First-year Project: Map of Denmark

Visualization, Navigation, Searching, and Route Planning (Spring 2022)

Course Code: BSFDVNS1KU

Group 27:
Rasmus Nielsen (rasni@itu.dk)
Lucas Alexander Bjerre Fremming (lufr@itu.dk)
Morten Falk Jonasson (mfjo@itu.dk)
Nicklas Lundsteen Jensen (nlje@itu.dk)
Jonas Westh Nielsen (jwni@itu.dk)

Date of hand in: 20. May 2022

Content

1 Preface and introduction

This report has been written in correlation with the First-Year project at the IT University of Copenhagen under the supervision of the course manager Troels Bjerre Lund and the TA (Teaching Assistant) Anne Mendelssohn Bartholdy-Falk.

In this course we have developed an interactive map of Denmark, which allows the user to move freely on a map and use many of the implemented features. These features, are among other things, a zoom which allows the user to focus on certain areas, search a desired address, find optimal pathways between two locations and so on. This project span across 11 weeks of continuous work with two progress presentations and a weekly TA update meeting.

2 Background and description of problem

2.1 Problem area

The application to be developed must be a visual representation of a map reminiscent of other map applications such as OpenStreetMap and Google Maps. One of the requirements must therefore be a runnable application which can execute map data retrieved from the previously mentioned OpenStreetMap. The specific requirements were furthermore described in the project description. It will primarily be these criteria in which the application will be developed from. The requirements can mainly be differentiated into two different sections. These are the usage requirements and the systematic requirements. The usage requirements are connected to the user experience and therefore are requirements for actions the user must be able to perform. Inversely are the systemic requirements the actions the application must be able to execute. These requirements were defined as follows:

2.2 Usage requirements for the program

#	**Usage requirements for the program**
1	Allow the user to specify a data file for the application.
2	Include a default binary file embedded as a resource of the program.
3	Allow the user to change the visual appearance of the map.
4	Adjust the layout when the size of the window changes
5	Feature a coherent user interface for your map.
6	Allow the user to search for addresses.
7	Allow the user to choose between routes for biking/walking, and cars.
8	Output a textual description of the route found.
9	Allow the user to add something to the map, e.g., points of interest

Table 1: Usage requirements for the program.

2.3 Systemic requirements for the program

#	**Systemic requirements for the program.**
1	Draw all roads in the loaded map dataset, using different colors.
2	Show any rectangle of the map, as indicated by the use.r
3	Show the current zoom level.
4	Where appropriate, draw other cartographic elements.
5	Unobtrusively show the name of the road closest to the mouse pointer.
6	Be able to compute the shortest route on the map between two points.
7	Be fast enough for convenient use.

Table 2: Systemic requirements for the program.

3 Analysis

3.1 Design choices for the programs internal structure

To keep the applications internal structure as simple and manageable, we decided to use the Model-View-Controller design structure in our project. That means the structure is divided and separated as *datamodel business logic (Model)*, *User Interface (View)* and *Program logic (Controller)*. This setup makes it easier to both maintain and expand the code. This also helps keeping the code with as low coupling and as high cohesion as possible.

3.1.1 Using a Parser

We started by using the parser from our hand-in assignments earlier in this course. This choice was made on the grounds that it would be easier to understand the parser whilst having it, rather than creating a new one. The parser was changed throughout the project, while still keeping the original structure. These changes were primarily additions to the `loadOSM` method. In the method we had to account for the different tags the OSM-file contained. The tags are responsible for a lot of the data we are using to create the map, so knowing all the tags would prove to be useful. This information was provided to us by *OpenStreetMap.org*.

3.1.2 Using Java to create a User Interface

The Model-View-Controller structure is applied through the `JavaFX` library. The library makes it easy to design a graphical user interface, by using .XML files to layout the design of an application.
Through our course `UX and Web-Programming` we have learned how to use markup-language to design a desirable user-interface. While working on the design of the application, our group have utilized `Scene Builder` which is a visual layout tool for `JavaFX` applications. This allowed us to quickly demonstrate different design concepts and solutions when we added new features to the application. We also chose to implement the library `ControlsFX` [3] which is an open source project for `JavaFX` that aims to provide useful UI controls and other tools to complement the core `JavaFX` distribution. This was implemented because they have a great method for textfields called `bindAutoCompletion(...)`, which takes the the textfield and a list as parameters and automatically adds a dropdown menu for auto suggestion when typing characters in the textfield.

3.2 Choice of IDE

We chose, unconventionally, to work in two different Integrated Development Environments. Those being IntelliJ IDEA and Visual Studio Code. While this might seem to cause problems, it was actually quite the contrary. The IDEs first of all were quite similar. Both are made to handle large projects and have a good implementation of controlling the system "Git". There were, however, a slight difference in, among other things, the way the debugger functioned. At times throughout development, we would encounter that the IntelliJ debugger would be more thorough and warn about e.g. possible NullPointerException where as the Visual Studio Code debugger would not care. This back and forth between IDEs resulted in a program which was more versatile, and surprisingly did not cause any issues.

3.3 Choice of algorithms

3.3.1 Address storing

At first we made a temporary solution, where we parsed all addresses from the given OSM file into a dedicated `Address` class, to be stored in an *ArrayList*. Each of these `Address` classes contains the street name, house number, postcode and city, as well as the x,y coordinate and id for each address node on the given map. We could then iterate through this *ArrayList* with the methods
Address.toString().contains().toLowerCase()
to find every corresponding address to an input string.
However we knew that this was a temporary solution since an *ArrayList* containing that many classes would take a long time to search through. given it's linear runtime. The upside to this solution was that it did not use that much memory compared to other solutions.
At the midway presentation, we saw that a lot of the other groups used a `TrieTree` to store their address data. After researching this type of tree we decided to use it, as we would rather have quick search and performance rather than a low memory usage, which the `TrieTree` fulfills. This would later become a downside to our program since our `TrieTree` used too much heap memory, because the size of a `TrieTree` ultimately is *O(N*C*K)*, where N is the number keys, C is the number of *Characters* in the `TrieTrees` alphabet, and K is the key length for the longest key [5].
We briefly discussed using a `Binary Search Tree`, a `Ternary Search Tree` or simply compressing our `TrieTree`, to reach a better combination of both performance and memory usage. Although, because of time constraints, we never got to test another tree structure to compare with our current one.

3.3.2 Path finding

Our path finding algorithm of choice is `Dijkstra's algorithm`. The benefits of Dijkstra is, once the algorithm has been run, you can traverse to every vertex from the source at little to no cost of extra runtime. Since we know the weight of an edge in our graph can not be negative, there is no need for our algorithm to detect negative cycles, like the Bellman-Ford algorithm does. However running the Dijkstra-algorithm can be time consuming since it traverses in every direction from our source vertex. This can be combated by upgrading the algorithm to A*, which could improve runtime if the right heuristics can be applied. Although, due to the time constraints we were not fully capable of applying the right heuristics, as we would rather have a more functional Dijkstra algorithm, than a weak A* with bad heuristics.

3.3.3 Storage of spacial data

Drawing a map this size is a big task. We started out with an OSM-file with a small cut-out of Copenhagen. With this small of a sample size, it is sufficient to run through all drawable content linearly. However, when working with bigger data sets, this method quickly fails. To store our data-points, we ended up with a choice between quad-trees and KD-trees. Troels Bjerre Lund went over these two structures in week 11, with his lecture. Both of the structures have their own strengths, however we found that a KD-tree would suit our project the best. A quad-tree splits into 4 equal regions, which can cause less efficiency since the points in a map are not equally spaced out. KD-trees on the other hand, can split data into 2 partitions based on some data analysis, which in our case would be the median node in a set of nodes. This makes it way more efficient at partitioning uneven distributed nodes, and is the main reason we chose to use the KD-tree as our data structure.

3.4 Design choices for the User Interface

3.4.1 Process of creating the layout

Before we decided how we wanted our UI to look, we went to big companies like Google and Apple to gain some inspiration. We noticed that they have their `Canvas` to fill the whole application, while the UI components are placed on top of it and to the sides. We found this to be a significantly good way of emphasizing the user to have their attention set to the canvas whilst still keeping the UI components straightforward and to the sides and top, just like Google and Apple. We started off by having a side-panel with every UI component that could be featured from what we currently had ready in the back-end. As the time went on, we kept expanding the side-panel with features and ended up liking it more than the "On top of the canvas" components. So we decided to let the final version have the simple and minimalistic side panel. The downside is less *Screen real estate* for the `Canvas`.

3.4.2 Design choices for the features

We noticed how companies like Google and Apple have their address searches in either the top corner or bottom corner. So from the beginning, we decided to have our address features in the top left corner, as that would be the most common feature for the user. For the simplicity of the design, we chose to have a switch button that determined whether the user were looking for a specific address or looking to create a route between two addresses. When the switch is turned off, only a single text-field is shown. If it is turned on, a new text-field, including transportation, travel time, and a travel distance. A direction list is then shown to make sure the user would not get confused in the process.
At the bottom of our side-panel we have our *Debugger Menu.* This includes 4 check-boxes: *FPS Counter, KD-tree Visuals, Nearest Neighbour Visuals and Dark-Theme mode.* Following up we have the graphical interpretation offer

the zoom values together with a zoom in- and zoom out button. These two features were added after loads of debugging hours trying to fix the very unstable `onScroll` method for zoom - especially on touch pad. These were later fixed for a more stable approach when zooming on a touch pad, but we saw no reason to remove the buttons anyway. The zoom visual increments with 10% per zoom scroll. At first we used the `getDeltaY()` method from the scroll event on the canvas. This was a smart move, to make the scroll more smooth and consistent across different display resolutions. The reason we chose to set a fixed value at *50* was to make sure the touch pad scroll and mouse wheel scroll would be more consistent between the two scroll types. Additionally we implemented a `Thread.sleep()` method, so there was more control over the touch pad scrolls, which would tick a lot faster than the mouse wheel. The downside for the `Thread.sleep()` method is the less responsive feeling it may give to the user as it creates a small delay between the scroll inputs.
When opening the program, the user is welcomed by a start page. The start page is very simple containing two buttons. The first button `Load Bornholm` opens a map of Bornholm as we weren't able to load the entire map of Denmark. The other button is a file-loader, so that the user is free to load another OSM-file if that would wish to. Pressing one of the buttons will trigger the splash page, which is a responsive loading page. This would ideally give the user a feeling that something is indeed loading in the background. At the start of our project, loading the program would make it "Not responding" until it was done loading the map. This could interpreted as something was off, even though it was just loading. This was done using different threads so the splash GUI, with the responsive progress bar, would not get affected by the program loading the whole map and "Not responding".

4 User manual

4.1 Opening the program

When first opening the program the user will be greeted by a splash screen asking whether to load the default map, Bornholm, or load a custom OSM-file. After choosing one of the two maps, a loading screen will open, and when done loading the intended map is displayed on the screen.
At the top of the map we have two drop down buttons, one is the *File* button, to load a new map from a custom OSM-file, and the other an *Info* button, that when pressed shows an *About*,- and a *Hotkeys* button. The *About* button will show a popup with the names of the creators of the map, along with a link to our GitHub repository.
The *Hotkeys* button will open another popup, which shows the different hotkey combinations that can be used on the map, either to create a route or a point of interest.

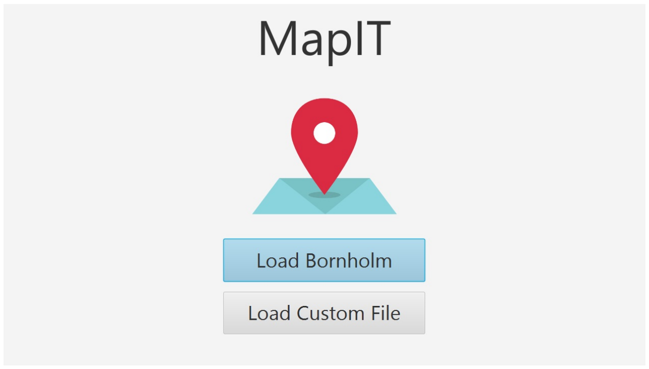


Figure 1: MapIT Startpage

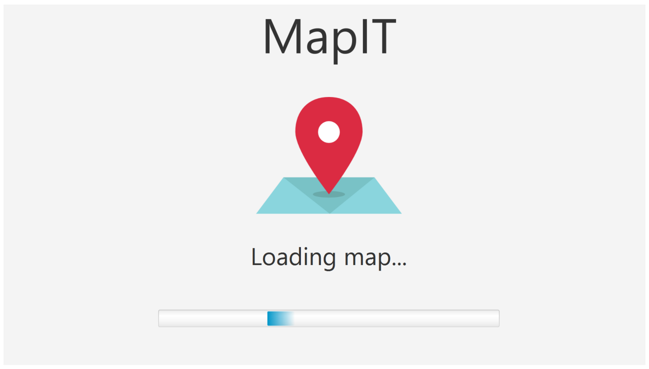
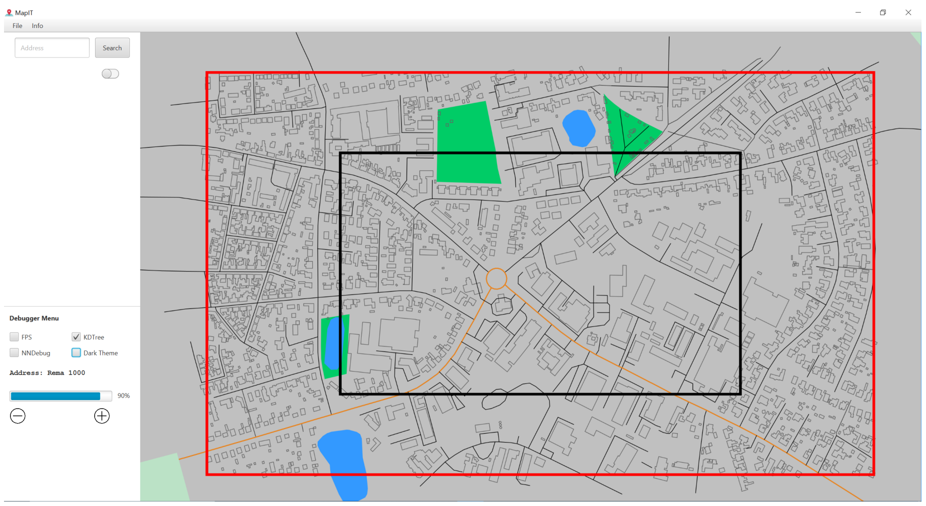


Figure 2: MapIT Splashpage

4.2 Debugger menu

On the left hand side of the map is where our *Debugger Menu* is presented. Here we display the zoom level, along with a plus and minus icon for zooming in and out which can be done with the mouse wheel as well. On top of this is the nearest street field, which at all times shows the closest street to the current mouse position on the map.
Above this we have our four debug settings in forms of buttons the user can switch on or off. Firstly we have a *FPS* counter which displays the current FPS of the map. This number is only recalculated when the map is repainted, which means it only updates when the map is being moved. Therefore it is not always precise, since you have to move the map around a bit before it displays the correct FPS-count.
To the left of this we have the *KDTree* debug function. When toggled two squares are displayed on the map. A smaller black square along with a bigger red square. The black represents the screen size, that meaning what the user can see. And the red square is the buffer, meaning the extra space being rendered for a more smooth experience when dragging the map.
The third button is the *NNDebug*, short for Nearest Neighbour debug. When toggled, every left click on the map will highlight the closest street to the mouse in a red color. This is to show the user how our routing works, since it uses this feature to choose start and end points.
Lastly we have our *Dark Theme* mode. This will make the entire map and sidebar dark themed, thus making it easier on the eyes in lower lighting.



FPS [✓ KDTree]
NNDebug Dark Theme

Address: Rema 1000

Figure 3: MapIT KDTree Debug

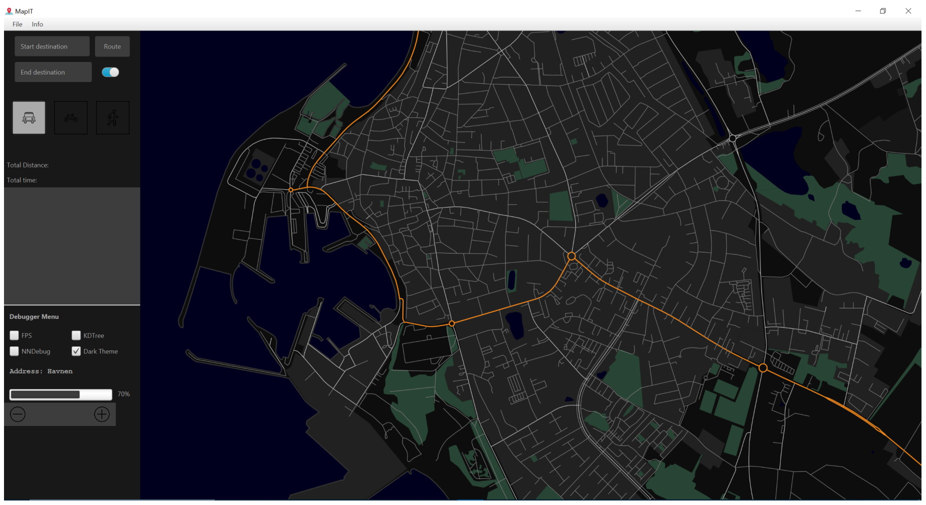
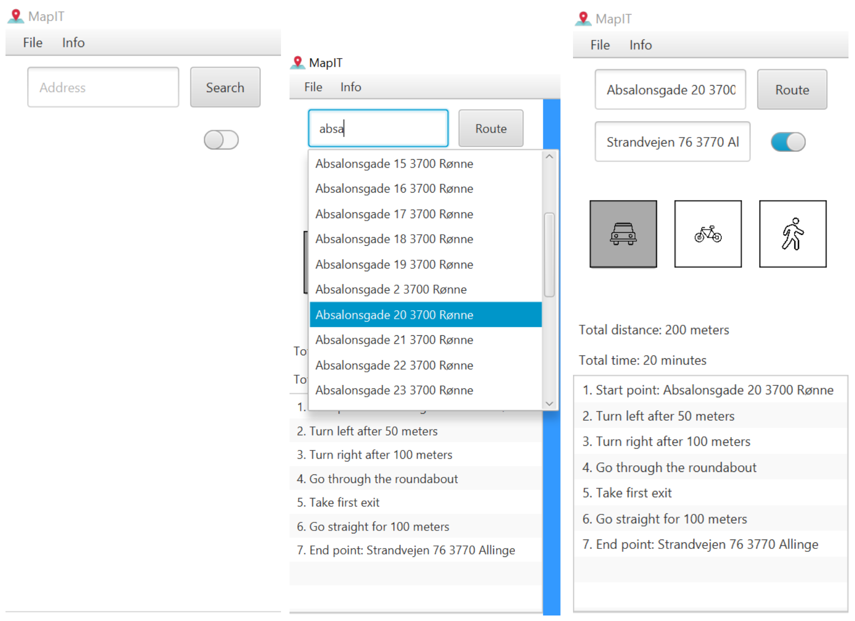


Figure 4: MapIT DarkTheme Mode

4.3 Searching for addresses and routing

When wanting to use our routing function, there are two ways to do it. The first way is to only search for one address. This is done via. the search bar at the top left, which will auto suggest every possible address with your current text input. After the user have chosen a full address from the drop down, the user can press the *Search* button and a red circle will indicate the chosen address on the map.
The second way is to toggle the switch under the search button. This will make a second search bar appear, as well as three buttons to choose your preferred transportation mode. When two different addresses have been entered, the user can choose their desired way of transport and hit *Route*. This will show a red circle for starting point, and a pink circle for end destination on the map, along with a yellow line to show the route. At the current moment in time, every transportation method will display the same route. We made a *Total distance* and *Total time* counter as well as a textual route guide. These are however not working at the moment and will just show a fixed distance, time and guide.
Another way to get a route between to points are by holding left control along with left,- and right clicking your mouse. This will, for left click, add a starting point, and right click, add an end destination. When doing this it is not needed to choose a transportation mode, and the search bars will not update with the new addresses.



Absalonsgade 15 3700 Rønne
Absalonsgade 16 3700 Rønne
Absalonsgade 17 3700 Rønne
Absalonsgade 18 3700 Rønne
Absalonsgade 19 3700 Rønne
Absalonsgade 2 3700 Rønne
Absalonsgade 20 3700 Rønne
Absalonsgade 21 3700 Rønne
Absalonsgade 22 3700 Rønne
Absalonsgade 23 3700 Rønne

Absalonsgade 20 3700

Strandvejen 76 3770 Al

Total distance: 200 meters

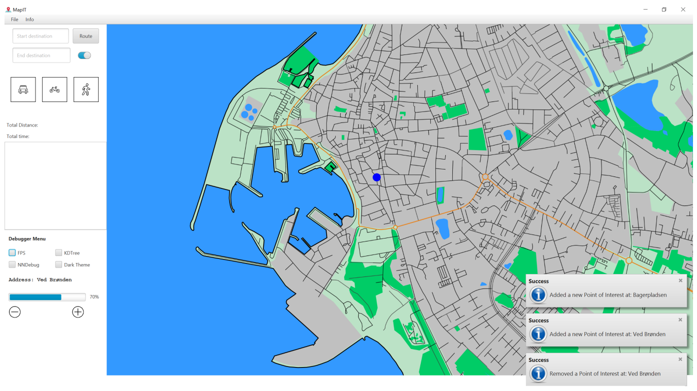
Total time: 20 minutes

1. Start point: Absalonsgade 20 3700 Rønne
2. Turn left after 50 meters
3. Turn right after 100 meters
4. Go through the roundabout
5. Take first exit
6. Go straight for 100 meters
7. End point: Strandvejen 76 3770 Allinge

Figure 5: MapIT Address Search

4.4 User pins

An additional feature for our map is user pins. These work by holding left shift and left clicking a point on the map. This will create a blue circle along with a text popup which informs you at which address you have added a personal point of interest. For deletion of a *POI* the user can hold left shift and left click on a blue circle, this will remove the *POI*, and a popup will show with a message of deletion.



Success
Added a new Point of Interest at: Bagerpladsen

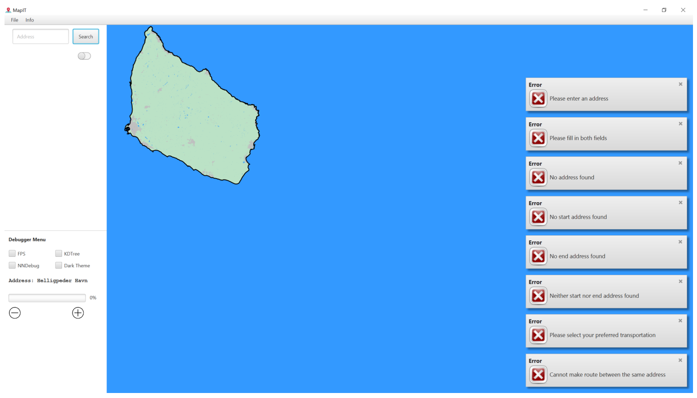
Success
Added a new Point of Interest at: Ved Brønden

Success
Removed a Point of Interest at: Ved Brønden

Figure 6: MapIT POI

4.5 Error messages

Our program uses popups as error messages. If the user have not entered an address and still hit the search or route button, a popup will appear with the message *"Please enter an address"*, or *"Please fill in both fields"*. When trying to search for a route without choosing a transportation mode you will get a *"Please select your preferred transportation"* error.



Error
Please enter an address

Error
Please fill in both fields

Error
No address found

Error
No start address found

Error
No end address found

Error
Neither start nor end address found

Error
Please select your preferred transportation

Error
Cannot make route between the same address

Figure 7: MapIT Error Messages

When trying to make a route between two addresses on the same street you will get a *"Cannot make route on the same street"* error, and searching with an invalid input will create a *"No address found"* error, a *"Neither start nor end address found"*, a *"No start address found"* or a *"No end address found"*, depending on whether the route switch is toggled or not.

5 Technical description of the program

5.1 OSM parser

Our `Model` class is responsible for the entirety of the parsing process, and is the foundation of our application. Firstly it recognizes whether the loaded file is a `OSM or ZIP` file, and then creates a *FileInputStream* from the file. This input stream is then sent to our *loadOSM* method which creates a *XMLStreamReader* from the input stream.
Next we iterate over every line in the file using a *While* loop, and a *Switch case* is then in charge of determining whether the read line is a `OSMNode, OSMWay, tag etc`.
The `Tag` case is in charge of the details of the program, and therefore is by far the longest. Here we use a nested `Switch case` to determine what the tags value is. It is in this case we assign all the `WayType enums`, get all address info for our address storing, as well as, get all highways in the map.
Lastly we have another `Switch case` for *END ELEMENTS* in the file. This creates all the highways, along with all the *multipolygons* we use to draw forests and lakes.
After the while loop ends we have two *System.out.println* statements that prints the time the parsing took, along with the time the filling of the `KDTrees` took. This is purely to our own benefit to see if changes within the code improves the loading time.
The last third of our `Model` class consist of smaller methods we use to create everything from the `TrieTree`, `KDTrees` and `Graph`, to get info like the *minlon* or *maxLat*.
Ultimately, we would have liked to have split this `Model` class up into two classes, a dedicated parsing class, and a class for all the other methods and variables we create in the `Model` class. We never got to implement this change however, since it got down prioritized to other, more important aspects of the project.

5.2 Address storing

When we figured out how to parse the address in from an OSM-file, we made a dedicated `Address` class for each address found. This was the easiest and quickest way to bind an address to an `OSMNode`, which gave us the exact x,y coordinates for each address, which we knew we would eventually have to use for our path creation.
Initially we stored every address in an ArrayList of `Address` classes, but knew we had to use a quicker data structure for searching through these as the list grew in size. We choose to use a `TrieTree` for this.

5.2.1 TrieTree

A `TrieTree` is a type of search-tree used to locate specific keys. It works by splitting every key up into characters which is then assigned to a node determined by the ASCII value of the character. Each node then has a pointer to the next character in the string. for an example of the structure refer to appendix 8. In order to then retrieve a key from the tree, a *Depth first search* is implemented. We made a few additions to the traditional `TrieTree`, by assigning a x,y coordinate to each key. As well as implementing a auto complete function using a *recursive depth first* algorithm.
The reason we chose a `TrieTree` is because it is one of the quickest data structure for inserting and searching through strings, since the worst case running time for a `TrieTree` insertion and search is: *O(n)*[5] where n is the length of the input key. Our `TrieTree` worked by inserting every complete address we had into it. This did later give us some memory issues, since the depth of the tree ultimately is the longest keys length. And since we used a complete address (Streetname + housenumber + postnumber + city) this key could potentially be incredibly long. Furthermore since we used numbers and whitespaces our `TrieTree` alphabet also were 37 characters instead of 26.
As mentioned the downside to using a `TrieTree` is that is uses a lot of memory. And we later ran into memory issues when loading bigger maps, such as a complete map of Denmark which gave us a memory exception:
java.lang.OutOfMemoryError: Java heap space. We realised that the way our `TrieTree` alphabet worked, as well as how we stored each address in it, were really poorly optimized and used way more memory than needed. This were because of, the way we chose to store each address, as a complete string with both, streetname, number, postnumber and city, which then resulted in very big keys. And because of this we had to increase the alphabet to both the English alphabet plus numbers and white spaces. This resulted in an alphabet size of 37 which resulted in the tree size being $alphabetSize * keyLength * N$, [5], N being the number of keys in the tree.
We briefly looked at how to compress our `TrieTree` so that each node could hold more characters instead of a single one: refer to appendix 8 And therefore minimizing the `TrieTree` size along with memory usage. To save even more memory we could have changed the input keys we used, and decreased the

alphabet size to make the tree as small as possible. We also realised that the dedicated address class is obsolete since it is only used as a middle ground before we parse into the `TrieTree`, when we could have parsed directly the addresses directly into the `TrieTree`. However because of time constraints we never got to implement any of these optimizations.

5.3 KD-Tree

K-Dimensional trees, otherwise know as kd-trees are data structures used for space-partitioning data. The data structure behave much like a binary tree, with each of its node representing a point in the multidimensional space. The main uses for kd-tree are nearest neighbor searching range queries, which we have been using our trees for in the project.

5.3.1 Structure

Like stated above, a kd-tree is similar to binary trees, which results in a structure with bigger nodes on the right side, and smaller on the left. In our project there was only a need for partitioning 2 dimensions: X, Y. What makes kd-trees different from binary-trees is that for every layer in the kd-tree, we compare a different dimension. In our case, with 2 dimensions it goes like: For depth 0, we compare the x value, then in the next layer, depth 1, we compare the y-values. tree.
"A kd-tree for a set of n-points uses O(n) storage and and can be constructed in O(n log n)."[1]
An example of a 2-dimensional kd-tree can be found at appendix 10.

5.3.2 Range Query

To figure out which parts of the map we should be drawing for the user, we made use of range queries in our kd-trees. The query starts by looking at the root node of the tree, checks if the node is inside the given range, and adds it to the list of it is. Afterwards it checks whether the 'less' side of the range is less than our current nodes value: if we are comparing x values we check of the left side of the range is less than the current nodes x-value, and if the right side is more than our current node's x-value. The Same thing happens when comparing y-values, just for the top and bottom of the range. If any of these are true, we will call the same function on the appropriate child and continue our search. All of this results in an ArrayList of things to draw, and we can run through that linearly to draw them. *"A rectangular range query on the kd-tree takes O(n + k) time, where k is the number of reported points."* [1]

5.3.3 Nearest Neighbor

Nearest Neighbor was used to find the closest road to a given point. This algorithm was used to find the closet road to the mouse, as well the closest road to an address. Like the range query, this algorithm starts with the root of the tree. Checks the distance to the target and stores it. It then looks to see if the target is less or greater than the node's x or y, based on the current layer of tree. If the target is less than, we check the left child, and vice versa if the target is greater. Every time we check a new node, we test if the distance to the target is smaller than the previously recorded shortest dist, and change it if that is the case.
As well as checking the closer branch, we need to determine whether a closer point could be in the farther branch. To do that we check if the distance between the target and the partition is less than our current shortest dist.
Nearest Neighbor Search *"has been shown to run in O(log n) average time per search in a reasonable model. (Assuming d a constant)*[2]
For a visual representation of nearest neighbor search, refer to appendix 11.

5.3.4 Limitations "TODO: Find correct word"

Line are not points, and that brings with it some complications in terms of filling a kd-tree. The solution we went with to fix this issue was to get the average x, and y of all the nodes making up a line. This average coordinate would be the representative for the drawable. This however, brought its own complication. Since a drawable is defined from its center, a part of it could be in the range query and not be drawn, since the center could be outside. To combat this, we implemented a buffer around the users screen. This bigger range would make sure that the amount of cases where this problem occurred would be minimized.

5.4 Graph-structure and pathfinding

The graph structure for the application is only applied to the highway-tag that is parsed from our data file. We chose to not include tags `path, track and cycleway` in our structure, because it gave us unexpected results from our Dijkstra-algorithm(*can be seen on figure 9*). The algorithm would sometimes choose a cycle-way, track, or path because it turned out to be a shorter path than just staying on the highway. Preferably we would have three graphs, one for pedestrians, one for bicycles and one for cars. We have not included it for pedestrians and bicycles because of time constraints.
When parsing information in the `Model` class and `loadOSM` method, we are checking different OSM tags through a switch case. When we get to the tag `highway`, we are setting the `isHighway` boolean to true (*can be seen on figure 12*), so we know later in the switch case to parse the information further into an arraylist of highways. Furthermore we collect data from tags `maxspeed and name` to create a new `OSMWay` object if `isHighway` is true.
Once we have parsed all information concerning highways into the `highways` arraylist, the `createGraph()` method is used. The method uses a foreach-loop

to run through all `OSMWays` in `highways` and runs through each `OSMWay's` list of `OSMNodes`. We are creating an `Edge` object with the appropriate information such as id's for vertices `from, to, from2, to2` and `weight, distance` for each node in a `OSMWay` through a for-loop. Currently the weight is the same as distance, because of an unexpected result, when debugging on a map of Fyn. Each `Edge` is then added to an arraylist of edges; `edgeList`.
When creating an object of the `EdgeWeightedDigraph` class, you need to specify the amount of vertices in the graph. We chose to create an index `id2` that keeps track of every OSM-node, because the graph should not contain more vertices than there are OSM-nodes. Even though all nodes are not used for highways, this was an easy way of solving the problem. Once the foreach-loop is done, we are adding all edges in `edgeList` to our graph.
The classes `Edge, EdgeWeightedDigraph and Dijkstra` were initially from the textbook:
`Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne`[4],
and eventually fitted to meet our demands. To traverse the graph we are using the `Dijkstra` class, which uses a source vertices and a graph. Initially we intended to use the A* algorithm, but due to limited time we prioritized on functionality firsthand. This is why the Dijkstra constructor also takes a second integer. This was intended to be the target vertex in the graph, and our plan was to use the target for heuristics in the `relax(Edge e, int t)` method. A path is eventually created when the method `drawablePath(int v)` is called. The method simply traverses the path to the given vertex v, and adds all nodes in the path to an arraylist. The arraylist of nodes is used to create a `PolyLine` object that is returned by our method, and displayed in MapCanvas. The total runtime for the Dijkstra algorithm is `O(ElogV)`[4] where E is the number of edges in our graph and V is the number of vertices. Since we created our graph, intentionally, with a higher amount of vertices than needed it would be desirable to look into how to create the graph with the right amount of vertices.

6 Testing, Debugging and Bugs

6.1 Debugging

Throughout the entirety of our coding process, we implemented a lot of debugging. This was entirely done by *System.out.println* statements within our code, to debug whether or not a chosen method were working as intended, or a chosen variable were being instantiated properly.
We used these for the bigger algorithms, the kdtree, trieTree and the dijkstra-algorithm. But also if we ran into unexpected exceptions. Overall this combination of reactive debugging, along with preemptive debugging helped out a lot. And we could quickly find and fix the issues within the code by using these *System.out.println* statements.

6.2 Testing

We never implemented the desired testing we wanted to, because of time constraints. Although, when looking back, we could have saved our self some time by implementing `JUNIT testing`, which would have made it easier and quicker to test the implemented methods, instead of using
System.out.println statements.

6.3 Bugs

#	**Bugs**
1	The `TrieTree` does not reset if a new map is loaded.
2	Draw fills might not be placed at the correct coordinate.
3	Holding shift while zooming only zooms out. And does not affect the zoom value.
4	Typo in the *Hotkeys* menu regarding deleting a POI.
5	Loading bigger maps is not possible because of memory usage.
6	When routing you get the route to a road and not a precise address.
7	Not possible to load in a map containing characters not in the `TrieTrees` alphabet.

Table 3: Known bugs in the application.

7 Process description

7.1 Form of work

7.1.1 Physical attendance

One of the first things we agreed upon in our group contract, was that we would physically meet at least once a week at ITU. This was typically on Wednesdays. We would meet at 10 am, and work until we were happy with the progress made. There were several aspects to a physical meetup, which were very pleasant. These were, among other things, that it was easy to get feedback from one another and show screens to get input on issues. Another big plus were the social aspects. We would grab lunch together and talk. There were, however, some downsides to this. Every group member preferred to work from our PC at home rather than a laptop. We would combat this by also having days were we would meet online.

7.1.2 Virtual attendance

Virtual days were typically on Fridays. Here we would use the communication software "Discord". This gave us the opportunity to communicate and share files with one another. We were able to emulate "looking at each other screens" feature by using the "Share Your Screen", thus having similar work flow to physical meetings. This made the group very effective, since we were able to be on our personal computer. We were all adjusted to working from home, duo to the previous years of Covid-19 lockdown. We were therefore also aware of the issues this could cause, such as lack of motivation and how easy it was not to be productive. We would face this issue by having a concrete plan and hand out work responsibility.

7.2 General issues

None of the group members were used to a project of this scale, let alone with this group size. This lead to an overwhelming start of the project, where the group was not as productive as intended. By the time of the second progress presentation, we realised that we were beginning to fall behind. This lead to the last few weeks of the project being very stressful and filled with tasks that needed to be done. Another issue we ran into, was that within a larger group, it was easy to lose track of what members have been working on and what needed to be done.

8 Product conclusion

Through our analysis we aspired to fulfill all the expected requirements. Whilst working on the project we realized that compromises were inevitable and we had to prioritize certain things above others. Things such as fastest path with Dijkstra, a different type of search tree, or more efficient storage of data, are certainly things that we wanted to look more into. Nevertheless we ended up with an application that is capable of loading somewhat bigger maps, in which you can search for addresses and find shortest paths efficiently. We could have saved a lot of time if we implemented testing instead of debugging every time we ran into issues. Overall our time management could have been better structured, as we experienced stagnation in our progress.
Through all this we can conclude that our group has made an application that is capable of drawing a map using OSM-data.

9 Sources

References

[1] Hermant M. Kakde, 2005, *Range Searching using Kd Tree.*
[online] Available at:
`http://www.cs.utah.edu/~lifeifei/cis5930/kdtree.pdf`
[Last visited 17/05-2022]

[2] Thinh, Nguyen, Oregon State University, *Lecture 13+: Nearest Neighbor Search*
[online] Available at:
`http://andrewd.ces.clemson.edu/courses/cpsc805/references/nearest_search.pdf`
[Last visited 17/05-2022]

[3] *ControlsFX is a library of UI controls and useful API*
[online] Available at:
`https://controlsfx.github.io/`
[Last visited 19/05-2022]

[4] Robert Sedgewick and Kevin Wayne, 2018 *Algorithms, 4th Edition*
[online] Available at:
`https://algs4.cs.princeton.edu/44sp/`
[Last visited 19/05-2022]

[5] geeksforgeeks, 2022 *Trie, Insert and Search*
[online] Available at:
`https://www.geeksforgeeks.org/trie-insert-and-search/`
[Last visited 19/05-2022]

10 Appendices

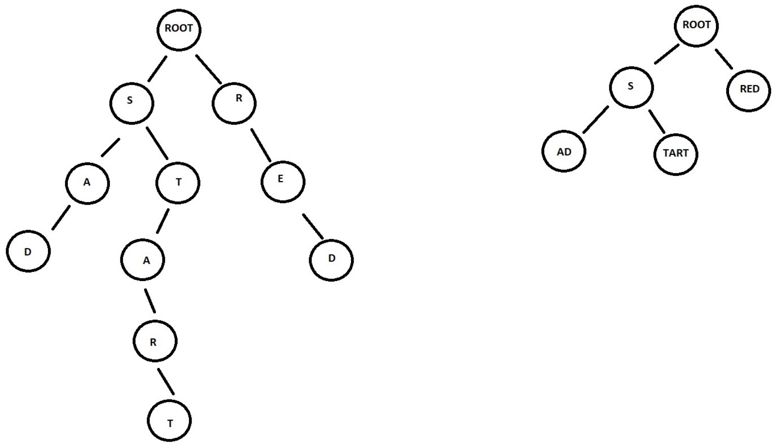


Figure 8: Example of a TrieTree and a compressed TrieTree

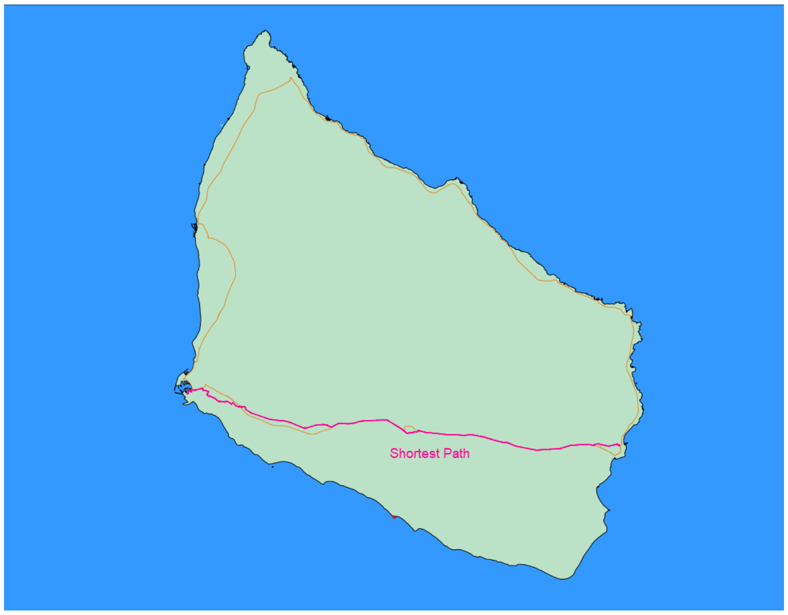


Figure 9: Bug for shortest path

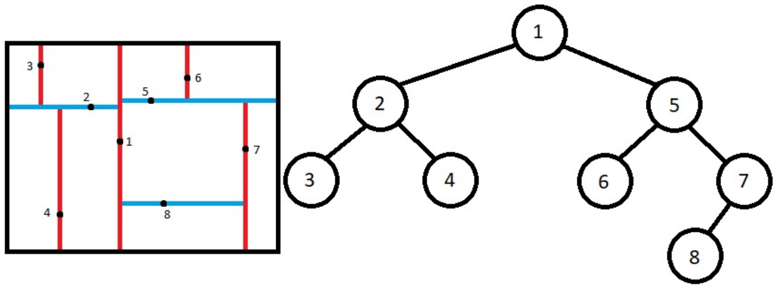


Figure 10: Example of a 2 dimensional kd-tree

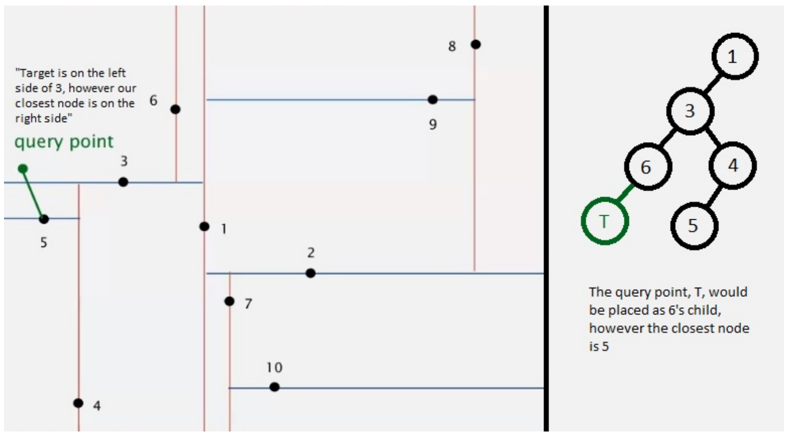
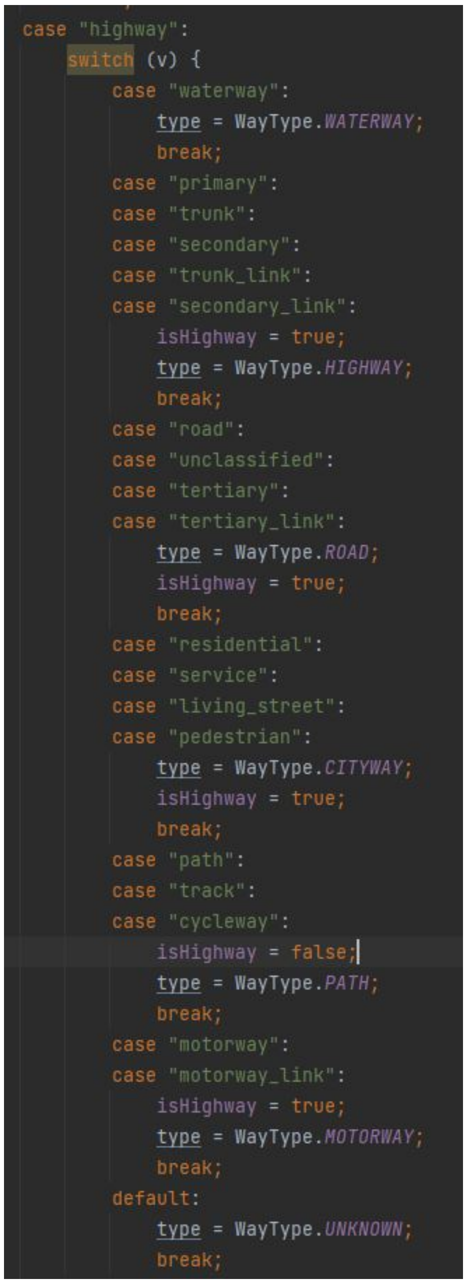

"Target is on the left side of 3, however our closest node is on the right side"
query point
The query point, T, would be placed as 6's child, however the closest node is 5
T
1
2
3
4
5
6
7
8
9
10

Figure 11: Example of nearest-neighbor search, and showing the importance of checking the other child of a node



```
case "highway":
    switch (v) {
        case "waterway":
            type = WayType.WATERWAY;
            break;
        case "primary":
        case "trunk":
        case "secondary":
        case "trunk_link":
        case "secondary_link":
            isHighway = true;
            type = WayType.HIGHWAY;
            break;
        case "road":
        case "unclassified":
        case "tertiary":
        case "tertiary_link":
            type = WayType.ROAD;
            isHighway = true;
            break;
        case "residential":
        case "service":
        case "living_street":
        case "pedestrian":
            type = WayType.CITYWAY;
            isHighway = true;
            break;
        case "path":
        case "track":
        case "cycleway":
            isHighway = false;
            type = WayType.PATH;
            break;
        case "motorway":
        case "motorway_link":
            isHighway = true;
            type = WayType.MOTORWAY;
            break;
        default:
            type = WayType.UNKNOWN;
            break;
```

Figure 12: Parsing through highway tag

Group formation and talk about how to work on the project throughout the course. Group contract was made and social channels established for easy communication concerning the project.

Week 11 `2022-Mar-16`

This week we started the coding process and got a working XML parser to read OSM files from our earlier handins.

We also setup a map canvas with some paint methods which are not working at the current moment in time.

Changelog:

- Made a working XML parser, which can read OSM files and save nodes etc.
- Setup a canvas with paint methods which does not work atm.

Working on:

- Getting lines painted on canvas - coloring.

Week 12 `2022-Mar-23`

This week we worked a lot on the basis for the application. We added a lot more tags to the parser and some light color coding for some of the tags like motorways and highways. We also fixed our draw methods in map canvas and made it so the draw method is depending on the amount the user is zoomed in.

Changelog:

- Added draw behaviour depending on the zoomedIn variable
- Replaced if statements with a switch case for parsing tags.
- Added more tags to the parser.
- Added color coding for motorways.
- Added more specific types of tags.

Known issues:

- A lot of lines defined as unknown.
- FPS problems, because of rendering.
- Water fill is not precise.

Working on:

- Fix the known issues.
- Setup kd-tree.
- Basic UI.
- Debug functions.

Week 13 `2022-Mar-30`

This week we started working on the KD-tree as we feel the route drawing is dependent on this. Furthermore we added a dedicated address class to all the addresses as well as adding the needed tags in the parser to fill this class. We expanded the Enum class to fit more types with the most important new tag being coastlines. We also started working on some visual debug functions and the template for the graph, as well as fixing the last of the issues we have with our drawing functions.

Changelog:

- Added address class.
- Added address parser so each address will get its own class.
- Added a List of all address classes.
- Added coastline.
- Added more enums.

Working on:

- KDTree.
- Javafx debug mode.
- Edge weighted graph.
- Fix water coloring.

Week 14 `2022-Apr-06`

We finished working on the KDTree, along with the classes it uses. We also got our first debug function running, the FPS counter as well as some basic GUI. We are still working on the KDTree, the rest of the visual debug functions, the draw functions which are not working, and the graph. We also started working on a slider to the zoom function as we are experiencing some issues with zooming on the trackpad of a laptop.

Changelog:

- Made somewhat functional KDTree:
  - Added KDTree class.
  - Added OSMNodeParser class
  - Added OSMNodeSorter class
- Added first version of GUI with few components
- Added FPS counter to Debugger menu

Working on:

- Finish KDTree.
- Fill out the rest of Javafx debug mode.
- Edge weighted graph.
- Fix water coloring.
- Make slider function for zoom.

Week 16 `2022-Apr-20`

Most importantly our KDTree now fills correctly and is working. To go with this we started working on a query or boundary box to make use of the KDTree. We transitioned the work from the graph to now work on both the graph and the Dijkstra algorithm simultaneously.

Changelog:

- Fixed kdtree (working and fills correctly)
- Added an attempt on query (boundarybox) (not working)

Working on:

- Query.
- Dijkstra.

Week 18 `2022-May-04`

The query function for the KDTree is now working as well. We merged our KDTree branch with our UI branch to combine the UI changes and the tree. For example to make a debug function for it to display the boundary box on screen. We dismissed the idea of a slider for zoom levels and instead made a progressbar for it. We also started working on a TrieTree for the address storing, and a function to be able to load a custom OSM file and .OBJ files. We are still working on the graph and dijkstra. Along with the faulty draw functions we still are experiencing. We also need to convert the KDTree to take ways instead of nodes.

Changelog:

- Finished KD-tree query function
- Merged KD-Tree branch and Niller's branch to combine some UI-changes and the tree
- Added progressbar for zoom-value

Working on:

- Adress-Trie
- Load Custom OSM-files aswell as object files
- Edge weighted graph.
- Fix water coloring.
- Convert KD-tree from OSM-Nodes to OSM-Ways

Week 19 - Part One `2022-May-12`

Probably the biggest progress in a week so far. We implemented a working TrieTree for our addresses, it works great but soaks a lot of memory which we are having some issues with on bigger maps. We also finally merged our coastlines and fixed the last issues with the drawing functions to draw waters and forests correctly. The graph and dijkstra got finished as well, and can now create a shortest path. The KDTree is now finalized and we implemented a nearest neighbor search as well. UI wise we made popups with both success messages and error messages. We are working on the final smaller tweaks now, like a few changes to drawing, what to draw and some fixes to nearest neighbor search.

Changelog:

- Implemented Adress Trie
- Merged coastlines
- Fixed water coloring
- Implemented Edge weighted graph
- Implemented Dijkstra
- Implemented Nearest Neighbor to find name of closest street
- Finalized the KD-tree class
- Tweaked some things for drawing
- Cleaned up the project with accessors and mutators
- Added pop-ups as error handling for invalid user-inputs

Working on:

- Load Custom OSM-files aswell as object files
- Tweaks in drawing (zoom level required and some more tags for cleanliness)
- Finish the reset method to restore default camera settings
- More accurate Nearest Neighbor (look for every node in way instead of average coords)
- What to draw on the map is still hugely in development:
  - We need to figure out in what detail we want to draw the map in.

Week 19 - Part Two `2022-Mar-13`

Final release logbog. In the couple of days since the last logbog we made our final refinements to our application. We made a function for the user to input personal points of interests. Refined our dijkstra and connected it to work with the address search, as well as mouse clicks. In the UI we added a loading page, and a dark theme for the map. Lastly we fixed the last drawing issues we had, with harbor filling, and the zoom values. This is the final release.

Changelog:

- Points of Interests added
- Djikstra refined

- Routing based on address input
- Routing based on mouse clicks
- Added points for end- and start-points for route
- Splashpage when loading
- Added Dark Theme
- Load custom OSM
- Tweaked zoomvalues for drawing
- Info and hotkey pages added in "info"
- Fixed harbors filling out
- Zoom has been fixed
- Nearest streetname to mouse is shown constantly

Known issues:

- Closest street is defined off of avg x, y for the street, resulting in inaccuracy
- Some streets are not connected to the roadmap since it includes car-only roads
- Too much memory usage, resulting in a cap on how big a loaded OSM-file can be.
- Trie does not accept non Danish-characters, resulting in failure when opening maps for other countries
- When creating a route from 2 addresses, the end point is pink.