Pathfinding for Persons with Reduced Mobility - Comparing the performance of different routing-algorithms given various physical and environmental restrictions

Final Report for CSM6960 Major Project

Author: Jostein Kristiansen (jok13@aber.ac.uk)
Supervisor: Dr. Myra Scott Wilson (mxw@aber.ac.uk)

September 30, 2016
Version: 1.2 (Release)

This report was submitted as partial fulfilment of a MSc degree in Intelligent Systems (G496)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Acknowledgements

I would like to thank my supervisor, Dr. Myra Wilson, for her invaluable technical and moral support during the duration of this project.

I would also like to thank Dr. Edel Sherratt for her continuous support during the entirety of my MSc degree. She has helped me resolve a wide array of issues – many of which were not insignificant.

I would also like to thank Stefan Klaus MSc for helping me structure this document and make sure that it complies with the requirements set for MSc dissertations.

And finally: I would like to thank Dr. Angharad Shaw for visiting my high school back in Norway and recruiting me to Aberystwyth University. She has not been directly involved with this project, but as she is the reason why I applied for my BSc, I recognise that I would not be where I am today without her.

Abstract

Route-planning applications are designed to help their users find good paths between two or more locations on a map. These systems are useful to a wide variety of people – like tourists travelling to unfamiliar areas, or companies wanting to minimise fuel-costs and/or travel-time for their delivery vehicles. Route-planning can even be a useful tool on planes and boats, as it can help guide vessels into areas where the forces of nature give less resistance, or away from restricted/dangerous areas.

Conventional route-planning software is often aimed at one or more large groups of specific users – like motorists (commercial and/or private), pedestrians, cyclists, etc. But very few of these systems are able to plan good – or even practical – routes for Persons with Reduced Mobility (*PRM*), as these users are often simply grouped together with pedestrians or cyclists; with no special consideration taken with respect to their physical limitations.

Built-up areas pose particularly difficult environments to navigate for PRMs, as many commonly encountered obstacles like stairs, non-automatic doors, curbs, and steep slopes are effectively impassable for people with certain physical limitations. Without the aid of a route-planner, PRMs would need to rethink their planned route to a location on the fly whenever they encounter an obstacle, which can often prove quite frustrating, as an accessible path to where they want to go may not even exist.

The route-planning system described in this thesis tries to address the aforementioned issues by identifying and avoiding inaccessible areas, and using accessible buildings as shortcuts in an attempt to shorten the routes returned to its users. A number of different pathfinding algorithms have been tested, each of which have their own advantages and disadvantages.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction, Background & Objectives

1.1 Introduction

Route planning has been a useful tool for thousands of years; even before we started drawing maps, and routes were planned relative to landmarks and star-constellations. In our modern society, route-planning systems are becoming increasingly more accurate and commonplace in our daily lives. These systems can be used by anyone: From tourists trying to find the nearest public toilet, to the Mars rovers [15] - boldly going where no one has gone before.

A significant amount of money and effort has been put into extensively mapping our world [1, 5, 27, 34], which in turn has provided route-planning systems even more data to work with – making them more accurate and useful than ever before.

Route-planning systems are of course only practical to use if they are able to plan routes that their users are able to follow, which is unfortunately often not the case for persons with reduced mobility (PRM). Most public route-planning systems focus on providing routing-services to very large, generalised groups of people - like pedestrians, cyclists or motorists, but very rarely consider PRMs as a separate group to these.

PRMs are often categorised as pedestrians, and are therefore given routes intended for people who are able to walk up stairs, cross streets with elevated curbs, climb steep inclines, etc. Many PRMs are able to pass these obstacles with some assistance from someone else, but this does not apply to everyone; their wheelchair might be too heavy to lift manually, they might be old and/or have a form of muscular atrophy that makes it hard for them to keep their balance, or there might not be anyone around able to give them the help they need.

PRMs could try following routes intended for cyclists, as these usually avoid stairs, but they might still contain tall curbs, steep slopes, etc. Bike-routes also tend to be noticeably longer than pedestrian routes, so these may not always be suitable for many PRMs either.

As built-up areas are often particularly challenging environments to navigate for PRMs, the project developed as part of this dissertation has focused most of its attention to areas like this. Additionally: Because Aberystwyth University has made a notable effort to make its campuses more accessible for everyone in recent years, and because there are more people with disabilities in higher education now than ever before [37], this project has put its main focus on the Aberystwyth University campuses.

This project builds upon the author's Major project for their BSc degree at Aberystwyth University ("Access Aber - Pathfinding"), completed in 2015, which in turn was inspired by a separate system developed by Aberystwyth University in the summer of 2014: "Access Aber".

1.2 Analysis

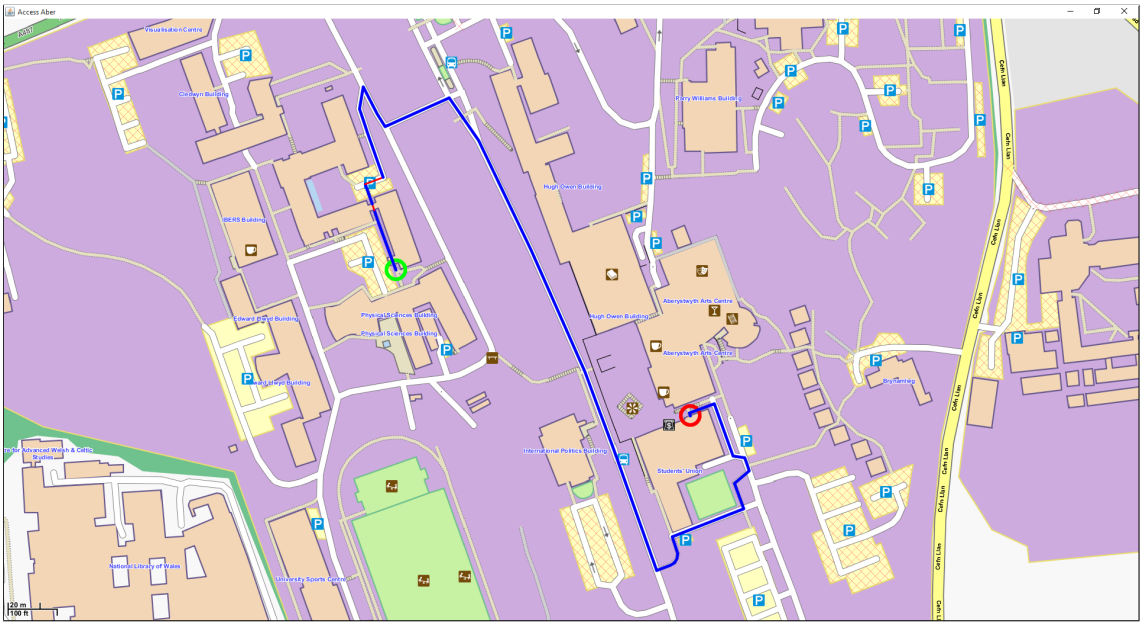
The goal of this project is to show that it is possible to plan accessible routes for PRMs, but more importantly: The project needs to show why more route-planning systems should consider PRMs to a much larger degree than they currently do, and that some areas on the Aberystwyth University campuses could still be made more accessible than they currently are.

The system developed as part of this dissertation has to identify the most important aspects of route planning, and replicate this in a route-planning system made specifically for PRMs. These aspects include, but are not limited to: calculating routes in near-real-time, having a low memory-footprint, employing optimal and complete routing-algorithms, and of course finding appropriate routes for the target users.

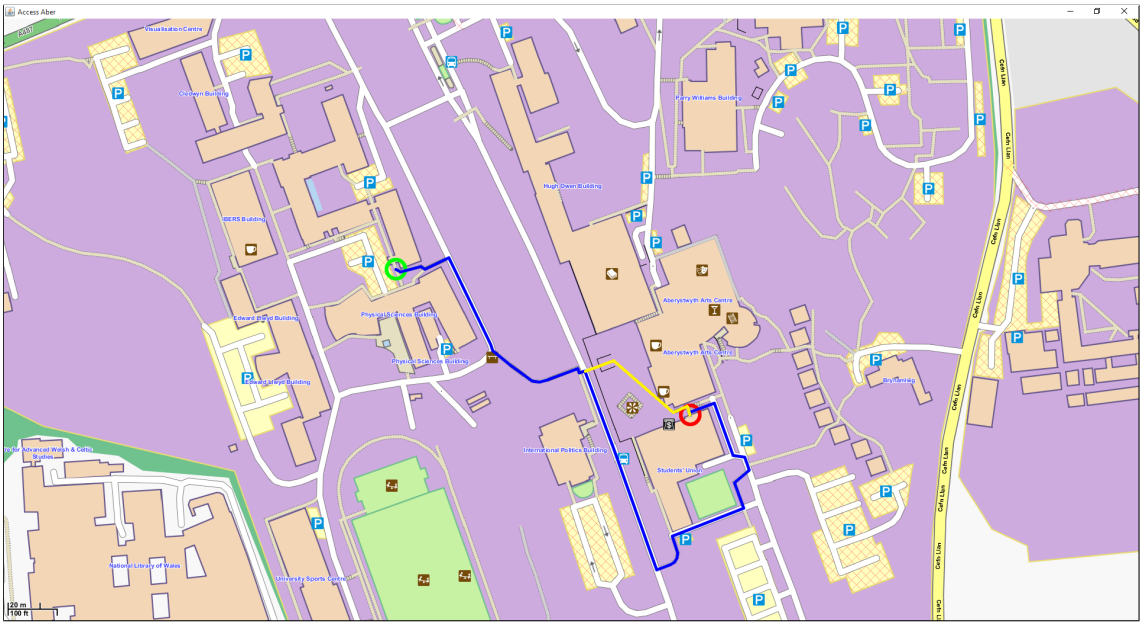
The process of extensive and accurate mapping of Nodes (or individual points of navigational data) is a task beyond the scope of this project, so this data will have to be retrieved from a third party. This will be discussed in more detail later in this report.

The A Star (A*) search-algorithm has been chosen as the default algorithm for route-planning (The reasoning behind this will be explained later), but three other search-algorithms have also been implemented – each of which can be selected by providing the system with a specific command when it is started; See the *README.txt* packaged together with the source-code.

Most commercial route-planners use search-algorithms able to abstract routing-data into large blocks connected to each other by only a few roads, which lets them plan incredibly long routes seemingly instantly by jumping directly from block to block, rather than looking at the road-networks inside in detail. This abstraction often leads to a loss of accuracy however, as the algorithms tend to favour main roads over side roads which may provide a slightly more time/fuel efficient route. For most people, this slight increase in path-costs do not matter too much in their day-to-day lives, but as explained later in this document, PRMs often need to follow significantly longer routes than able-bodies pedestrians, due to their physical limitations making it hard to pass certain obstacles. Because of this, the routes returned to PRMs by a route-planning system should always be as optimal as possible to avoid any further inconvenience. This does not mean that the routing-data cannot be abstracted at all however – and you will see later – so it is still possible to take certain measures to reduce runtimes and memory-requirements.



(a) Route for Persons with Reduced Mobility



(b) Route for able-bodied pedestrians

Figure 1.1: These images show two different routes from the computer-science department (*IMPACS*) on Penglais campus, to the Students' Union. As there are many stairs in the way, many PRMs are forced to follow a far longer route than an able-bodied person would. The yellow line in (b) (hand-drawn by me – starting where (a) and (b) intersect) shows the actual path an able-bodied pedestrian would take. The route in (b) is likely sub-optimal because data is missing from OpenStreetMap's database; Mapzen on `openstreetmap.org` finds the same route as (b) (29.September.2016).

1.3 Process

The life cycle models used for this project are Test Driven Development (More on this later), and the Spiral Model.

The Spiral Model makes it easier to select specific parts of the system to develop within a certain time-frame, and to develop these parts properly, as it encourages more detailed planning. The following are the four main steps in the Spiral Model:

1. Determine objectives

   Clearly specify what functionality we expect to be present at the end of this iteration.

2. Identify and resolve risks

   Find the best solution(s) for solving the problem at hand, but which will still allow us to finish within the planned time-frame

   This might mean sacrificing a good solution for a slightly worse one if we expect that the best solution would take too long to implement – ie. lowering the risk of not being able to finish the project on time.

3. Development and Test

   Implement the solution(s) found in the previous step, and test whether they break any new or existing functionality.

4. Plan the next iteration

   Determine which part of the system needs the most attention, and make plans relative to that.

Extreme programming (XP) was also considered for this project, but was decided against because the author knew from experience that they needed a slightly more plan-driven approach to development to make sure that the system was finished on time. The Spiral Model makes it easier to focus on one aspect of the system at a time, and makes sure that that aspect is implemented properly. It does this by having a step dedicated to finding and evaluating alternative approaches based on the risk involved in implementing them. This risk could be: time vs memory requirements, expected difficulty of coding, etc.

After one iteration of the spiral is complete, we are able to move on to another part of the system. This makes it much easier to stop optimising every aspect of the system, and instead prioritise creating a finished product.

Testing has been a crucial part of development, and every method in the source-code has one or more tests associated with it, but again: This will be discussed in more detail later.

Chapter 2

Design

2.1 Overall Architecture

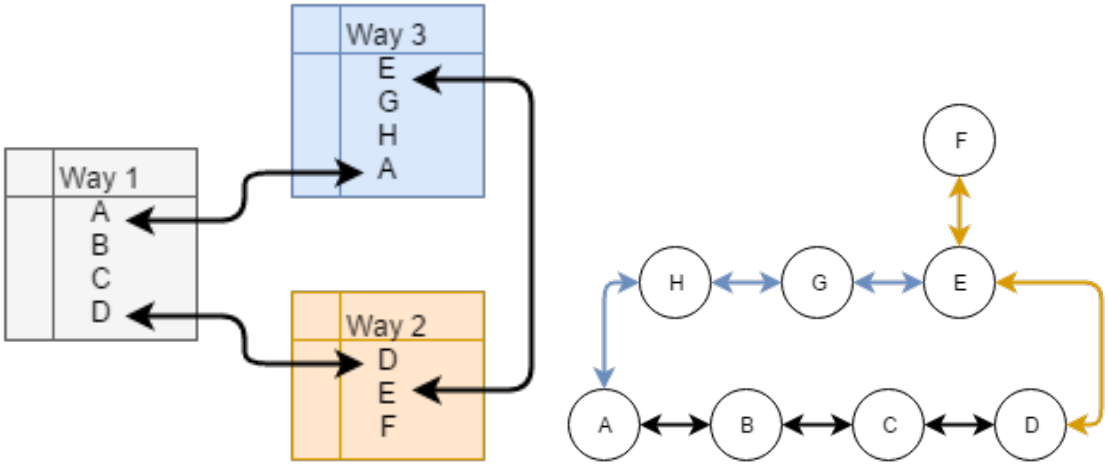
Route-planing systems find routes by continuously expanding Nodes connected to other Nodes; starting at a "Start Node", and ending at a "Goal Node". A Node is a single point on the map – defined by its Latitude and Longitude coordinates. Nodes can be isolated points of interest like statues or monuments, or they can be part of one or more larger structures called Ways. Ways contain an ordered list of Nodes, each of which define its shape, and direction. The ordering of Nodes inside the Way is important in cases where a degree of inclination is indicated, or whenever a route passes through a one-way street.

If the ordering of Nodes inside a Way is not respected, then its degrees of inclination may get flipped from positive to negative, or a route may become illegal to follow if it goes the wrong way on a one-way street. See Figure 2.1 for an illustration of how Nodes and ways may be represented, and Figure 2.2 for an illustration of how Ways may be expanded.

Some search-algorithms find the shortest path between a Start- and Goal Node by using some knowledge about the search-space like the path-costs between Nodes or a Node's estimated distance to the Goal Node, while others simply expand Nodes in a specific order until the "Goal Node" is found, at which point the route is returned – no matter how good or bad it might be. Both kinds of algorithms will be discussed here, along with their advantages and disadvantages.

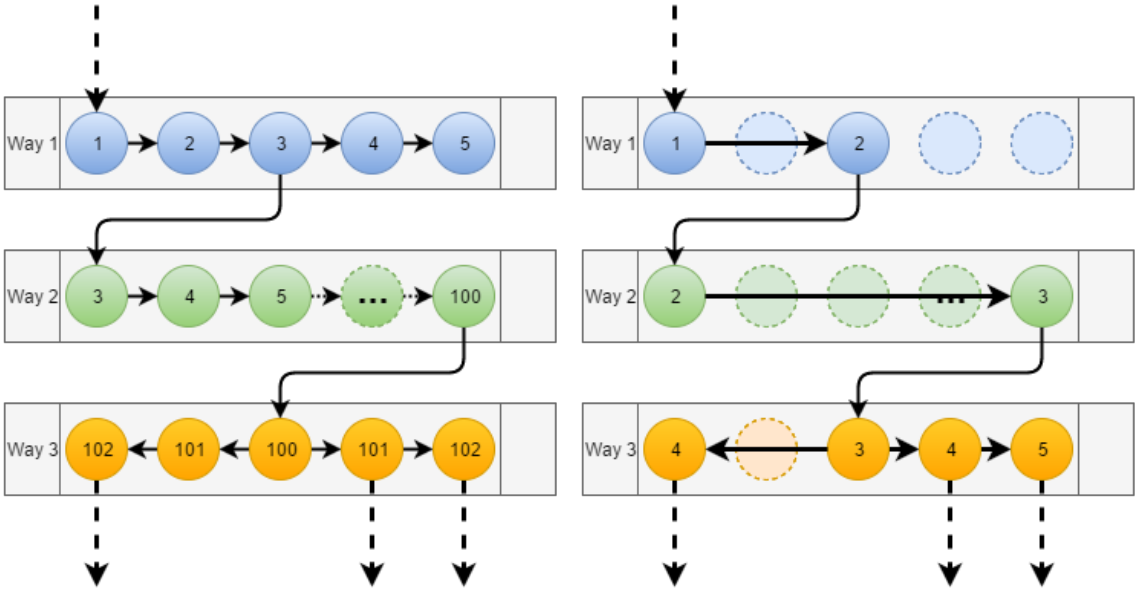
Nodes shared between two or more Ways can be thought of as intersections between those Ways. These Nodes are called "Tower Nodes", while unshared Nodes are called "Pillar Nodes"; See Figure 2.2.

Only Tower Nodes are expanded when planning routes, as this speeds up searches, and reduces memory-requirements significantly. Some Ways can contain more than 100 Nodes, but may only have 2-3 Tower Nodes, as you can see in Figure 2.2. By ignoring the large majority of a Way's Nodes when planning routes, searches are sped up significantly, and less memory has to be used to keep track of which Nodes were expanded to get to another Node.



(a) Individual Ways (b) Graph

Figure 2.1: These images show how Ways may be connected by their Nodes. Note that a connection does not have to originate from the first or last Node in the list.



(a) Expanding both Pillar- and Tower Nodes (b) Expanding only the Tower Nodes

Figure 2.2: This illustration shows the order in which Nodes may be expanded inside a Way, and the number of steps required to get from one Way to another. Every Node points to the Node adjacent to it, and so on. Note how Nodes are expanded starting from the entry-point into the Way, which is not always the first Node in the list, and that Nodes may be expanded going in either direction. The abstraction seen in (b) is very similar to how many other hierarchical path-finding systems abstract their data into entry- and exit-points only, and thus reduce the number of steps required to go from one area (or Way) to another; Notice that a constant number of steps are taken in *Way 2* in (b), no matter how many Pillar Nodes the Way contains, but that an additional step has to be taken in *Way 3* because there is a Tower Node between the Nodes in steps 3 and 5.

2.2 Classes and methods

Every class has been sorted into one of three packages for database-indexing, route-planning, or running the system. Mapsforge provides the system with many new packages to handle the map, but as these were made by a third party, and accessed via a *.jar*-file, they won't be mentioned here.

The Node- and Way-classes implement an interface to ensure that any new Node/Way classes implemented into the system in the future will still be compatible with the rest of the system, while still allowing them to implement new functionality or store more data.

There is also an interface for *"Uninformed Search"*, which provides every search-algorithm with the methods they need to expand Tower Nodes, change start/goal positions, keep track of run-times and memory usage, and unwind the path found by the algorithms (which only includes Tower Nodes) so that it also contains the Pillar Nodes on that particular path.

Additionally: There is another interface for informed search-algorithms, which extends the aforementioned *"Uninformed Search"*-interface to also keep track of path-costs, and provide methods to estimate the total path-cost to a goal Node.

There is one class for every search-algorithm in the system, each of which inherits its methods either from the *"Uninformed Search"* Interface or the *"Informed Search"* Interface. Because most of the functionality required to run a search-algorithm is already provided by the interfaces, the search-algorithm-classes only need to implement a single method for defining how the search is carried out.

There are two enums which define what constitutes an accessible or inaccessible Way, as well as an enum that defines which Ways represent larger areas or structures (eg. buildings and parking-lots) rather than a concrete path (eg. footways and roads). These three enums act as filters on the routing-data used by the rest of the system to plan routes, and only Nodes and Ways that make it through these filters can be part of the routes returned to the user. For this reason, it is very important that the filters use the same labels as the routing-data, as any changes to the contents of these filters can have a massive impact on the routes returned to the user, and care should be taken to make sure that the filters are neither too strict nor too permissive. The only difference between the filters used to find the routes in Figure 1.1 is that stairs were considered accessible in one, but inaccessible in the other – the effects of which are two very different routes.

The filters, routing-data, and map-tiles are the only parts of the system that need to be updated from time to time to make sure that the system is able to plan relevant routes indefinitely; the routing-data and map-tiles can theoretically be updated automatically via a third-party server, while the filters need to be updated manually in the source-code (if the labelling of the routing-data changes [20, 21, 23, 24]).

2.3 tests

This project has been developed using Test-Driven Development (TDD), which means that unit-tests have been written for every class and method in the system. To make it easier to run and keep track of all of these tests, separate test-classes have been created for every *"normal"* class, in which all of its associated tests are kept.

There is also a separate top-level test-class made for running every test-class in the system together, so that the system's functionality can be tested as a whole, rather than testing individual parts of it.

A custom test-graph has also been developed, which is meant to verify that the search-algorithms implemented in the system do not plan inaccessible paths, and are Optimal and Complete (if they are supposed to be); See Figures 4.1, 4.2, and 4.3.

2.4 Routing-data

The routing-data used by this system has been downloaded from OpenStreepMap (OSM) [27], and is free to use for the public as long as we adhere to their license-regulations [19].

As all of the routing-data from OSM is stored as text in an *.osm* file, the system has to copy the relevant data into memory to make sure that it can be accessed quickly. Because *.osm* files are formatted as XML, the process of reading and copying the routing-data is done by an XML-reader made by Lars Vogel [40]. This particular XML-reader is free to use for anyone.

The system is more than capable of reading and using data from other sources though – as long as the file is formatted similarly to the OSM data currently used; see Table 2.1. The tags used to identify Nodes and Ways, as well as those used to identify the fields contained within them, can easily be changed in a designated enum where all of these tags have been hard-coded.
Let us say that a hypothetical dataset from Google was used to build our collection of Nodes and Ways, and their XML-file renamed Ways to 'Relations', then the label 'way'can easily be changed to 'relation'in the aforementioned enum to let the system know that this particular data has a different name, but should still be handled in the same manner as before.

After the relevant information in the *.osm* file has been copied into the system's memory, three filters are applied to this data, each of which are defined in separate enums. One filter removes any Ways that are deemed to be inaccessible (eg. steps), another filter retains only the Nodes that are relevant for route-planning, and deletes the rest (eg. any Way that is **NOT** a footway, building, piazza, etc.), and the third filter removes any inaccessible doors in buildings, as well as all isolated Nodes and Ways without any connections to the rest of the data-set. These three filters remove a large chunk of unusable data, which frees up a lot of memory, and makes Node-expansion much faster as there are fewer connections and therefore fewer alternative routes to explore.

As mentioned already: It is possible to dynamically download routing-data while the system is running, based on which Nodes the algorithms want to expand. This requires less data to be stored locally on the system, and ensures that the data is always up-to-date. The downside to downloading routing-data from an external source while the system is running is that Node-expansion may be significantly slower per Node, and any downtime on the servers or interruption

to the user's internet-connection[1] would bring the entire system to a halt. The third party may be able to give a guarantee that their servers will be available over extended periods of time if they are paid to provide this service, but as this system is not going to generate any money: paying for data is out of the question.

The best thing about downloading routing-data from OSM is that it is kept up-to-date by a large community of volunteers, and anyone can add more data; whenever the routing-data used by the system becomes outdated, we can just replace the old *.osm* file with a newer version, and the system will be able to find new routes right away. If tags are added or removed from the OSM data, then the three enums used as filters can easily be updated with this new information, and the changes should be reflected in the routes straight away.

Node	`<node id="1"...lat="52.4" lon="-4.0"/>`

Node	`<node id="2"...lat="1.0" lon="1.0">`
	`  <tag k="entrance" v="yes"/>`
	`  <tag k="wheelchair" v="yes"/>`
	`</node>`

Way	`<way id="1"...>`
	`  <nd ref="1"/>`
	`  <nd ref="2"/>`
	`  <tag k="highway" v="footway"/>`
	`</way>`

Table 2.1: The file read by the system has to be in the XML-format, and its Nodes and Ways have to contain the data listed in this table. The fields are allowed to have different names, but all of the data has to be present; information other than that listed here is simply ignored.
Nodes can be formatted in either of the two ways shown here, but only the tags listed are currently accepted – with the exception of 'v="designated"', and 'v="limited"'acting as alternative values after 'k="wheelchair"'.

[1]The author's internet connection was unreliable during the development of this system, which is another reason why all routing- and map-data is stored locally.

2.5 Algorithms

Routing-algorithms used for route-planning should be both Complete and Optimal. This means that the algorithm should always find a path from one point on the map to another if such a path exists, and that it always returns the best possible routes.

It is also important that the algorithms have low time- and space-complexities, but this will be discussed further in Section 2.5.5.

In order to better understand the importance of implementing Complete and optimal routing-algorithms, as well as the impact that time- and space-complexities have on the system as a whole, algorithms that fit one or more of these criteria– but also some that do not– have been implemented; See Table 2.2. By comparing and examining the routes found by each algorithm, as well as the resources required to run them, it is possible to determine which algorithm appears to be the most efficient in this particular route-planning-system – out of the algorithms tested.

The algorithms that have been implemented in the system are: A Star (A*), Greedy Best First Search (GBFS), Depth First Search (DFS), and Breadth First Search (BFS).

It was quite hard to accurately measure the average-case complexity of each of these algorithms, as just moving the start- or goal Node a little bit on the map could result in a wildly different result, especially for the uninformed search-algorithms (DFS and BFS). Greedy Best First Search only expands one child for each parent, and never retraces its steps, so it usually ended up hitting a dead end very quickly – thus making it hard to measure average-case complexities (Usually $O(10)$ when expanding both Pillar and Tower Nodes, but this could probably be reduced to $O(4)$ if only Tower Nodes are expanded inside Ways.

A class has been implemented into the system to measure the time- and space-complexities of each algorithm, but it didn't generate very reliable results, so finding average-case complexities using it was not possible. The class is called: "ComplexityAnalysisSearchTest" if you want to take a look. It runs warm-up rounds and everything, but the results were not conclusive.

2.5.1 A Star

The A* search-algorithm is both Optimal and Complete, and works in polynomial time. This is the algorithm chosen as the default for route-planning, as it is faster and more memory-efficient than the other algorithms that have been implemented.

A* is a best-first search algorithm, which means that it is able to prioritise certain Nodes over others when expanding the graph. It does this using the following formula:

$$f(n) = g(n) + h(n)$$

Where $g(n)$ is the path-cost accumulated to reach Node n from the start Node, and $h(n)$ is an estimate of the shortest path-cost from n to the goal Node. The algorithm uses a priority queue to sort the Nodes it has expanded, where Nodes with a lower f value are placed before those with a higher value, making sure that more promising Nodes are expanded before others.

$h(n)$ is calculated using this formula, which is the straight-line distance (ie. shortest possible path) from Node n to the goal Node:

$$h(n) = \sqrt{(n_{Latitude} - goalNode_{Latitude})^2 + (n_{Longitude} - goalNode_{Longitude})^2}$$

A* is also able to guarantee that no Nodes are visited twice as long as the following is true for all Nodes n:

$$pathCost(n_1, n_2) > 0$$

$$h(n_1) \leq pathCost(n_1, n_2) + h(n_2)$$

This means that once Node n has been expanded, there cannot be any shorter path to it in the graph, as all other Nodes will have a higher $f(n)$ than n. This avoids infinite loops and could let us eliminate the need for a list of visited Nodes, but this list is still present in this system because the easiest way to retrieve the route found by the algorithm is to keep a list containing the parents to every Node, and then just iterate through it starting at the goal Node.

This algorithm is the only one it was possible to measure the average-case performance of, as the others produced quite unreliable data. The average-case space-complexity of this algorithm depends on the branching factor of the Nodes in the graph. If the search is performed on both Pillar and Tower Nodes, then the space-complexity looked like this: $O(W * b)$, where W is the number of Ways expanded to reach the goal Node, and b is the branching factor of the graph. The branching factor before removing inaccessible Ways from the routing-data is higher than afterwards, so the space-complexity was therefore higher before than after applying the three filter mentioned earlier. But because W also depends on the branching factor, it was a bit hard to get an average value for it.

2.5.2 Greedy Best First Search

Greedy Best First Search is very similar to A*, but with the key difference that it only considers a Node's estimated path-cost to the goal Node, and only expands one child for each parent. This means that the algorithm uses this formula to determine which Node to expand:

$$f(n) = \sqrt{(n_{Latitude} - goalNode_{Latitude})^2 + (n_{Longitude} - goalNode_{Longitude})^2}$$

If the algorithm hits a dead end (ie. a child without further children, also called a leaf Node), it is unable to retrace its steps and retry a different path because the algorithm is 'greedy', and always 'consumes' the best Node at each step in the graph – making it impossible to return to a previous parent Node.

There are also implementations of Greedy Best First Search where the algorithm is allowed to retrace its steps at a dead end; making the algorithm complete if it makes sure to not revisit Nodes. But because the algorithm is greedy, it will return to the last parent with additional children, and expand the most promising Node left. This does not guarantee optimality however, as the path-cost through this Node might be higher than a path going via a different Node closer to the start Node.

2.5.3 Depth First Search

Depth First Search expands the first child for each Parent it encounters in the search-tree until it reaches the goal Node. This means that it goes in one direction until it hits a dead end, at which point it returns to the last Node it encountered which has additional children. This algorithm can get stuck in infinite loops if it does not keep track of which Nodes it has already visited, which results in it visiting the same chain of Nodes forever. So the algorithm keeps a list of visited Nodes, which also acts as a record for retracing the steps of the algorithm to find which route it found. This is performed by finding the goal Node's parent Node (ie. The Node the goal Node was expanded from), then finding the parent of the parent, etc. Until the last parent is the start Node – This is a very easy way to generate the route.

2.5.4 Breadth First Search

Best First Search expands every child of a Node before it goes on to expand every child of those children, etc. Until it find the goal Node. This can sometimes be a very slow process, but the algorithm always returns the path from the start Node to the goal Node with the fewest steps (Not necessarily the route with the lowest path-cost). This algorithm cannot get stuck in an infinite loop, as it explores the entire search-space rather than just an isolated part of the graph. The algorithm still keeps a record of every Node's parent though, as this makes it much easier to retrieve the route the algorithm found – as explained in the "Depth First Search"section above.

Algorithm	Complete	Optimal
A*	Yes	Yes
GBFS	No	No
DFS	No	No
BFS	Yes	No

Table 2.2: This table shows which algorithms are Complete and Optimal. Every algorithm can find an optimal path under the right circumstances, but only algorithms that can guarantee this behaviour every time can be considered Complete or Optimal.
All of the algorithms have a worst-case space-complexity of $O(|N|)$, and worst-case time-complexity of $O(|N| + |E|)$, where N is the total number of Nodes in the graph, and E is every connection between them.

2.5.5 Complexity analysis

The complexity of a graph (or set of interconnected Nodes / Ways), can be determined by the following formula:

$$Measure\ of\ problem\ difficulty = |V| + |E|$$

Where 'V' is the complete set of Nodes in the graph, and 'E' is every link or connection between them [32, p.82][2]. Because every Node inside a Way is connected to two other Nodes (except for the first and last Node in non-circular Ways), the formula to calculate 'E' looks like this:

$$E = \sum_{n_i=\{Nodes \in Way_i\}}^{Ways} (2 * (|n_i| - 1))$$

The complexity of an algorithm can be measured in how much its run-time increases depending on the amount of data it is handling (time-complexity), and how much memory or storage-space it needs in order to complete the search (space-complexity). Both of these measurements are usually written using "Big Oh" notation (written like this: $O(\ )$ [32, P.82 & P.1037]), which is a general way of presenting how an algorithm's complexity increases in relation to increasing amounts of data or traffic.

Only Tower Nodes are used for route-planning, as this has been proven to have a significant impact on runtimes and memory-use [2,41]. Some Ways may contain more than a hundred Nodes, but only a couple of Tower Nodes; by only routing between Tower Nodes, the algorithms are able to jump from one Way to another by just expanding a single Node, rather than a long chain of Pillar Nodes leading to an eventual Tower Node; See Figure 2.2 for an example. This is comparable to the hierarchical path-finding systems discussed in the papers by Yang et al. [41], and Botea et al. [2], with the distinction that my system groups Nodes by their relation to individual physical objects or structures (eg. stairs, buildings, or roads), while those systems group Nodes by where they are located in the environment (eg. by splitting the map into ten separate squares, and defining a limited number of entry and exit points for each square).

Other Hierarchical path-finding systems are usually able to split their Node-groupings into different layers with larger or smaller groups depending on how accurate the planned route needs to be, while my system only has two layers: One for looking at Ways without their Pillar Nodes (used for route-planning), and one for looking at Ways with their Pillar Nodes included (used for path-cost-calculations and drawing routes on a map).

Hierarchical path-finding systems like the ones described by Yang et al. [41], and Botea et al. [2] are usually able to plan routes using less time and memory than other optimal algorithms like A*, but lose some precision in return ("up to 10 times faster [when compared to A*], while finding paths that are within 1% of optimal." [2, page. 1]), resulting in sub-optimal paths. Because the routes returned to PRMs are already relatively long and many PRMs (especially the elderly) can tire quite quickly from extended periods of physical activity, the routes returned to them should really always be as optimal as possible.

The "hierarchical path-finding" performed by my system does not save as much memory or speed up searches as much as the systems described by Yang et al. [41] and Botea et al. [2], but it does preserve the optimality and completeness of its algorithms, while also reducing runtimes and memory-use.

[2] I forgot to note which page the graph-complexity formula was on before I returned [32] to the library, but I believe it was on page 82

2.6 Map and Coordinates

The map used in this project has been provided by Mapsforge [13, 14], as it is free, open source, and easy to implement. Many other map-providers were considered as well [3,5,6,8,11,17,18,29], but many of them cost money, were not written in Java, or provided too little/much functionality. GraphHopper [8] in particular was discarded because it is itself based on Mapsforge [13], but required that a prominent attribution to GraphHopper be displayed to the user, and required that the system be made publicly-accessible via a web-page or app-store, which this system is not going to be.

Creating a map without using third-party code was also considered, but ultimately decided against because of how mammoth this task seemed to be. The area of the Penglais Campus selected for use in this project contains 3492 individual Nodes, and 466 Ways, with the earliest entry timestamped in 2008. The routing-data kept by OSM has in other words been built up slowly over time, and it has taken a large community of volunteer mappers years to get it to the state it is in today. Trying to map all of these Nodes and Ways again in a completely new dataset would have taken away a lot of valuable time that could be spent on programming the route-planning system instead.

Routes are calculated either from two sets of latitude- and longitude-coordinates representing the start- and goal-Nodes (shown as a green and red circle respectively), or from a location on the map clicked on by the user. The start- and goal-Nodes are not placed on the exact coordinates specified by the user however, but rather on the Node closest to those coordinates. This means that the system is robust to coordinates placed outside the expected coordinate-range of $(-90 \rightarrow +90)$ for Latitude, and $(-180 \rightarrow +180)$ for Longitude [16, 26, 28].

Because only Tower Nodes are expanded while planning routes, the intermediate Pillar Nodes have to be looked up after a path has been found, so that a line can be drawn through every Node in the Way, instead of drawing a straight line from one intersection to another – which would make the routes look like they cut through buildings, roads, fields, etc. This process has been illustrated in Figure 2.2, where you can see that only the Nodes that bind different Ways together are expanded, and Nodes are expanded from the Tower Node used as an entry-point into the Way.

Because the interior layouts of the buildings on the Aberystwyth University campuses have not been mapped, it is impossible to know the actual path-cost of planning a route through any particular building. Adding to this, many Ways like buildings, parking-lots and squares use their ordered list of Nodes to define the outline of an area rather than an actual path that should be followed. Because of this, it has been decided that any routes that pass through buildings or across open spaces should be drawn with a narrow red line; See Figure 2.6. This line indicates that both the entrance and exit points of the Way are accessible for PRMs, but the area in between may be significantly longer and cannot be guaranteed to be accessible. There is no way to get around this issue, short of mapping the insides of every building manually like a few other projects have attempted to do in the past [38].

2.7 Programming

This system has been written in Java, as this is the language most familiar to the author. Many other route-planning applications and map-services have also been developed in Java, so there were plenty of useful examples, discussions, and libraries available on the web. The IDE used to developed the system is Eclipse [35]. Eclipse has great debugging, auto-complete, and library-import tools, making it very useful in this project – especially with regard to testing and debugging.

2.7.1 Classes and Methods

The classes and methods in the system are heavily commented, which should make it easier to follow the execution-flow of the program while debugging. Javadoc has been generated for every class and method (with a few exceptions), as this makes it much easier to use the auto-complete functions provided by Eclipse, but also because this can act as its own sort of system-documentation that is always kept up-to-date.

The classes and methods are loosely coupled for the most part, but complete loose coupling has not been achieved. It should be possible to change the execution of one method or module without breaking another, but just to make sure that all functionality is still intact after a change has been made, a test-class has been written for every class, and error/exception handling ensures that errors do not propagate further into the system.

2.7.2 Data-structures

The data-structure most suited to storing and indexing all of the Nodes and Ways is a Hash map/table. Provided that the hash-function is quite good, a Hash table has a best-case time-complexity of $O(1)$, which means that it is able to retrieve any data in constant time, regardless of how much information it stores. A poor hash-function can result in the far worse time-complexity of $O(n)$, which is the same as a standard Array (with unknown index), but worse than that of a binary search-tree with a complexity of $O(\log n)$. A Hash table's space-complexity (memory requirements) is $O(n)$, which is the same as a standard Array or binary search-tree, meaning that their space-complexities grow linearly to the number of elements stored. [31]

Because a Node might be referenced by more than one Way, the Ways store a reference (or pointer) to the memory-addresses of each of its Nodes in the array of Nodes. This makes sure that every Way references the same Node, and that references to non-existing Nodes can be safely discarded. The Nodes could be stored as Strings, but because an empty String takes up 40 bytes of memory, and a reference/pointer only uses 32/64 bit (depending on the operating system), pointers seem like a better choice. There might be an issue with cache-misses when lots of references to Nodes are read whenever a Way is expanded, but by only expanding Tower Nodes we can reduce the impact this has on the system's runtime.

2.7.2.1 Priority Queue

In order to sort the expanded Nodes by their weights (path-cost, distance to goal, etc.), a priority queue had to be implemented for each of the algorithms that were able to consider these weights.
The priority queue used in this system was imported from the Java-library: `java.util.PriorityQueue`.
This priority queue is based on a Heap, which is a maximally efficient data-structure to use for queues, and has a time-complexity of $O(log\ n)$ for insertion, and $O(1)$ for retrieval of the head of the queue (the best Node).

Priority queues automatically sort elements as they receive them, but this particular implementation does not reorder Nodes if their weights change after they have been added. This was a potentially big problem, as shorter paths to already discovered Nodes may be found by algorithms that are not consistent, and this lack of reordering can make searches significantly slower by not pushing good Nodes to the front of the queue.
This problem was solved by first removing the rediscovered Nodes from the priority queue, then adding them again to force a reorder; removing and then adding Nodes has a worst-case time-complexity of $O(2n)$ for each Node though, as the entire queue may need to be searched through to find and remove the Node, at which point we might need to go all the way to the back of the queue again if its path-cost didn't improve much.

Reusing routes between runs can be an effective tool to reduce run-times, as a Node expanded into the old route will indicate that the old path presents the shortest possible path starting at that Node. If path-costs are the same going in either direction startNode→goalNode and goalNode→startNode, then a new path just has to be planned starting at the Node that was moved until either the other Node or the previous path is found.

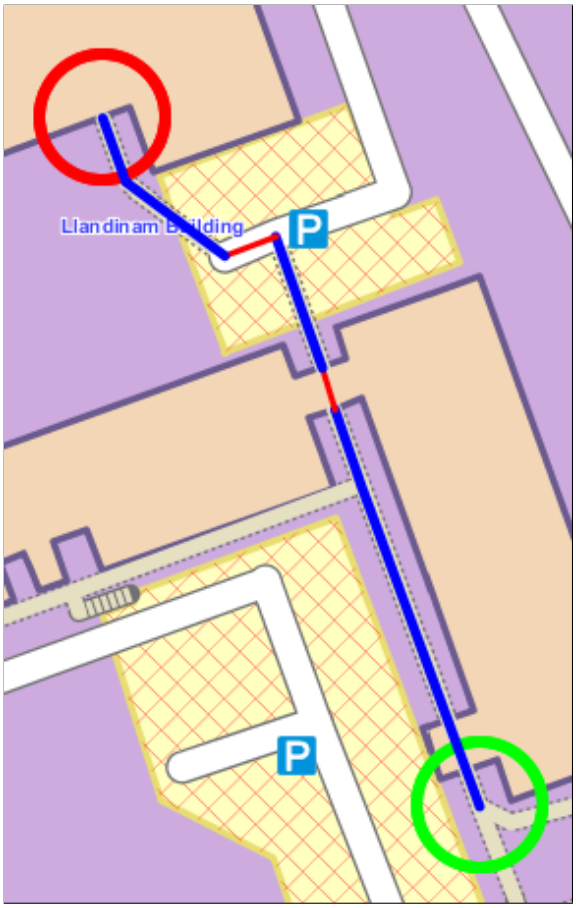


Figure 2.3: This image shows how the route is drawn as a thin red line whenever it passes through a larger area like a building or parking-lot. These red lines are meant to indicate that the actual path through the area is unknown, may be longer than shown on the map, and cannot be guaranteed to be accessible throughout; only the entry- and exit-points are guaranteed to be accessible.

Chapter 3

Implementation

3.1 Routing-data

Using routing-data downloaded from OSM while the system is running proved to be much more challenging than first expected. Because paying for access to this data was out of the question, only free services remained as a viable option. But because these services are free, there are heavy restrictions put on how much data we are allowed to download at a time, and also restrictions put on the speed of the connection to the server. Without these restrictions, the servers would obviously be bogged down really quickly by the constant high demand for data from its users, so it is quite easy to understand why the restrictions were put in place. But because these services were so restricted, they proved to be unsuitable for the amount of tests run on this system during development – which is a direct result of programming using Test Driven Development. Another issue with downloading routing-data from a third-party, is that the functionality of the system is made dependent on several third parties like the routing-data-provider and user's ISP. As mentioned a couple of times already: The author's internet-connection was not always reliable during the development of this system, so a decision was made to store routing- and map-data locally in an attempt to guard against further delays in development.

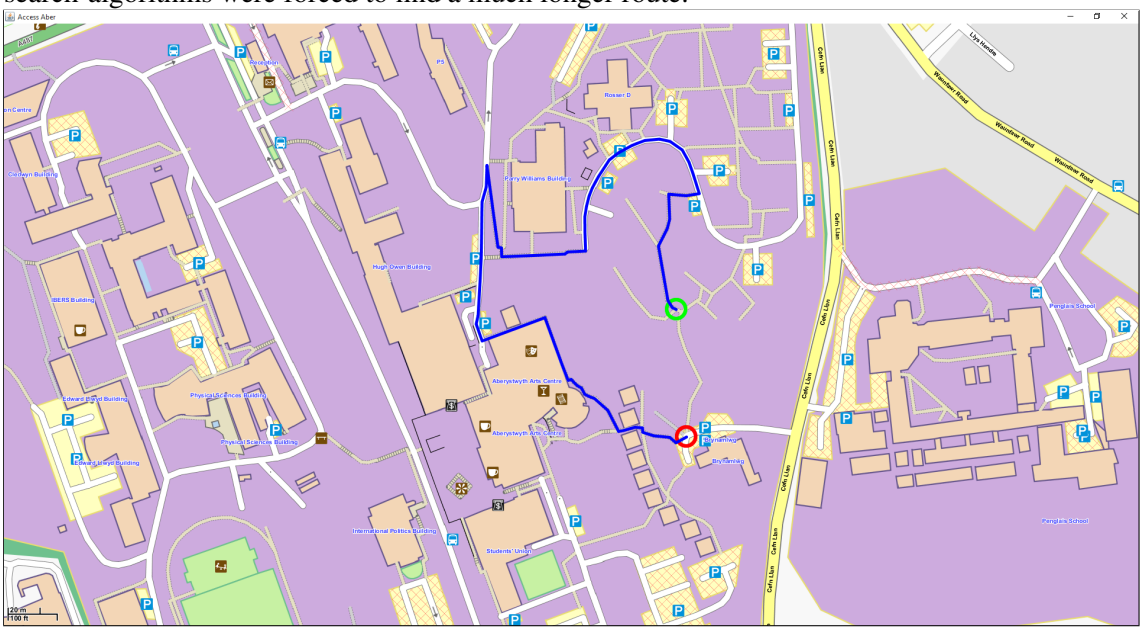
All routing-data is stored locally[1] in a *.osm* file downloaded directly from OpenStreetMap's website [27]. This download was restricted to a relatively small area, so the Penglais campus of Aberystwyth University was chosen as the testing-grounds during development because it is (almost) small enough to fit inside the boundaries of the download (See Figure 3.1) – and because the author is very familiar with this area, erroneous routes were easy to spot.

It is possible to download routing-data covering much larger areas from other servers like Geofabrik [5], but these files are usually massive, and use a lot of memory when unzipped, so the *.osm* file downloaded directly from OSM seems like a good compromise between space-complexity (when compared to Geofabrik), and time-complexity (when compared to downloading Nodes and Ways while the system is running).

The *.osm* file is read by an XML-reader developed by Lars Vogel [40], and copied into separate arrays for Nodes and Ways. Connections between Ways (not to be confused with the routing-data-

[1] All map-tiles and routing-data is stored locally. This is because it was too risky to rely on third parties to keep their servers running at all times while the system was being developed. The author's internet-connection was also quite unreliable at times, so making the system rely on online-data would make this project very vulnerable to further delays.

Figure 3.1: This image shows what can happen to a route if the algorithms are working with bad data. The path between the start and goal positions is not included in the .osm-file, so the search-algorithms were forced to find a much longer route.



type "Relations" [22]) are formed whenever two or more Ways reference the same Node, which is easy to detect because every Way links to specific Nodes in the array of Nodes, thus ensuring that they all reference the same data.

The *.osm* file also contains information about "Relations", but as these represent abstract areas like the outline of a forest or housing-block, they are not useful for pathfinding, and can therefore safely be ignored by the system.[2]

After all of the data has been copied from the *.osm* file, it is run through three filters to remove various types of data. One filter removes Ways that are inaccessible to PRMs (eg. stairs, residential housing, or Ways marked explicitly as inaccessible), another filter removes Ways that are unrelated to pedestrian pathfinding (eg. forests, railings, or traffic signs), and the third filter removes inaccessible doors from buildings, as well as isolated Nodes and Ways. These filters reduce the amount of data stored by the system, as well as reduce the number of possible paths available for route-planning – reducing run-times.

[2] "Relations" also represent things like bus-routes, which could make a route-planner aimed at PRMs even more useful if considered – but this will be touched on later in this report.

3.2 Search

This route-planning system will always plan accessible routes, as any inaccessible Ways are removed as the routing-data is loaded, and before any routing-algorithms are run.

Runtimes and memory-requirements are reduced by only planning routes via Tower Nodes, and skipping Pillar Nodes altogether (except when calculating path-costs). This abstraction of routing-data into only a subset of path-connections is comparable to the systems described by Yang et al. [41], and Botea et al. [2], with a few key differences.

This system groups Tower Nodes by the physical structures they represent, while most other hierarchical path-finding systems group Nodes by their geographical position, with respect to some artificial divide in the data. The latter form of hierarchical path-finding reduces the number of stored Node/Way-connections to an almost constant number of connections between areas, no matter how large the dataset becomes. My system will store at least two connections for each grouping (one entry and one exit point), and the number of total connections stored will therefore increase with the number of Ways in the routing-data. This means that other hierarchical path-finding systems scale much better than my system when given increasing amounts of routing-data, but as already discussed: Those systems sometimes find somewhat sub-optimal paths, whereas my system always finds truly optimal paths (provided that an optimal search-algorithm is employed).

A quite big issue encountered during development, was the way Java's priorityQueue-library handled an item's weights being updated after it had been added to the queue. If an item is in the priorityQueue, then its position will not change even if its weights do – unless it is removed and then added again. This problem was solved by first removing and then adding the Node to the priorityQueue again, but this could – in the worst case – result in iterating to the last position in the queue twice ($O(2n)$) if that Node's path-cost is worse than that of any other Node in the queue before and after being updated.

My system is not able to reuse old routes when planning new ones, which would be a very beneficial, and time and memory saving feature on longer routes. Because the user is only able to move either the start or goal-marker before a new route is planned, the new route could be planned from whichever marker was moved to the other marker, or alternatively a Node in the old route. If the new route enters the old route, then we know that there cannot be a better path to/from the Node it is connected to, so the rest of the new path can safely reuse the remainder of the old path to the marker that was not moved.

The issue with this type of reuse is that it assumes that the path-cost going in one direction is the same as the cost of going in the opposite direction, which is not true when considering the inclination of a path. The inclination of roads is not currently considered by the system when planning routes, but because this was a planned feature, there was no reason to consider reusing old routes[3].

[3] If the start-marker is moved, then the old optimal path to the goal-marker can be reused if it is found, but this is not true if the goal-marker is moved.

Figure 3.2: Algorithms avoid stairs

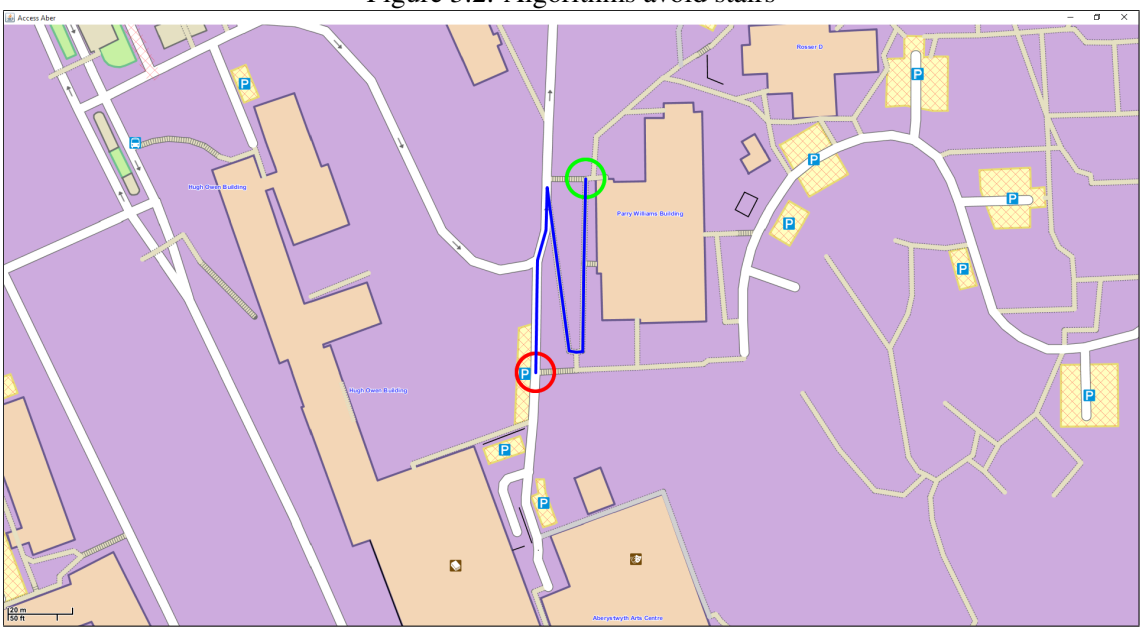
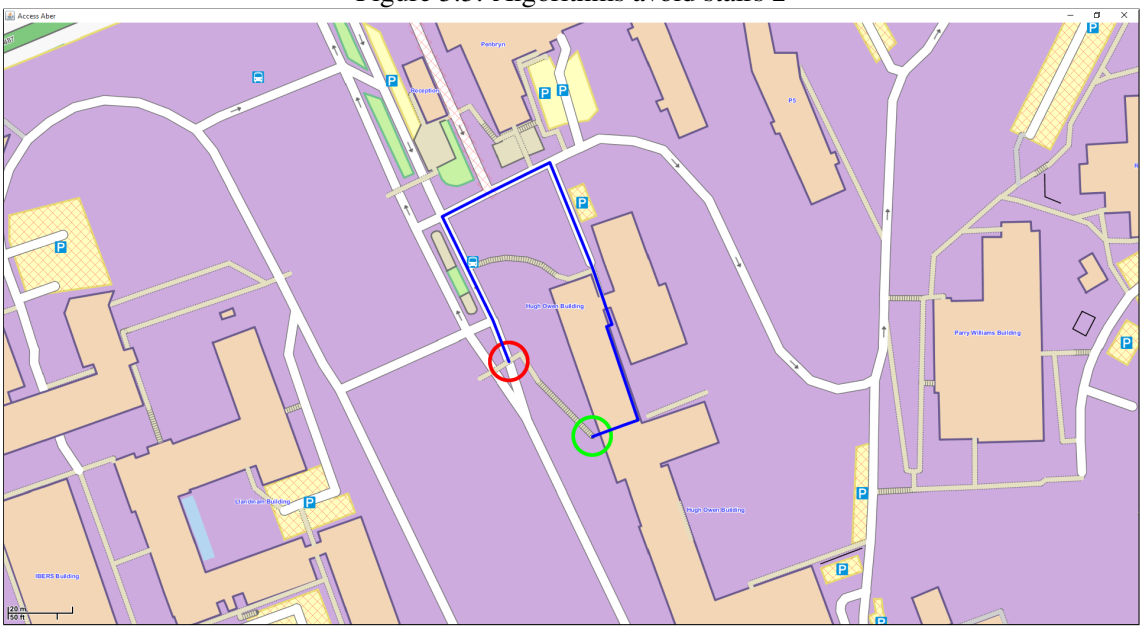


Figure 3.3: Algorithms avoid stairs 2



3.3 Map

The map used in this system has been provided by Mapsforge [13, 14]. Both the map-tiles and classes and methods used to display them come from Mapsforge[4]. The map-tiles are stored locally, as opposed to being downloaded whenever a route is planned in an area, because it takes some time to download this data without a dedicated server. A couple of free online map-tile providers were tested during the development of this system, but as they all restricted download-speeds quite heavily, displaying the map took a little too long, not to mention zooming or moving the map-view around.

The map-tiles stored locally cover the entirety of Wales, Great Britain, while the routing-data stored in the *.osm* file only covers the Penglais campus of Aberystwyth University, so these map-tiles are more than enough to display every possible route planned by the system.

In order to plan a route, you can either input five commands as arguments to the system before running it ((text)algorithm, (decimal number)latitude1, (decimal number)longitude1, (decimal number)latitude2, (decimal number)longitude2), or alternatively start it without any input-commands. If the system is started without any input-arguments, it chooses A* as the default routing-algorithm, and waits for the user to click on two separate locations before it plans a route between them. After a route has been planned and displayed to the user, they can click anywhere else on the map, and the start- or goal-Node will be moved to that location depending on which is closer.

3.4 Fulfilment of requirements

1. Make it clear that PRMs need to be considered by more route-planners

   This project has uncovered routes that may be fine for able-bodied pedestrians, but are completely inaccessible for PRMs; See Figure 1.1, where an able-bodied pedestrian would be able to walk in an almost straight line up the stairs to the Student Union, a PRM would need to take several detours.

2. Distinguish between accessible and inaccessible routes

   Three filters have been created to remove inaccessible and irrelevant data. The system is also able to plan routes through buildings, finding much shorter and practical routes for PRMs than most other conventional route-planning systems are able to.

   It should be very easy to add and remove labels in the filters, making sure that keeping the system up-to-date is relatively painless. The filters are only made for wheelchair-users though, except for the Building/Area-filter which can be a useful addition to any route-planner for pedestrians.

3. Find optimal and complete algorithm(s)

   A* is optimal and complete, and paired with the fact that the routing-data is abstracted by only using Tower Nodes for route-planning, the algorithm and system as a whole get many of the same benefits presented by a hierarchical path-finder.

[4]Third party mapping-services that were considered but decided against for various reasons: GraphHopper [8], Mapbox [11], Leaflet [39], Cloudmade [3], Omniscale [17], OpenLayers [18], and Google Maps [6].

This implementation of A* is still much slower than other hierarchical path-finding-algorithms when used to plan longer routes however, and works best on smaller areas. Most PRMs are unlikely to travel great distances though, at least on foot or in a wheelchair – which are the methods of locomotion that this route-planner is made for.

4. Algorithm(s) have to plan routes in near-real-time, and use as little memory as possible.

   The routing-data is filtered to only contain information relevant to routing, and paths are sped up by only using Tower Nodes while routing – making it possible to jump from one Way to another in a single Node-expansion. All of this reduces the time- and space-complexity of the system.

5. Display the routes on a map

   Mapsforge [13, 14] displays the map, using map-tiles stored locally on the system.

6. No localisation (eg. GPS) implemented, but added functionality to find the Node closest to some coordinates and set it as the start/goal position, thus making it relatively easy to use localisation to set the user's position in the future. The coordinates need to be in the same coordinate-system as OSM's routing-data though [26].

Chapter 4

Testing

4.1 Overall Approach to Testing

This project has been developed using Test Driven Development (TDD), which means that there is at least one test associated with every method implemented in the system. Instead of collecting every test in one single test-class, they have been split into separate test-classes for each class being tested. A top-level test-class has also been created to make it easier to make sure that every part of the system works as it should, not just isolated classes and methods.

As soon as the map was implemented, it became possible to visually analyse the routes planned by the system to make sure that they follow the expected path. The map also made it possible to compare the routes planned by this system to the routes planned by other route-planners like GraphHopper, Mapzen (both at: [27]) and Google Maps [6]. These comparisons showed that PRMs usually have to follow much longer routes than other pedestrians, but also that my route-planning system was able to plan routes through areas where other systems cannot – like through buildings or across open spaces like a piazza.

4.2 Automated Testing

Most of the Junit tests are fully automated, and only require the programmer's attention whenever they fail. Some test like the complexity-analysis tests do require that the user looks over the output however, as the runtime of each algorithm tends to fluctuate somewhat depending on the IDE, available system-resources, etc. There are Java libraries made exclusively for testing the time/space complexities of specific parts of the system, but they were a little tricky to import and implement into the system, so the simple complexity-analysis performed by the system at the moment will have to do. Those libraries made it sound like they provided higher accuracy than standard measurement-methods by eliminating variables like automatic code-optimizations, IDE-quirks, and fluctuating available system-resources, but they also seemed to be aimed at professional systems where minor optimizations down to the millisecond or nanosecond could have a massive impact on the runtime of the system.

A final automated test is the top-level test-class which runs every other test-class in order, working with the same data as the test-classes before it used and possible modified. This top-

level test class is useful because it lets us test the system as a whole, rather than as many separate modules.

4.3 Integration Testing

The top-level test-class made to test every other test-class in the system acts as a sort of "Big Bang" Integration test, where most of the system's modules are coupled together to form a complete system. There is also a special test-class made for the Main-class, where the entire system is run, and the programmer has to look at the routes created in order to spot any issues. This test-class does not interact with any other test-class, so if something fails, then the other top-level test-class has to be run in order to find which class and/or method the issue is located in.

The system has also been tested manually by placing the start- and goal-positions at different coordinates in an attempt to break the system, and force a crash.

Another integration test has been performed with a custom graph created specifically for this project, where the optimality and completeness of every search-algorithm can be tested, as well as ensure that all routes are accessible; See Figures 4.1, 4.2, and 4.3 for illustrations of what this looks like. The custom graph also contains a few Nodes meant to act as loops and dead ends, but because these were hard to include in the illustrations without making everything look really messy and hard to follow, the paths to them have not been included.

4.4 User Testing

The system has unfortunately not been tested in a real environment, so any routes planned by the system has just been assumed to be accessible for PRMs. There are a few errors in the OSM data used for route-planning (See Figure 3.1), where things like stairs that exist in reality are not marked in the dataset (for example on the path between the Llandinam building and Physical Sciences Building), but routes planned through these areas are not the fault of the system, but rather the fault of missing data.

As already stated: Routes passing through buildings cannot be guaranteed to be accessible on the inside of the building, but every entry- and exit-point is guaranteed to be accessible. This is because the interiors of buildings are rarely mapped in the OSM-database – evident by the lack of interior mapping inside buildings on the Penglais campus of Aberystwyth University – so there was no reason to develop this functionality in the system; it could be a very useful addition to further improve the functionality of the system however.

Figure 4.1: Inaccessible path in the custom graph. This route passes through a Way with the tag: *highway=steps*, and is therefore inaccessible. It is the shortest path in the custom graph, but should never be expanded by any algorithm.

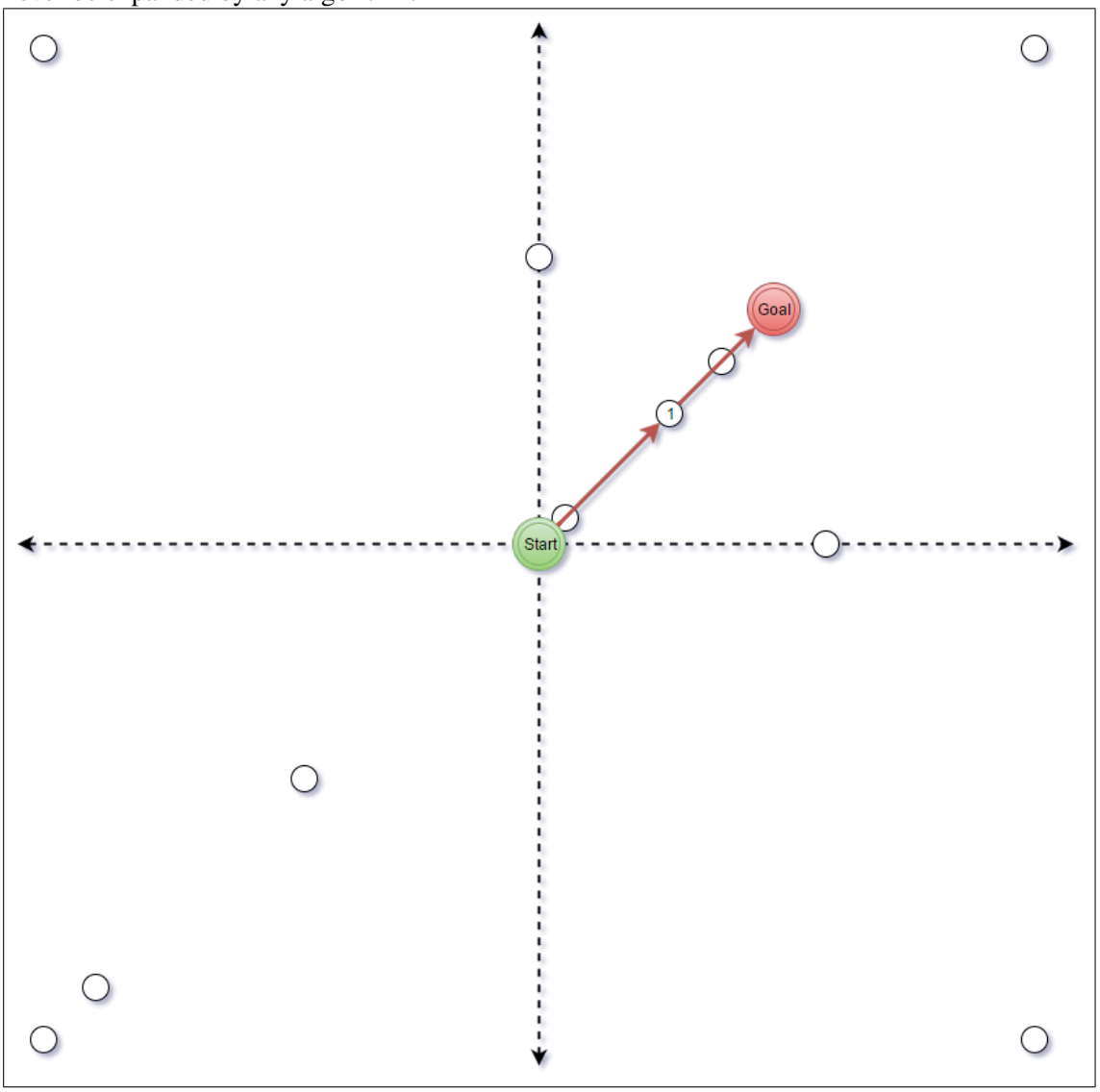


Figure 4.2: The longest path in the custom graph. This route is the longest with respect to distance, but passes through fewer Nodes than Figure 4.3. Uninformed search-algorithms like BFS and DFS, and greedy algorithms like GBFS are likely to follow this path.

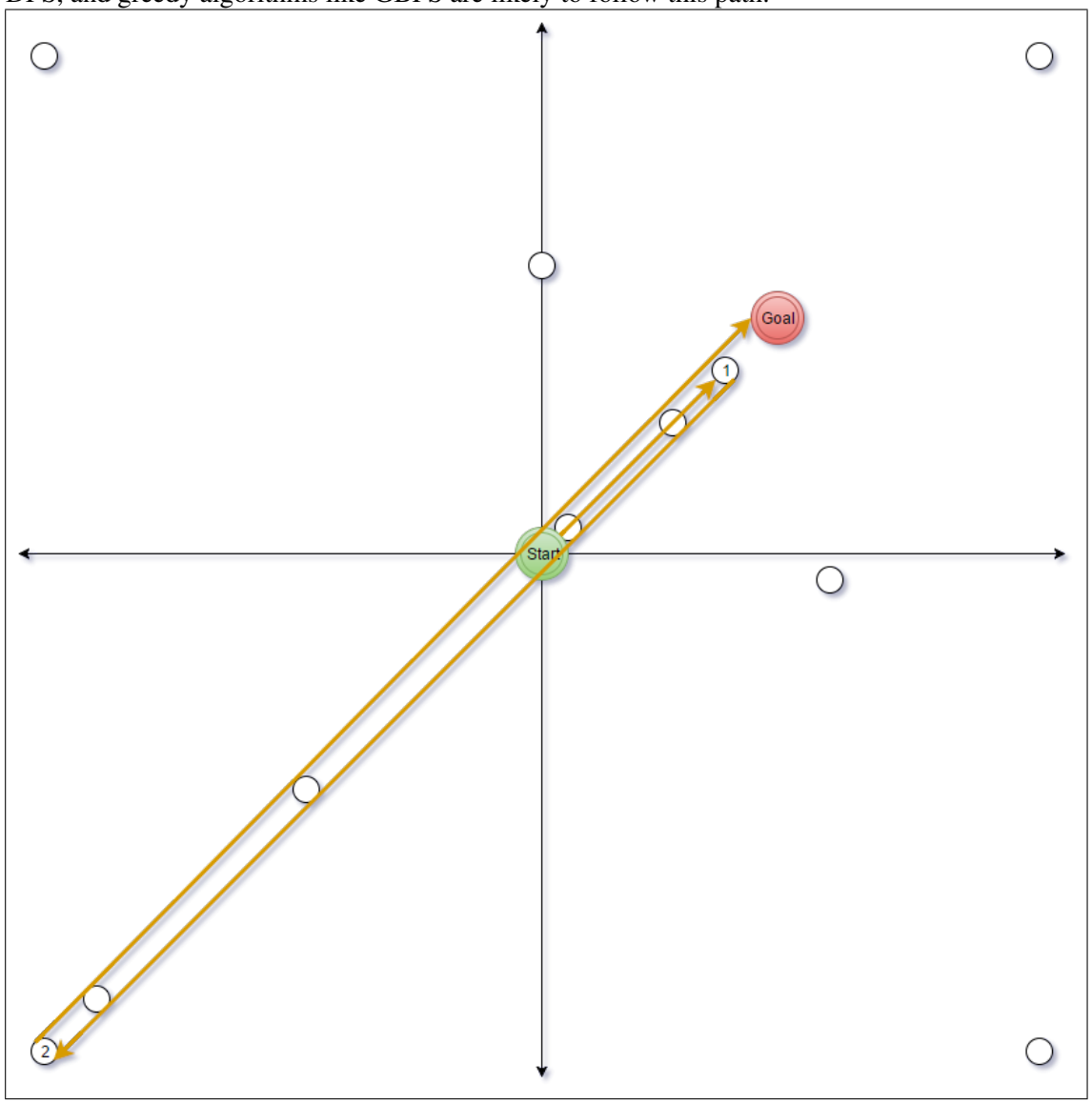
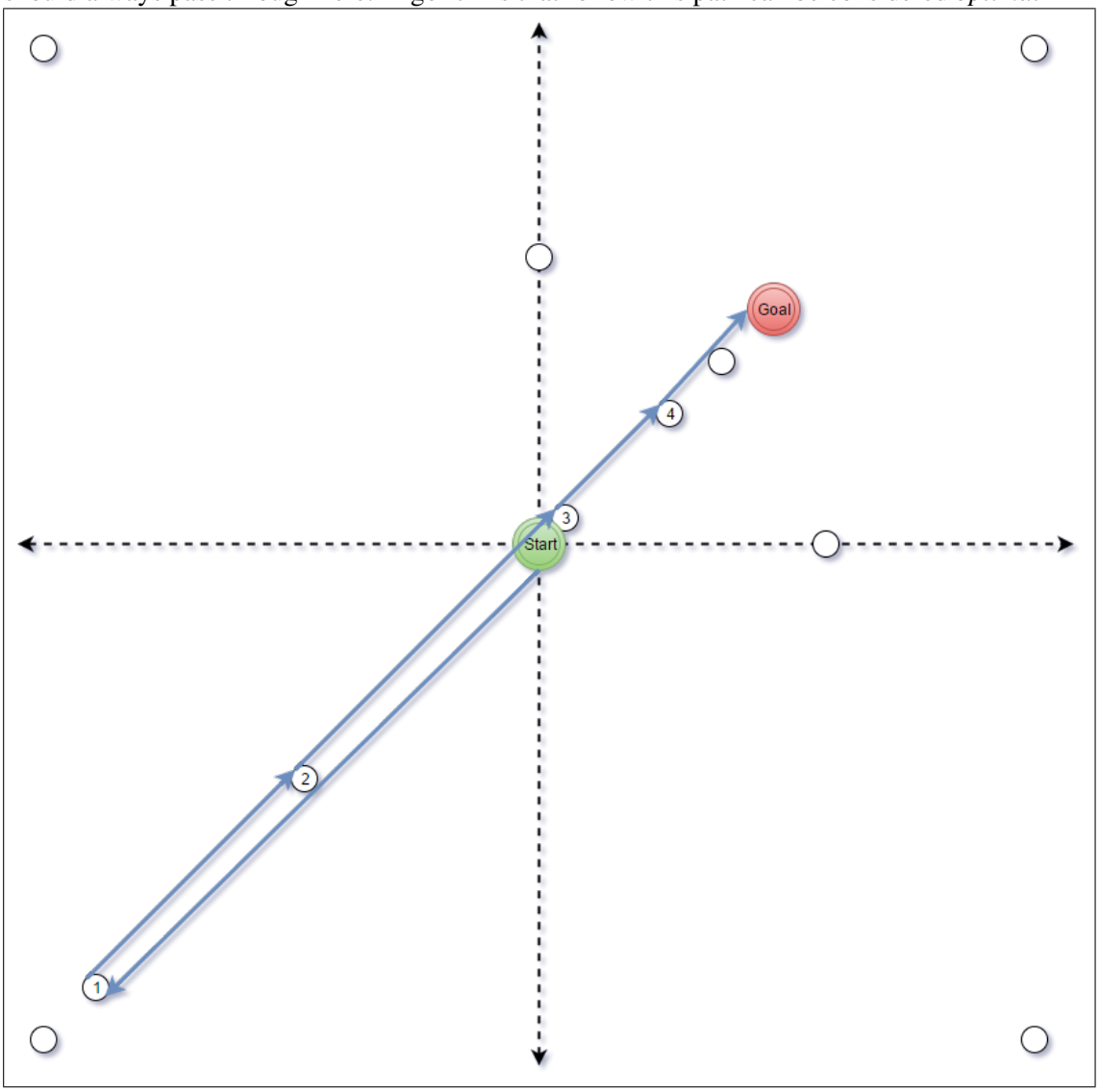


Figure 4.3: The optimal path in the custom graph. This route is the shortest with respect to distance, but passes through more Nodes than Figure 4.2. Informed search-algorithms like A* should always pass through here. Algorithms that follow this path can be considered *optimal*



Chapter 5

Evaluation

5.1 Completed work

I have made sure that I am allowed to use all of the data and third-party libraries used in this project, and have included most of the licenses in the bibliography at the end of this document, or in the Javadoc of my source-code.

I was able to fulfil the large majority of the functional requirements set forth at the start of this project, but I did not have time to ask Aberystwyth University for their records on accessible and/or inaccessible areas on their campuses, which would have let me make sure that the OSM data is updated. This does not have any adverse effects on my system though, as it is still fully capable of planning routes.
I did not have time to look into implementing any tracking-functionality either, which would have let me plan routes relative to the user's physical position. This would include running the system on an emulated mobile environment with a simulated GPS-signal, but this functionality is not needed in order for the system to be able to plan routes either.

Route-suggestions (especially through buildings) could probably be significantly improved if OSM's routing-data reflected the university's records on accessibility. Right now the route-planner just assumes that it is possible to travel from one accessible entrance to another within the same building, but when I walked around the Penglais campus to try to identify possible problem-areas, I noticed that some accessible entrances were on floors only connected to other floors via stairs (Like the new automatic doors on the bottom of the Llandinam builing, opposite IBERS). This could prove potentially problematic, but I don't know how to rectify this issue without mapping the interior of buildings, or marking the door as leading to an inaccessible area in the OSM database.

This system is most likely unable to plan routes quickly (and using little memory) if the start and goal Nodes are very far apart. This is because path-finding algorithms that abstract the search-space into larger blocks or sectors usually can't guarantee optimal routes, but justify this by being able to find routes quickly, using little additional memory. This system is aimed at PRMs, a group which includes people like the elderly and wheelchair-users, both of which may struggle to follow longer, sub-optimal routes – So I have made a concious decision to preserve the optimality of my algorithms and routes by abstracting the search-space less. This does lead to improved runtimes and memory-requirements, but could be improved even further by calculating path-costs between Tower Nodes before searches are performed, and ignoring Pillar Nodes altogether until the route

needs to be drawn.

The system is not always able to plan optimal routes. This is usually caused by missing routing-data (See Figure 3.1 and 1.1), or mislabelling of Nodes and Ways in the OSM database. These routes are optimal in the sense that there exists no better path in the available routing-data, but appear to be sub-optimal when viewed on the map where we are able to visualise the optimal routes ourselves.

The routing-data used to test the system is about a year old, and should probably be replaced by a new, more updated *.osm* file, but I don't want to update this file before handing the project in because this would result in the markers being unable to replicate the routes I show in many of the figures in this report.

If you want to see just how easy it is to update the routing-data used by this system, follow these steps (Last checked 27.September.2016):

1. Go to `www.openstreetmap.org`
2. Click on "Export"
3. Select any area of the map

   You do not have to select the Penglais campus of Aberystwyth University; the system should work on data from any urban area, but it has been developed specifically for the Penglais campus.
4. Click on "Export"again

   This will download a *.osm* file covering the area you selected.
5. move the *.osm* file into the folder containing my source-code, and replace the old *map.osm*.

   make sure that the new *.osm* file is also called "*map.osm*".

   It should be given this name by default, but it is best to make sure.

The Pillar Nodes inside Ways are not removed or skipped until the Way is expanded. This means that the path-cost from Tower Node to Tower Node via the Pillar Nodes between them has to be calculated while the routes are being planned. If the path-cost from Tower Node to Tower Node was calculated before the searches, then the system would resemble a hierarchical path-finder more than it currently does, and runtimes and memory-use could be decreased even further, while still ensuring completeness and/or optimality for complete and/or optimal algorithms.

5.2 Future work

The map does not currently display the distance of the routes it returns. It stores this value in a variable, but I was unable to find out how to print it on the map.

The system works with routing-data and map-tiles stored locally. Retrieving this data from a third party [5, 18] would make the system able to plan routes over greater distances, and make sure that it always works with updated data.

Nodes and Ways cannot currently be found and retrieved in O(1) time, as the system has to search through the arrays of Nodes and Ways in order to find the data instead of just jumping

straight to the correct index. A Hash map/table with a good hash-function could solve this, as they are able to search for and retrieve data in O(1) time by calculating the correct index, given a good hash-function.

Both Nodes and Ways are currently stored, but as long as inaccessible Ways and Nodes are filtered out when loading the data (As my system currently does), we only really need to store Nodes and Node-connections. The Node's type (eg. road, stairs, tree) would not matter, as any inaccessible or obsolete data will be removed before any path-finding is performed.

The user is currently not able to choose their method of locomotion or which type of PRM they are. Some PRMs like the elderly are are able to walk up a couple of steps, while others like those in wheelchairs would prefer to avoid stairs altogether. My system currently only considers wheelchair-users, but because it is very easy the change the three filters used to remove/retain routing-data, any other type of user could in theory be represented – even motorists.

My algorithms are not very well optimised for use on longer routes, and use a lot of memory and time compared to other hierarchical path-finding systems [2, 41]. The data is abstracted down to only Tower Nodes, but this is possibly still a lot more Nodes than other hierarchical path-finding systems need to store. My system can guarantee that optimality and completeness is preserved though.

There are a few places on the Penglais campus of Aberystwyth University that are labelled incorrectly, which can possibly result in inaccessible routes for certain PRMs. This can be corrected by updating these problem-areas manually, but I have not had the time to do this myself. The university probably has a few records pertaining to accessibility that could be useful to this end.

The system is not able to track the user and plan routes relative to their position. A route-planner should be able to recalculate a route if the user stops following it, as this could indicate that parts of the route might be inaccessible. The system is able to find the Node closest to a set of coordinates, which could be used to follow the user, but there is no localisation-functionality currently implemented.

Users currently have to click on or specify the coordinates of both the start and goal position before a route can be planned. The user might not know where it is, or might not know where the place they are trying to get to is. This could be fixed by letting the user search for building-names or similar identifiers, and plan routes relative to this information.

The data-type "Relations"is currently not used by the system to plan routes. But because "Relations"also represent abstract areas like bus-routes, they could serve as a great addition to a route-planner for PRMs who might not be able to travel over great distances.

Appendices

Appendix A

Third-Party Code and Libraries

Lars Vogel's XML-reader written in Java has been used to read and store the routing-data downloaded from OpenStreetMap [40]. Most of the code has been altered heavily to better fit my system and the data in the *.osm* file downloaded from OpenStreetMap.
The tutorial is open content under the CC BY-NC-SA 3.0 DE license [4]. The source-code is distributed under the Eclipse Public License [35].

Mapsforge's sample-code for displaying a map in Java has been used to display the map used by this system. Accessed 26.August.2016. The Mapsforge project is open source and is available on GitHub [12].
The sample-code is released under the GNU Lesser General Public License (GNU LGPL) [36].

Appendix B

Code samples

2.1 calculating the distance between two Nodes

```
public static double distanceBetweenPoints(double latitude1,
  double longitude1, double latitude2, double longitude2){

//Distance=squareRoot((x1-y1)^2 + (x2-y2)^2)
double distance = Math.sqrt(Math.pow(latitude1-latitude2,2)+
  Math.pow(longitude1-longitude2,2));

/*This ensures that all distances are >0 (CRUCIAL FOR PATH-COST
  CALCULATIONS),
* but some accuracy is lost when dealing with nodes really close to
  each other.
* UPDATE(21.July.2016): These calculations seem to work now - at least
  with the path I have tested everything on. -
* - finding the distance from a node to itself will still return >0
  (as it should)
*/

return Math.max(distance, Double.MIN_VALUE);
}
```

2.2 PermittedKeys

```
public enum PermittedKeys {

WHEELCHAIR(”wheelchair”,”yes”),
WHEELCHAIR_LIM(”wheelchair”,”limited”),
WHEELCHAIR_DES(”wheelchair”,”designated”),

FOOT_ACCESS(”foot”,”yes”),
FOOT_PERMISSIVE(”foot”,”permissive”),
BIKE_ACCESS(”bicycle”,”yes”),
BIKE_PERMISSIVE(”bicycle”,”permissive”),

FOOTWAY(”highway”,”footway”),
PEDESTRIAN(”highway”,”pedestrian”),
CYCLEWAY(”highway”,”cycleway”),
LIVING_STREET(”highway”,”living_street”),
RESIDENTIAL(”highway”,”residential”),
//PATH(”highway”,”path”),
//EQUESTRIAN(”highway”,”bridleway”),

BUILDING(”building”,”yes”),
UNIVERSITY_BUILDING(”building”,”university”),
PUBLIC_BUILDING(”building”,”public”),
CIVIC_BUILDING(”building”,”civic”),
//BRIDGE(”building”,”bridge”),//Are bridges always accessible?

AREA(”area”,”yes”),
SQUARE(”place”,”square”),//Is this the same as the Piazza in Aber?
CROSSING(”highway”,”crossing”),
SIDEWALK_LEFT(”sidewalk”,”left”),
SIDEWALK_RIGHT(”sidewalk”,”right”),
SIDEWALK_BOTH(”sidewalk”,”both”),

PARKING(”amenity”,”parking”),
//PARKING_DISABLED(”capacity:disabled”,null),
//DOCTORS(”amenity”,”doctors”),
PARKING_AISLE(”service”,”parking_aisle”),
ENTRANCE(”entrance”,”yes”);

String key, value;

PermittedKeys(String key, String value){
this.key=key;
this.value=value;
}
```

```
String getKey(){
return key;
}
String getValue(){
return value;
}
}
```

Annotated Bibliography

[1] BBC. Google Maps uses Ground Truth project to battle Apple. [Online]. Available: www.bbc.com/news/technology-19536269

[2] A. Botea, M. Müller, and J. Schaeffer, "Near Optimal Hierarchical Path-Finding," *Journal of Game Development*, vol. 1, no. 1, pp. 7–28, 2004.

[3] CloudMade. Bespoke routing-service provider. [Online]. Available: cloudmade.com/solutions/maps-portals/

[4] Creative Commons. CC BY-NC-SA 3.0 DE. [Online]. Available: http://creativecommons.org/licenses/by-nc-sa/3.0/de/deed.en

[5] Geofabrik GmbH. Geofabrik. [Online]. Available: download.geofabrik.de

[6] Google. Google Maps. https://maps.google.com/.

[7] ——. Google maps - terms of service. [Online]. Available: https://developers.google.com/maps/terms

[8] GraphHopper GmbH. Graphhopper. [Online]. Available: graphhopper.com

[9] JGraph ltd. free-to-license web graph-drawing application. [Online]. Available: draw.io

[10] A. Kishimoto, A. Fukunaga, and A. Botea, "Scalable, Parallel Best-First Search for Optimal Sequential Planning," in *Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09*, Thessaloniki, Greece, 2009, pp. 201–208.

[11] Mapbox. Mapbox. [Online]. Available: mapbox.com

[12] Mapsforge. Map-sample awt. [Online]. Available: https://github.com/mapsforge/mapsforge/blob/master/mapsforge-samples-awt/src/main/java/org/mapsforge/samples/awt/Samples.java

[13] ——. Mapsforge - homepage. [Online]. Available: mapsforge.org

[14] ——. Mapsforge - map-tiles. [Online]. Available: download.mapsforge.org/maps/

[15] NASA. Autonomous Planetary Mobility for the Mars Exploration Rover Mission. [Online]. Available: mars.nasa.gov/mer/technology/is_autonomous_mobility.html

[16] National Geospatial-Intelligence Agency. Wgs84. [Online]. Available: http://earth-info.nga.mil/GandG/wgs84/

[17] Omniscale GmbH & Co. KG. Omniscale en. [Online]. Available: omniscale.com

[18] Open Source Geospatial Foundation. Openlayers 3. [Online]. Available: openlayers.org

[19] OpenStreetMap Foundation. Openstreetmap license. [Online]. Available: www.openstreetmap.org/copyright

[20] ——. OSM - Features. [Online]. Available: wiki.openstreetmap.org/wiki/Map_Features

[21] ——. OSM - Nodes. [Online]. Available: wiki.openstreetmap.org/wiki/Node

[22] ——. OSM - Relations. [Online]. Available: wiki.openstreetmap.org/wiki/Relation

[23] ——. OSM - Tags. [Online]. Available: wiki.openstreetmap.org/wiki/Tags

[24] ——. OSM - Ways. [Online]. Available: wiki.openstreetmap.org/wiki/Way

[25] ——. OSM - XML file format. [Online]. Available: wiki.openstreetmap.org/wiki/osm_xml

[26] ——. Wgs84 conversion in openstreetmap. [Online]. Available: wiki.openstreetmap.org/wiki/Converting_to_WGS84

[27] ——. (2004) OpenStreetMap Homepage. http://wiki.osmfoundation.org/wiki/Main_Page.

[28] Ordnance Survey. A guide to coordinate systems in great britain. [Online]. Available: http://www.ordnancesurvey.co.uk/docs/support/guide-coordinate-systems-great-britain.pdf

[29] OSRM. Open source routing machine. [Online]. Available: project-osrm.org

[30] Route4Me, "Route4Me Route Planning SDK," http://route-planning-blog.route4me.com/2014/08/route4me-releases-c-sdk-for-route-planning-api-2/, Aug. 2014, accessed February 2015.

[31] E. Rowell. Big O cheat sheet. [Online]. Available: bigocheatsheet.com

[32] S. J. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Custom Library, 2014, pearson New International Edition. 978-1-292-02420-2.

[33] P. Sanders and D. Schultes, "Sanders/Schultes: Route Planning," http://algo2.iti.uka.de/schultes/hwy/, Feb. 2008, accessed February 2015.

[34] The Atlantic. How Google Builds Its Maps - and What It Means for the Future of Everything. [Online]. Available: www.theatlantic.com/technology/archive/2012/09/how-google-builds-its-maps-and-what-it-means-for-the-future-of-everything/261913/

[35] The Eclipse Foundation. Eclipse public license - v 1.0. www.eclipse.org/legal/epl-v10.html.

[36] The Free Software Foundation. GNU LGPL. [Online]. Available: http://www.gnu.org/licenses/lgpl.html

[37] T. Tinklin, S. Riddell, and A. Wilson. Disabled Students in Higher Education. [Online]. Available: http://www.ces.ed.ac.uk/PDF%20Files/Brief032.pdf

[38] Various. Osm routing-research projects. [Online]. Available: wiki.openstreetmap.org/wiki/Research

[39] Vladimir Agafonkin of Mapbox. Leaflet. [Online]. Available: leafletjs.com

[40] L. Vogel. Java and xml - tutorial. [Online]. Available: www.vogella.com/tutorials/JavaXML/article.html

[41] S. Yang and A. K. Mackworth, "Hierarchical shortest pathfinding applied to route-planning for wheelchair users," in *Proceedings of the Canadian Conference on Artificial Intelligence, CanadianAI 2007*, Montreal, PQ, May 2007, pp. 539–550.