Contents

1 Introduction

This report is the result of a semester project at ITU in the fall of 2012. It is based on a piece of software written by five first-year students from Software Development. The report contains an exhaustive overview of the program including requirements, discussion, testing, problem analysis, and evaluation.

During the course of the project, we developed an interactive map of Denmark based on data from a well known and widely used map service provider, Krak maps. The program can be used to navigate on a map of Denmark as well as planning the shortest route between two points. The program is a client-based map service which saves the user the tedious process of manually planning routes on a non-digital map.

The program is intended to give the user a responsive and easy way to navigate around the streets of Denmark, helping the user to plan travel routes by vehicle through Denmark (and the southern part of Sweden).

This report contains sections;describing and discussing the obligatory requirements of the program, as well as additional extensions. Each implemented extension is explained separately.

2 Problem description

In this section we will describe and discuss the various needs of a user that constitute the core requirements of the program. We will also cover the background and context of certain software details through a thought up typical user scenario. On the basis of this, we will write a list of requirements for the program. Moreover, we will discuss any relevant framework and theories that will help explain the context of the program. Finally, we will consider decisions regarding our general system design.

2.1 Background

The need for accurate navigation is a need as old as the need to travel. Man have developed many systems for navigation with cartography, the making of maps, being one of the most important. However, maps have traditionally been very inaccurate, as they were designed by account of experience and/or by cartographers measuring the land by foot. The development of flying machines helped to make maps more accurate; however, maps were still manually drawn and thus subject to human error.

The last fifty years have made the computer an invaluable tool of the cartographer. Now, one can create detailed maps based on geographic information gathered with great precision via satellites. One could argue that the maps created now are not only far more accurate than earlier; their correctness can also be demonstrated through the use of satellite images and complex calculations.

The problem of planning a route arises when someone wishes to go from A to B in a way that is efficient and thought out. Using a map to plan a route seems obvious as it is easy to convince oneself that a given route is the shortest, if not the fastest. Planning a long route with many different roads and turns can be a very tedious process, subject to various measuring errors. A computer performs this process much more efficiently, being able to process a large number of possibilities and guaranteeing the shortest route from A to B, assuming the calculations for doing so are correct.

The improved means of transportation throughout the country combined with the low cost of such means quickly forms the need for efficient route planning. It is this need we are trying to address with this project in an effective and user-friendly way.

2.2 Scenario: Using the program

The following is a thought out scenario where a fictive user attempts to use the route planning system of our program. The description is not a typical use-case scenario, instead we will loosely describe problems when they arise and derive a list of requirements from the scenario.

2.2.1 Navigate the map

The user wishes to navigate around on the map to make sure he is looking at the relevant parts of the route. In addition, he wants to navigate the map after he has searched for a shortest route, so that he can see different parts of the route at various scales, giving a visual impression of the travel length of a route.

Navigating the map should be fast and intuitive. One could use buttons or the scrollwheel, but whatever the implementation, it should be fast and scalable.

The user could run into several problems when navigating the map:

- The program could be **slow to respond**, which would make navigating very frustrating, if not impossible.
- The program could show **too many details on the map**, making sections like large cities seem very clustered and impossible to navigate through.
- The user could lose his **orientation** entirely, making the map seem out of focus or perhaps end up somewhere in the ocean. The user would then need to be able to easily navigate to appropriate part of the map.
- The user could end up not being able to **distinguish road patterns** in the map, for example not recognizing highways from pathways.

These problems are central when using the map and have been taken into consideration when constructing our list of requirements that the program should follow.

2.2.2 Planning the route

The next step for the user would be - when he is done navigating the map - to plan a route. The user looks for input fields where he can input his starting address and his destination into the system. The user could experience a range of problems during this step:

- The user could mistype the address searched for, making the program unable to process a valid route.
- The user could have an invalid address as original input, resulting in the same problem as in the above point, but requiring a different solution.

When the user is done with the input, the program reaches a critical step: computing a shortest route from the input and returning the result to the user. Again, a range of problems could occur:

- There would be no shortest route available - in this case, the user may or may not get an explanation as to why there is no route.
- The user could not find the shortest route on the map (e.g. no graphical representation of the route was available).
- The user was not satisfied with the shortest route and wishes to re-compute another route with new inputs.

If none of the above mentioned problems occurs, the user should now be done with the route planning and can proceed to use the information elsewhere.

To summarize, the basic concepts of navigating the map easily and planning a route easily are necessary if the program is to be considered successful. However, some additional steps are required if the operations are to run smoothly. These steps will be discussed in the next section.

2.3 Requirements

Based on the user scenario, we will now derive a list of requirements. These requirements are what we will measure our softwares performance up against. They are also used to maintain a common thread throughout the report.

The requirements that we show in this section are comprised of two parts. The first part is a list of technical and descriptional requirements that are obligatory to the project. The second part is a list of requirements that we have thought out, and which we believe will improve the core functionality of the program as well as provide some additional features for the user.

2.3.1 Core requirements

The core function requirements are partly given in the project description and is partly a result of our analysis of the problem. The full description of the requirements given by the project description can be found in the project description[5].

- The user should be able to navigate the map by using mouseonly, zooming in by scrolling and panning by click-and-releasing the mouse.
- The user should experience a responsive environment (once loaded), at all zoom levels, with all operations.
- The user should be able to easily navigate throughout the GUI, intuitively knowing where to click to generate a route.
- The user should have a visual representation of the route when it has been computed.
- The user should be able to easily distinguish road types from each other, by color as well as thickness.
- The user should be able to repeatedly use any of the functions mentioned above.

2.3.2 Additional features

These requirements above are linked to the requirements in the project description. Moreover, we have derived a list of requirements that are suitable to develop extensions from:

- Should the user misspell the address in the search fields, the user should be notified from the GUI that it is an ineligible address.
- Should the user have a wrong address from the get-go, the user should receive suggestions as to what to write in the address field.
- When the user receives a list of addresses, the list should contain multiple matches for addresses with same name but different postal codes and postal names.
- The program should be faster to load than what the original input files from Krak allows.

- The program should be safe, not causing crashes. While loading, the program should never give the impression of freezing.

These requirements combined with the guidelines in the project description should allow us to build a fast, responsive, user-friendly program. We have a narrow margin of optional requirements so that we dont promise too much, rather aiming to implement additional requirements thoroughly and well-documented.

2.4 Theory and input

In this section we will discuss the theories required to understand the program. We will explain the terminology related to coordinates and the re-calculation of these to fit our program.

2.4.1 Terminology

The following sections will make use of terminology thats more relevant than 'roads' and 'road sections'. We will briefly describe it here.

Consider a map built from a system of roads. Each road consists of one or more sections describing the length of the road, the angle, the type etc. Henceforth, we will denote roads as these sections, also called *edges*. Each edge begins and ends somewhere on the map. These "endpoints" will now be denoted as either *points, nodes* or *vertices*. Each name describe a different function in the program, but they are all representing the same coordinate sets.

2.4.2 Geometric coordinates and the coordinate system

The input data uses coordinates measured in System 34, a danish coordinate system. The system is built upon a north-directed Y axis and west-directed X axis. The coordinate system covers most of Denmark but not Bornholm, which uses its own coordinate system. The error margin of the system is at most 5cm/km. System 34 has been deprecated since 2006, being replaced by a joint European coordinate system. This may make the program incompatible with newer coordinate systems such as UTM/ETRS89.[2]

2.4.3 Interpreting input coordinates

When being presented with coordinates that follow certain geometric standards, and one wishes to convert these coordinates to a different coordinate system, one needs to transform the old coordinates so they conform to the new coordinate standards.

The method we used to figure out this transformation was to consider the original coordinates as being contained in a square (original square) in an arbitrary coordinate system. We then added another square (transformed square) inside of the original square, that was to contain the new coordinates. The challenge was then to figure out a formula to transform all coordinates from the original square to coordinates in the transformed square, while keeping the same relative position between the coordinates intact. This is illustrated below.

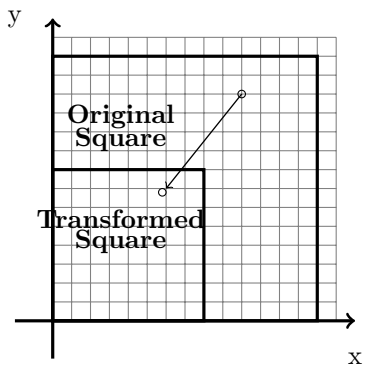


Figure 1: `Transforming coordinates from one system to another`

In our case, it was necessary to transform coordinates from a large system to a small one, but the same principle would apply if it was the other way around.

The resulting formulas are as follows.

$$transformed_x = \frac{original_x * transformed_{width}}{original_{width}}$$
$$transformed_y = \frac{original_y * transformed_{height}}{original_{height}}$$

$transformed_x$ and $transformed_y$ represent the transformed coordinate.
$original_x$ and $original_y$ represent the original coordinate.
$transformed_{width}$ and $original_{width}$ represent the width of the transformed square and the original square respectively. $transformed_{height}$ and $original_{height}$ represent the height of the transformed square and the original square respectively.

This approach works well between coordinate systems, where the axes are in the same direction. The problem we ran into was that coordinate systems in Java use a downward facing y axis instead of upward facing like in classical, mathematical coordinate systems. This is fixed with a small addition to the above formula for the y coordinate:

$$transformed_y = \frac{((original_y * -1) + original_{height}) * transformed_{height}}{original_{height}}$$

We used this approach in our Viewport class to transform the coordinates. The implementation details of this formula vary slightly from what is seen above, but the principles are the same.

3 Problem analysis

In this section we will analyze and discuss how to design a program that fulfills our requirements. This section is intended to give the reader an impression of *why* we have designed our program as it is. First, we will discuss our choice of user interface platform. Then, we will analyze different data structures and choose a suitable one. And finally, we will describe how to solve the route planning problems along with the data structure needed to do so.

3.1 Creating the GUI

This section will discuss the important decisions regarding our user interface. We will cover our choice of platform on which to display our user interface. Moreover, we will discuss how we have decided to design our GUI in terms of component placement etc.

3.1.1 Java vs JavaScript

The project description states that our choice of user interface platform lies between a Java based GUI and a JavaScript/HTML-based interface[5]. Another choice we had to consider was whether to implement our graphical user interface using Java with the Swing toolkit or a HTML-based solution using JavaScript and SVG. The upsides of using JavaScript were that we would learn a new programming language and probably get a lot of experience. Moreover, SVG (Scalable Vector Graphics)[1] has some interesting features. SVG looks smooth and scales well when resizing, a requirement we would have to implement manually in Java. On the other hand, learning SVG and JavaScript would be very time consuming. We also experienced that SVG had some performance issues when drawing many roads.

Developing for a browser-based interface also contains a lot of browser-specific coding, which is very time consuming. JavaScript is not a typesafe language; neither does it support alot of exception handling, contrary to Java. Seeing as security were an important part of our requirements, the characteristics of JavaScript played a big role in the decision. Doing the graphical interface in Java Swing seemed like the safer choice, as we were all a lot more experienced with the Java Swing toolkit.

Doing it the easy way allowed us to focus on building a good data structure and secure stability. Moreover, we had considered an auto-completion feature which seemed more readily implemented in Java than in JavaScript. To summarize, here is a brief comparison chart covering our estimated pros and cons when choosing our user interface strategy.

Local Java	
Pros:	Cons:
Experience with development Easily accessible API (Swing) Supports planned feature	Ugly user interface Swing may be hard to debug

JavaScript/SVG	
Pros:	Cons:
Platform independent New relevant technology SVG has a nice look and scales well	No experience SVG may have performance issues Very hard to debug

3.1.2 Designing the GUI

This section discusses our thoughts regarding the design of a Java GUI, using the AWT and Swing framework. We will discuss how we think the GUI should look in order to support our requirements.

As decided in the problem description, we will use Java code both to manipulate the data from Krak and also to present it to the user. Hence, we need the common Java AWT and Swing framework so that we dont have to build the entire GUI from scratch. This framework allows us to select from a wide range of components when building the GUI. We will skip the discussion of alternatives to choose from, as Swing is

an industry standard and as such is the most widely used GUI framework. In addition, Swing components are not only easy to implement, they are also platform independent and generally easy to customize.

When first creating the frame, we discussed how we wanted the visuals to look and where to place the canvas, the buttons and the like.

We also looked at different map services for inspiration on how to do it, but that did not help as much as we thought, as many of the services had very fancy GUIs which would be hard to reproduce in Java. Also, all of the GUIs were on browser based platforms, which gave them other possibilities than us. In the end we liked Kraks GUI design the best, and our layout is thus heavily influenced by it.

Krak has a pane in the top containing a search field and a search button. We found that this was a smart way to do it, as the search field and button will not interfere with the map and it will also - in addition to this - become the first thing the user sees when the program starts.

Furthermore Krak also have GUI related elements in front of their canvas. This would be much more difficult to implement in Java, as we would be unable to use components like we did in the top pane, but would instead have to implement mouse listeners on specific areas of the canvas, draw the icons manually and many other details that would be too comprehensive to create. As such, we had to think of alternate ways to implement the functionality that we now lacked.

In the end our decision fell upon creating a top pane separate from the map contaning search functionality, and having the map just below. We decided to also add a pane in the bottom of the screen containing buttons for navigating the map in order to make up for the lacking interface features. These involved zoom and panning buttons in addition to a map start and a map reset button.

The zoom and panning buttons were located on different tabs, since they took up a lot of space. This proved to be very inconvenient for the user, as they needed to switch tabs every time they changed from zooming to panning.

Because of this inconvenience to the user and the fact that we made it possible to zoom and pan with the mouse directly on the canvas, we chose to remove the panning tab all together and only keep the zoom buttons and the map start and map reset buttons. This also made for a more simple and easy to use interface.

3.2 Choice of data structures

3.2.1 Data in multiple dimensions, choosing the KDTree

The KRAK dataset consists of two files; one file containing data on the coordinates as a list of points, the other containing data on connections between these points, representing edges. The responsibility of the model in the light of this would involve:

- Retrieving the data and constructing road objects containing all the information needed for a visualisation client to draw the roads.
- Answering client method calls, returning the appropriate data in reasonable time

Based on this crude description of the model, it was possible to formulate a number of requirements.

Writing a visualisation client that draws every edge, regardless of the zoom level would be a waste of resources. Thus implementing a datastructure that supports range searching in the multiple dimensions was necessary, in order to create a responsive program. Furthermore the running time of the search operation would have to be reasonably low, as this operation would be performed whenever a user panned the map or

zoomed in or out.

These requirements implied a number of possible data-structures. Initially, using an SQL database was considered, as all of the members of the project group had previous experience with this technology. An SQL database would certainly accommodate the requirements for retrieving data based on queries in multiple dimensions, as well as doing so in a reasonable running time. It would also make it easy to implement a number of clever future features, like finding a section of the map containing a road with a specific name, or auto-completing user search addresses, provided that the database had been organised into appropriate entities.

The frequency of the queries that would be necessary to pass to the database however, made the SQL solution less than ideal; a visualisation client would have to ask the model for the relevant data each time a user panned or zoomed in the map, resulting in a truly huge amount of queries, creating an unreasonable dependency on the database connection. To solve this, another data-structure would still have to be implemented to hold the KRAK data to answer the requests from a visualisation client.

We still had two other possible data-structures that we had knowledge of remaining. A grid file or a multi dimensional search tree, be it binary or otherwise, of which we were aware of two; a k-dimensional tree (kd-tree in short) or a quad tree. The material that we were able to obtain on the grid file data-structure did not suffice for making us confident enough in our understanding of its concepts, for us to take on writing an implementation within the given time frame of the assignment, leaving the quad tree and the kd-tree.

The quad tree supports searching in two dimensions[8] (another similar data-structure, the octree supports three-dimensional search), whereas the kd-tree supports searching in any number of dimensions. It was clear that the data-structure had to support at least two dimensional search in order to solve the visualisation task, but a number of other potential challenges could be addressed by choosing the kd-tree. As the kd-tree is scalable to more than two dimensions, it would be possible to filter roads by name, by road type or something else entirely in addition to the start and end points of the road. The question remaining was whether or not the kd-tree would provide range search functionality that was fast enough for purposes, which will be discussed in the next paragraph.

The binary search tree data-structure upholds the invariant that all keys in the left sub-tree rooted at any given node are smaller than the key of the root node, and all keys in the right sub-tree are greater than the key of the root node. The kd-tree only differs from that structure in that, as the depth of the tree increases, the different dimensions of the keys are used for comparison in a cycle.

In a 2d-tree for example (a kd-tree where $k = 2$), where keys are essentially points containing x and y values, keys would be compared in their x-value at the root, their y-value at the children of the root and again in their x-values at the children of the children of the root.

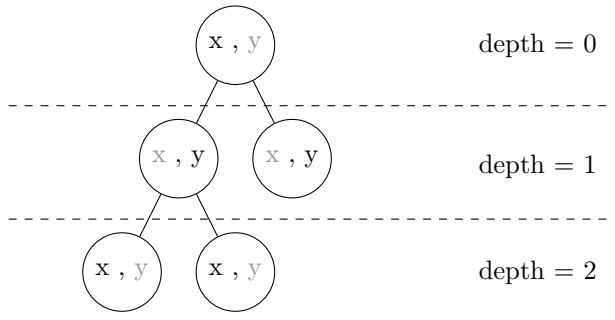


Figure 2: A two dimensional binary search tree

The running time of a range search that reports all the nodes would be $\sim N$ in a kd-tree with N nodes, regardless of how well balanced the tree is. However most range queries in our program would rarely,if ever, result in traversing the entire tree, as a GUI client should not attempt to draw all edges at once in the first place.

A more refined analysis takes into account not only the total number of nodes, N, but also the reported points of the range search, k. As the search method visits each of the reported nodes, the running time should include the time spent at each reported node, which would be k. The remaining nodes visited would be the nodes on the paths of the reported nodes, which in a balanced tree would be at most $\sim logN$. Therefore, the total time spent would be $logN + k$[3], which for our purposes would be an acceptable running time.

Considering the pros and cons of each data-structure, we chose the kd-tree for the following reasons:

- We felt confident that we could write a concrete implementation within the time frame of the project.
- It supports range searching in more than two dimensions.
- It carries out range searching with an acceptable running time.

3.2.2 A new direction, the directed graph and its implementation

In this section we would like to discuss the choice of shortest path search algorithm and underlying data-structure required to support this feaure.

From our course in algorithms and data-structures we had gained sufficient knowledge regarding the distinction between different data structures to know that pathing problems are solved by graphs[6]. In our case we needed a structure that maintained the boundaries set in the model while providing an efficient structure for searching. In our case this meant a structure which maintained connections between points[1] in addittion to contain information about whether these connections were unidirectonal or not.

A simple graph implementation would suffice for keeping track of connections, but we needed more information such as which edges were reachable, in case we had to traverse one-way streets. What we needed in order to satisfy this was a structure known as a Directed Graph. Its basic implementation is based on maintaining an array with the total number of all the vertices, where each vertex refers to a bag that contains the vertices' adjacent edges. With this structure every edge from our multi dimensional tree would be processed and mapped in arrays where the source would be that edge's starting vertex.

An alternative structure for implementing a graph could be a 2-dimensional array list of edges(adjacent[1][2]). In this case the index accesses would refer to the starting vertex and end vertex of the edge at that current position ie. adj[100][200], would contain the edge going from vertex 100 to vertex 200. Even though this structure is fast and efficient for looking up single edges - the problem lies in looking up references. The Directed Graph maps to all edges originating from the vertex, where the 2D-array only returns the edge between 2 vertices. Furthermore in a 2 dimensional array if you want to find out which edges go from [1][0..N] you have to iterate over the entire array, even through the null references - this is rather time consuming for large collections where the directed graph just returns a rather simple bag containing ONLY the edges linked directly to it, no null references. So in the construction the directed graph has already been reduced to containing the required information.

So to sum it up, an obvious advantage of using a Directed Graph is, as the name implies, the directional information in the structure which a simple graph does not contain. In relation to advanced planning, where

[1] Connections between points are referred to as edges

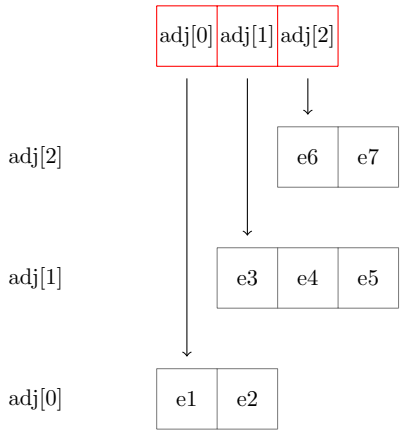


Figure 3: Example of directed graph adjacency array

y x	0	1	2	3
0	null	e1	e2	e5
1	e1	null	e3	null
2	e2	e3	null	e4
3	null	null	e4	null

Figure 4: Example of 2D-edge array

multiple edges have to be linked for computing routes, the bag structure restricts the algorithm to only iterate over relevant edges, so we avoid having to iterate over the entire collection every time or the entire dimension in a 2D-array.

3.2.3 Dijkstra

Dijkstras algorithm works by keeping track of the shortest path from a source vertex to a target vertex by maintaining an array(distTo[]) with the shortest distance from the source to the target through all other vertices in the graph. Then by using a priority queue it keeps relaxing edges(similar to a stretched rubber band that keeps getting loosened) till we end up with the shortest possible path which is then returned.

Dijkstras algorithm was our main choice for computing the shortest path. We basically went with a text book implementation and altered our graph and our search algorithm to support the specific data implementation we already had. Rather than looking up the vertices, we looked up the edges and then accessed their starting vertex' ID.

Same as when choosing the graph structure, we went with the knowledge we had gained from our BADS course and Dijkstras search algorithm was the algorithm we got introduced to as being superior for route planning. We mainly chose Dijkstra over bellman-ford due to the guaranteed worst case runtime - which is its major strength.

Algorithm	Restriction	Average	Worst-case
Dijkstra	Positive weights	E log(V)	E log(V)
Bellman-Ford	no negative cycles	E + V	E V

Figure 5: Table derived from Algorithms 4th. Edition p. 682

3.3 Design patterns

When designing a system like ours that includes 35+ classes, it becomes necessary to follow good coding principles. For example due to testing and maintenance, it's important that the interaction between classes are well defined. Early in development we experienced problems, that would easily be solved with the use of design patterns, adapting them to fit our needs while keeping their intended purpose intact.

The design patterns in our system can be divided into two categories: some small, encapsulated patterns that are used to secure well defined functionality in one or a few classes: and some large, integrated patterns designed to secure low coupling between major parts of the program.

3.3.1 Overall system pattern

When deciding on a pattern that would affect the flow and structure of our entire program, we devised a list of attributes that we would strive to attain:

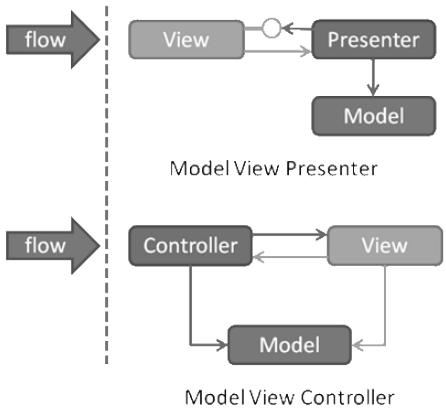
- Low coupling between class packages.
- Simple control flow between classes.
- High degree of separation between domain logic and interface.
- Clear, understandable terminology.

With these attributes in mind, we chose to adapt a Model-View-Presenter pattern to our program.

3.3.2 Model-View-Presenter

The Model-View-Presenter pattern is a pattern concerning the system structure and the flow of data in the program as a whole. It is derived from the Model-View-Controller (MVP) pattern, and the MVP pattern's general purpose is to make it easy for the programmer to later change the way the program is presented to the user.

Most calculations that formerly were in the view, according to the MPC pattern, are in the MVP pattern moved to the presenter. This is why it should be easier to make test cases for some of the view functionality, because it is in fact situated in the presenter.



All in all, this pattern describes a programs flow.

In our program the control flow is as follows: An action happens in the view, for instance a button is pushed. Most of the calls first goes through the view to change different fields and afterwards calls the presenter (our presenter class is called Controller) to let it update the information the view needs. The presenter does this by going to the model to fetch the needed information from one of our data structures. When the presenter gets its hands on the data it either processes it and then send it to the view or plainly passes it along. When the view receives updates, it updates itself to fit the new data.

3.3.3 Patterns throughout the model

In the model we have used a variety of different design patterns. Because one class can implement several pattern properties the patterns we have used will be briefly described, after which each pattern-affected class and its function will be described together with an explanation of, why we found the specific mix of patterns useful for that particular class.

Overview
Patterns are generally broken down into three categories: creational, structural, and behavioral.

Creational
Creational patterns are patterns that deal with the mechanisms involved with object creation, controlling how the creation of objects should be performed. Without creational patterns, object creation and instantiation could quickly add to the complexity of the program, making it difficult to spot exactly where, which objects get created and why, which in turn could substantially increase the coupling between classes. The creational design pattern we make use of in the model is the Singleton pattern.

Singleton
This is the most used pattern in the model, it ensures that only a single instance will ever exist of a particular class. As a side note, all classes implementing the Singleton property in our program do so through the use of the enum type.

Structural
Structural patterns are patterns that deal with the overall design of class interaction, specifying ways to easily identify relationships between classes. Employing structural patterns can be a way to decrease coupling between classes which in turn increases modularity.

The structural design patterns we make use of in the model are the Facade pattern, the Adapter pattern, and the Proxy pattern.

Facade
This pattern specifies how clients only have access to one interface (the facade) in the model, and everything else is completely hidden. We found this to work well with the behavioral mediator pattern.

Adapter
Also referred to as wrapper, this pattern is all about modifying an existing class methods to work with other classes. This also helps to hide unwanted functionality from the existing class to its clients. We found this to work well with the singleton pattern.

Proxy
This pattern is about having clients operate with a proxy instead of operating directly with a specific class. We have used this mainly in relation to serialization (see section 10).

Behavioral
Behavioral patterns are used to specify common communication patterns between objects. By doing this, communication between objects become more flexible.

The behavioral design pattern we have used in the model is the Mediator pattern.

Mediator
This pattern is about creating a class that encapsulates how a set of objects interact with each other. This can vastly increase maintainability.

3.3.4 Implementation of design patterns

Class: Search
The Search class is an attempt to implement the Facade pattern property by creating a single interface to be used by all clients outside the model. This means that any calls to the model will have to go through the Search class. The idea behind this is to make other parts of the program independent of model specific implementations. An interesting detail is that the Search class does not actually contain any methods in itself, but rather functions as a gateway to mediator enums that again functions as gateways to select parts of the model. Since all calls from outside the model was to go through the Search class, it was an obvious choice to make it an enum, automatically gaining the Singleton property.

Class: Search.Data
This class is also a Singleton, whose function is to act as a gateway to what weve chosen to call data specific classes in the model. These classes involve any implementor of the DataSearch interface, and they involve storing/containing the data used by the program. Currently our KDTree is implemented to also function as a container for all our data related objects. The Data class is also an attempt to use the Mediator pattern, and the Adapter pattern.

Class: Search.Text
Acts as a singleton, whose function is to act as a gateway to all text specific classes and data structures. These classes involve any implementor of the TextSearch interface, which is currently only our TST class. The Text class is an attempt to use the Mediator pattern, in addition to the Adapter pattern.

Class: Search.Coordinate
This is a small utility class with the Singleton property. During development, it became clear that having access to the highest and lowest x and y coordinate pairs in our data was useful, and when serialization got introduced to the project, we had to avoid using the KrakLoader class as we had done up to that point, as it would result in parsing unnecessarily from our text files, effectively negating any performance gain coming from serialization. The solution was to add this utility class whose job is to contain the highest and lowest artificial points in our data.

Class: Search.Graph
The last of the nested enums in Search, and also acting as a singleton, it functions as a gateway to all graph related classes. Currently, that is the DiGraph class. The Graph class is once again and attempt at using the Mediator pattern, together with the Adapter pattern.

All the classes Data, Text, and Graph contain the method initialize(), whose function is inspired by the Builder design pattern. It has its flaws in its current implementation, as one can call methods on the classes before calling the initialize() method, which can produce unwanted behaviour. But for our needs, especially with concurrency and loading visuals (see section 11) we found it a necessary evil.

Below is a crude illustration, depicting the Search class as a facade, with the nested enums as mediators:

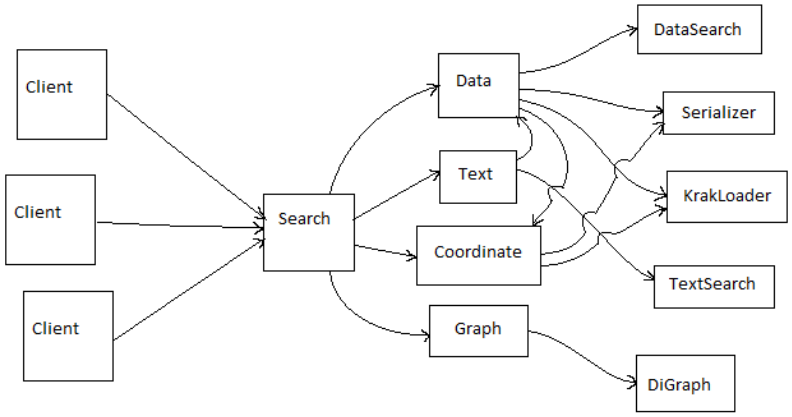


Figure 6: The data flow as we intended it to be

Class: RoadType
This class contains a fixed number of road types that will not change during runtime, it is therefore a good candidate for getting the Singleton property.

Class: KrakLoader
This class is used as an Adapter for the KrakDataLoader class, only implementing the methods needed. It was a good candidate for getting the Singleton property, as KrakDataLoader is our parsing factory (not to be confused with the design pattern Factory), which is not suited for multiple instantiations (which would not make much sense either).

Classes: KDTree, KrakTreeKey, Point, Edge
All these classes are using the Proxy pattern. The reason for this is because of how well the pattern works with serialization (see section 10). So while the classes themselves have no proxy function, they all contain a private nested class called SerializationProxy, this nested class functions as a proxy for the enclosing class during write and read operations. Vastly adding to the security of write and read operations. Using the proxy pattern this way allows immutable fields in the classes to stay immutable, which would not otherwise have been possible (see section 10).

3.4 Discussion of extensions

The last part of our sections covers our choice of extensions. Through the requirements we created in the problem description(see section 2.3) we have selected some extensions that will suit meet these requirements. We will also discuss some alternatives that could fill the requirements just as well.

3.4.1 Address validation

Early in this project's course we were introduced to a task concerning address parsing. We already then experienced the multitude of problems concerning user input validation. We decided then that we needed a good strategy for user input.

A naive implementation would be to design a series of fields where the user could type in address information. Each separated field would constitute a part of an address (for instance Postal Code, Address Name etc.) This would make parsing the address extremely easy. On the other hand, we would have no easy

way to validate the input before the address was final and ready to be used in route planning. This would mean that the user could experience a lot of search misses (not finding the address in the system) before he would find a valid adress. Also this implementation would take up a lot of screen space, because of all the extra search boxes.

A more sophisticated approach would be a search field which validates the input from the user while he types it. Moreover the field could write suggestions to the address in a drop-down list that the user would then be able select. This approach is harder to implement but a lot more sophisticated and its a widely used feature. The approach also gives the user the impression of selecting a correct address every time and it doesn't take up a lot of screen space.

For a detailed description of the validation selection, please review Extension 1: Auto-completion 8 and Extension 2: Postal Map 9.

3.4.2 Faster load times

Another feature we wanted to work on was the improvement of our programs initial load time. Dialogue with other groups, who were working on similar projects showed that many found the load time of their programs bothering and cumbersome. However testing during developing showed that the most time-consuming aspect of our program was parsing the input file we used. Hence we needed to change the way this was done.

At a glance building a database from the input file once and then accessing this every time seemed like a probable solution. This would mean that once the database was created, the information would be accessible with a fast database implementation. Also our data would be accessible anywhere with an internet connection and our client would lose a lot of data weight.

However, this solution would require a considerable amount of time to implement. Each new function of our system that needed access to the data would need a database-specific implementation. Moreover we had no real experience with performance with such a data mass, as the one we were to use in this project.

On the other hand, we had the solution of keeping the data locally but converting it to another, easily readable format. This meant that we could keep the solution local and Java-based, as with everything else in the program. Additionally we had access to design patterns that were relevant from the course literature [1, Chapter 11-12].

For a more detailed description of the serialization extension, please review section 10.

3.4.3 Threading and thread safety

The last attribute we wanted our program to have was threads. Our slow load times made the program look as if it wasnt starting. We wanted our program to launch instantly, when asked to do so by the user. The consequence though, was that the program was not usable until after a certain period after launch time - when the system was fully loaded behind the GUI. We needed to signal the user that the program was loading and had not crashed.

This signal could be a simple text saying Please wait.. until the system was loaded. On the other hand, we wanted to do something gimmicky, and we came up with the loading bar.

But because we wanted to use threads, we also had to make sure that the threads didn't come in the way of each other. This is why we also had to look into thread safety, if we wanted to implement threads.

To summarize, we wanted the program to quickly appear on the screen before the user, while displaying a loading bar. Doing so meant we had to order and change the execution flow of our program. The only possible way of doing this in Java is to use threads. Or at least we havent come up with any alternatives.

But with threads also comes the problem of thread safety. If threads were to be implemented, so was thread safety.

For a more detailed description of the thread implementation, please review section 11.

Our extension planning were based partly on our requirements and also what we thought would be educational and fun to implement. We have developed them with the intention of still allowing the program to be modified through future releases.

4 User manual

4.1 Getting Started

To run the program properly, you need to run the executable jar file with extra heap space (see the README on the CD).

Once the program is running, you should see a loading pie chart, which tracks the loading progress of all modules and additional MapTastic features needing to be initialized.

Figure 7: Loading chart example

4.2 Navigating the map

4.2.1 Panning

Panning the map is done by clicking and holding the left mouse button while dragging the map around.

4.2.2 Zooming

Zooming can be done either by scrolling the mouse wheel or using the bottom panel.

Figure 8: Zooming bar

4.3 Planning Routes

4.3.1 Autocompletion

Our advanced auto completion will provide search matches as you type to ease the input process for you.

Once both fields are filled, you can compute the route by clicking "find route". Keep in mind that you have to fill both fields to be able to use the route planning feature.

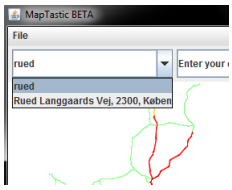


Figure 9: Auto Completion

4.3.2 Visual appearance

When the computing of the route is done the path will be highlighted with a blue line.

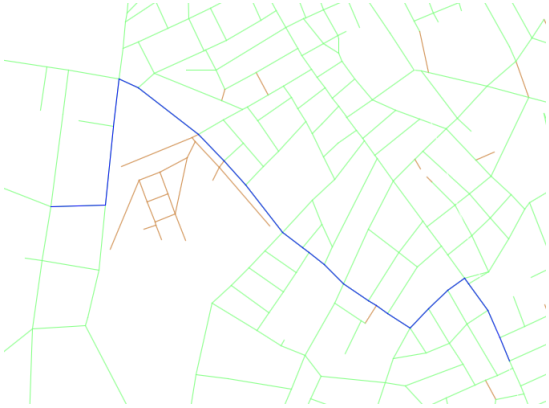


Figure 10: Route visualisation

5 Technical description

The following section is an in-depth description of the internal structure of the program where we will describe important classes and methods. The descriptions will be grouped by program responsibility. First, we will describe the GUI part of the program. Then, we will go through the model, and finally we will describe the controller part as well as the thread classes used to synchronize the system.

The `View` class is the main constituent of the view package and is responsible for building the frame of the GUI and the components. Calling the constructor in `View` initializes all the components, including menus, panels and the map. `View` is responsible for passing information regarding the map to `MapCanvas`, as well as address information to the custom boxes.

`View` implements most of the event handlers used in the system. It listens for scrolling and mouse dragging events. Overall, `View` can be considered as the gateway to the other components in the package, which helps to maintain a reasonable level of encapsulation.

Controller parts of the system can subscribe themselves to the `View` as observers. This obligates the controllers to implement the `CanvasObserver` interface, which we will describe later. This observer design pattern is intended to let other parts of the system listen for user input and handle accordingly. As of now, there is no limit on the amount of observers that can subscribe to `View`, although only one observer should respond to the requests from `View`. The reason for this strategy is to allow logging of activity through a separate controller.

5.0.3 The MapCanvas and Viewport classes

The part of the view that draws the map consists of two classes: `MapCanvas` and `Viewport`.

If we look at the approach used to implement the view, we figuratively create a map, which is stationary. The map in this case is the data structure containing all the edges, and the viewport is essentially a lens

through which we look at the map.

To explain how panning and zooming works, more specific details on the viewport and map are needed. The map is a rectangle, defined by two points M0 and M1 that correspond to the upper left corner and the lower right corner respectively. The viewport is also a rectangle, which is defined two points - V0 and V1 - that corresponds to the upper left and the lower right corners respectively. (See figure 11).

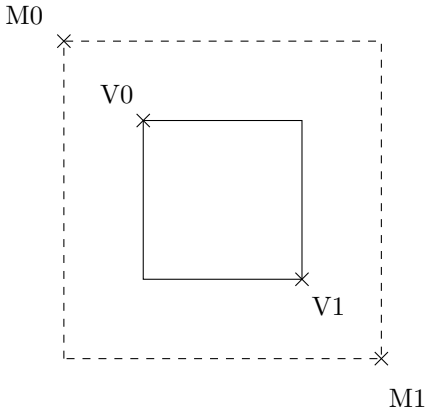


Figure 11: `A graphical representation of the map and the viewport`

What the viewport does when panning around is to move V0 and V1 horizontally or vertically, while maintaining their relative position to each other.
With zooming the two points move a distance away from or towards each other, which means that the number of edges visible within the viewport increase or decrease respectively.

5.1 The model

In this section we will review the implementation of a few of the most important classes in the model package. We will attempt to give the reader an overview of how various technical challenges have been handled.

5.1.1 Implementation of the KDTree

We will now proceed by examining the java implementation of the kd-tree for solving the visualisation task. You will be introduced to the problems encountered during the implementation, and to the solutions and compromises utilised as response to those problems. The implementation you will see here is largely based on the java implementation of a binary search tree found in the book *Algorithms, 4'th Edition*[6] .

Keeping the Invariant
In order to maintain the invariant of the kd-tree, we use a similar approach as Sedgewick and Wayne[6], in that the key is a generic type that implements an interface with a method for comparing the value of the key to another, in order to place each node appropriately in the tree.

Where the key in SedgeWick and Wayne's binary search tree implements the `Comparable<T>` interface provided by the java API, the kd-tree key implements an interface called `KDTComparable<T>`, which among others defines the method; `int compareInDimension(T o, int dimension)`. The kd-tree will then have to rely on the concrete implementation of `KDTComparable` that correctly implements comparing the right values, given an integer representing the dimension.

The implementation of the kd-tree made for this assignment does not support removing keys and values

from the data-structure, as this is not imperative for the functionality of the program given that the KRAK data is static in its nature, that is, it does not change during runtime. So the only method that performs any kind of mutation on the data structure is the `put(K key, V val)` method of the `KDTree` class, shown in appendix A.

In line 29 you will see how the dimension used for comparison is found for this particular depth and in line 30 how the data-structure depends on the `compareInDimension` method of the key.

These two lines in combination ensures that the invariant of the kd-tree is maintained. The method, much like the `compareTo` method of the `Comparable` interface, returns a negative integer if the object that its being compared to is greater than itself, 0 if the two objects are equal and 1 if the object is greater than the object it is being compared to.

You should also note that unlike in a binary search tree, the kd-tree does not replace a value when two keys are equal in the dimension being compared in at a particular depth. Instead, the key is inserted as the right child of the equal key by convention, as the fact that two keys are equal in one dimension does not suggest that they are equal in all other dimensions.

Balancing the Tree

The running time of the `getRange` method is dependant on the depth of the tree, as explained in section 3.2.1. This means that optimizing the time efficiency of the tree is best done by keeping the tree as balanced as possible.

The usual way of doing this is with the tree rotation technique which involves rotating nodes in manner that keeps the tree balanced each time an element is added to the tree. This approach however, is very difficult to implement in a kd-tree, as it can easily break the invariant of the data-structure.

As earlier mentioned, the KRAK data used to draw the map does not change during runtime, and only needs to be loaded when the program starts. Because of this we were able to use the following approach:

1. Load the entire set of KRAK data from the disk, and construct a collection of edge objects containing the data needed by a visualisation client.
2. Sort the entire list with respect to the first dimension.
3. Calculate the mean value of the first dimension.
4. Find the element in the collection with the first dimension value closest to the mean, and add that element to the tree as the root.
5. Sort the sub-collections to the left and right of the mean element in the first dimension with regards to the second dimension, find the mean of those two collections and add the elements with the value the closest to the mean, and add them to the tree as the children of the root.
6. Keep dividing the array, sorting with regards to the dimensions in a cycle, finding the mean, identifying the elements closest to the mean, and add them to tree until all elements have been added to the tree.

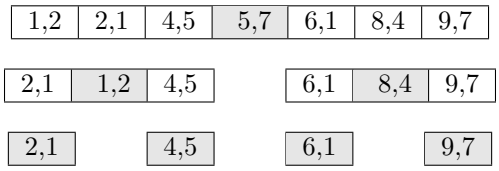


Figure 12: Process of creating a balanced kd-tree

Note that figure 12 does not depict the tree itself, but simply the order of which elements are added to the tree in order to keep it balanced.

Building a balanced tree is a very time consuming process, but as this would only have to be done once, we figured that whether or not investing the time at program startup would be worth it, was dependant on how unbalanced the tree would become without sorting in the different dimensions, finding the mean and so on.

By adding all the edge objects read from the KRAK data file directly to the tree, in the order they were read from the file, and adding code to calculate the depth of the tree, we measured a depth of around 60. This would not be a problem if the main operation of the tree was to search for single nodes. But when applying range searching, which is the primary job of the kd-tree, the search will often hit the maximal depth of the tree, meaning that keeping the tree as balanced as possible, impacts the running time of the range search operation considerably.

To avoid the cost in startup time, the data representation on the disk could be rearranged, so that the edge objects would be read and added to the tree, in an order that would make the tree balanced.

5.2 Graph

The graph is a sub package in the model, this is due to its big role as a stand alone component in the route computing. The package itself is rather small and consists of 3 classes.

5.2.1 DiGraph.java

`DiGraph` is an implementation of the graph structure. The major difference is our custom edge implementation and the inverting of edges because our kd-tree only stored 1 edge per 2 vertices.

5.2.2 DijkstraSP.java

This class contains the algorithm used to convert the data in the directed graph to the shortest path possible between 2 vertices. Since our implementation uses edges as search criteria this means we generate paths between both the start and end vertex and compare them to one another.

5.2.3 Route.java

This is an addition to the general model - especially with the extensions of route description we had in mind, this new class will make implementing that at a later time easier. Its main function is to contain information about the edges that make up the route, from speedlimits to distances.

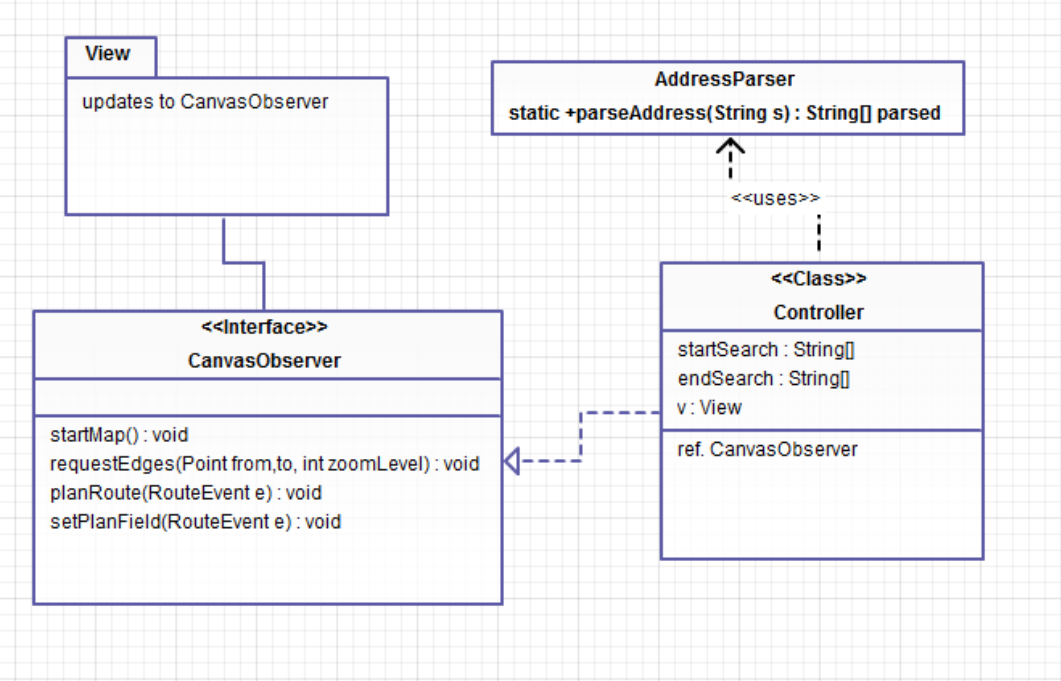


Figure 13: Class diagram of the controller package

5.3 The Controller

The controller package handles input from the view. It is responsible for translating the user input into a request the model can understand.
The typical control flow is as follows: The user inputs text in the search fields. The view notifies the canvas observers registered, which is the controller by default. The controller then parses the input, searching for address patterns, after which it asks the TST for relevant information regarding the address. It then updates the view with the changes from the model. If it is route information, the controller will first query dijkstras algorithm, then update the view with information regarding a highlight of the route.

6 Testing

In this section, the reader will be introduced to our approach for systematically testing crucial parts of the program. We will try to give an introduction to our concrete methods for testing java classes, as well as our philosophy regarding test input.

6.1 Black-box

6.1.1 The model

The approach used for testing correctness of the various classes in the model has primarily been black box testing using the JUnit test framework. The focus of the tests have been identifying bugs in classes handling non-trivial tasks, and to a lesser extent bugs in various helper classes as well as classes handling control flow.

The concrete test cases are based on the black box test philosophy described in a document by Peter Sestoft [7], under which units of the program are exposed to both typical and extreme types of input data. To give an example of the test philosophy, we will now show the testing of the KrakDataLoader Class. The KrakDataLoader class handles the construction of all the basic objects needed in the rest of the program when parsing from txt files. One important part of that task is to construct the KrakTreeKey objects used for correctly being able to search in the primary data-structure used for visualising the map.

The problem which is addressed with the key generation in KrakDataLoader is the problem of searching for lines in two dimensions, which in a geometrical context are defined by their start and end points, that intersect a rectangle requested by a visualization client. The problem occurs when a line is sufficiently long, so that neither the start or endpoint are within the area being drawn. Naturally, the line representing a piece of road should be drawn if it intersects with the requested area, but how will the the data-structure used for searching know that the line is to be returned to the client, when the values used to define the line are outside the requested range?

The solution implemented in KrakDataLoader, is to generate additional keys for lines longer than a certain constant, so that even at a close level of zoom, at least one key will be within the range being requested, and the line will be included in the collection being returned to the client, even though its original starting and ending points are outside the requested rectangle.

The JUnit test case KrakLoaderTests tests this functionality using the black box approach. As input data, a test document mimicking the format of the KRAK data is used. The test input file define 9 two-dimensional lines of different direction and length. The KrakDataLoader is then used to read the file, and produce the number of keys for these lines. The test then asserts that each line can be found by all the keys that KrakDataLoader was expected to produce, using a java standard library hashmap.

The test data contains lines that can be viewed as extreme input data, as they include lines for which the length is defined by only the difference in x-coordinates of the start and end point (horizontal lines), in y-coordinates (vertical lines) and both (diagonal lines). The test data-set also contains lines for which only one key should be generated, as such representing typical input data.

This approach exemplifies the black box testing philosophy, as the tests are carried out without regards to the internal mechanisms of the KrakDataLoader class, but merely tests the overall functionality of the unit, by using typical and extreme input data, as well as expected output data, and lastly comparing the two.

6.1.2 The GUI

We used black-box testing for the view, since that is what makes the most sense. Tests we did was for instance to test the buttons. Did the correct action happen, when the button was clicked? A specific example could be the zoom in button. The input is to click the button, the expected output was that the map did actually zoom in. During the test, we did not look at the implementation.

These kinds of tests of the view included zooming, with and without the buttons, panning the map, starting the map and resizing the map. Also we did use black-box testing for the search menu in the top of the screen and also for route planning, where we checked if searching for a route gave the visible feedback on the map like we wanted.

The black-box test helped us during development of the view to point out where our implementation was missing or faulty. When a function was fully implemented it did not mean that we did not continue testing.

During the whole development process we tested features periodically to ensure that new changes to the program did not ruin the previously working code.

Although black-box testing does not focus on the implementation, you can still easily with some tricks find the position in the code, where a mistake was created. System.out.println is an easy way to do this, but you will have to manually remove them, before delivering the program, because printing to the console is really slow.

6.2 White-box

Although most of the formalised test cases use the black box testing approach, white box testing has been applied to one crucial class; the KD-Tree data-structure. As opposed to black box testing, a white box test case takes into consideration the inner mechanisms of the unit being tested, in order to guarantee that all statements in the code have been run at least once[7]. Ideally a white box test case also makes sure that every possible path in the program is executed at least once. However, as the KD-tree uses recurrence in almost all its methods, and upholds a relatively complex invariant, achieving path coverage in a white box test would be close to impossible. Instead, we try to achieve statement coverage by identifying all conditional statements in the code and create data-set which will cause the program to enter every code block. The coverage table shown below depicts the six conditionals determining the path through the KD-trees getRange method:

Choice	Input propertey	Data-set
1. true	root is null	None (root is null by default)
1. false	root is not null	C
2. true	searchkey < root in dimension	E
2. false	searchkey > root in dimension	E
3. true	searchkey < root in dimension	E
3. false	searchkey < root in dimension	E
4. zero	not possible	4 iterations no matter the data
4. once	not possible	4 iterations no matter the data
4. more	every run	4 iterations no matter the data
5. true	root outside range	E
5. false	root in range	C
6. true	root in range	C
6. false	root outside range	E

The tests are then automated using the JUnit test framework, by creating expected output from the specification, and asserting the output from the unit being tested.

7 Process evaluation

The following section will describe the group's process, while creating the program. Initially, we have placed an overview of who contributed to the different parts of the system.

We will discuss both formal and informal tools used internally in the group to make cooperation easier, as well as some technical solutions used to make programming smoother. Finally, we will reflect on the iterative process and how we can improve our usage of the process.

7.1 Contributor overview

Each point includes the testing that was necessary to implement the parts correctly.

- Controller and Observer interface - Magnus and Phillip

- KD-tree and parsing the Krak Data - Morten and Sune
- Building the Graph - Ivan
- Implementing Dijkstras shortest-path algorithm - Ivan and Magnus
- Implementing GUI - Magnus and Phillip
- Making the Viewport and MapCanvas - Phillip and Magnus (Sune and Ivan helping)
- Implementing auto-completion fields - Magnus
- Building the underlying TST - Morten and Magnus
- Serialization - Morten and Sune and Magnus
- Threads and thread safety - Morten and Phillip

7.2 Group ethics

When forming a group, it is necessary to make some formal arrangements that allows participants of the group to recognize positions regarding when to meet, how to work etc. The formal document we arranged is known as a cooperation agreement and formally describes these rules. For a closer inspection, please review the appendix C.

In the initial stages of our group process, we discussed how our group should work together, what the consequences of breaking our agreements should be etc. But the transition from the informal discussion to a formal document went too slow, and it was never reviewed. It seems clear that a more rigid meeting structure as well as some more project-oriented tools could help make the project from start to end more structured.

Examples of these tools could be: a Gantt diagram, displaying each task in the project as well as each group members progress on their particular assignment; or a formal introduction to each meeting, where the last meeting is reviewed as well as the agenda for the days meeting. The project have shown us the usability of such tools in a clearer way.

7.3 Technical tools

The following is a brief description and discussion of the technical solutions we used to make the programming process easier.

7.3.1 DropBox

All group members used a shared folder on Dropbox to share files that were not a part of the source code. Dropbox is fast, easy-to-use free and can be recommended to everyone.

7.3.2 IDE

When developing a program in Java, we found that when choosing an IDE (Integrated Development Environment), two good options were NetBeans and Eclipse Java Edition. Considering that most of the group used Eclipse already, the version control we were using was integrated in the IDE, and the fact that Eclipse is industry standard, the choice fell on Eclipse.

7.3.3 Version control

A version control system allows programmers to efficiently develop a system, while maintaining a good workflow. We used eGit and a regular Git terminal along with repository hosting at GitHub. However, merging different versions of the projects proved to be quite an advanced problem, a problem which could have been avoided had we gone through a more thorough introduction to Git. A better tutorial or a more elaborative lecture on the terminology used in version controlling could have greatly improved our workflow, when dealing with conflicts in the system.

7.4 Iterative process

We decided early in the process that we would use an iterative process and not the waterfall process when developing the program. This meant that some of the meetings during the project had to be specifically appointed to evaluating components, design decisions, redesigning control flow etc. However, work pressure and general inexperience resulted in a late start using the process. The choice of process also meant that it should be possible to write beta parts of the report early in the process. However, this was also delayed due to the previous reasons.

Overall, we still think that the iterative process was the right choice, but that you need a more explicit way of defining the steps in the process. For example, the group could make an Iterative Day, which is a variant of code-freeze, where no code is allowed to be written, and all decisions are made on a higher level of abstraction. That way, the implementation can be looked at with fresh eyes and big refactoring decisions can be made in the group.

8 Extension 1: address searching and custom GUI box

In this section we will describe and analyze the auto-completion extension we planned and implemented.3.4.1

8.1 Concept

The idea behind the extension is that we wanted a safe way for the user to type in his start position and end position, while route planning. This led to a number of ideas of which the auto-completion box seemed the most ideal. Shown below is a brief overview of the pros and cons of implementing such a box.

Pros	**Cons**
Widely used feature, very user-friendly	A lot of customization required
Input validation	Swing has no native support
(a selected address must exist in the map)	Confusing bug fixing

8.2 Requirements

Creating a search box as the one in the program, which allows the user to type in parts of addresses and retrieve lists of matching addresses that can be used for route planning can be divided into three parts. First, we need to design the box that the user interacts with in a user-friendly and smooth fashion. Second, we need a data-structure, from which to fetch the matching addresses as well as the edges associated with these. And third, we need an address parser which splits the input given by the user into valid data for the data structure to use. To implement this efficiently, we needed to develop some requirements for each part:

Input field
The requirements for a responsive input field were defined as following:

- The user should be able to select from a list of addresses after typing two characters.

- The user should be able to navigate the suggestions in the box with the arrow keys and the mouse.
- The interaction with the box should happen with little to no delay.

Data-structure
The requirements for a fast data-structure were defined as following:

- The requests from the search box should be very fast with no noticeable delay.
- The requests for addresses should support full requests and partial requests.
- The data structure should return the edges associated with the address and be able to filter by them by postal code.

Address Parsing
The requirements for parsing the address were defined as following:

- The address should be parsed into a postal code and a street name.
- The implementation of the address parser should be easily configurable and readable.

8.3 Analyzing requirements and implementation

The above mentioned three parts of the implementation will be discussed in the following: a discussion of the component used to create the custom search field; a theoretical description of the data structure; and a description of the address parser along with a regular expression.

8.3.1 The interface box

The address search box is custom made but is extended from the Java component called JComboBox. The box receives items to show in its dropdown list (suitable address matches) from the controller. The box is designed to make the use of more than one in a gui as easy as possible. It should be noted as well that the box requires updates from the controller every single time a user types a character.

8.3.2 Algorithm analysis

When choosing a data structure to support the interface box, the most important requirement was that the structure should be so flexible as to allow the user to search for only parts of the key (the address). When entering an address into the box, the user should feel that the results retrieved are instant and character-dependent. This leads to a search-based structure, where the internal mapping is shown below:

- **Keys:** address strings.
- **Matches:** if the string matches an address in the dataset or the wildcards have any matches.
- **Values:** The edges of the map that corresponds to the address, filtered or unfiltered (by postal code).

Considering that the keys are strings and only strings, the choice of data structure fell upon a trie, as they only contain strings and have efficient retrieval functions when using prefix or wildcard matches. Analyzing the space and time consumption of the trie structure was central in the decision to use it. However, we present only a simplified discussion in this report due to the complexity of the analysis. The trie builds a tree where nodes are characters contained in the search keys. Each node have a list of references to all characters in the alphabet used in the tree. Hence, a standard unicode node have 256 links. The upside of this data model is that searching for addresses is extremely fast. For example, a tree containing the key

abcdefg with the search abdefg would be a search miss already at the node b. Moreover, a search hit only travels a maximum of search string length + 1.

On the other hand, containing all these null links prove to be extremely time consuming. A theoretical approximation of the number of links we would have in our system is 180+ million. Considering the rest of our application already used a rather large amount of memory, this makes the algorithm inapplicable for our program.

The consequence of the space problem was that we needed a new data structure. We still needed a representation close to the trie, as it allowed key-by-key searching. However we found another closely related data structure called a ternary search trie, discussed in the section below.[6, p. 746]

8.4 Ternary Search Trie (TST)

A ternary search trie is a data structure where each character in the address set is represented as a node in a search tree. The TST is a variety of the traditional trie structure. Each node has three links: the left link of the node contains characters that are less than the node, the middle link contains characters that are part of an address that was previously inserted, and the right links contains characters greater than the node characters.

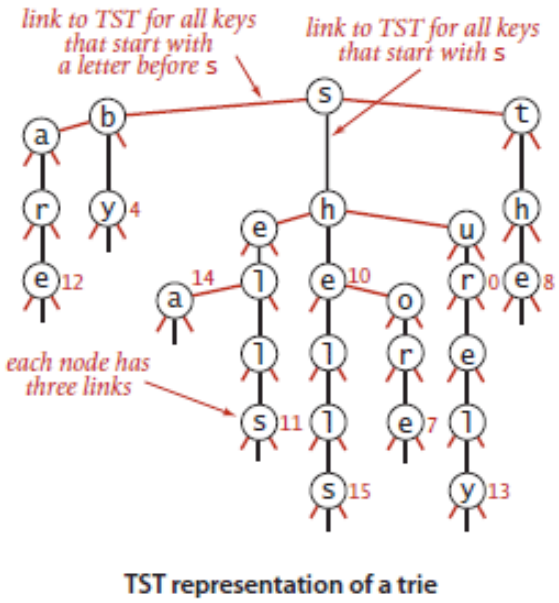


Figure 14: Graphical representation of a TST[6]

Compared to the ordinary trie, the order of growth when searching/inserting is not nearly as fast. However, black-box testing on different laptops showed no visible delay when searching, which is in accordance with our requirements. Hence, the assurance from elsewhere (REFERENCE ALGS 752.)[6, p. 752] that the TST had a reasonable order of growth along with our testing gave us enough reason to make the data structure choice permanent.

8.5 Adress parsing

Since the search box contains a freely editable text field, the need to parse addresses into different segments arose quickly. However, our previous work with parsing addresses was extremely tedious and code-heavy, so we decided to look for an alternative. We found that the most readily accessible API for parsing strings and finding patterns in Java was the Matcher and Pattern classes[4]. These utilize regular expressions, which are flexible ways of expressing a set of characters. In fact, they are so flexible and usable that the only real challenge in implementing a good parser were creating suitable requirements for the expression. Consider the following requirement:

- **Address:** The address must be a maximum of three whole words, contain no numbers and contain all characters in the danish alphabet (not case sensitive).

A regular expression containing this requirement would look something like this:

- `\\b[a-zA-Z]*\\b\\s*[a-zA-Z]*\\s*[a-zA-Z]*`

The escape character denotes word positions, and the characters inside the brackets denote three words containing only danish characters. This is an example of a regular expression used in the code. The advantages of using regular expressions compared to manually analyzing the strings are many. First, the code is very concise and compact. Secondly, the java API is well defined and easy to use. There are also a few disadvantages, though. The API is not very transparent, so weve got no real clue about the algorithms used by Java. This leads to the second disadvantage, being unable to debug anything else than the regular expression themselves.

For more information concering address parsing, please review section3.4.1.

9 Extension 2: postal map

In this section we will discuss a very small extension, however noticeable. Due to the shortness of the extension, we will not divide this section into paragraphs such as concept, implementation etc., but just discuss it briefly.

Due to the working with the search fields, we realized that no input in the Krak data gave associations between postal codes and their district names. However, when choosing an address with many duplicate entries among different postal codes, the user is potentially more comfortable when having a name to associate with the postal code. Therefore, we created a custom postal info document which we build into a map at runtime. The program maps postal codes to their associated names. The class responsible for retrieving suitable addresses for the user to choose from then uses the map to make the proper association.

The postal map was very fast and easy to implement, but it makes a noticeable impact on the address format in the search field.

10 Extension 3: serialization

Serialization was introduced in the project about half-way through. The original argument of serializing the datastructure was to improve the loading time of the program, which at the time was around 40 seconds for an unbalanced datastructure and about 200 seconds for a balanced datastructure.

Balancing the datastructure was not fully functional at the time, but as we were determined to use a fully balanced datastructure in the release, it seemed that we would benefit more from serialization the sooner we implemented it.

The first attempt at implementing the serializable feature was a naive one, simply making all affected objects implement the Serializable interface (including inner classes) using the default serialized form without explicit serial version UIDs. This method was definitely flawed, as it was necessary to delete the serialized files and write them anew with each modification of the affected classes.

It quickly became apparent that using the default serialized form was inadequate, and after consulting various websites and the book Effective Java we realized that our naive implementation had severe security flaws. We also became aware of the exact pros and cons relating to serialization using the default serialized form:

Pros	Cons
Reading from byte streams is faster than reading from text files Portability	Extremely low flexibility No security Not backwards compatible Can cause StackOverflow Can consume excessive space Can consume excessive time Hard to maintain

Continuing with this type of serialization, we would risk making the load time slower than without serialization, which was the main argument for implementing serialization. A further consequence of our naive implementation was that inner classes had to implement the Serializable interface too. This is discouraged, as inner classes contain compiler-generated synthetic fields containing references to the enclosing instances. These fields are unspecified in the class definition, and as such, we would have no way of knowing whether the inner classes behaved as expected or not after deserialization.

After this realisation we had no choice but to refactor all affected classes.

10.1 Instance control

It also turned out, that maintaining the singleton property of many of our model classes would be impossible to guarantee with the way deserialization works, as any readObject method, returns a new instance, which will be different than what was created at class initialization time. Another consequence is related to security of the program:

> If a singleton contains a nontransient object reference field, the contents of this field will be deserialized before the singletons readResolve method is run. This allows a carefully crafted stream to steal a reference to the originally deserialized singleton at the time the contents of the object reference field are deserialized. [1, p. 309]

Fortunately, our class design completely eliminates that possibility, as we use enums wherever theres a need for the singleton property, as encouraged by Effective Java.

10.1.1 Defensive copying and validity check

Making a class serializable, can roughly be considered equal to adding the default constructor. This means that any field can be assigned its default value upon deserialization. To prevent this, the class is required to provide a readObject method that not only deserializes the object, but also overwrites all fields containing object references with new object references, after which invariant checks are performed.

10.1.2 Serialization Proxy

The idea behind this design pattern is that you never write or read an instance of the class you serialize. Instead, you write a nested proxy class containing just enough information to create a new instance of the enclosing class with the exact same object references and field values as it originally had. This means that

the instance itself cannot be modified between reads and writes. As the nested proxy class is private, only the enclosing instance knows of its existence.

Implementing this design, requires a writeReplace method in the enclosing class that should return a reference to the proxy class, effectively writing the proxy instead of an instance of the enclosing class. Upon deserialization, the proxy class readResolve method is invoked, which is used to create and return a faithful instance replica of the enclosing class.

This even ensures that fields can be final in the enclosing instance, which would not have been possible without the proxy due to the defensive copying.

A last method is required in the enclosing class, and that is readObject. This method should simply throw an exception, as the only scenario in which it is called, is if someone attempted to read a serialized instance of the class. Since one such was never written in the first place, and considering that the developer(-s) know this, it would likely be an attempt to destabilize the program. In any case, it greatly adds to the security and reliability of the program.

In fact, it is only at times where the programmer adds a field to the enclosing class, that contains information necessary for the reconstruction of the particular instance of the class, that he is required to modify the serialization proxy. If the enclosing class constructor is modified, the code will not compile until the serialization proxy has been updated. A final note is that all fields in the enclosing class can be declared transient, which means immutable fields can stay immutable, which would not be possible if we used the readObject method for deserialization due to defensive copying.

This practically eliminates all drawbacks of serialization.

10.1.3 Compatibility and speed

Something as simple as adding an explicit serial version UID to each implementing class saves expensive computations with each write operation. And it even makes serialized files with same UID completely compatible. In combination with the serialization proxy pattern, maintaining the program will be just as simple as without serialization, adding more methods, and even changing object references have no impact on compatibility.

10.1.4 Security

If done properly, it will be impossible to break serialized classes invariants, as it is not the classes themselves that are actually serialized. Even if one were able to break whatever invariants the SerializationProxy has, which requires one to know of it in the first place, the deserialization will cast an exception. This is because of the fact that a replica of the original instance is constructed anew by the proxy instance with every deserialization, and because this reconstruction happens in the readResolve method, which is the last step before deserialization is complete.

The only scenario in which invariants can be broken is if the enclosing instances constructor has inadequate validity checking with defensive copying. In that case, the program was already flawed before serialization.

If we take a look at the table regarding pros and cons with this new way of attaining the serializable feature it looks alot better:

Pros	Cons
Reading from byte streams is faster than reading from text files	
Portability	
Flexible	
Backwards compatible	
Secure	
Easy to maintain	

This goes to show that serialization without consideration can hide disastrous security holes and lead to decreased performance. But if done with careful consideration, can lead to increased performance, portability, and with no loss of security.

11 Extension 4: threads and concurrency

11.1 Thread programming

A thread can be compared to a time line that executes code statements. Programming with multiple threads allows for several code statements to be executed in parallel. This can be helpful in many of situations, e.g. when processing a lot of data, while wanting to show a smooth user interface.

As soon as multiple threads are introduced, the concept of thread safety is crucial in order to maintain program integrity. Thread safety will be explained in the following section.

11.2 Thread safety and concurrency

When doing concurrency programming, there are numerous things to take into consideration. If concurrency programming is introduced in a project without careful consideration, unpredictable deadlocks and behavior is guaranteed to follow. These deadlocks and the bugs associated with concurrency are extremely hard to reproduce, as theres no way of knowing which sequence of statements is executed when.

The answer to these problems are locks. When programming with several threads, using the keyword synchronized on a method or object assures that no more than one thread can be inside the method at one time. If the synchronized keyword is used inside a method, it is called a synchronized block. This has the advantage that as soon as a thread reads the statement, it will only enter the method or block if theres no other thread inside. This is useful for getting reliable behavior with shared mutable data.

For an implementation example, see appendix A

The example in above mentioned appendix illustrates the use of object locks. If there were no object locks surrounding the call to the private put method, and there were two threads accessing the method at the same time, it would be possible for the first thread to pause at line 24 in the recursive call, and the next thread could then pause at the exact same line in the recursive call, making it possible for the first thread to proceed and create a new Node object as the right child of node h. After thread one has finished the call, thread two now proceeds and creates a new Node object as the right child of node h, effectively overwriting the old node.

This would have numerous unwanted effects, one being that the object created by thread one have now disappeared, another effect is that the size of the subtree rooted at node h is now 1 unit larger than it should be, as thread one and two both incremented the field containing the size of the tree, while only adding one object. This has now made the size method of the entire tree invalid as it no longer returns the number of nodes in the tree.

It is impossible to say how often this would happen, but even if it happens only once in a million, it could be that the node that was overwritten contained information vital to the proper function of a nuclear reactor, or if all nodes in the tree represented the identities of all the people in Denmark, there would now be a person with no identity, and all persons below node h in the tree would have gotten another identity.

Note that we have an additional synchronized block in the private put method. If this additional synchronized block was left out, and if other methods in the class called the private put method, similar effects would occur.

This goes to show how important thread safety is. In our example method above, the scenarios just explained is impossible, as thread two would have to wait until thread one has exited the synchronized block before proceeding. This time the new object from thread two will be correctly appended to the node from thread one.

Another important part of concurrency programming is to ensure that the latest mutated value is used by every thread. E.g. with an integer field with value 10, if thread one adds the value 100 to the field, and thread two is supposed to add 50 to the field afterwards, theres no way of knowing whether thread two would read the old or the new value of the field. This can happen if thread one pauses after reading the value 10,

then thread two adds 50 to the original value of 10, after which thread one continues and adds 100 to the original value of 10, resulting in 110 instead of 160.

Again, synchronization is the key. If the methods used to access or mutate the integer field are synchronized, or even better, if they contain synchronized blocks using the same object lock, the second thread cannot read the value of the field before the first thread is done mutating it.

The reason why object locks are preferable to synchronized methods is that as soon as a thread enters a synchronized method, it locks the entire instance of the class until it exits the method. But with object locks, as soon as a thread enters a synchronized block, it only locks access to all other blocks synchronizing with the same object. This ensures that methods that have nothing to do with the integer field can still be accessed by other threads while methods that depend on the value of the integer field can only be accessed by one thread at a time, effectively increasing performance and decreasing the risk of deadlocks.

11.3 How to use threads

One of the reasons for us to make threads in the first place wasnt actually to support the program and GUI during the use of the program, but rather during the loading. The loading times of the program are long enough to make a user wonder whether the program is running or not, especially since the user couldnt see if the program had started, before it was finished loading. This made us implement threads.

One way the thread helped us was to make our GUI to start up while loading. The loading would happen in the background in a separate thread instead of in the main one. The GUI now appeared like it should, but the loading times were still a problem, since there was no visualisation to indicate if the program had had frozen or if it actually was processing something.

The solution for this was loading bars. Loading bars are made to show the user how far the program is in the process of an action or to show that the program hasnt frozen. Often this is shown as a percentage of how finished the process is and that can easily be shown with a line, rectangle, circle or any other shape.

We used several threads to create a loading bar that showed both activity and progress. When loading three lines circle around inside a circle to imply that the program is working. Moving the lines is done by a thread that just counts. This count is then added to the lines original positions, which then looks like they move. To show the progress of the loading we made a pie diagram that displays the loading in three different colours according to each part of the program that is loading. When initialising we load/create the KDTree, the TST and the DiGraph. Each of these have a different colour that fill up exactly a third of the circle each. The progress of each part is gathered by a thread, which at certain intervals gathers the progress for each. The representation of the progress works like the lines used for showing action. They all have some start values and the progress is added on top.

At some point we discussed the use of SwingWorker for fetching data from the model. We just didnt get to it due to time constraints and we needed to refactor and change a lot of things in our view classes for it to work. Also it would disrupt the flow of control and the flow of data that we had agreed on. It would have been easier, if the program was built around the use of the SwingWorker, but we didnt think much about threads until later in the process.

The threads in our program allows the GUI to load instantly and and visualise loading to make the user aware that the program is processing. This is meant as a service for the user, since when the program has loaded, we actually dont utilize threads.

12 Conclusion

In this section, we will try to summarize our overall experience of the project. In addition, we will present a list of features that we have worked on, some of which were not ready for implementation at release and some of which could be interesting to implement in future patches. Furthermore we will present a list of known bugs in our initial release.

Through the course of the project, we a gained a deeper understanding of the complexities of carrying out a large scale project. In particular, the features that proved challenging to design and implement, yielded the most valuable experiences. For example, when we were implementing the mathematical parts of the GUI, we found it surprisingly tricky to rid the program of logical errors that did not show up at compile time. We all learned that a large portion of these problems could have been caught earlier if our testing had been more consistent and formalised. In a future project, utilising test driven devopment to a larger extent, would be an interesting experiment.

In addition, the scale of the project called for extensive management of data flow and instance control, in order to avoid an unmaintainable degree of complexity. The approach we used to meet those challenges was utilising design patterns. We learned how relevant design patterns in the source code can signifcantly increase mantainability, structure, and modularity of a program. This goes to show that knowing which combination of design patterns to use, and when to use them, is invaluable knowledge in any large project.

We also learned the importance of carefully planning the interaction between modules in the program, as when we were implementing the graph data-structure, we experienced a severe miscommunication between the graph and the rest of the model, resulting in a lot of wasted time.

Several times during the course of the project, we were faced with making decisions regarding what datastructures to use. It is our assesment, that these challenges have made us more capable in assessing which data-structures are most relevant for the problem at hand.

All in all, we feel that our project has been a success, even though it has been chaotic and unorginazed at times, as we feel that the educational yield has been significant. If nothing else, we have learned the importance of planning ahead, in order to benefit from the iterative process that we had originally planned for.

12.1 Known bugs

- The Reset Zoom button should possibly be named Map Reset for better understanding
- When entering addresses, the search box may for unknown reasons enter an infinite document-listener related loop. There is a current bugfix which resets the box a number of times; however, this is symptom treatment and does not fix the origin of the bug.
- Resizing the window may bug in some situations. When testing extreme use of the resizing, we found that some strange behaviour.

12.2 Future improvements

During the course we thought up several ideas for extension that we abandoned due to either time constraints or difficulty. This is a list of these along with some others we think would fit in nicely with our program:

- Ivan worked with doing a detailed route description in a separate window. However, we ran into time constraints and there was some major bugs. However, the code is still available (out-commented) and could be implemented. Then, a window to display the information in the GUI should be added and the control flow from model to controller should be updated.

- Early in development we added functionality for selecting what transport type one would use for route planning (vehicle or bicycle). The feature was implemented and seemed to work. However, we did not have time to sufficiently test it for correctness. Hence, the feature was removed from the release. This could be a feature thats easy to implement and adds a nice look to the gui (two clickable icons).
- The address parsing is developed so that it only covers the most basic route planning. When planning a route from fx Tagensvej, one could wish to specify which part of the road one wanted to travel from. The feature would be to parse house numbers from input and then use route planning with the selected road-section.
- Our early implementation supported road names that displayed in the same angle as the correspondent roads. However, this feature proved harder to implement with our current map and had to be removed. This could be a very user-friendly feature. Moreover, highways could also display their highway number, which is very useful when driving.
- The last feature we would like to suggest is a Via option for route planning. This would just require a few more calls to our route planning algorithm and a box in the GUI. However, implementing it also requires testing and a thorough analysis of the features correctness, which we did not have time to do.

References

[1] Joshua Bloch. *Effective Java, Second Edition*. Addison-Wesley, Santa Clara, California, 2008.

[2] Danish board of map and land registry. Definiton of system 34. `http://www.kms.dk/Emner/Referencenet/Referencesystemer/System_34/`.

[3] Hemant M. Kakde. Range searching using kd tree. `http://www.cs.fsu.edu/~lifeifei/cis5930/kdtree.pdf`, 2005.

[4] Oracle. Java documentation for pattern class. Technical report, Oracle, 1993.

[5] Rasmus Pagh. First year project: Map of denmark. 2012.

[6] Robert Sedgewick. *Algorithms, Fourth Edition*. Addison-Wesley, Boston, Massachusetts, 2011.

[7] Peter Sestoft. Systematic software testing. 2008.

[8] Wikipedia. Quad tree. `http://en.wikipedia.org/wiki/Quadtree`, 2012.

A `put()` method of the KD-tree

```
public class KDTree<K extends KDTComparable<K>, V> {

    private transient final Object stateLock = new Object();

    ...

    /**
     * Searches for a node equal to searchkey. If found, adds searchkey as the right leaf of that node.
     * If not found, searchkey is added as a node with value val.
     * @param searchkey
     * @param val   Can be null.
     */
    public void put(K searchkey, V val){
        if(searchkey == null) throw new NullPointerException("parameter searchkey was null");
        synchronized(stateLock){
            root = put(root, searchkey, val, 0);
        }
    }

    private Node put(Node h, K searchkey, V val, int depth){
        synchronized(stateLock){
            if(h == null){
                keys.add(searchkey);
                values.add(val);
                // Standard insert
                return new Node(searchkey, val, 1);
            }

            int dim = depth % DIMENSIONS;
            int cmp = searchkey.compareInDimension(h.key, dim);

            if      (cmp < 0)               h.left = put(h.left, searchkey, val, depth+1);
            else if (cmp > 0)               h.right = put(h.right, searchkey, val, depth+1);
            else                            h.right = put(h.right, searchkey, val, depth+1);

            h.n = size(h.left) + size(h.right) + 1;

            return h;
        }
    }
    ...
}
```

Listing 1: `put()` method of the KDTree class

B Log book

Logbog - Projektgruppe 13

Den sidste måned:

Den sidste måned i projektarbejdet bestod af hårdt arbejde alle ugens dage. Vi mødtes alle dage vi havde mulighed for, weekender inkluderet, for at færdiggøre vores program og vores rapport.

Vi havde ikke samme mødeform som tidligere, hvilket betød, at vi valgte ikke at skrive logbog for møderne, da de kom i rap session. Dette betyder dog, at vi ikke har dokumentation for den sidste del af vores process over den sidste måneds tid.

03-05-2012

Ivan, Morten og Phillip havde møde over Skype fra kl. 10.00 til ca. 14.00 Ivan fortsatte på arbejde på Directed Graph, som også er blevet merget ind i Develop-branchen og gerne skulle virke, men skal kobles til resten af programmet. Morten har ændret, hvor TST'et bliver lavet. Der er tilføjet et interface kaldet Progress, som Search.Data og Search.Text, som muliggør at få en progress status i procent og har implementeret koden til det i visse modelklasser. Phillip har arbejdet på visning af loading progress. Status hentes fra modellen og tegnes både som cirkeludsnit og som linjer. Systemet er lavet, så det er let at tilføje nye loadtyper end dem, som allerede eksisterer. Vi har ikke aftalt specifik mødetid, men vi mødes mandag d. 7/5-2012.

30-04-2012

Møde på itu kl 10:00 til ca. 18:00 Magnus samt Sune gik tidligere hhv. ca. 16 og 17 Ivan mødte 10:45 Magnus arbejdede med serialisation af softwaren Phillip arbejdede med tweaks til viewport samt repræsentation af vejnavne på kortet Morten arbejdede med omstrukturering af KD-træet samt ændring af KrakData filen til ny struktur (search.java). Ivan arbejdede med LaTeX rapport samt, fixede tests til Directed Graph træet. Edge-set kan sagtens returneres nu og fungerer med de nye implementationer ydermere blev et ødelagt GIT repository repareret. Sune arbjededede med pan samt zoom funktioner.

28-04-2012

Møde på ITU fra kl. 12.00 til 16.00 Code-freeze! Idag skrives rapport samt iterativ proces. Vi evaluerer komponenter og skriver JavaDoc til koden. Alle er mødt op undtagen Philip, som er på Skype.

25-04-2012

Møde fra 09.00 til 17.00 På grund af krise i ViewPort land, går Sune, Ivan og Philip i bugfixer mode og implementerer kortet samt retter zoom og panning. Magnus og Morten implementerer TST som tillader auto-completion! Vi har

yderligere et par timer til iterativ proces, hvori vi gennemgår komponenter samt hvor tæt de er på at være færdige (+ redesign)

23-04-2012

Møde fra 10.00 til 17.00, Magnus ankommer kl 12.00 Philip arbejder med at implementere den nye ViewPort, som er en refaktorering af logik fra tidl. MapCanvas Magnus bygger GUI og skriver en ny AddressParser med regular expressions Sune laver tests til KD-træet. Philip bugfixer ViewPort Ivan implementerer Graf og begynder på at implementere Dijkstras SPT algoritme

18-04-2012

Møde fra kl 09.00 til kl 17.00 - Titel: "Fuld smæk!" Gruppen har haft et lille wake up call efter visualiseringen! Sune og Morten: Arbejder med at bugfixe KD-træets range check, hvilket viste sig at have inkonsistente Edges. Philip: Arbejder med at finde en bedre løsning på view delen og refaktorering af vigtig logik fra MapCanvas til en separat struktur. Magnus: Arbejder

16-04-2012

Alle undtagen Magnus mødtes efter forelæsningen for at snakke om den nye opgave (resten af projektet) og for at arbejde lidt videre. Demoen af programmet gik nogenlunde set i perspektiv af at vores program reelt ikke virkede. Vi har aftalt at mødes på ITU på onsdag d. 18/4 kl. 9.00 for at komme videre. Desuden har vi sat en deadline for, hvornår visualiseringsdelen skal være færdig. Nemlig søndag d. 22/4.

09-04-2012

Skype møde planlagt til kl 10.00. Ivan er på kl. 12.00 Magnus og Phillip arbejdede på View for at få zoom og panning til at virke. Mere specifikt arbejdede Magnus med at hente de korrekte edges og Phillip arbejdede med reelt at få bevægelsen rundt på kortet til at virke. Morten lavede om, så man søgte på zoom-level i stedet for vejtype.

02-04-2012

Skype møde planlagt til kl 10.00 Der arbejdes videre med implementation af kd-træets samt bugfixing af view. Magnus ordner bagudrettet logbog. Philip (og Ivan?) arbejder videre med at bugfixe view-delen, især problemet med spejlvendt kort: Kortet er nu ikke længere spejlvendt. Der bliver dog stadig kaldt til modellen med alle edges. Morten og Sune har arbejdet med kd-træet: Vi har droppet at sortere dataen og gøre det balanceret da det er unødvendigt i forhold til vores køretid KrakLoader bygger nu et map af edges og KrakTreeKeys KrakData kan nu filtrere i veje efter vejens størrelse Cleanet code i modellen, og slettet unødvendige klasser.

28-03-2012

Møde planlagt til 09:00 - Magnus møder kl 10.00 og Ivan er fraværende Morten og Sune har implementeret et kd-træ som et rødt-sort BST. Arbejdet pr. den 28 går med at implementere metoder, der kan range-query i træet samt behandle de forskellige strukturer, der returneres. Magnus og Philip arbejder med Swing men er på det overordnede niveau bremset af ikke at have testet paint() metoder med det samlede kort. Magnus implementerer zoom metoder, mousescroll metode, refaktorering. Philip udfører design af strokes på veje, samt rotation af vejens tekst. Derudover arbejder begge med at tilpasse SCALE konstanten så den passer til modellens opdateringer. Ivan arbejder hjemme. Ved afslutning af mødes konkluderes: at range query nu virker i kd-træet, og at GUI'en kan tegne hele kortet samt zoome frem og tilbage i kortet. Men kortet er på spejlvendt! Og denne omformning viser sig ikke at være triviel pga g2d.translate

26-03-2012

Sune melder sig syg. I forlængelse af dette aftaler vi at arbejde hjemmefra, da Philip og Magnus kan skype om Swing, og Morten kan arbejde videre alene mht. implementationen af kd-tree. Med gui'en opnås følgende: Endelig en færdig repræsentation af test-input Tekst og farver tilføjes kortet Plan for onsdagen. Mødet næste gang: Troubleshooting af View-del. Implementering af range query. Implementation af basic funktionalitet vedr. GUI.

21-03-2012

Møde planlagt til 9:00 - reelt fremmøde flextid i tidsrummet 9-10 Phillip, Sune, Ivan og Morten mødt indenfor flextid - Magnus planlægger at komme kl. 11 Vi har fortsat arbejdet i de mindre arbejdsgrupper: hhv. swing og model pt. modellen ændres i forhold til implementering af et KD træ. swing udbygges så vi nærmer os vores målsætning med grafisk representation og drag funktion til at vælge et område i java Vi regner alle med at få mere styr på datastrukturer inden weekenden er omme, specielt ved Thores foredrag fredag Mødet næste gang: Implementering af datastruktur API af View-delen Midlertidigt valg af design pattern

19-03-2012

Møde afholdt fra: 12:00 til 16:15 Fremmøde: Magnus, Phillip, Morten, sune, Ivan Det første møde efter Projektets del 1 er frigivet. Overvejelser: swing vs browser JavaScript virker fuldstændig uoverskueligt på nuværende tidspunkt. Men da vi tidligere har arbejdet med Swing vælges dette. Vi er dog ærgelige over at skulle droppe SVG som en konsekvens af dette, men Java har heldigvis også en udemærket vektor grafik. database vs. kd-tree, parsing struktur som udgangspunkt er gruppen positive overfor at oprette en database. men udfordringenen vedrørende kd-træet samt Thore's opfordring til anvende enten dette

eller en gridfile gør vi beslutter os for at undersøge dette nærmere. Det viser sig dog at artiklen omkring grid-filen er ALT for besværlig at læse. Kravspecifikation - anden fil hvor delopgaver er specificeret Vi afholder en introduktion fra Sune omkring LaTeX. nyt møde aftalt onsdag 21/03 - api skal gennemføres

07-03-2012

Mødet begynder klokken 9, Sune er forsinket Tilstedeværende Philip, Ivan, Morten, Sune Arbejder på afleveringen af KRAX formatet, SVG repræsentationen af krak-data og indlæsning af disse i en browser vha. javascript. gennemgår fremlæggelser af Effective Java.

27-02-2012

Mødet påbegyndtes efter forelæsningen som aftalt. Tilstedeværende: Ivan, Sune, Phillip, Morten og Magnus. Overvejelser omkring BigDecimal kontra double værdier - køretid versus målenøjagtighed.

25-02-2012

Mødet påbegyndtes kl 10:00 - over skype Tilstedeværende: Ivan, Sune, Phillip og Morten Vi fik en fejl: OutOfMemoryexception og løste denne. Vi færdiggjorde programmet, så det virker og har udskrevet en outputfil som specificeret i opgaven. Der er ændret en del ift. programmets struktur, vi har tilføjet en del metoder og lavet nogle flere skridt for at sikre at vores output ender i rigtigt format jf. opgavebeskrivelsens specifikation.

27-02-2012

Efter forelæsning mødes vi til “final touch” og afleverer. Skype mødet fortsatte mellem kl. 15:00 - 17:30. Tilstedeværende: Magnus og Morten Her fik vi testet transition mellem doubles og BigDecimal og belyst nogle problemer til næste gang: Der var en fejl på 20000 meter i summen før og efter bearbejdning. Datatyper vedr beregninger? - BigDecimal, sumberegning etc. Vi mangler stadig at erstatte KVD-ID med ID der går fra 1 ¡ nodeMap.length Gør det noget at dubletter (dobbelte kanter repræsenteres i outputtet?).

22-02-2012

Tilstædeværende: Sune, Magnus, Phillip og Morten Ivan udeblev grundet sygdom. Mødet var planlagt til kl. 9, men begyndte reelt først kl. 9.30-10.00. Morten brugte en del tid på at prøve at komme på nettet, så vi kunne kode sammen. Vi andre arbejdede igennem imens. Vi prøvede 2 indgangsvinkler til problemet, og til sidst fandt vi en fungerende rekursiv metode, som kunne finde connections (dele mellem kryds(1 eller mere end to edges)). Dog fungerede den ikke på de store filer, grundet stack overflow. Vi gik i gang med en

tredje metode, doubleHashMap, som ikke fungerer rekursivt, men løber listen af nodes'nes edges igennem. Note omkring den opbygning ligger på google docs: "Ny strategi - Metoder, programopbygning mm" Vi kan derfor kort sige, at vi er i gang med omskrivning nummer 3 af vores kode. Vi ser om vi kan holde et skypemøde lørdag d. 25. feb. og ellers på mandag d. 27. feb. Nærmere tidspunkt planlægges, når vi når nærmere derhen.

20-02-2012

Tilstedeværende: Morten, Sune, Phillip, Magnus og Ivan Mødet begyndte ved 12 tiden Den nye opgave er påbegyndt Stor del af tiden er gået på overvejelser og diskussion af hvordan implementationen skal foregå mht. knudepunkter - en bred enighed om en datamodel med "nodes" og "edges". Morten har kodet en MapReader og Magnus har arbejdet på en testklasse med tilhørende test dataset. Sune samt Phillip arbejdede med at opbygge datastrukturen. Ivan har blandet sig lidt over det hele, hovedsageligt ved Morten og Readeren. Ændringer på datastrukturen: FindConnections skal ændres således edges ligges sammen per connection og ikke ved hvert kryds for at undgå for mange gennemløb ift. rekursion - StackOverflow pt. Næste møde aftalt til kl 9.00 onsdag 22-02-2012

19-02-2012

Skypemøde med følgende tilstedeværende: Morten, Sune, Phillip, Magnus og Ivan. Påbegyndt ca. 13:00 Vi har arbejdet videre på afleveringen. Vi troede ikke der var så meget tilbage, men opdagede en del bugs, som gjorde at vi brugte en del ekstra tid. Det der mangles at blive færdiggjort er: Whiteboxtests af: findHouseNb, findPostalCode Expectancy tables osv. Det bliver færdiggjort i aften og sendt af Sune.

15-02-2012

Tilstedeværende: Morten, Sune, Phillip, Magnus, Ivan. Påbegyndt ca. 9:30 Vi Arbejdede videre med anden aflevering. Vi færdiggør afleveringen på søndag (d. 19.) over skype. Lav findHouseNb om, så den kan skille etager fra husnumre Test alt igennem

13-02-2012

Tilstedeværende: Phillip, Sune, Morten, Ivan. Påbegyndt fra ca. 12:00 Der er begyndt på anden aflevering. Se mere nedenunder: Koden er blevet opdelt til mindre moduler, med tilhørende test cases.

- findStreet() - Phillip
- findHouseNb() - Ivan
- findHouseLetter - Sune

- findFloor - Fælles indsats (planlagt, dato onsdag d. 15-02-2012)
- findPostalCode - Morten
- findCity - Morten

Næste møde afholdes d. 15-02-2012, kl. 9.30. Alle skal læse op på testing (teksten til mandag d. 13. feb) og kigge på expectancy (og coverage) tables.

09-02-2012

Tilstedeværende ved mødet: Phillip Phoelich, Ivan Naumovski samt Morten F. Terkilsen Start tidspunkt ca. 14:00 - med ekstreme programming via. TeamViewer. Der blev udarbejdet ændringer mhb. på at optimere parsingen og enkelte fejl blev elimineret og revurderet. Især "singleton" bogstaver der behandles som del af husnumre. Array til specialtegn er også blevet tilføjet for at sikre at illegale karakterer ikke bruges Næste møde er fastsat til mandag eftermiddag efter forelæsningen. Næste opgave tages op til mødet Slagplan udarbejdes, evt. påbegyndelse på udarbejdelsen. intet hjemmearbejde, dog er overvejelser altid velkomne og tages gerne op i det fælles forum (facebook gruppen)

08-02-2012

Mødet startede omkring kl. 10. Mødetiden havde ikke været ordentligt oplyst her første gang, hvilket betød at ikke alle mødte til tiden. Vi har aftalt en samarbejdsaftale, som også ligger på google docs. Vi har arbejdet på den første opgave til timerne. Næste mødes holdes torsdag 09-02-2012, hvis opgaven ikke er færdig. Morten, Magnus og Sune laver videre på opgaven og ser om de kan nå, at blive færdige.

C Cooperation agreement

1 Cooperation agreement

1.1 Samarbejdsaftale

1.1.1 Mødetider:

- Man skal møde til tiden og for sent er for sent.
- Man skal give besked så tidligt som muligt, hvis man ikke kan komme.
- Hvis man kommer for sent, noteres det i logbogen.

1.1.2 Logbog:

- Der skal være en referent til hvert møde, som efter mødet skriver ned omkring mødet
- Hvad er der sket på mødet
- Hvad er der blevet aftalt
- Hvad har vi lavet
- Hvornår er næste møde
- Hvad skal der laves til næste møde og af hvem

1.1.3 Pauser:

- Pauser tages fælles
- Man byder ind, når der er brug for pause
- Der regnes med en pause på 5-10 min. Hver time

1.1.4 Ambitionsniveau:

- Vi skal have noget ud af kurset, lære det vi skal.

1.1.5 Planlagte møder:

- Vi mødes mandag efter forelæsning og frokost til mellem 16-17 stykker
- Onsdag fra 8 til 16-17 stykker