find their way around the university campus.

The time used to generate paths is vastly improved over doing the work manually. This was possible through the use of the A* algorithm to create paths, in addition to our own methods to connect the paths into one network. Due to some randomness when creating paths and connecting them, the quality will not necessarily be on par with a human doing it manually. It is important to note that even though this system automates much of the work involved with creating paths, it is not a replacement of tools such as QuantumGIS allowing the job to be done manually, but rather a

system that can be used in supplement with these manual tools to achieve a good result in a much shorter amount of time.

8.2 Future work

To further improve the system, some future changes will be interesting to carry out. Being able to expand this system to buildings using a different floor plan format than NTNU, without manually having to do any set-up would be a solid improvement. This could perhaps be done by involving computer vision to make out information from floor plans stored as images. The visuals of floor plans usually follow a standard that should be possible to generalize for the majority of buildings.

Another improvement that should be made is to tweak the path generation algorithms to avoid detours. We have previously shown that generated paths at times could benefit from connecting to additional paths to create shorter routes between some points of interest. Creating more connections between different paths, leading to a larger grid, will likely solve this.

Currently manual work is required to connect paths across more than one floor in a building. If there are stairs, elevators or similar on a building floor, there are likely stairs above or below those points, which can be used to automate the process of connecting floors, saving additional time.

Most of the errors occurring when generating paths are due to doors being misplaced. A more solid method for recognizing where doors should be placed would solve this problem, which in turn would lead to less work to do by hand.

The system has been tested on a large amount of different building sizes, and it is apparent that larger buildings take a while longer to have paths generated for it. We think this time could be reduced by avoiding to check all lines when we want to connect a new point of interest to the path network. For example by only trying to connect to paths within a certain radius around the point in question.

Bibliography

[1] W. Trondheim, "The CampusGuide." http://www.campusguiden.no/?lang=english, 2012. [Online; accessed 27-August-2012].

[2] S. Andresen, J. Krogstie, and T. Jelle, "Lab and research activities at wireless trondheim," in *Wireless Communication Systems, 2007. ISWCS 2007. 4th International Symposium on*, pp. 385–389, 2007.

[3] J. Krogstie, "Bridging research and innovation by applying living labs for design science research.," in *SCIS* (C. Keller, M. Wiberg, P. J. Ågerfalk, and J. E. Lundström, eds.), vol. 124 of *Lecture Notes in Business Information Processing*, pp. 161–176, Springer, 2012.

[4] Google, "A new frontier for Google Maps: mapping the indoors." http://googleblog.blogspot.com/2011/11/new-frontier-for-google-maps-mapping.html, 2011. [Online; accessed 10-October-2012].

[5] Autodesk, "DXF Format." http://www.autodesk.com/techpubs/autocad/acad2000/dxf/dxf_format.htm, 2012. [Online; accessed 20-September-2012].

[6] École polytechnique fédérale de Lausanne, "EPFL - Orientation Tool." http://plan.epfl.ch/, 2013. [Online; accessed 17-October-2012].

[7] Google, "Google Maps Indoor Maps availability." http://support.google.com/gmm/bin/answer.py?hl=en&answer=1685827, 2012. [Online; accessed 10-October-2012].

[8] I. Sutherland, "A method for solving arbitrary-wall mazes by computer," *Computers, IEEE Transactions on*, vol. C-18, pp. 1092 – 1097, dec. 1969.

[9] L. Ran, S. Helal, and S. Moore, "Drishti: an integrated indoor/outdoor blind navigation system and service," in *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pp. 23–30, March.

[10] J. Orkin, "Three States and a Plan: The AI of F.E.A.R.," in *Proceedings of the Game Developer's Conference (GDC)*, 2006.

[11] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, pp. 100 –107, july 1968.

[12] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Q.*, vol. 28, pp. 75–105, Mar. 2004.

[13] pygame, "pygame - python game development." http://www.pygame.org/, 2013. [Online; accessed 5-January-2013].

[14] D. Varrazzo, "Psycopg." http://initd.org/psycopg/, 2013. [Online; accessed 10-January-2013].

[15] T. P. G. D. Group, "PostgreSQL: The world's most advanced open source database." http://www.postgresql.org/, 2013. [Online; accessed 10-January-2013].

[16] PostGIS, "PostGIS: Spatial and Geographic objects for PostgreSQL." http://www.postgis.net/, 2013. [Online; accessed 10-January-2013].

[17] ezscroll, "Pygame scrollbars - so easy to use." https://code.google.com/p/ezscroll/, 2013. [Online; accessed 23-January-2013].

[18] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd ed., 2009.

[19] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25 –30, 1965.

[20] C. Eastman, P. Teicholz, R. Sacks, and K. Liston, *BIM handbook: A guide to building information modeling for owners, managers, designers, engineers and contractors*. Wiley, 2011.

Appendix A

Line generation pseudo-code

Algorithm A.1 Line generation pseudo-code:

```
function MINIMIZEPATH(path)
    path ⇐ path.reverse()
    currentPosition ⇐ path[0]
    validSquare ⇐ path[0]
    for square in path do
        if isValidLine(currentPosition, square) and not isValidLine(currentPosition,
path[-1]) then
            validSquare ⇐ square
        else
            for line in lines: do
                for point in line: do
                    closestValidPoint ⇐ Find the closest valid point
                end for
            end for
            if closestValidPoint exists then
                return Make a connection to this point and split the existing line
the point was on.
            end if
            if closestValidPoint does not exist and validSquare exists and current-
Position ≠ validSquare  then
                Make a line from currentPosition to validSquare
            end if
        end if
    end for
    Make sure to connect currentPosition to the last square if this position in code
is reached.
end function
```

Appendix B

Contents of ZIP-attachment

This report is bundled with a ZIP-file and here is the explanation of its contents.

Source code

In the folder called *Source code* the system implementation can be found.

- **importer.py** - Contains the code for the system created, written for Python 2.7. Note that the database information has been stripped as it is considered sensitive information.
- **readme.txt** - Contains instructions required to run the system.

Output

The *Output* folder contains screenshots from running the system on the first floor for all buildings at NTNU Gløshaugen. They can be seen in connection with Table 6.2. The files are named as **build-*(Building ID)*-*(Floor)*.jpg**, meaning building ID 33 on floor 1 will be named *build-33-1.jpg*.