

Minion Map



Group 11.

Johannes Jørgensen	jgjo@itu.dk
Elias Lindholdt	lild@itu.dk
Andreas Løvsgren Nielsen	anln@itu.dk
Marius Cornelius Wisniewski Larsen	coml@itu.dk
Kevin Skovgaard Gravesen	kegr@itu.dk

First-Year Project, Bachelor in Software Development, IT Univ. of Copenhagen

BSFDVNS1KU

May 14, 2024

Table of Contents

1	Preface	3
2	Introduction	3
2.1	Data set	3
2.2	Requirements	4
3	Technical Design & Analysis for Interface	5
3.1	Interface	5
3.2	Zoombar	6
4	Algorithms & Data Structures	7
4.1	KDTree	7
4.2	Trie	10
4.3	Minimum Priority Queue	14
4.4	Pathfinding	18
5	Technical Description of the application	23
5.1	Application Structure	23
5.2	Data Flow	23
5.3	Data Processing	27
5.4	Visuals / Frontend	27
6	Systematic tests & Benchmark	29
6.1	Unit Testing	29
6.2	Benchmarks for general operations	30
6.3	Benchmarks of pathfinding	31
7	Dissusion	31
7.1	Unachieved implementation of chunks	31
7.2	Trie and searching	35
7.3	Memory optimization	35
8	User Manual	37
9	Conclusion	39
10	Process Reflection	40
11	Appendix	41
	References	53

1 Preface

During a 10 week period, the Geographic Information System (GIS) called “Minion Map” was developed as part of the ‘First Year Project: Map Of Denmark‘ course at the IT-University of Copenhagen. Minion Map was developed using Java and the graphics Library JavaFX. This project report aims to communicate the thought process and design choices throughout the development of Minion Map.

2 Introduction

For millennia, humans have understood and recognised the importance and utility of maps and cartography. The history of mapping can be dated back more than 5.000 years, where humans would map places of interest on clay tablets or delicate maps on silk. In the modern era satellite systems, surveying techniques and contemporary cartographers are used to map the earth with very high precision and consistency. [1] This has as a result made the cartographic data of the earth widely available to the general public, both from web-map services such as Google Maps and Apple Maps or from open source databases as OpenStreetMap.com.

While there already exists an abundance of digital maps, they generally adhere to many common features and design standards. Developing a straightforward and user-friendly digital map demands a substantial amount of coding and attentive use of algorithms and data structures.

2.1 Data set

Minion Map uses data from the online cartographic database OpenStreetMap.com. The data is uploaded and maintained by volunteers worldwide, and is freely available to the public.[2] The data comes in the form of .OSM files that are written in XML format, which contain all the necessary data needed to compile and run Minion Maps.

The .OSM data consists of three data elements, which is nodes, ways and relations. Each of the three elements consist of a unique id, but the data elements have different structure for their data.[3]

- Nodes(TagNode): A node is a pair of latitude and longitude.
- Ways(TagWay): A way comprises a list of at least two reference nodes, describing a linear feature. Closed ways exhibit identical first and last reference nodes.
- Relations(TagRelation): Relations are a group of zero or more data elements with an associated role.

All of the three data elements can contain a sub-data element named *Tags*, which can describe additional information about the data element.

2.2 Requirements

Through the project description, from our course BFST F2024, we were given a set of specific requirements that our final product have to include [A.1]. The formal requirements were then processed, interpreted and split up into the following system- and user-based requirements.

2.2.1 System-Based Requirements

1. Compatibility for the “.zip” format, included default file for the program to use out of the box.
2. Drawing roads different from what type they are.
3. Graphical illustration of the map data contained within the bounds of a given rectangle chosen by the user.
4. Showing the current zoom level.
5. Multiple color themes for the user to pick from.
6. Dynamic layout listening to window changes.
7. Allowing address search.
8. Pathfinding.
9. Allowing the user to have multiple choices of transportation and allowing the user to place point of interests.

2.2.2 User-Based Requirements

1. Ensuring a clean GUI which is reasonably easy to navigate in and be fast enough for convenient use.
2. All of the system requirements will be developed having the user-based requirements in mind and by that not interfering with their fulfillment.
3. By having the User-based requirements in mind, we gave ourselves some informal requirements for the final product. These include autocompletion of addresses in the searchmenu, the color scheme of the map shall be simple and clear, and all of the fulfilled requirements should be supported by a simple GUI.
4. The formal and informal requirements prompt a multitude of different problems with the modern hardware and the chosen language, which we will try to solve with our final product.

3 Technical Design & Analysis for Interface

In this section. There will be analyzed some of the main features of the interface, and how they hold up in terms of time complexity.

3.1 Interface

A good way to improve performance when drawing the map, is to reduce the amount of things you draw every frame. When parsing the .OSM file, the different types of structures are split up into different types in the Type enum. Here they are assigned a hierarchy which is directly specified in the Type file. In earlier iterations of the program, some roads were assigned to a quite low hierarchy, which meant that they would only be drawn while the map was zoomed in a lot.

In the end it was decided to move quite a lot of types a layer down in the hierarchy, as it vastly improved performance, since the program did not have to draw such a large amount of items. When zoomed further in, the KDTree allowed for better performance since most of the items outside the view simply were not drawn. There was much consideration when deciding which hierarchies the different types were assigned to.

The point when different hierarchies are drawn are hardcoded. They were changed throughout the different iterations of the program.

3.2 Zoombar

One of the formal requirements stated that the program should include a graphical representation of the current zoom level. From that the following list of possible solutions came up:

- Displaying the current percentage of the zoom level
- Displaying the current range of a static ruler

The solution related to showing the current percentage of the zoom-level was ignored because of the disassociation with the value. Therefore, the first revision of our “zoom bar” was made by showing the width according to the map of a static picture of a ruler relative to the map. Which was made possible by using the haversine formula for calculating the distance.

After interacting with the program ourselves we decided to make the ruler that represents the distance to be dynamic, reasoned by the metric representation not being easily readable due to long obscure numbers and we liked the similar feature implemented in Google Maps. The dynamic zoom bar was set to have a class of its own, due to the benefits of securing low coupling and high cohesion.

It is instantiated through its constructor requiring parameters such as: number of intervals, maximum zoom-level and minimum zoom-level. Due to the calculation of the dynamic zoom bar being compute intensive, we decided that the methods should be called during the loading of the map. These methods being `setZoombarScales()` and `setZoombarMetersInterval()`, which calculates the intervals at which zoom levels would indicate even numbers of me-

ters. As illustrated in Figure 1, both X and Y calculations are made by using exponential functions made from applying exponential regression to tables having zoom Level per mouse scroll. From that the amounts of mouse scrolls can be evenly distributed throughout the zoom intervals. Although this makes an imperfect representation of the distance, we deemed it to be accurate enough

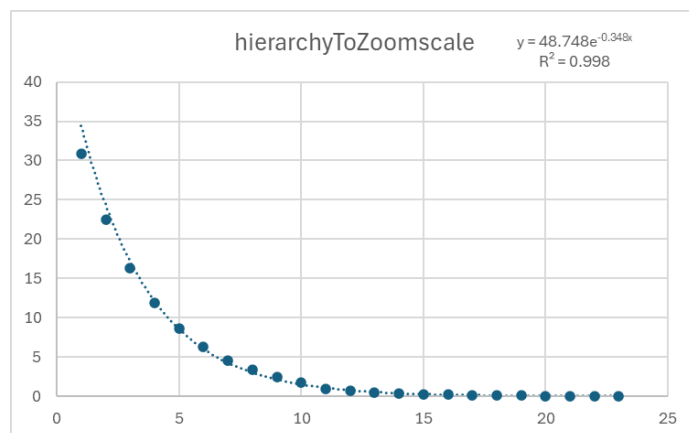


Figure 1: Regression analysis of zoombar hierarchy levels

for our purposes. Next we have the hierarchy level to meters that determine how many meter options the zoom bar should have on runtime. Both allocated in the ram and secured that the calculation for the current amount of meters to be displayed on the zoom bar is cost efficient in runtime. It is calculated by calling `setRange` which does two things. It calls `setZoombarHierarchyLevel` that costs linear time to calculate what hierarchy level the current zoom amount corresponds to with a for loop $O(n)$, a comparison and an access of an array $O(1)$. Then `setRange` executes a return of an array access costing furthermore $O(1)$.

```
1 double hierarchyLevelToMeters(int lvl)
2     return 4.5925*Math.pow(Math.E, 0.7689*lvl);
```

Listing 1: Code of `setZoombarHierarchyLevel()`

	Running time
Best case	$\Omega(3)$
Worst case	$O(2 + n)$

Table 1: Zoombar's running time

4 Algorithms & Data Structures

In this section, there will be analyzed several algorithms in the program, by going in depth with how they were implemented, and how they hold up to algorithms with similar functionality, by comparing their time complexity and memory usage.

4.1 KDTree

One of the most important parts of making a performant cartography application is to keep the loading and drawing time of the map as low as possible. To achieve this we can eliminate the need to draw unnecessary parts of the map. Such as elements outside of the user's viewpoint or elements that would be too small to be worth drawing. This is where k-dimensional trees become a great utility. K-dimensional trees, or KDTrees, is a data-structure that organizes the dataset such that getting a specified point in space or a multitude of points does not require the computer to read through every datapoint in the dataset.[4] We wish for a data-structure that reads through the smallest amount of data for a correct result. In other words we need the data structures' time complexity

to be smaller than $O(n)$, where n is the number of all points of the map.

There are two ways of illustrating a KDTree. One way of illustrating, is to demonstrate the structure of the code, where one can see that the structure is based on a binary tree. This is because of the nature of a KDTree, where every layer of the binary tree switches from the x-axis being the key to the y-axis. This means if a node is being inserted, it compares its values to the tree's value based on the current layer. Therefore in the first layer it compares only the x-values and travels to the right if it is higher, and left if it is lower. An example of input is the yellow circle in figure 2.

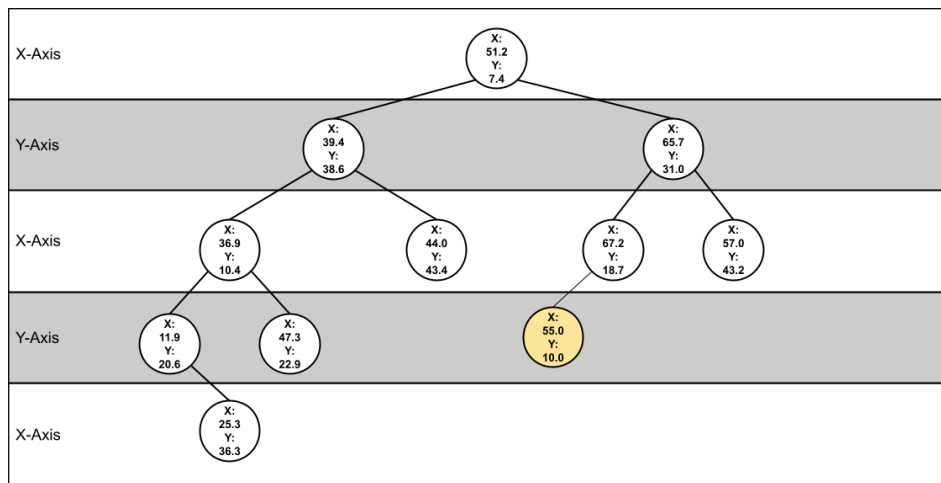


Figure 2: Demonstration of a KDTree with the structure of a binary tree

Another way of demonstrating a KDTree is through the inserted node's physical position in a 2 dimensional space. Where every layer is a "cut" in space, such that every child node is either one side or the other. The cut is made alternately in the x-axis and in the y-axis. As seen in the "root" node (marked in the white layer in figure 2), it has a cut in the x-axis.

The need for a spatial data structure is to cull the unneeded parts of the dataset to only use the data points within the viewpoint of the user. Thus the KDTree is chosen, since it implements a `range()` function that only returns data points within a specified rectangle on the map, according to the viewpoint. The use of a KDTree also implements a `nearest()` function that finds the nearest point from an arbitrary point. This can be useful for the user to be able to place points on the map and get information about the nearest point, such as addresses and its coordinates. However we did not get the feature to work through the KDTree, which means that we have made a separate component to find the nearest point.

We instead made the choice to look through every point in the dataset to find the nearest point.

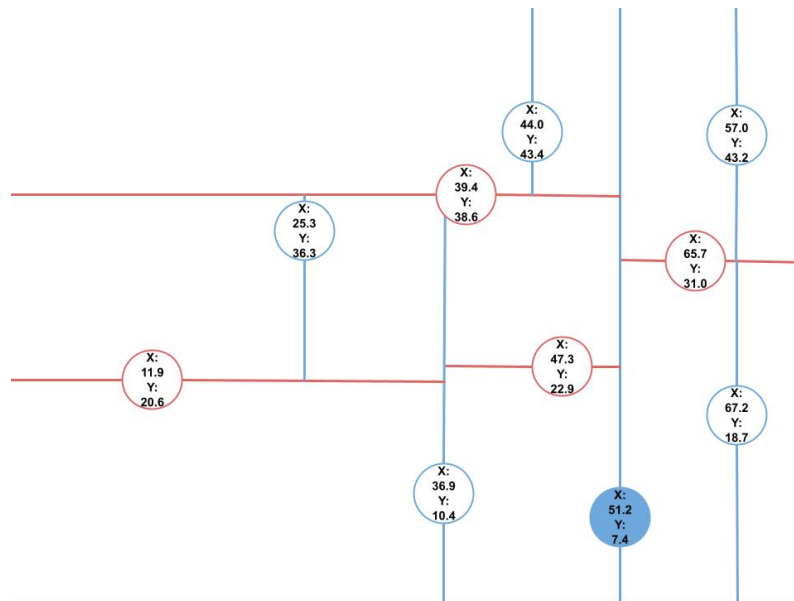


Figure 3: Demonstrates how a KDTree partitions segments in space

This was done because of time savings, as we had struggled to get the nearest function to work through the KDTree, and we did not have any disposable time left. The choice did not decrease the performance for every frame rendered, because we only used the nearest function to find the nearest road of a point. For example when going from an address, which is a building in the .OSM files, to a road that can be used in pathfinding between two points, which is acceptable as it does not happen every frame.

When calling the `range()` function on a 2 dimensional KDTree we get everything within the specified range, but as we implemented hierarchies that only show a specific level of detail (LOD), it would be more efficient to incorporate this hierarchy in the KDTree. This is where a 3 dimensional KDTree would be able to integrate LODs by placing the hierarchy in the third dimension. Which makes the `range()` function only return data within the specified rectangle and within the LOD group, specified by the zoom level.

The time complexity is the most important part of KDTrees. We are mostly interested in the time complexity of the `range()` function as it is called every updated frame. If we analyze the complexity of the function we get an average of $O(\log(n))$ and in the worst case $O(n)$, which is greater than manually checking if a point is within a rectangle for all points on the map, which would be an average time complexity of $O(n)$. This is because of the inherited behavior of a KDTree, where the data is sorted according to their position. Other than the `range()` function, we also use the function `insert()` when inserting a point into the KDTree. This function has a time complexity of $O(\log(n))$

in the best case and in the worst case $O(n)$, but as we only insert elements at the start of the map this does not impact our frame time.

Function	Average Time Complexity	Worst Time Complexity
<code>range()</code>	$\Theta(\log(n))$	$O(n)$
<code>insert()</code>	$\Theta(\log(n))$	$O(n)$
Search through every datapoint	$\Theta(n)$	$O(n)$

Table 2: Table over time complexities for KDTree

4.2 Trie

Trie is a data structure for storing strings, which provides a convenient way to search for the different stored strings. It is set up in a ‘tree’ structure, with each node of the tree being a single character. When a string is inserted into the trie, every character is, one by one, put into a node, and as the following set into a child of said node. If a node of said character already exists as a child of the current node, no nodes will be created, and the iteration continues. [5]

As an example, you could add the strings “stringone”, “stringtwo” and “string” to an empty trie, in that order. First you would start with the root node, which is empty. You then check the first letter of the string, which is “s” in this case. You search in the “branches” hashmap of the empty root node, and see if it already contains the character. If it does not, it creates a new node, and adds it to the hashmap. It then changes the current node to the new node, and repeats the process. When the last character is reached, the node is marked as an end node. On to the second string, “stringtwo”. It would again start off in the root node, and check the branches for any characters matching the first character. In this case you would find it. Instead of creating a new node, it simply sets the current node to be referencing the node containing “s”. It then searches that branch to find any matching characters for the second character, and so on. This repeats until the second “t” (stringtwo) is reached, which is not in the branches of the “g” node. A new node containing the character is then created, followed by two new nodes, the last of which is designated as an end node. As for the string “string”, it will start the same way as the other two, but when it gets to “g”, it will then just mark that as an end node. An end node can still have branches, as it just means that it qualifies as a valid search input.

Searching in a trie works in almost the same way as inserting. When trying to find a specific string,

the first character is searched for in the branches of the root node. If found, this process repeats until an end node is reached, which is a possible answer to the search. In this particular program we are mostly interested in getting a list of search results that fit into what is currently searched for. If no nodes in the branches match the specified character, it would indicate that the string that is searched for is not in the trie.

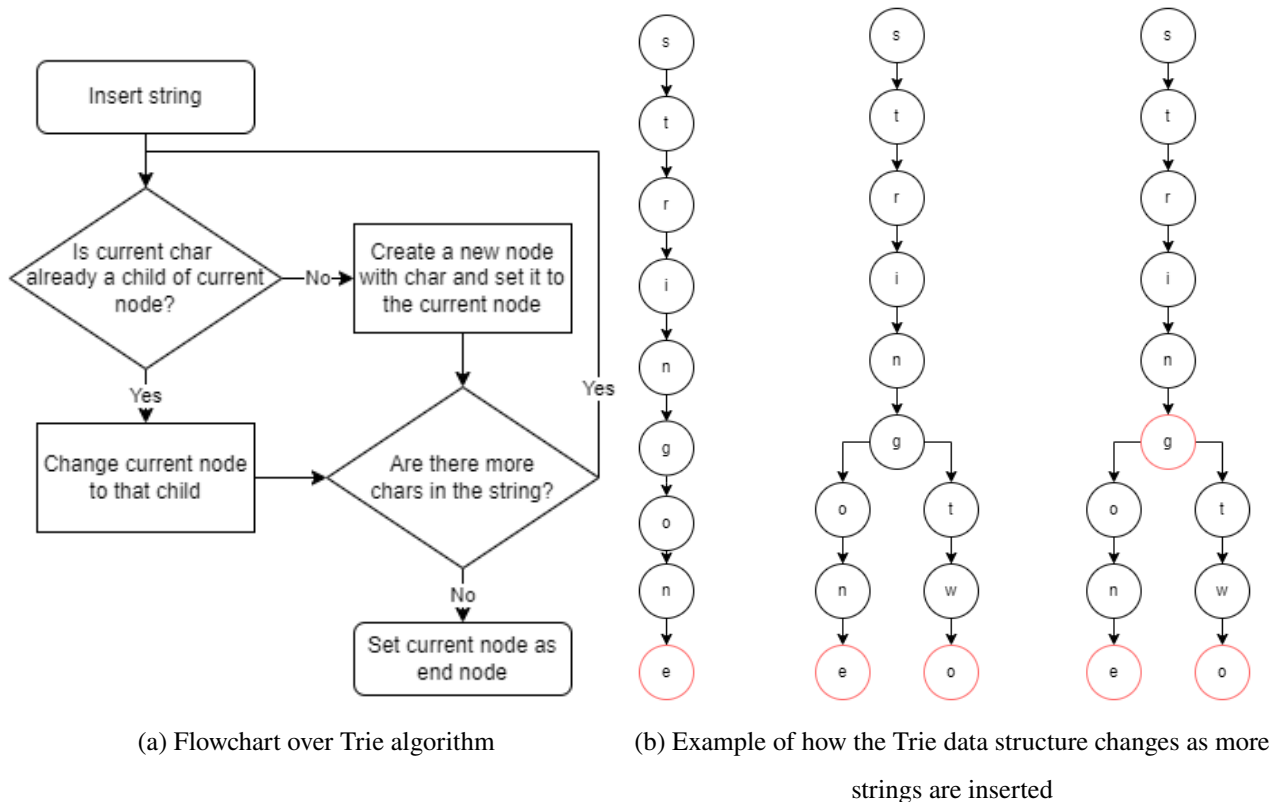


Figure 4: Illustrations of Trie algorithms

4.2.1 Why would we use Trie?

It was a requirement for the program to be able to handle searches for an address as a single string, and for it to be able to “handle ambiguous user input appropriately, e.g., by displaying a list of possible matches.”^{4b} A way to do this would be to simply store all the addresses in a hashmap, and use regular expressions to separate the different parts of the address. An algorithm, Levenshtein distance, could be used to approximate what strings the user could have meant instead of their exact input. This is mostly useful in cases where the user misspelled the input, in which case the program would ideally suggest the actual address they were searching for.

An alternative for this solution is the trie. A trie allows for the program to make suggestions on

what address the user might be searching for. When the user then tries to find a specific address, the program only has to search through the hashmap of the house numbers connected to the specific road, instead of a hashmap of every single address.

4.2.2 Trie Implementation

When a string is added to the trie, all of the spaces are removed. This means that the strings “Hvidovrevej” and “Hvidovre vej” would be treated the same. We decided that we wanted to store the addresses in the format of house number, postcode and city, for instance “Hvidovrevej 2650 Hvidovre”. The spaces are then removed, and everything is made lowercase. What is inserted into the trie will thus be “hvidovrevej2650hvidovre”. All end nodes have a hashmap which contains the different house numbers connected to the road, city and postcode. In the case just mentioned, the end node would be the final “e” in “hvidovre” (the hvidovre after the postcode). That node would

then contain a hashmap which contains the TagAddress objects corresponding to the house numbers present on Hvidovrevej 2650 Hvidovre. This makes searching for an address a lot faster, than trying to go through a whole hashmap of addresses. The different possible addresses are added to a list, which is displayed in the search bar. Only the street, postcodes and city names are shown initially, but after a specific street has been chosen by the user, the different house numbers are then displayed, again allowing the user to pick the specific address they desire.

A remove method was not implemented in the Trie class as it was deemed unnecessary in the circumstances of the program.

Implementing Trie into the system, there is a search bar which allows the user to type in whatever they desire. Whenever the user types in a new character, the program will search through the Trie tree, which contains all possible addresses for the given map.

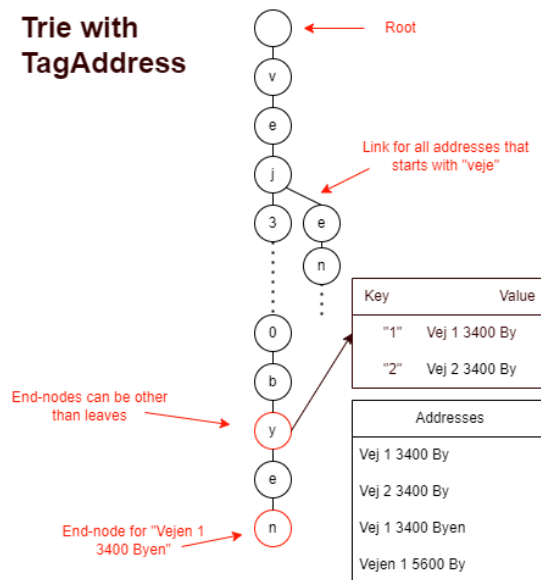


Figure 5: Example of Trie data structure with fictional addresses

If the user would type “hvidovre” into the search bar, Trie would look up at most 5 different suggestions for addresses that start with the given input. The user will now see that 2 suggestions pop up. These are TrieNodes set as an end-node, which each contain a hashmap, linking house numbers to real addresses. If the user were to click on “Hvidovrevej, 2650 Hvidovre”, the user would get all the endnode’s values. Now the user can click on whichever address they desire, and searching will hereby be complete.

If they were to continue writing an address instead of clicking on “Hvidovrevej, 2650 Hvidovre”. The program will detect that only one TrieNode can be found in the Trie tree. Since this is the case, the program will then lookup house numbers as the previous example.

4.2.3 Memory and performance

Trie will be stored into the heap as soon as Bornholm has been read. The size of Trie does not change during runtime afterwards.

The following benchmarks are performed on a pc with intel-i7 6700 processor and 16 GB 3200 MHz ram. The results on memory and performance suggest that Trie works for the program.

Object	Count of instances	Size
Heap used for Bolmholm	~ 7.5 million	344.118 kB
TrieNode	~ 27.000	868 kB
TCharObjectHashMap	~ 27.000	1.735 kB
Key, value pair in Hashmaps	~ 60.000	1.920 kB
HashMap	~ 27.000	1.296 kB
Sum of Trie algorithm	~ 141.00	5.819 kB (1, 69%)

Table 3: Memory usage of Trie, compared to the rest of the program, which is the entire heap of Bornholm

Operations	Avarage running time
Running time for finding five address suggestions	$\leq 1\text{ms}$
Inserting all Addresses for Bornholm into Trie	~ 100ms

Table 4: General operations for Trie

Optimizing the data structure would only be able to improve memory with small per-milles. Any larger mathematical sequences such as getting suggestions or inserting all addresses into Trie, is not slower than instantaneous.

It can thereby be concluded that Trie's functionality does not get in the way of the larger systems performance and memory usage. If the program were to handle larger maps, Trie would still be rather effective, due to its time complexity.

Operations	Average running time
Running time for finding an address suggestion	$\Theta(\log n)$
Inserting all addresses	$\Theta(n)$

Table 5: Time complexities in Trie, where n is the number of addresses

4.2.4 Improvements

The implementation of Trie did not allow for addresses to be found without entering the whole string. This prevents it from being used in the case that you want to find an address based on something else, like position. This resulted in the need for saving the addresses in some other place, if the program were to support such cases.

It also does not consider spelling mistakes made by the user. To fix this, a merge between the Levenshtein algorithm and Trie would be beneficial. Simply whenever the user types in a street, city or postcode in and it does not give any results. The Levenshtein algorithm will be used to find any possible string that is closest to the user's input.

4.3 Minimum Priority Queue

The Minimum Priority Queue (PQ) has two is being used in Minion Maps pathfinding and in layer sorting. The priority queue is a heap-ordered complete binary tree, where the root will be the minimum value, and the following will continuously be the next smallest value.

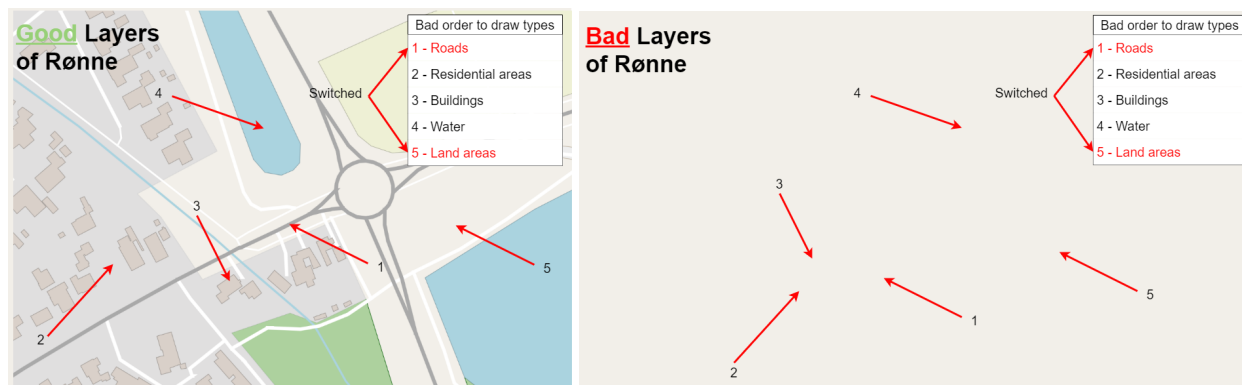
The Priority queues in Minion Maps, have been inspired from SedgeWick and Wayne's book on Algorithms, and implements many of the features from the book. The priority queue sorts with the `swin()` and `sink()` operations, meaning that values within the tree exchange index position with one another, up and down they go, until their parent has a smaller value and their children a larger value (vice versa for MaxPQ).

4.3.1 Priority Queue in Pathfinding

In Minion Maps, The pathfinding algorithm Dijkstra is implemented with the use of a modified Minimum Priority queue, called an Index Minimum Priority Queue (IndexMinPQ). The modified priority queue supports the same methods as the normal priority queue, but also has the methods `decreaseKey()` and `increaseKey()`. These operations support updating the priority of a key at a specified index. To implement this in Minion Map, we maintained a hash-map where each key represents a key-value pair's key and points to the index of the key. We then update the value at the specified index and reposition the current element based on its priority, and then update the index value in the hash-map accordingly. This ensures that the two methods have a time complexity of $O(\log n)$. [6] This is useful in the case of dynamically changing the priority of an element, and that comes very much into play when we update the best candidate vertex in the priority queue.

The priority-value of a given element can be changed, depending on what the search-term the user selects. The search-term changes if the priority queue should be prioritized by the distance (shortest path) or time (fastest path).

4.3.2 Priority Queue in Layers



(a) Example of layering Rønne in different type, where a type with the smallest value is behind everything else

(b) If the island of Bornholm were to be drawn in front of everything else (building, water etc.)

Figure 6: Illustrations of layering of Rønne

When drawing maps, it can be useful to define in what order, different types of polygons and lines should be drawn first. Otherwise, when reading an .OSM file, there would be a risk of drawing large polygons such as the island of Bornholm above everything else on the island. Instead, it is necessary to ensure that the island of Bornholm gets drawn *first* where the following polygons will be drawn afterwards, thus, polygons such as buildings will be *on top* of Bornholm.

The solution to this problem would be to **assert a value to a type of polygon or line**, which a priority queue can use to compare its data. When obtaining a whole list of objects that should be drawn on the screen. All instances will be redirected over to the minimum priority queue with the insert() method.

Here, an object will be put as the last index of the queue, and will begin to “swim” upwards appropriately using its **layer value**, until its parent’s value is smaller than itself.

Once every object to be drawn has been inserted, the minimum object will be deleted from the queue and will be drawn. This happens by exchanging the root (the minimum value) with the last index. Once this has been done, there is a good chance that the new root is misplaced. Here the sink() method will be used. Which exchanges the given index with one of its children, preferably the child with the smallest value, until none of its children is smaller than the value to sink.

MinPQ for layering

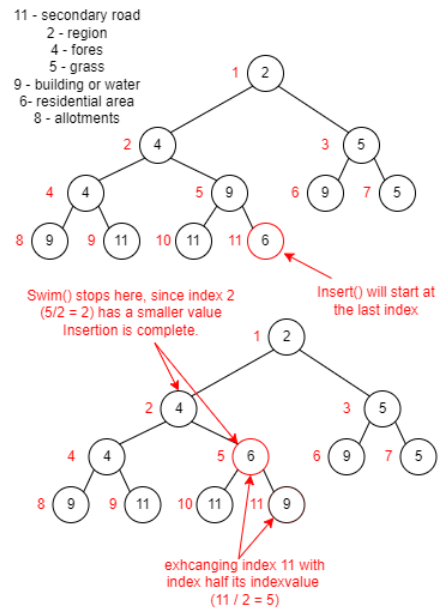


Figure 7: Inserting a value into the Minimum priority queue

Getting the object to draw

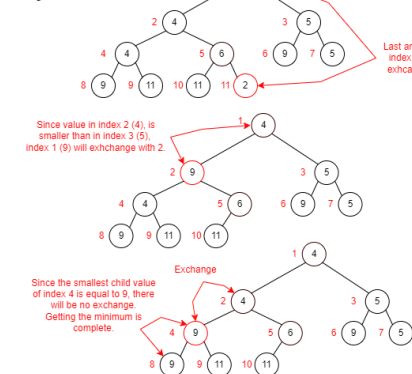


Figure 8: Deleting the minimum value of the MinPQ

4.3.3 Memory and performance

An instance of a MinPQ has

- An array of the given type, the size in memory the chosen type's size time with the length of the queue.
- An integer, which is the amount of objects stored on the pq.

The size of the array on this project's instance is precisely based on the amount of lines and polygons to draw. And an integer is only 4 bytes.

Object	Count of instances	Size
Heap used for Bolmholm	~ 7.5 million	344.118 kB
TagWay	~ 79.000	3.792 kB
Sum of MinPQ for layering (worst case)	~ 79.000	~ 316 kB (0.09%)

Table 6: Table of Memory usage with MinPQ compared to the entire heap, while running Bornholm.

Although each TagWay takes up 48 bytes. MinPQ will only take up 4 bytes per TagWay, since TagWay is a reference type, the array will store references to an object, which takes up 4 bytes per reference. $4 \text{ Bytes} * 79,000 = 316 \text{ kB}$.

The following benchmarks are performed on a pc with intel-i7 6700 processor and 16 GB 3200 MHz ram. When running Bornholm, it is on average 5,000 lines or polygons, no matter where you

Operations	Average running time (~ 5.000 objects)
Inserting all ways into the MinPQ	$\leq 1 \text{ ms}$
Obtaining and deleting the entire MinPQ	$\sim 2 \text{ ms}$

Table 7: Operations for Minimum Priority Queue

zoom or pan, and looking at these operation times, makes it clear that using a minimum priority queue for layering, takes almost no resources on the computer, considering the other parts of the program.

4.3.4 Comparison with alternative sorting-algorithms

If we were to consider other sorting methods. No other alternative (from the book) seems to work more efficiently, as far as our research brought us. MinPQ, which is arguably similar to heapsort (since heapsort is built from PQ's `delMax()` method), will use $O(N \log N)$ time for sorting all TagWays, where N is the number of TagWays.[7] This process is almost identical to deleting the minimum value in MinPQ until there are no elements left. Sorting-algorithms similar to MinPQ's sorting running time would be mergesort, quicksort and 3-way quicksort. The problem with all these three algorithms is their extra use in memory. Heapsort is the only one of these four algorithms, of which extra space is a constant size (Object header, padding, size integer), while the others use at least extra space of $\lg N$. [7] This extra memory can be found in the stack whenever the algorithm sorts, since quicksort and 3-way quicksort are recursive algorithms, whereas heapsort is just iterative.

4.4 Pathfinding

Minion Maps implements a edge-weighted digraph for representing the road network, with support for the A* pathfinding algorithm.

In order to figure out what the best way to represent the road network of Minion Maps, and give the user the ability to find the shortest route. We will need to discuss the use of the directed edge-weighted graph or edge-weighted digraph data structure and pathfinding algorithms.

4.4.1 Edge-Weighted Digraph

To make a good representation of a real life road network, we need to take multiple considerations into account. How do we represent roads, intersections, one-way streets and more as a data type? In Minion Maps, we have chosen to use a edge-weighted digraph to address all of these considerations.

A directed graph (or digraph) is a set of vertices and a collection of directed edges that each connects an ordered pair of vertices. In the case Minion map the directed edges are weighted, to help distinguish what road is faster to go through. In the terminology in the case of Minion Maps, the vertices are a TagNodes and the weighted edges are a pair of TagNodes with distance as the weight. The digraph in Minion Map uses the adjacency-list representation, where we have a vertex-indexed array of lists which contain the vertices connected by weighted edges. The representation can be constructed and take up $\Theta(E + V)$ time and space, where E is the number of edges and V is

the number of vertices. The actual data structure in the adjacency-list is a list of links to weighted edges containing the references to other weighted edges.

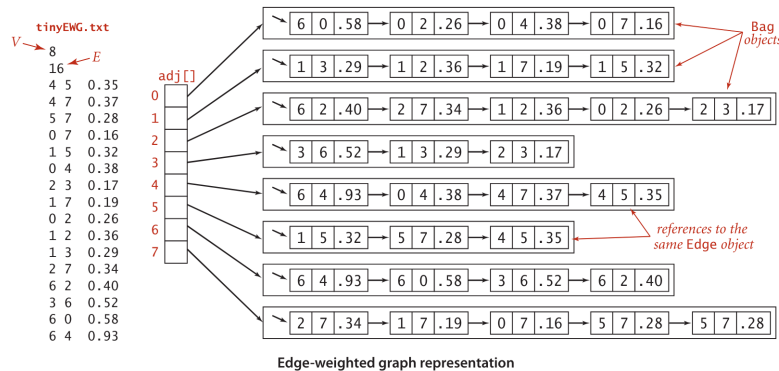


Figure 9: Edge-weighted graph of the adjacency-list representation[8]

When implementing the Edge-Weighted digraph in Minion Maps, we implemented the vertex-indexed array as a TreeMap instead of a normal list. This is to achieve a search time of $O(V \log E)$ time instead of getting a search time of $O(V + E)$ when using an array representation.[9] The reason why we get sequential search time on the array representation, is because the vertices are stored unordered and as objects and we thereby do not know their index in the array. We therefore use a TreeMap that implements a natural ordering of its keys and use a balanced Red-Black search tree to gain a search time of $O(V \log E)$ in the worst.

4.4.2 Dijkstra & A*

Dijkstra is a famous pathfinding algorithm that finds the shortest path between two nodes in a weighted graph. Minion Maps supports both Dijkstra and a variant of Dijkstra called A* (pronounced “A-star”), that uses heuristics to find the shortest path more efficiently. The main mechanism in Dijkstra’s algorithm is to recursively find the best candidate vertex for creating the shortest path from one vertex to another.[10]

Algorithm 1 Dijkstra's Algorithm[11]

```

procedure DIJKSTRA( $G, s$ )                                ▷ Digraph  $G$  and source vertex  $s$ 
    Initialize  $Q$  as an empty priority queue
    Initialize  $d[v]$  to  $+\infty$  for each vertex  $v \in V[G]$ 
    INSERT( $Q, s, 0$ )                                       ▷ Insert source into queue with distance 0
    while  $Q \neq \emptyset$  do
         $(u, k) \leftarrow \text{DELETE-MIN}(Q)$                  ▷ Extract vertex with minimum distance
         $d[u] \leftarrow k$                                    ▷ Update shortest distance to  $u$ 
        for all  $(u, v) \in E[G]$  do                       ▷ Iterate over neighbors of  $u$ 
            if  $d[u] + w(u, v) < d[v]$  then
                if  $d[v] = +\infty$  then
                    INSERT( $Q, v, d[u] + w(u, v)$ )
                else
                    DECREASE-KEY( $Q, v, d[u] + w(u, v)$ )
                end if
                 $d[v] \leftarrow d[u] + w(u, v)$              ▷ Update shortest distance to  $v$ 
            end if
        end for
    end while
end procedure

```

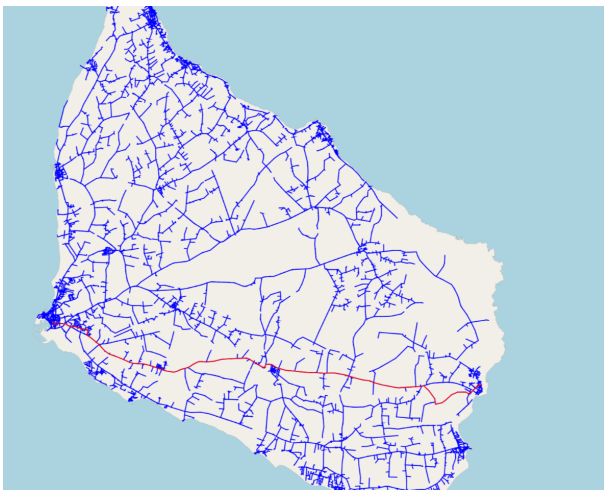
Dijkstra keeps track of the distance for all edges between the start s vertex and any given vertex v through the `distTo[]`, while simultaneously updating the priority queue with the element's priority-value, which is the distance between the two vertices divided by the speed limit of the road. Dijkstra uses relaxation of an edge $u \rightarrow v$ to test whether the best-known way from s to v is through $s \rightarrow v$, or by going through the edge $u \rightarrow v$. If going through $u \rightarrow v$ is the best way by comparing the edges weights, then update our data structures to indicate that this is the case [12]. As this is a recursive method, the process starts again with the newly assigned best edge until a route is found from the starting point to the end point.

A* takes a heuristic distance into account when prioritizing the vertices. A* utilizes heuristic values to assert how to prioritize the given vertices. A* takes into account the distance from any given

vertex to the end vertex ($hScore$) and the distance from any given vertex to the start vertex ($gScore$). A^* then prioritizes the vertex with the lowest $fScore$, which is the addition of the two heuristic distances $fScore = hScore + gScore$

The heuristic of A^* can be implemented in different ways and the quality of the heuristic and how to approximate the $hScore$, can have a big impact on the running time of the algorithm. If the heuristic in theory did not help cutting down the possible branches that the algorithm could search though, we could calculate the time complexity by estimating the amount of vertices the algorithm has explored. Here the time complexity of A^* would be $O(b^d)$, where b is the average amount of outdegrees a vertex has, and d is the depth of the shortest path. If the heuristic is well implemented, the heuristic would decrease the amount of outdegrees the algorithms would have to search though. The Psudecode for A^* would almost be identical as Dijkstra, except when inserting or decreasingKey of the priority queue, the heuristic value is being added on top of current weight.

Algorithm	Function	Average Time Complexity	Worst Time Complexity
Dijkstra	<code>shortestPath()</code>	$\Theta(E \cdot \log(V))$	$O(V^2)$
A^*	<code>shortestPath()</code>	$\Theta((\sum b)^d)[13]$	$O(b^d)$



(a) Illustration of Dijkstra's Algorithm



(b) Illustration of A^* Algorithm

Figure 10: Illustrations of the Dijkstra and A^* pathfinding algorithms in Minion Map, where the red line is the shortest path and the blue is visited paths

4.4.3 Douglas Peucker

An experiment that was attempted was to reduce running time for drawing the map by reducing the amount of points to draw lines to. This could be done with the Douglas Peucker algorithm which iteratively reduces the point between an endpoint and a startpoint. [14]

Say there is a line with n amount of points. Then the Douglas Peucker algorithm removes points based on the length of a tolerance distance. At the start of the algorithm. A line is drawn from the start-point to the end point. Then the point that is furthest away from that line is checked whether it is within the tolerated distance or not. If it is all points except the start and the end point gets removed. Else, this point will be added to the line, and the point that is closest to this line will then be checked. This iteration continues until a point is within the tolerance distance.

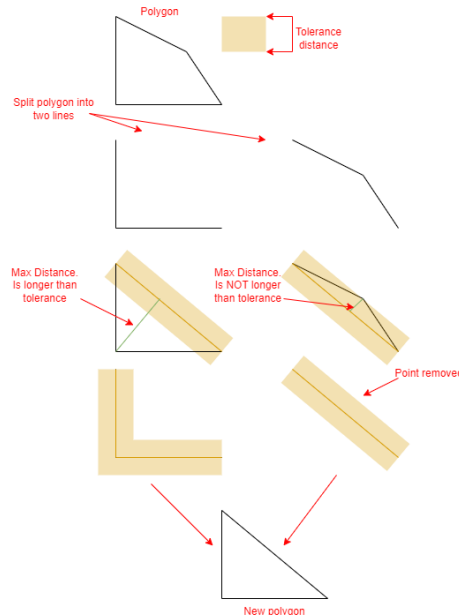


Figure 11: Process for Douglas Peucker over a polygon

Douglas Peucker would be implemented by simplifying all polygons and lines based on how far the user zooms out on the map. As the user zooms further out, the tolerance distance would become larger, and thus lines and polygons would become simpler and simpler. This was however not a very optimal idea, as the algorithm would have to run whenever the user would zoom out, and since the amortized time complexity is $O(n \log m)$ [14], where n is the number of points and m the number of points for the new, simplified line. The algorithm would simply have to handle too many polygons and lines at once, and too often as well. Additionally the algorithm conflicts with the newly structured LinkedList foundation, where each point is chained to the next and its previous, and breaking this chain may require additional calculations. It is possible that there could be a work around, where the linkedlist itself would not be changed, but only some of the points would be marked as drawable. If there was more time to work on the project, Douglas Peucker may be a possible implementation, but due to its many challenges, the algorithm simply was not fitting enough for our problem to be prioritized higher than other algorithms.

5 Technical Description of the application

This section will explore the technical aspects of the application. This is done via a data flow, and separating the system into an MVC-architecture.

5.1 Application Structure

- **Parser:** The parser has the job of parsing and structuring the .OSM data, into individual classes. This component goal is to pass the parsed and structured .OSM data onto the next component.
- **Data processing:** After the data has been parsed, the data needs to be processed. This component's goal is to further structure the data to be able to handle the data in an efficient way, so the newly structured .OSM data efficiently can be drawn to the user.
- **Visuals:** The visual component controls all the components that are displayed to the user and drawn from the different parsed objects. The goal of the view component is to display the right data to the user and manage the logical states of the JavaFX elements.

5.2 Data Flow

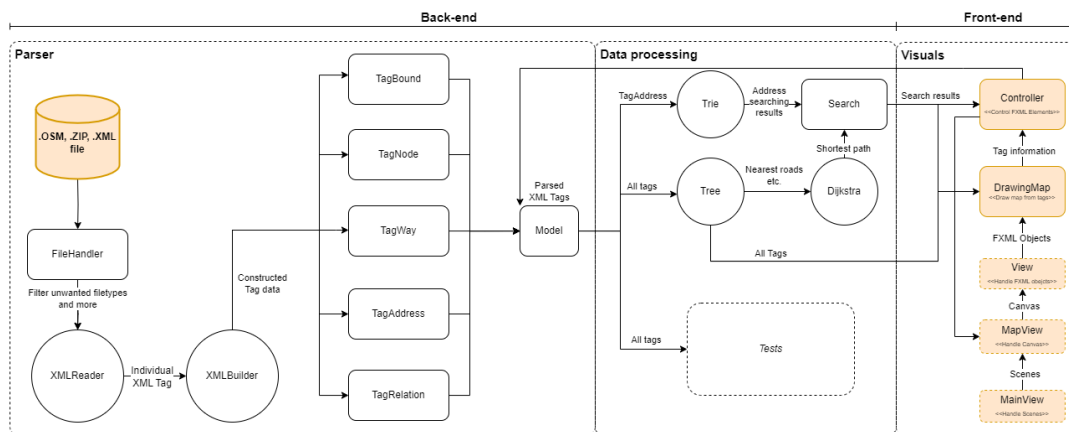


Figure 12: Flowchart of the flow of data in the application

5.2.1 Parser

The parser is the first of the three smaller structural components to handle the inputted .OSM data. Firstly the data passes through the FileHandler to check inputted files for unsupported file types,

unzips inputted .ZIP files and more. When accepted, the .OSM data is passed onto the *XMLReader* file. The *XMLReader* extracts individual XML elements to be read by the *XMLBuilder*, which reads the element and parses the data into objects. When the XML elements attributes and subtags have been read and parsed, all the data go into their respective Tag classes [see 2.1]. It would also be in this step where the constructed Tag class would be appended to the binary pool, to be written into chunks. See 7.1 for the implementation.

- Model class: intentional design to be a main exit point for all the parsed data, and an entry point to write the parsed data to binary.

Tags

Every .OSM XML element has specific characteristics that can be split into five *Tag*'s. All of the tag classes inherit from the abstract superclass *Tag*, which handles common attributes and methods for all tags. Each of the tags was implemented with the intention of being used in chunks. This has some major drawbacks when it comes to Minion Maps current implementation, without the functionality of chunks (see 7.1).

TagBound

The *TagBound* class represents the *bounds* element in the .OSM file. The bounds element is the first .OSM element in the file. The bounds define a rectangular area of *maximum*, *minimum* latitude and longitude. All of the XML elements that have their latitude and longitude are within the area that is within the .OSM file. There are only a few exceptions where there are references to XML elements that are without the bound area, this mostly occurs in the way and relation elements.

Minion map uses this class in both for the functionality to pan and zoom on the map, but was also supposed to be used as a header for chunk files to define the bound area.

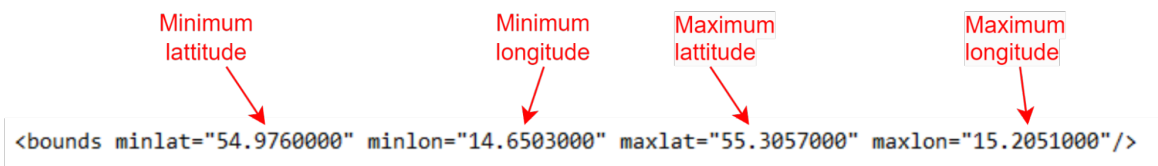


Figure 13: Illustration of TagBounds when parsing

TagNode

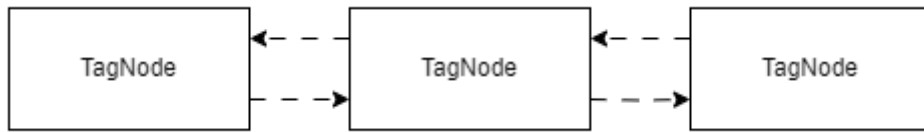


Figure 14: Illustration of TagNodes representation

TagNode is a class that represents a *node* element in the .OSM file and is a sign point on the map. The node has a unique id and a latitude and longitude as attributes.

In Minion Map we have made the *TagNode* compatible to be used as a double linked list, for the benefit of linking all of the nodes together if they are a part of a *TagWay*.

TagWay

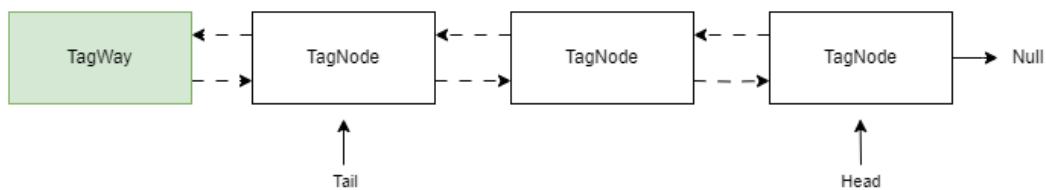


Figure 15: Illustration of TagWays representation

TagWay represents the *way* tag in the .OSM file and is used to display lines or polygons. A *TagWay* is constructed via a sequence of *TagNodes*. The *TagWay* itself does not contain latitude or longitude, however, its referred *TagNodes*, implicitly defines where the *TagWay* is drawn. The *TagWay* is stored in memory as a linked element for the tail *TagNode* that the way represents.

A *TagWay* also contains a *Type*, which defines whether the way should be a line or polygon, which layer and hierarchy it should be drawn on, and what color it should be drawn with.

TagAddress

TagAddress is similar to the native nodes, but the node has a series of subtags that contain the information of an address.

In Minion Map, each node with the right subtags is parsed as an address to be later used in Trie and the KDTree. The reason to parse this kind of node into its own class is to ensure easy use throughout the application and fast instance checking of the class.

TagRelation

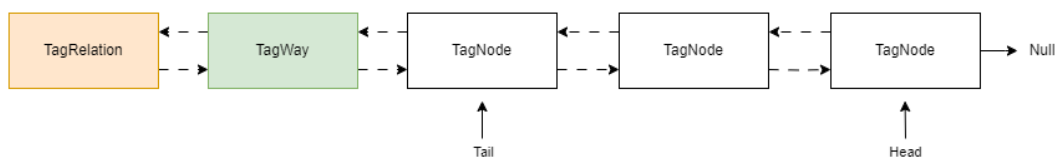


Figure 16: Illustration of TagRelations representation

TagRelations are similar to TagWays, however instead of containing a list of *TagNodes*, TagRelations contain a list of *TagWays*. TagRelations can be a set of larger polygons "multipolygons", or it can be a network in for example public transport.

The TagRelation is stored as a link to a TagWay, just as the TagWay is linked to the tail TagNode in the list of referenced nodes.

In Minion Map, TagRelations are only parsed if they are polygons. The list of TagWays are typically set up as either inner or outer ways. The entire collection of outer ways represents the larger polygon(s). The inner ways are polygons which are within the outer polygons. These inner ways are typically a completely normal TagWay as represented above. The outer TagWays however do not contain a type themselves. Instead, these should inherit the type of the relation.

MecatorProjection

One of our design choices was to project everything using Mercator projection. Due to some of the data being delivered in OSM as degrees which would mean that if anything distance related was required it would have to be converted by calculation in runtime. That led us to implementing this. With Mercator projections we already translate the necessary data in the loading phase. By calling the projection function in the parsing process and then storing the processed data inside of the heap instead.

5.3 Data Processing

5.3.1 All Tags

In order for the program to run smoothly. It is necessary that all of the tags can be obtained quickly, since we may end up working with millions of tags at once. Thankfully, since a computer screen only can be up to a certain size, only a selected number of tags need to be handled at once, the tags that appear on the screen. Since all tags implicitly or explicitly contain coordinates. The tags can be set up in a *Tree* (see 4.1), which sorts the objects based on their latitude and longitude.

This *Tree* can also be used by the pathfinding algorithm *Dijkstra*, to determine which roads or addresses are close to each other.

5.3.2 Tag Addresses

Defines the addresses for all the buildings on the map. This data needs to be processed, so that the user can search on these addresses. Additionally the program needs to be capable of auto completing the user's input into a valid address. That's why the Trie algorithm is implemented, which consists of "TrieNodes", constructed by all the TagAddresses individual characters (see 4.2).

5.4 Visuals / Frontend

Visuals is the part of the program that is responsible for everything that the user experiences while traversing the map. Here, both **view** and **controller** in our Model-View-Controller-architecture reside (MVC-architecture). The model is solely responsible for the entirety of backend processes. When working with the frontend, it is important for the view and controller to give and receive information to each other. Whenever the user *controls* by panning over the map, information will be sent to MapView about whether a button should be marked blue, or a new tab should be opened. Whenever the view gets changed, the controller will have to behave differently to the modified interface. The controller also sends information back to the model, where data once again can be processed through the different algorithms. For example, whenever the user *controls* by panning over the map, the model needs to know how much the screen has changed in its position. Then the model will use that information to obtain which new polygons and lines should be drawn and which should not. This information will inevitably have an impact on what the user *views* on the screen.

5.4.1 View

The View class is a super class with two child-classes Lobby- and Mapview bound up to it. Each holds crucial information for the two screens that reside inside of the program: the start screen and map screen. Each view-class has its own controller, lobbyview: Start Screen Controller and MapView:Controller.

The lobby view stores information such as what map the user wants to load, when pressing the button “Launch Map”, it sets the model object to a new instance of it from the file chosen by the user which is then created in the superclass View.

The superclass View is also responsible for changing Views, loading their controller, fxml sheets and then storing protected data from the child classes in itself. Ensuring the data is only retrievable from the View-class itself which is the active data.

5.4.2 DrawingMap

The DrawingMap class is mainly responsible for the cartography itself. This means that it obtains all relevant polygons or lines (TagWays) and displays them appropriately by getting their individual types and their points (TagNodes). The TagNode’s points are handled with the help of JavaFX’s transform. This ensures that the coordinates are handled differently with each frame, as the viewport gets panned and zoomed by the user.

6 Systematic tests & Benchmark

6.1 Unit Testing








To ensure that each individual component of the program works as intended, systematic unit testing was required. This was done by Whitebox [15] testing using our knowledge of the program to create tests with Junit tests, asserting that each specific action of the module gave the expected result. Thereby ensuring that every component inside of the class works as intended and thereby having the ability to rule it out during the debugging process of a component.

6.1.1 Jacoco






Jacoco was a tool used by us to determine the fulfillments of our tests made to the class. From Jacoco we extracted reports on what was tested, to what degree and what needed. Helping us determine if the testing had been done thoroughly enough.

For the overall test coverage, we received 45 %, which to an extent can be seen as less than adequate. But most of the uncovered programs are from the controller and the drawing map, which were difficult to test, because of how these classes were dependent on the main application class. Instead the resources were spent on testing various utility classes and the XMLReader with an osm file for testing.

The most tested classes are also amongst the most central. The TagWayBuilder is responsible for building all TagWay classes. Same for the TagAddressBuilder with TagAddresses. The XML-Reader reads the osm file and translates data to tags. The model stores all tags and is responsible for Trie and the KDTTree. And the Tag superclass to all the different tags is also almost completely covered.

Element	Missed Instructions	Cov.
gui		0%
pathfinding		16%
parser		65%
structures.KDTTree		48%
parser.chunking		49%
structures		61%
util		88%
default		0%
Total	8,135 of 14,979	45%

(a) Total Test coverage in Minion Map

Element	Missed Instructions	Cov.
TagWay.WayBuilder		97%
TagAddress.AddressBuilder		91%
XMLReader		95%
Model		94%
Tag		84%

(b) Most test covered classes in Minion Map

6.1.2 Improvements

Getting more coverage, by being able to test the main application would certainly be preferred. This could be done more easily if the various classes such as drawingmap had a default constructor that could be called without the need of starting main. Another thing would be to make more of the methods static so that the tests would not need to initialize a DrawingMap to a field, and instead use the one constructed in the mapView.

6.1.3 Un-synced Tests

Before incorporation synchronization for the tests, there were alot of issues getting them all to parse at once in one round which was required by jacoco to file the report. This was probably caused by improper Garbage collection after each and every one of them. The tests passes individually. Which is why you see SOME/ALL? of them being synchronized in the program.

6.2 Benchmarks for general operations

The following benchmarks are performed on a pc with intel-i7 6700 processor and 16 GB 3200 MHz ram with an SSD.

Operations	Time
Drawing (through panning or zooming)	2–40 ms
Loading zipped file	~ 4500 ms
Pathfinding A*	≤ 1124 ms

Table 8: Table for operations over Bornholm

Although the drawing method seems effective compared to other general operations, it has to be run for every frame when panning. Which means that for 40 ms, the program would run at 25 frames per second. Loading is only called once a map is chosen in the LobbyView. For a Zipped version of Bornholm, 4.5 seconds is up to our standards as it takes additional time to unzip the map. For a larger map it will take longer to load, but as the cost of loading a map only happens once on startup and not every frame loaded, the cost is not detrimental to our program.

6.3 Benchmarks of pathfinding

The following benchmarks are performed on a pc with AMD Ryzen 7 5800H processor and 16 GB 3200 MHz ram.

Route	Dijkstra	A-Star	Gain	Gain percentage
Bagergade 2, 3700 Rønne to Nexø Busstation, 3730 Nexø	832ms	101ms	-731s	~ 87.9%
All of the edges scanned through	10962 edges	2131 edges	-8.831 edges	~ 80.6%

Table 9: Table for operations over Bornholm

As we can see in the table above, the performance gains from using A-Star instead of only Dijkstra are quite substantial. It takes 87,9% less time to calculate the shortest route. The routes found from both techniques, in this example, are exactly the same and therefore there is no loss in accuracy. But it can vary from route to route.

7 Dissusion

7.1 Unachieved implementation of chunks

Memory efficiency and optimization has been a high priority since the start of Minion Map. We knew from the start of the project, that to be able to load Denmark would require efficient handling of heap space. Therefore it has always been a goal to filestream the inputted .OSM file into smaller *chunks*.

7.1.1 What is the problem?

The main objective with the chunks was to minimize the heap allocated from the KDTree, and only load in what part of the map the viewer sees. As mentioned in 4.1, the current implementation of Minion Map uses the KDTree to draw all the content the viewer sees. But a big problem with this is that the KDTree always stores all parsed .OSM data in the heap at all times, even if the user only sees a small part of the .OSM data. This can lead to memory overflow when trying to load bigger .OSM files, such as Denmark. We knew this drawback from the start of the project, and therefore started the development of chunks at the same time as KDTrees.



(a) Representation of current KDTree implementation.

(b) Representation of chunks

Figure 17: Illustrations of intended implementation of chunks

The red box in figure 17 is the viewbox of what the user can see. Outside the viewbox in figure 17a there are undrawn parts of the map that are still in the heap. The viewbox within the bounds of nine chunks in figure 17b is, where all the map parts are loaded into the heap and drawn. The grey boxes are unloaded map chunks, that are loaded in at runtime, if the viewbox is within the bounds

7.1.2 How did we implement it?

When we implemented file streaming the data into chunks, we had multiple considerations for how the data should be stored into the chunks. To achieve as little memory allocation as possible, we chose to only store the TagBound of the chunks area and all the Tag-Nodes and Address' that were within the bound.

But this solution leads to a problem when we want to draw ways and relations. How do we draw a way if only we have loaded a single TagNode in from the chunk?

The path in figure 18a is not drawable because the isolated TagNode has no reference to the other nodes in the TagWay, so how did we solve this?

As mentioned in section 5.2.1 about the parser, we have made the TagNodes compatible to be in a double linked list, to solve this exact problem. This would allow a single isolated TagNode to draw its corresponding TagWay, just by going back to its previous node until the tail node has been

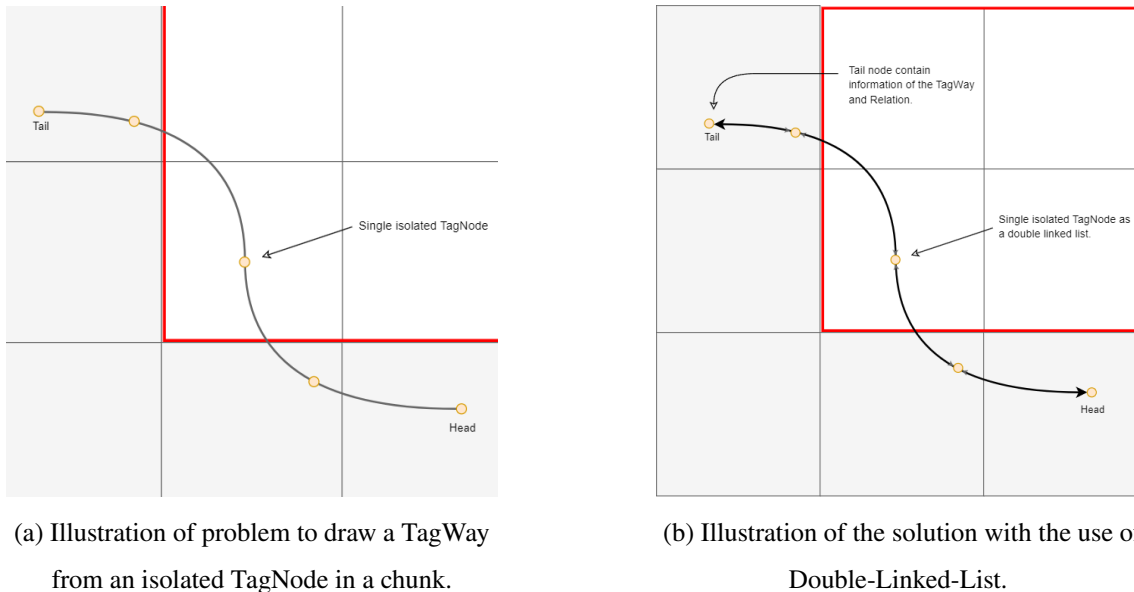


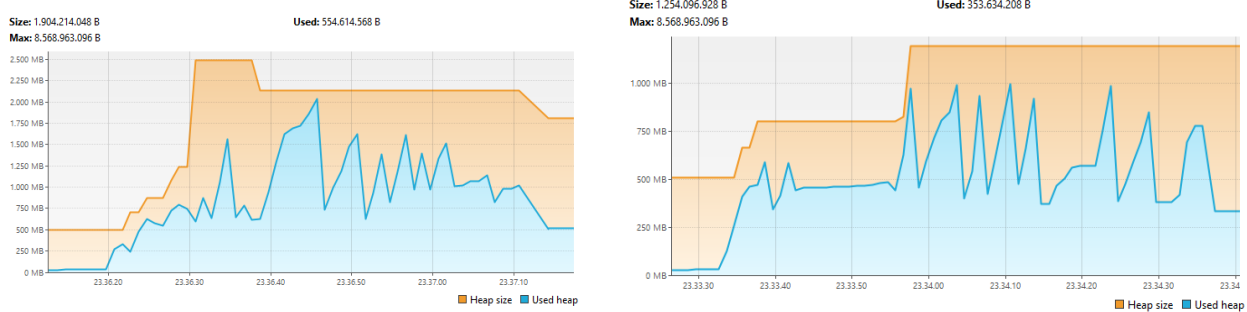
Figure 18: Illustration of problem cause and solution for chunks

reached. Because the TagWay object is connected to the tail node of a way, can the full list of tags in the way be read with all the attributes of the TagWay. TagRelations are at the same time connected to TagWay's the same way as TagWays are connected to TagNodes. That is how all the information of the corresponding TagWay and TagRelation can be read from one single isolated TagNode in the chunk.

Another problem to face is being able to load the right chunks quickly. Instead of iterating through all of the 256 chunks that were made from the original .OSM file, we chose to use the KDTree to chunk what chunks were inside the viewport. This would allow us to read what chunks needed to be loaded in $O(\log n)$ time.

7.1.3 What went wrong and was it beneficial?

When the code was ready to be run a week before the due date, we found a lot of trouble with the new implementation. The ability to filestream and write the objects to binary worked great, but not when it came to reading the data again. We quickly found out that the amount of references in the TagNodes came at a cost. When reading the TagNodes from the chunks, the stack overflowed with references. We therefore could not load any larger .OSM files, without manually increasing the maximum stack size that the program was allowed to use. Only then were we able to load large .OSM files, and benchmarks looked promising.



(a) Benchmark of Copenhagen without chunks, tested one week before the due date. (b) Benchmark of Copenhagen with chunk, tested one week before the due date.

Figure 19: VisualVM benchmark tests of Copenhagen, with or without chunks

Heap-dump of Benchmark	Heap Size	Classes	Instances
Benchmark 19a	522.994.496 B	6.030	12.233.209
Benchmark 19b	320.652.472 B	6.079	5.564.363

Table 10: Heap-dump for the two benchmarks in figure 19

The benchmarks show the chunked version of the application to be significantly more memory efficient than with the version that uses the KDTree. The chunked version's average memory usage is around 450-500 MB, with spikes that do not go above 1 GB when new chunks are loaded in. The chunked version used >75% less average memory and had a reduction of 60%-66% less memory from the peaks of the two benchmarks. The real reason for this large memory reduction can be seen when comparing the two heap dumps- The difference between the amount of instances in the two benchmarks was also reduced by 87%.

7.1.4 What are we left with?

The implementation with chunks seemed promising from a memory perspective, but was not from a practical standpoint. The stack overflow problem was the largest one, and if that needed to be solved efficiently, we would need to refactor a large amount of the application. If we refactored the double-linked TagNodes into primitive arrays or another type, we would need to refactor many other components in the application. This would mean that we also need to refactor how the KDTree handles ways, how relations are drawn, the chunk builder and other components. This would at the same time set us back to having the same problem that made us switch to double-linked-list in

the first place, and there was just no time for that. We therefore changed our focus away from the chunking implementation and focused on optimizing the KDTree to handle and draw data from the .OSM file.

7.2 Trie and searching

Throughout different iterations of development, different ways of searching were used. Initially it was almost entirely based on regex. A regex which split a string into different matcher groups was used to determine what part of the searched string would correspond to the street, house number, city and so on.

Levenshtein distance is a way to measure how close one string is to another. An algorithm to find the Levenshtein distance and comparing two strings was implemented to help this, as it improved the searching, when the user didn't write the address exactly as it was saved in the program. To utilize this, every address street and city were saved in ArrayLists, which were all compared to the searched string, until a suitable suggestion could be made.

After implementing the Trie, much of the former search class functionality was removed, as it was not compatible with the current vision. A lot of the functionality of the search class was moved to the Trie. Most of the address searching functionality was split up between the controller, the Search class and the SearchAddress class. It ended up being a lot more condensed, with the Search class doing significantly less, while the SearchAddress was removed entirely.

In hindsight, there could probably have been a way to preserve the Levenshtein distance algorithm, as it still might have improved the user experience when searching, as it would have enabled the user to still see results regardless of a misspelling. However it would also make searching slower, as more computations would be needed for every search. As the search method is called every time a character is added and then the Trie search, it could potentially cause performance issues when the user is typing. The method comparing Levenshtein distances takes two strings as arguments, which means that all addresses would have to be compared with the specified string. This could again be compared with specific streets saved in ArrayLists as was previously tried.

7.3 Memory optimization

One problem with objects created in the Java language, is the object overhead. When objects are created in Java, it contains something called object overhead, which is a term used for the data

created along with the objects. The data included in the object overhead is for our case mostly unused. Therefore it would be best for us to utilize primitive data types frequently as possible. This means that when using the built in `java.util.HashMap` we are forced to use objects as both key and value. Whereas with the library `GNU.Trove.map.TLongObjectHashMap` library, the keys are stored as primitive data types in one array, which eliminates the object overhead for the keys.

Generally most of Java's standard library uses objects in their data structures instead of primitive types, which through the eyes of memory optimization is not ideal. Therefore when it is possible to use a data structure that utilizes primitive data types instead of objects, we are obligated to do so. This implies when using `HashMaps`, `ArrayLists` and `LinkedLists`.

If we want to calculate the saved byte size when using primitive data types instead of objects, which is important when measuring the memory saved. We will have to tally the byte size per instance. For the primitive data type `long`, it is a 64-bit signed integer, which means the amount of byte it takes up in memory is 8, whereas the calculation for a `Long` object is different. Every object has a header which is 12 bytes and the accompanying data, which in this case is the primitive data type `long` that takes 8 bytes. Besides the object, we also need a reference to the object as it gets stored in a different place in the heap, this reference takes up 4 bytes of memory. Therefore the sum of a `Long` object is 24 bytes.[16] As seen in the table above there is a ~66% saving of memory when

Type	Bytes in total	Comparison
Primitive Long	8	~ 33%
Object Long	24	300%

Table 11: A table of the comparison between object `Long` and primitive `long`

using the primitive data type `long` instead of object `Long`.

Other than selecting the correct data structure to store data, it is important to only store data once and not referring to an object multiple times. When we read the data from the `.OSM` file, the data is stored in the `XMLReader` class, which is then transferred to the `KDTree` and deleted from the `XMLReader`. Which eliminates the duplicate references inside of our program.

Model The primary thing found in the heap is our parsed data and our data structures filled up with that data. Thus, it had to be secured that there were no unused duplications of that data in the memory. This is why the class is a singleton object which is only able to have one single instance and is currently the only class where an instance of the `XMLReader` is constructed. The

data within the tree consists of our relations and ways. These two collections are the ones that fill the most within our heap, therefore we clear them from the XMLReader after parsing them into the static Tree class fields. All remaining static fields are brought up to the singleton instance of Model and then cleared in the XMLReader, ensuring no possibility of having unused duplicates in the heap.

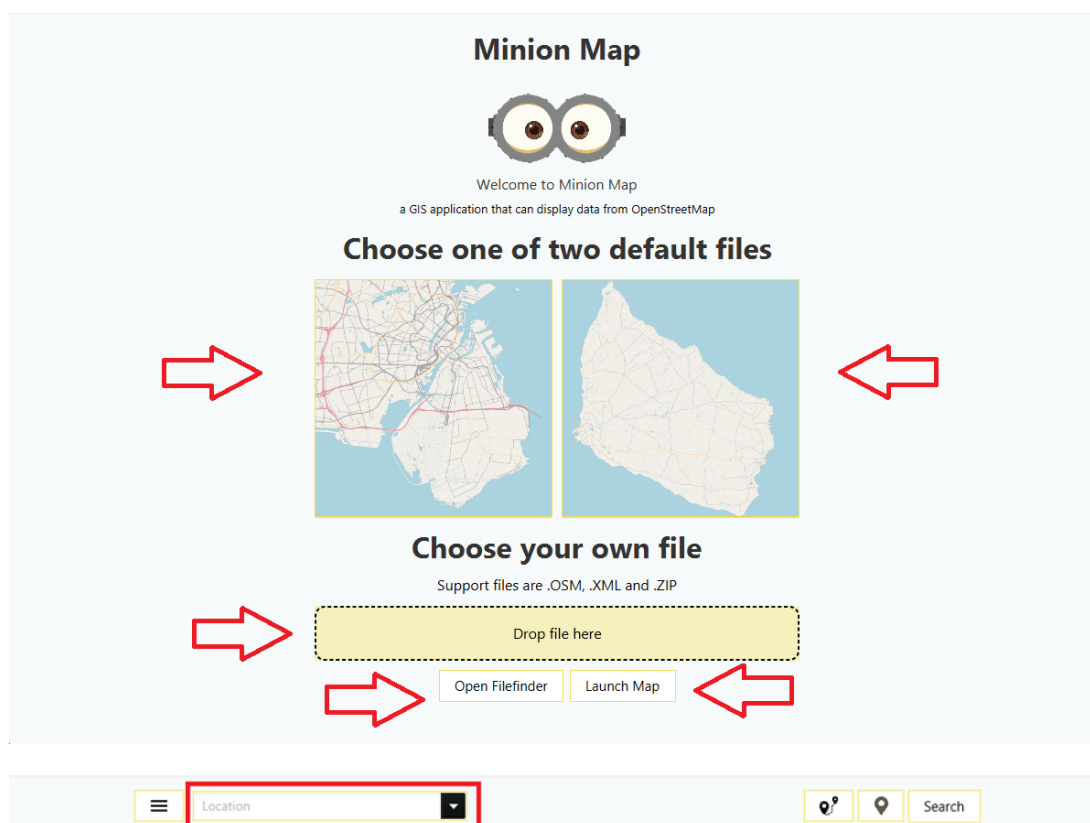
This initialization of the tree itself is very time consuming and fragile due to the amount of data. Therefore we use multi-threading inside of the Model class to load the KDTree. Each thread has its own stacksize and multiple threads can run at the same time, which has allowed us to load multiple datasets into the tree at the same time, cutting costs on load time for the tree.

Pre-loaded model in Binary-file One of the formal requirements was to include a default binary file, which was somewhat achieved, just not the way we wanted it to be done. The model class has been created because of the many benefits there could have been by serializing the class into a file. As previously stated the model class ties up everything that has happens outside the view of the user. By having connected to all components, we could have serialized and outputted all of the needed processed data into a binary file, having substantially better load time on the included default files. But it did not make it to the final product due to serialization of our relations causing stack overflows. We suspect it might be caused by the use of LinkedLists. But due to the lack of time, we chose to focus on other parts of the program.

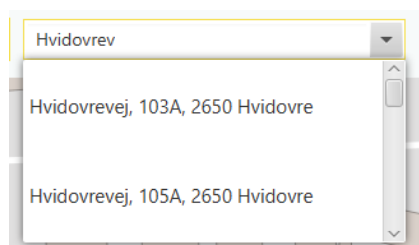
8 User Manual

When launching Minion Map, you will be greeted by a welcome-screen where the following launch options are provided. Launching Copenhagen, Bornholm or selecting your own file to display in the program by dragging and dropping the file into the program. Alternatively, you can manually navigate to a file in File Explorer by pressing Open Filefinder, choosing the file and confirming your choice by pressing “Launch Map”.

This will take you to the loaded map of your choice, providing you with options such as address finder, pathfinding, point of interest marking and theme chooser. Finding an address is done by selecting and typing in the textfield found at the top of the screen.



As you type in your address, you will dynamically give suggestions, saving you from typing the rest of the address. When the correct address is selected, the map will pan to your requested destination.

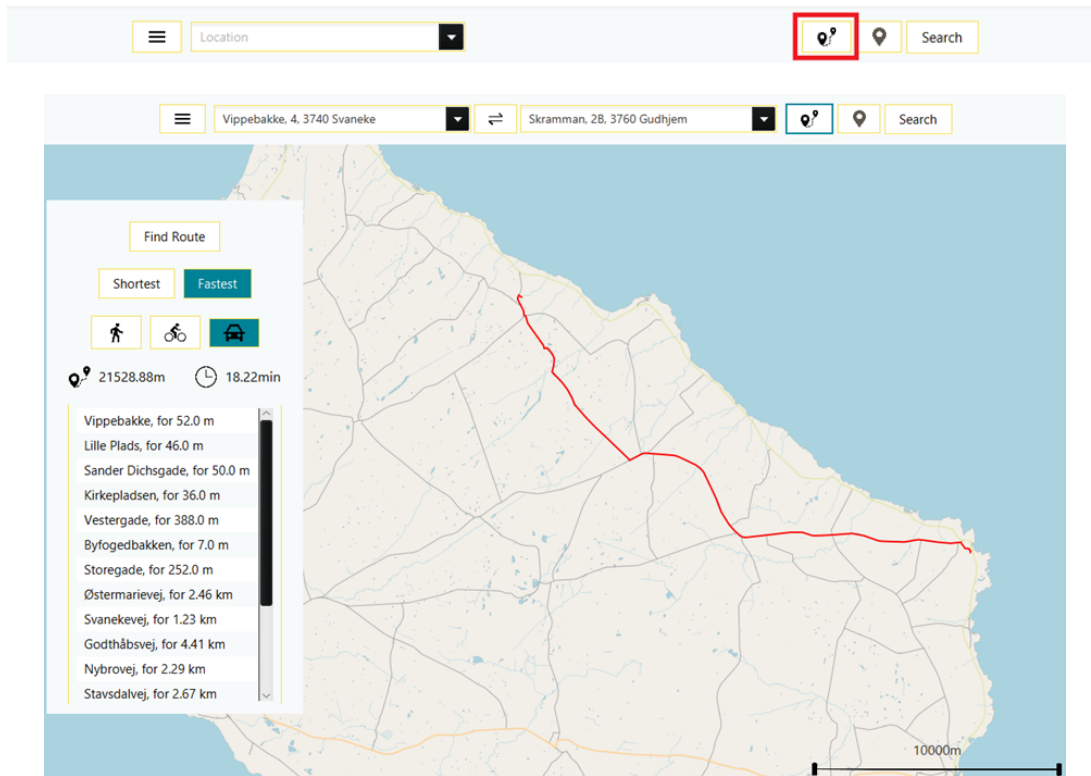


Pathfinding between two addresses can be done by pressing the "Route" button found at the leftmost location of the three buttons in the upper right corner.

From here on you type in your two addresses that you wish to find a path between and press "Enter" or the "Find Route" button.

Also provided is the Transportation menu, giving you options such as quickest or shortest path and choice of transportation. You will also be provided with a route guide, distance of the route and minutes of travel.

The feature for changing the color theme is found in the "Hamburger" menu in the upper left



corner of the screen. Press that and you will find a dropdown menu displaying your options.

9 Conclusion

In the beginning of the project we set out to fulfill the given formal requirements of which we interpreted into the requirements found in **1.3** found section 2.2. Many of which we succeeded in fulfilling - but due to one of our biggest projects, chunking, not making it into the final version, all tags are stored in the heap, which prevents us from loading Denmark. Additionally, there was not enough time to implement auto correction with the Levenshtein algorithm for address-autocorrection and polygon simplification with the Douglas Peucker algorithm.

Despite this, many aspects of the project went well. Having user-based requirements made fulfillment a debate on whether we succeeded or not. As for the system-requirements we can conclude the following.

The program runs the entirety of Bornholm smoothly thanks to the K3DTree sorting everything, both in latitude, longitude and hierarchies. The map is being drawn with a simple aesthetic made possible by our Type enumeration of TagWays and TagRelations. Any address can be searched upon, there is even autocompletion as the user types an address. Two addresses can be linked

through pathfinding, giving the user a general idea of which roads to choose, when driving, cycling or walking. The map has three different styles, ranging from default, darkmode and grayscale.

In terms of what could be improved upon, there is the implementation of chunking. Which ensures that only a limited amount of tags have to be stored in the heap at once.

Implementing Douglas Peucker and Levenshtein, would also be a qualified extension for the program.

Additionally, having more coverage of tests on classes such as DrawingMap and the main application, would be preferred to improve upon.

Regarding memory and benchmarks, it would make sense to possibly reduce the loading time of the map, as it already took 4 seconds loading Bornholm. Loading the entirety of Denmark would take minutes, since it's more than 50 times the size of Bornholm, in terms of Megabytes.

10 Process Reflection

As we now have created our map of Denmark, we do have some improvements to our workflow. Through the project, we spend a lot of time on researching different solutions to our problems, which is great when searching for the best solution. But our ability to recreate a certain algorithm within our project is deemed to be difficult, because of the personalized code written earlier. Where implementing and trying out multiple solutions to a problem creates a lot of time wasted on debugging why a specific implementation did not work. If we wanted to recreate the project, we would have made more concise time-frames for how much time we could spend on researching a specific problem, where if we deem to have used too much time, we would have just taken the best solution from the concurrent knowledge. Another problem that arose from this is that we would spend a lot of time on a specific requirement while neglecting other equally important requirements, until the final week. Instead, it would make sense to make more lazy solutions, that would be quick and allow us to finish all requirements earlier on in the project, following a long process of optimizing and refactoring. What we did rather well, was working at a consistent amount of hours throughout the process, of course we spent a little extra in the final week, but overall there were no weeks where we were not productive. We were also great at helping each other, cooperating on making algorithms, which would ensure that each individual would have a broader understanding of the entire system.

11 Appendix

A Formal Requirements

A.1

1. Allow the user to specify a data file for the application. The system should support loading a zipped .OSM file, but you may support other formats as well.
2. Include a default binary file embedded as a resource of the program, which is loaded if the user does not choose another input file at start-up.
3. Draw all roads in the loaded map dataset, using different colors for the different types of roads in the dataset.
4. Show any rectangle of the map, as indicated by the user.
5. Show the current zoom level, e.g., in the form of a graphical scale bar.
6. Allow the user to change the visual appearance of the map, e.g., toggle color blind mode, or customize which elements to show.
7. Adjust the layout when the size of the window changes.
8. Ensure a clean GUI which is reasonably easy to navigate.
9. Allow the user to search for addresses typed as a single string, and show the results on the map. The program must handle ambiguous user input appropriately, e.g., by displaying a list of possible matches.
10. Make the computation of a shortest route on the map between two points for specified by the user (e.g., by typing addresses or clicking on the map).
11. Allow the user to choose between at least 2 types of routes: for example, for biking routes and cars routes. Car routes should take into account speed limits/expected average speeds. A route for biking/walking should be the shortest and cannot use highways.
12. Allow the user to add something to the map, e.g., points of interest.

13. Be fast enough for convenient use, even when working with a complete data set for Denmark.

The load time for .OSM can be hard to improve, but loading a binary file should be fast. After the start-up the user interface should respond reasonably smooth.

B Group of Conduct

B.1

The working policy for the group.

1. Any breach of the following rules will result in cake-penalty.
2. Academic quarter.
3. Don't be late.

The frequency with which you will meet each other.

2 times a week at a minimum

Usually Monday after lectures and the other day will depend on when the TA decides to have his meeting. So at the moment: Monday, Tuesday.

Bufferday if needed: Friday before lectures.

Where you will work (normally)

2 times a week at the school or another physical place. 1-2 times a week from home.

How well as a group do you plan to do in this course.

We don't have any goal we strive for academically. But we do have a goal on having a great time with the project which will probably result in average performance. We will try our best, but not stress over striving for an exceptional grade.

Progress plan for the first two weeks.

Github workflow, Discuss the idea of group rotations with last member, Contact David.

C Log

C.1

Monday 04-03-2024

Meeting time: 12:00-16:00

We started looking into the project description, and have discussed the different requirements for the project. We have started discussing different ways to render the map in chunks. We have discussed how relations outside our view-box could affect what chunks are rendered, and how we should render them. We decided that this discussion would be better suited for later.

We used a lot of time trying to get a simple version of the program running on all of our machines. We had some issues with the people using IntelliJ not being able to run the program, while the Visual Studio Code users had none of those issues. People were generally fine with switching to VSCode to alleviate any issues, although we did try to fix the issues with IntelliJ.

Tuesday 05-03-2024

Meeting time: 10:00-16:00

The first meeting was made with our guidance counsellor, David. As much has yet to be implemented in our system, we mainly spoke about what was about to come in the following weeks, as well as what to expect in the exam.

After the meeting, we mainly focused on the technical aspects of our work environment. Ensuring that our base project works on all computers, and ensuring that Visual Studio Code handles javaFX correctly.

Otherwise, implementations have not been our highest priority, however, the parse related features have been further developed, and some previous features were moved from IntelliJ projects into various branches in our git.

Thursday 07-03-2024

Meeting time: 10:00-14:30

Today we worked from home, at 10:00 we had a meeting where we discussed the schedule of the day. After that we split up into 3 groups and worked on these issues: Regex, GUI and XML Parsing. We called it at 14:30 ish.

Monday 11-03-2024*Meeting time: 12:00-14:00*

Under this meeting we concluded on using milestones within git forward on. By using milestones we can set some clear boundaries for what we have to achieve every week and by knowing that having the ability to coordinate or estimate when we will take up issue #x.

In this week we will finish up making the first revision of the GUI for the map, Regex- and XML-parser.

Tuesday 12-03-2024

Minor changes to the parser and refactoring.

Meeting time: 10:00-14:00

Due to disease, two of our members were unable to attend today's meeting. We had a meeting with our TA, where we talked about GANT diagrams, and spoke about some of the theory from a prior course we could make use of. We worked on implementing the Levenshtein distance algorithm to give us a sort of "autocorrection" in case the user made a spelling mistake when searching for an address. We implemented matching input with city and street names. We also refactored the parser.

Thursday 14-03-2024*Meeting time: 12:00-16:00*

As we had a lecture, most of the group met up afterwards, although some members were prevented from being physically being at ITU. We split up in the same three groups as last Thursday, and worked some of the same things. We fixed bug so address parsing is possible. We also added types for OSM data.

Monday 18-03-2024*Meeting time: 12:00- 16:00*

Our meeting began with us having an internal meeting about what we accomplished and to what extend we did it last week. Then we talked about what's next, which issues would make the most sense to solve at the moment and after that we delegated the work out between us.

Tuesday 19-03-2024*Meeting time: 10:00- 16:00*

We started the day off with a meeting with our TA. We got some feedback on the project structure and guidance on what our different discussed ideas would benefit the project the most to prioritize. After that we had a clear goal in mind for the week. To display Bornholm.

Thursday 21-03-2024*Meeting time: 10:00- 16:00*

Make the XMLWriter to initialize chunk files.

Other changes to how the map is displayed and more.

Tuesday 02/04-2024*Meeting time: 10:00- 16:00*

The day started off with us having the first progress presentation. The other groups presentations solved some roadblocks or inspired us to do things different. Things that we began working on after this meeting was, GUI-rebuild, K-D Trees, Weighting of different elements which will be used to decide if the element should be displayed at a given zoom-level.

Thursday 04/04-2024*Meeting time: 10:00- 16:00*

Switched to FXML for the UI. Added a few new types to parsed elements.

Monday 08/04-2024*Meeting time: 12:00- 16:00*

KD-tree group from home, agreed on what to prioritize this week. Kd-trees, relations, filestreaming, adress-searching.

Tuesday 09/04-2024*Meeting time: 10:00- 16:00*

Missed meeting, with TA David. But stilled worked together on tuesday.

Thursday 11/04-2024

Meeting time: 10:00- 16:00

We worked from home, had a meeting at 10. ish. and then split up into our groups. Kd-Trees, Relations, Map-painting, File-streamer.

Monday 15/04-2024

Meeting time: 12:00- 16:00

Grouped up and agreed on to do the following this week. - Implementation of custom collections library, that takes primitive datatypes directly. So no reference of object, instead just object. - Address-searching and UI - Pathfinding-implementation - Finishing up KD-trees. Closest node and comment/documentation

Tuesday 16/04-2024

Meeting time: 10:00- 16:00

Fix major bugs in the Mercator Projection code. The core problem was that we didn't negate the latitude in the parser, so originally we did it in the Mercator Projector. This was a mistake because it messed up calculations with everything else.

This also came with a small refactor of the DrawMap class.

Wednesday 17/04-2024

Meeting time: 10:00- XX:XX

On Wednesday we experimented with making TagWay's as a linked-list, that could benefit us in the long run.

This will mean hopefully better chunking capabilities and more. # Wednesday 17/04-2024

Thursday 18/04-2024

Meeting time: 10:00- XX:XX

Used most of the day to refactor code in Mecator Projection and other parts of the code. Also a lot of focus for memory optimization.

Monday 22/04-2024

Meeting time: 10:00- 16:00

We merged a lot of code into main from the previous week. Work-on KDTree and Chunks was also in focus, where we fixed alot of problems with writeing to chunks and improved the KDTree to be used later.

Tuesday 23/04-2024

Meeting time: 10:00- 16:00

Start to work on 3D KDTrees, where the their dimension is for a hierarchy system for what to draw and when.

Start to workon how to read from chunks.

Achived functional address searching.

Thursday 25/04-2024

Meeting time: 10:00- 16:00

Made the complete switch to linked-list based TagWay's.

Memory improvements and a new feature for a zoom-bar has been made.

Monday 29/04-2024

Meeting time: 10:00- 16:00

Working to improve the searching for an address and more!

We also try to use a new tree data structure for searching, the structure is Trie.

Tuesday 30/04-2024

Meeting time: 10:00- 19:00

Trie is now fully functional, and allows us to search for a given address, where the program will autocomplete throughout. Additional development has been made on pathfind, which now has had a working instance.

Wednesday 01/05-2024*Meeting time: 10:00- 22:00*

New color palette, inspired by OpenStreetMap's color theme, and added more ui for searching and pathfinding.

Thursday 02/05-2024*Meeting time: 10:00- 18:00*

Functional incorporation with a*.

Friday 03/05-2024*Meeting time: 10:00- 23:00*

Tests added, refactoring directory.

Saturday 04/05-2024*Meeting time: 10:00- 19:00*

Most branches merged into main.

Sunday 05/05-2024*Meeting time: 10:00- 20:00*

Documentation, tests, css style.

Monday 06/05-2024*Meeting time: 12:00- 01:30*

Final changes, testing, functional zipping.

D Git Log

D.1

Officiel Release version 1.0

What's Changed

- init project by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/1>
- Java fx init by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/15>
- Initial OSMParser by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/21>
- Address search by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/23>
- GUI by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/25>
- Address search integration by @Spurberino in <https://github.com/johannes67890/MinionMap/pull/28>
- Map drawing by @AndreasLN in <https://github.com/johannes67890/MinionMap/pull/31>
- Add MecatorProjection class and update TagNode constructor by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/32>
- Zip file and pathfinder implementation by @MessiGames30 in <https://github.com/johannes67890/MinionMap/pull/30>
- Merge from main by @AndreasLN in <https://github.com/johannes67890/MinionMap/pull/33>
- Merge with FXML rebuild by @Hopsasasa in <https://github.com/johannes67890/MinionMap/pull/37>

- OSMParser Version 2 by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/38>
- Osm parser v2 by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/39>
- Filestreaming by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/40>
- Import alg4 lib by @AndreasLN in <https://github.com/johannes67890/MinionMap/pull/42>
- Map coloring relations by @AndreasLN in <https://github.com/johannes67890/MinionMap/pull/43>
- K d tree implementation by @MessiGames30 in <https://github.com/johannes67890/MinionMap/pull/44>
- Map coloring relations by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/45>
- Zoom scale made, comments made, want to merge into main by @MessiGames30 in <https://github.com/johannes67890/MinionMap/pull/47>
- Macatorprojection fix by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/49>
- KDTree implementation with Mecator by @Hopsasasa in <https://github.com/johannes67890/MinionMap/pull/50>
- Memory opt by @johannes67890 in <https://github.com/johannes67890/MinionMap/pull/51>
- Merge pull request #51 from johannes67890/memoryOpt by @AndreasLN in <https://github.com/johannes67890/MinionMap/pull/52>
- Kd tree optimization into main by @AndreasLN in <https://github.com/johannes67890/MinionMap/pull/53>

- main into search by @AndreasLN in
<https://github.com/johannes67890/MinionMap/pull/55>
- Kd tree memory opt by @johannes67890 in
<https://github.com/johannes67890/MinionMap/pull/57>
- Zoombar scaling by @MessiGames30 in
<https://github.com/johannes67890/MinionMap/pull/56>
- relations fix into main by @AndreasLN in
<https://github.com/johannes67890/MinionMap/pull/58>
- Color optimization into main by @AndreasLN in
<https://github.com/johannes67890/MinionMap/pull/59>
- Updated when to load smaller streets by @Spurberino in
<https://github.com/johannes67890/MinionMap/pull/60>
- Zoombar fixed by @MessiGames30 in
<https://github.com/johannes67890/MinionMap/pull/61>
- Zoominonrelation into main by @AndreasLN in
<https://github.com/johannes67890/MinionMap/pull/62>
- Testing into main by @AndreasLN in
<https://github.com/johannes67890/MinionMap/pull/63>
- UI by @johannes67890 in
<https://github.com/johannes67890/MinionMap/pull/64>
- Docs by @johannes67890 in
<https://github.com/johannes67890/MinionMap/pull/65>
- Swap button fix by @johannes67890 in
<https://github.com/johannes67890/MinionMap/pull/66>
- Bin last chance by @johannes67890 in
<https://github.com/johannes67890/MinionMap/pull/67>

- Zip file fix by @johannes67890 in
<https://github.com/johannes67890/MinionMap/pull/68>
- Tests by @johannes67890 in
<https://github.com/johannes67890/MinionMap/pull/69>

E New Contributors

E.1

- @johannes67890 made their first contribution in
<https://github.com/johannes67890/MinionMap/pull/1>
- @Spurberino made their first contribution in
<https://github.com/johannes67890/MinionMap/pull/28>
- @AndreasLN made their first contribution in
<https://github.com/johannes67890/MinionMap/pull/31>
- @MessiGames30 made their first contribution in
<https://github.com/johannes67890/MinionMap/pull/30>
- @Hopsasasa made their first contribution in
<https://github.com/johannes67890/MinionMap/pull/37>

Full Changelog: <https://github.com/johannes67890/MinionMap/commits/1.0>

References

- [1] Intergovernmental Committee on Surveying and Mapping. *Fundamentals of Mapping*. Maj 2024. URL: <https://www.icsm.gov.au/education/fundamentals-mapping/history-mapping> (visited on 11/05/2024).
- [2] OnlineStreetMap.com. *About us*. Maj 2024. URL: <https://www.icsm.gov.au/education/fundamentals-mapping/history-mapping> (visited on 11/05/2024).
- [3] FME.com. *OpenStreetMap (OSM) XML Reader/Writer*. Maj 2024. URL: <https://docs.safe.com/fme/html/FME-Form-Documentation/FME-ReadersWriters/osm/osm.htm> (visited on 11/05/2024).
- [4] Russell A. Brown. *Building a Balanced k-d Tree in $O(kn \log n)$ Time*. Journal of Computer Graphics Techniques (JCGT), 2020.
- [5] Wayne Sedgewick. “Algorithms - Fourth Edition”. In: Princeton University, 2021. Chap. 5.2.
- [6] Geeksforgeeks.com. *Indexed Priority Queue with Implementation*. 2022-29-11. URL: <https://www.geeksforgeeks.org/indexed-priority-queue-with-implementation/>.
- [7] Wayne Sedgewick. “Algorithms - Fourth Edition”. In: Princeton University, 2021. Chap. 2.5.
- [8] Wayne Sedgewick. “Algorithms - Fourth Edition”. In: Princeton University, 2021. Chap. 4.3.
- [9] Geeksforgeeks.com. *Graph representations using set and hash*. 2023-08-03. URL: <https://www.geeksforgeeks.org/graph-representations-using-set-hash/>.
- [10] Stuart J. Russell Peter Norvig. “Artificial Intelligence: A Modern Approach”. In: University of California, 2022.
- [11] Vijaya Ramachandran Mo Chen Rezaul Alam Chowdhury. *Priority Queues and Dijkstra’s Algorithm*. 2007-12-10. URL: <https://www3.cs.stonybrook.edu/%7Erezaul/papers/TR-07-54.pdf>.
- [12] Wayne Sedgewick. “Algorithms - Fourth Edition”. In: Princeton University, 2021.
- [13] Stuart J. Russell Peter Norvig. “Artificial Intelligence: A Modern Approach”. In: University of California, 2022. Chap. page 82.
- [14] Elmar de Konning. *Douglas-Peucker*. 2011. URL: <https://psimpl.sourceforge.net/douglas-peucker.html>.

- [15] Scaler.com. *White Box Testing*. 2023-16-11. URL: <https://www.scaler.com/topics/software-testing/white-box-testing/>.
- [16] Baeldung.com. *Memory Layout of Objects in Java*. 2024-08-01. URL: <https://www.baeldung.com/java-memory-layout>.