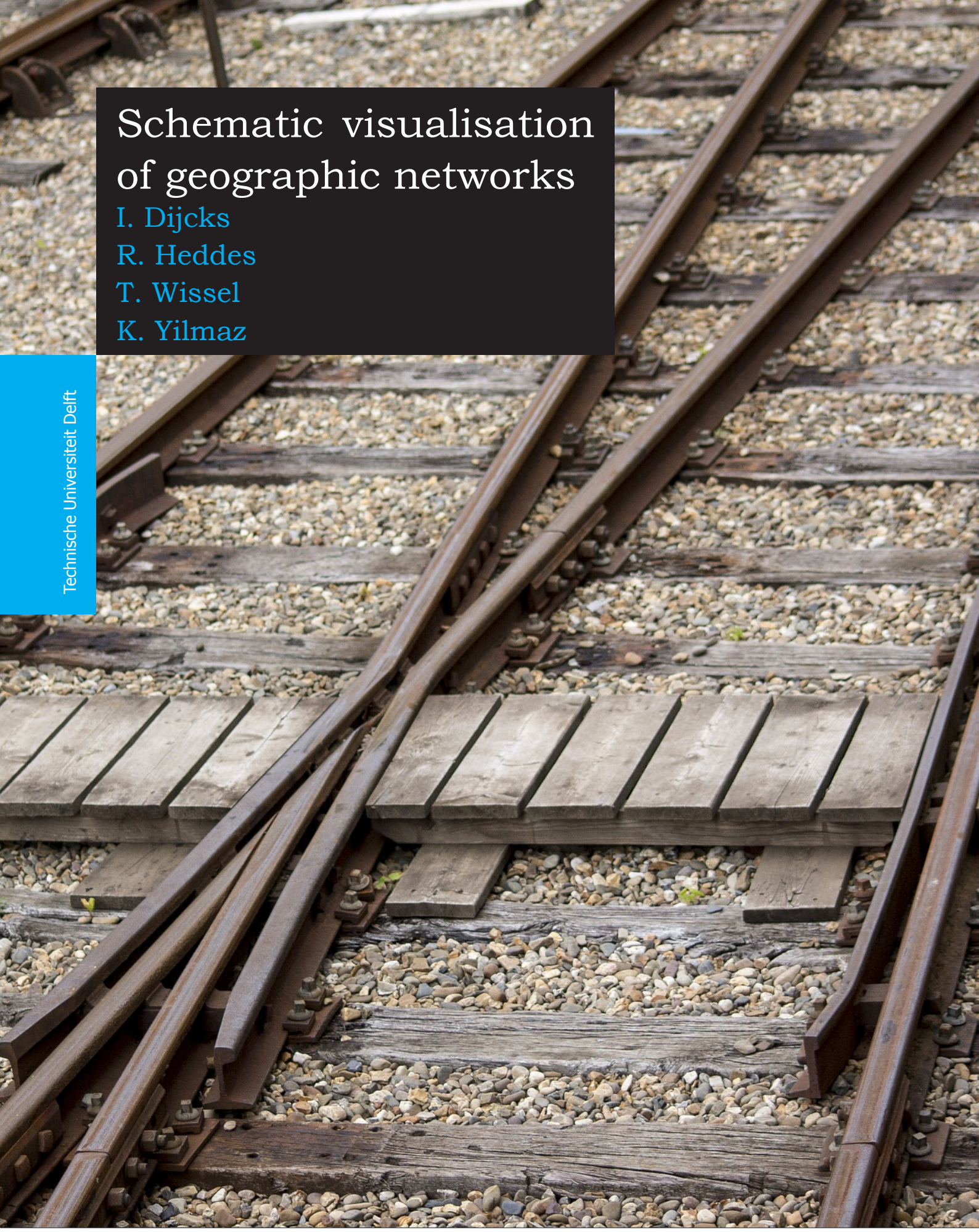
Schematic visualisation of geographic networks

I. Dijcks
R. Heddes
T. Wissel
K. Yilmaz

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

Schematic visualisation of geographic networks

by

I. Dijcks
R. Heddes
T. Wissel
K. Yilmaz

Bachelor's Thesis
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Project duration:	April 2018 – July 2018	
Presentation date:	July 4, 2018	
Coach:	Sohon Roy	TU Delft
Client:	Hylke van der Kolk	Moxio B.V.
	Matthijs Bon	Moxio B.V.
Bachelor Project Coordinators:	Ir. O. W. Visser,	TU Delft
	H. Wang,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Preface

This the Bachelor thesis created by Isha Dijcks, Rens Heddes, Tim Wissel en Kaan Yilmaz as part of the Bachelor Computer Science at the Delft University of Technology. Over the course of eleven weeks we have developed an application that can automatically generate a schematic representation of a geographical network.

We would like to thank our coach Sohon Roy for his guidance and feedback given during the project. He has taught us the value of a well-structured report.

We also want to thank Matthijs Bon who was always available for discussion about problems we experienced. Finally we want to thank Hylke van der Kolk for his advice and realistic views on our planning.

I. Dijcks
R. Heddes
T. Wissel
K. Yilmaz
Delft, June 2018

Summary

The focus of this project is to develop a web application that automates the process of drawing schematic networks from geographical networks. It allows users to upload geographical networks and inspect the schematic representation in the browser.

During the two week research phase we found a Master's Thesis which explains a method for modelling railway tracks and junctions and attempts to draw schematics. We improve upon the findings of this thesis.

We wrote a transformer that can transform real-world GeoJSON data of railway networks to abstract input usable by our algorithms. If our application is to be extended to other infrastructure networks, a different transformer can be implemented while using the same underlying algorithm.

We performed weekly sprints. At the end of each, we presented the improvements to our client to receive feedback. With this feedback we created a sprint plan to assign and prioritise the tasks and responsibilities of the next sprint.

The testing of our application is based on extensive unit tests and end-to-end tests. We evaluated the results of our application and documented recommendations for improving the algorithm.

Our application serves as a proof-of-concept to our client.

Contents

1

Introduction

Moxio[1] is an innovative software company in Delft based near the TU Delft campus. They create high-level software for managing, visualising and validating information for large infrastructural projects. The Spoorzone Delft is an example of one of these projects.

Moxio's clients include large companies which plan and analyse geographical networks such as infrastructure and power networks. For this purpose Moxio has created a tool called Objectbrowser which is a Geographic Information System (GIS) in which these geographical networks are displayed on a map.

When the properties of interest described in the geographic network are distant from each other, it is cumbersome to analyse the network. When focusing on one property, the other more distant properties are not shown on screen, and therefore the overview of the network is lost which makes it difficult to analyse the relation between distant properties. In this case a schematic representation of the network would be more suitable. Since a schematic representation shows only relevant information by emphasising the interconnecting paths and discarding physical details, properties of interests can be displayed in detail while maintaining an overview of the entire network.

These schematic representations are currently drawn manually, which is a costly process. Each time the physical network changes, these schematic representations need to be updated manually. Due to this being such an intensive task this is often neglected, resulting in outdated schematic diagrams.

The purpose of this project is to design and implement an application which can automatically generate a schematic representation from a geographical network. With this application, users can upload files containing geographical network to receive an interactive schematic representation. The main problem we aim to solve during this project is:
To what extent can a geographical network be automatically transformed into a schematic representation, ensuring that its properties are correctly represented?

A set of algorithms which transform geographic network into a schematic representation is presented in the paper *Automatic generation of schematic diagrams of the Dutch railway network* [1]. We build upon the ideas introduced in this paper by improving the algorithms and adding functionality, such as presenting the schematic representation in an interactive web application.

In this report we explain the development of our application in its entirety in the following structure. In Chapter 2 we describe the problem, present the functional and non-functional require-

[1] https://moxio.com

ments, discuss the findings of our research phase including previous work and introduce notation for modelling the network. In Chapter 3 we discuss the planning and the deliverables as well as which methodologies we use within our development team. In Chapter 4 we design the architecture of our application and present all steps of the conceptual algorithm we use to generate schematic representations. Chapter 5 contains the design choices regarding the technical implementation, including its components and their interactions. Chapter 6 describes how we can assure the quality of our code remains high throughout the development process by using automated tests and continuous integration. Chapter 7 contains an evaluation of our output images and to what extend they correspond to our aesthetics criteria and current manually drawn schematic diagrams. In Chapter 8 we reflect back on the project as a whole and give recommendations for future work. Finally we conclude this report in Chapter 9.

2

Research Report

The first two weeks of the project were dedicated to literature research to get a better understanding of the problem and its possible solutions. In this chapter we discuss the main problem and divide it into multiple subproblems which assist in solving the main problem. We explain why we limit our application to Railway networks and give a requirement analysis of the functionality of the application.

2.1. Problem Description

Geographical networks can be difficult to analyse when properties of interest are distant from each other. When focusing on one property, the distant properties are not shown on screen which causes the user to lose the overview of the network. This can be solved by analysing schematic networks which only show relevant information and discard physical details such as distance.

The drawing of schematic representation of geographical network is a time consuming task. Updates to a geographical network are not always reflected in the schematic representation. This can happen because the geographical network and the schematic drawing are duplicate sources of information, so when the geographical network is updated, the schematic would need to be redrawn manually, which can be neglected. The main goal of this project is to develop a web application that can automatically generate and display an interactive schematic representation of a geographical network. Users should be able to upload geographic networks and explore the schematic output.

2.2. Restriction to Railway Networks

The problem posed by Moxio requires the application to make a schematic representation of different types of geographical networks. During the first stages of the research phase we came to the conclusion that making such an application would be too complex for the duration of this project. Extensive research has been conducted on the topic of generating schematic representations of railways. For these two reasons we decided to limit the scope of our project to railway networks.

2.3. Requirement Analysis

To get a better understanding of the main problems, we create requirements for our application. We differentiate between functional requirements, which define features of our application, and non-functional requirements, which describe how these features should be implemented.

2.3.1. Functional Requirements

We divide these requirements according to the MoSCoW method[1] in the categories: Must have, Should have, Could have and Won't have.

Must haves

- The application must be a web-based application, meaning it must be viewable and usable in a web browser.
- It must be possible to upload a geographical network as a geoJSON input file.
- The web application must be able to show a corresponding schematic representation for a geoJSON input file.
- It must be possible to zoom in/out and explore the output graph. This means moving the graph in any direction to view more information.
- It must be possible to view the intermediate steps of the schema generation.

Should haves

- The application should have a set of filters that can be enabled/disabled to obtain a different view of the output. For example show/hide real-world distances of paths as labels.
- All objects should have a tooltip that displays information about them.
- It should be able to support different geographical network types.
- The application should be usable with only a short manual (one page description) by users who have knowledge on the domain.

Could haves

- The application could have interactive methods for doing calculations and visualisations of algorithms applied to the output. For example, two vertices in the graph could be selected, after which a shortest path algorithm is applied. The shortest path algorithm between those edges could then be represented as a set of coloured edges.
- It could be possible to submit a custom asset set using the web application, such that custom assets can be connected to nodes. This way, new network types can be defined by the user.

Won't haves

- The application won't support crossings where more than four tracks intersect.

2.3.2. Non-functional Requirements

We introduce non-functional requirements to deliver a codebase which is easy to extend and maintain such that our client can continue building on the ideas introduced during this project.

- **Maintainability**
  This is an important aspect of the final product in order to build further upon the concepts introduced in this project. Maintainability is helpful towards reaching a comprehensive and extensible code-base. We achieve this by applying static analysis tools (Section 6.3.2) and making sure the code has a high level of cohesion and a low level of coupling. Additionally, human code reviews are performed before every pull request, to make sure the standards of the written code remains high.

[1] Clegg, Dai; Barker, Richard (2004-11-09). Case Method Fast-Track: A RAD Approach. Addison-Wesley. ISBN 978-0-201-62432-8.

- **Usability**
  A crucial aspect is usability, an end user should be able to use the final application with relative ease. To achieve this requirement, weekly feedback sessions with our client will be held and front-end design will be a high priority.

- **Testability**
  An integral goal is to achieve a code-base that contains and supports automated tests. To reach this requirement, test-driven design is applied. Writing failing tests before implementing the functionality results automatically in a testable application in order to make the failing tests pass.

2.4. Subproblems

In this section we use the requirements from the previous section to divide the main problem into manageable subproblems.

2.4.1. How to present the schematic output?

We explored SporenplanOnline which shows schematic views of different railway networks throughout Europe. Unfortunately we do not have permission to use the drawings from SporenPlanOnline, but they are published on their website[2]. As these views are human made they define a format of relevant information to show in the output scheme. From these views we conclude that the following objects are relevant in the schematic representation:

- track switches
- track crosses
- railway signals

From the SporenPlanOnline views and (Battista et al, 2009) [2] we extract the following rules for the aesthetics of the schematic output.

Criteria 1. Avoid edge crossings where tracks do not intersect.

When two tracks are separate in the real world but cross in the output, it gives the impressions that they also cross in the real world. This should be minimised as much as possible to keep the schematic representation reliable.

Criteria 2. Edges should have a maximum of two bends.

To minimise the visual clutter we want to keep the amount of bends as small as possible. With two bends it is still possible to display parallel tracks.

Criteria 3. Keep edge lengths uniform.

Distance does not matter in a schematic representation. To keep the schema evenly spaced we give each edge a uniform distance.

Criteria 4. All angles between tracks should be the same.

For uniformity we set every angle between edges to 45 degrees.

2.4.2. What similar technologies exist?

During our research phase we found a number of papers that use ArcGIS Schematics [3] for drawing schematic diagrams. ArcGIS Schematics is software that can help map (geometric) datasets into schematic diagrams. We studied to what extend ArcGIS Schematics is suitable for (semi-)automatic transformations of a geographical railway network to a schematic network for the Swedish Transport

[2] http://www.sporenplan.nl/

Administration [4].

ArcGIS differentiates diagrams in three different categories: geographical, schematic and geo-schematic (see Figure 2.1). Geo-schematic diagrams use geographic locations with straightened lines.

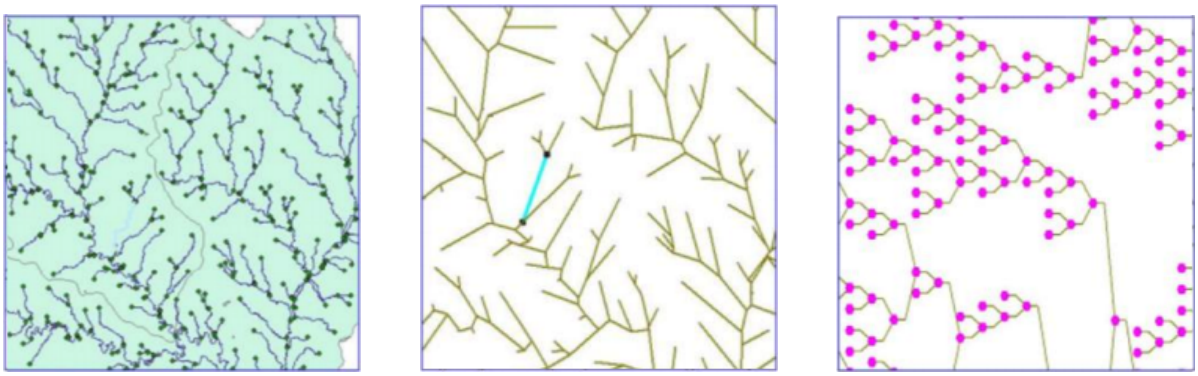


Figure 2.1: Different diagram types in ArcGIS Schematics (geographic, geo-schematic and schematic respectively) [5]

This brings up the question for us whether a schematic or geo-schematic network is fitting. Concluding from the documentation provided by [5], creating a geo-schematic network is very time-consuming because it requires manual editing of the converted schema. Our goal is to use purely automation for schema generation and therefore this method is not feasible.

Many different companies use ArcGIS Schematics, including Dutch and French national railway companies. The French Rail Network (RFF) argued that the algorithm "...does not work on the complex part of their network because the complex part is non-linear, as the routes have many curves, switches, overlapping, etc. and they are not only straight lines." [4]. So according to them, the algorithm is useful, but only for a small part of a network.

The study also concluded that, for it to be useful for the Swedish Transport Administration, new features must be added to keep topological ordering after running the algorithm. Currently, this needs to be done manually which is very time consuming [1].

2.4.3. How can we model a railway network?

A railway network consists of tracks, switches, crossings and stops. In this section we introduce mathematical notation for all objects so we can use them in our pseudocode in Section 4.2

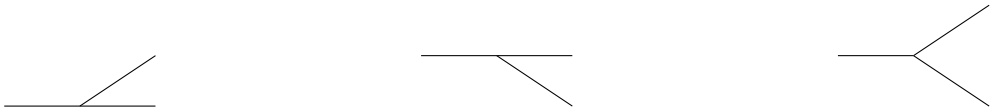
Definition 2.4.1. Track *uv* is a track, such that u is its source junction and v is its target junction.

Definition 2.4.2. *Completely straight track*: Track uv is completely straight if and only if uv is straight out of u and straight into v.

Definition 2.4.3. *Straight line*: Maximal sequence of connected completely straight tracks.

Definition 2.4.4. A *baseline* is a centroidal axis of a geographical network that indicates the main direction of the tracks.

Definition 2.4.5. A *single switch* allows a train to choose between two forward directions. A single switch has one or more diverging directions.



(a) Left diverging (b) Right diverging (c) Right and left diverging

Figure 2.2: Single switches and their diverging directions.

Definition 2.4.6. A *crossing* is an intersection between two tracks where it is not possible to switch directions.

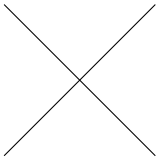


Figure 2.3: Crossing of tracks.

Definition 2.4.7. A *full slip* is a crossing of two straight tracks with additional tracks so the train can go to either direction. A single slip is similar but only has one additional track.

(a) Full slip (b) Single slip

Figure 2.4: Slips and their diverging directions.

Definition 2.4.8. A *stop* marks the end of a track.

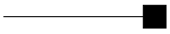
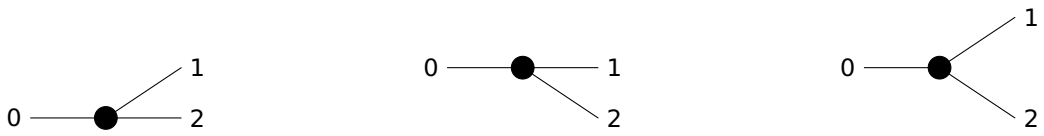


Figure 2.5: A stop.

These components have functional similarities. From now on we will consider them as a *junction*. All tracks belonging to a junction will be given an index starting with 0, and increasing clockwise. The track with index 0 is the one pointing to the left relative to the baseline. The *target index* is defined as the index a track has at its target junction and the *source index* as the index it has at its source junction. This is made clear in Figure 2.6. uv_s and uv_t are used as shortened notation for the respective source and target index of track uv.



(a) (b) (c)

Figure 2.6: Three single switch junctions with adjacent tracks and associated indices.

2.4.4. How can we add additional assets?

After the tracks and junctions of a railway network are generated, additional assets such as signals and railroad crossing should be added to the output. The location of assets are defined geographically in the input. To display the assets at the correct location in the schematic output, the assets

should be matched to the edges of the geographical network. For this matching linear referencing is used.

Linear referencing is a “support system for the storage and maintenance of information on events that occur along (or within) a transportation network” [6]. Before the algorithm is run, the locations of certain assets along edges, like signs, crossings, need to be matched using linear referencing. Then the edges are transformed to get the correct schematic output, afterwards the assets are placed back at the correct relative position to the edge it was mapped with.

2.4.5. How can we extend this model to handle different networks?

As described in Section 2.2 we limit the scope of our algorithm to railway networks. However we define a set of algorithms which map an abstract input to a schematic representation, together with a set of algorithms which map the geographic input from a railway network to the input of these algorithms. By splitting up these responsibilities, this system can be extended to apply to other geographical networks.

2.5. Conclusions of the Research Report

We decide to focus on developing an application that can generate schematic representations of railway networks instead of networks in general.

To aid the development, we have created the following criteria that our algorithm has to follow.

- Avoid edge crossings where tracks do not intersect.
- Edges should have a maximum of two bends.
- Keep edge lengths uniform.
- All angles should be the same.

During the algorithm design we make decisions based on these criteria to make sure they are met in the final output.

There exist technologies which convert geographic networks to a schematic representation. However these technologies are proprietary and therefore we can not build further upon those technologies. Furthermore it does not comply to our aesthetic criteria.

We introduced mathematical notation to model the railway network. This makes it possible to create algorithms operating on this model.

Lastly we defined two parts of the application so that it can be improved to work with different type of geographical networks, while using the same core set of algorithms for generating the schematic representation

3

Process

In this chapter we discuss everything related to the process of our project. More specifically, we present the way we approach the problem and the workflow we use during our project, as well as the responsibilities for each team member.

3.1. Workflow

We work according to the agile methodology, which means we have frequent meetings with our coach and client. We work in weekly sprints. Starting in week 3, immediately after the research phase. Every Monday we have a deadline at 15:00 when our issues have to be finished. After that, at 15:30, we will have a meeting with our client to show the work performed that sprint. Then we ask for feedback and use that as input to plan the issues of the coming sprint. When issues are not finished on time they are evaluated and may be scrapped or carried over to the next week.

At 9:30, every day, the group members have a daily internal stand-up meeting. During this meeting, every member shortly summarises what he has done the previous day, what he is planning to do today and where he might find any difficulties. After this, general points of discussion are tackled. We log notes of this meeting on Google Drive[1] and add tasks to Trello[2].

Every friday afternoon, we create an overview of announcements and points to discuss during the weekly meeting on monday 15:30. On friday we send this overview (document) to the client and TU coach.

3.2. Roadmap

3.2.1. Deliverables

Week 1	*25/04*	Project Plan
Week 2	*04/05*	Research Report
Week 6	*01/06*	SIG evaluation 1
Week 9	*22/06*	SIG evaluation 2
Week 10	*27/06*	Final Report
Week 11	*04/07*	Presentation

[1] https://drive.google.com/
[2] https://trello.com

3.2.2. Research Phase

Week 1 Creating a Project Plan and forming research questions.
Week 2 Finish the research report and final research questions.

3.2.3. Implementation Phase

Week 3-9 Weekly Sprints carried out as described in Section 3.1.

3.2.4. Report Phase

Week 9 Final SIG review and first draft of Final Report.
Week 10 Finish final report and implement SIG feedback.
Week 11 Presentation.

3.3. Responsibilities

We assigned each member a list of tasks that they are responsible for. This does not mean that they are the only member working on that task, but they have the responsibility to make sure it gets done.

Team leader	*Isha*	Communicating with Moxio and keeping the team on schedule.
Secretary	*Tim*	Taking notes during daily standups and updating Trello.
Web Application Frontend	*Rens*	Responsible for creating the interactive viewer.
Web Application Backend	*Kaan*	Responsible for the server that runs our algorithm.
Algorithm	*Isha*	Designing and implementing the main algorithms.
Scrum master	*Tim*	Making sure we follow the correct SCRUM principles and we stay Agile.
Report	*Rens*	Making sure the final report is kept up to date throughout the project.
Testing	*Tim*	Providing proper testing and CI tools to maintain code quality.

3.4. Division of Tasks

By the use of an integration between Trello and GitHub issues, tasks are divided over the members of the group. Furthermore, the expected workload (in terms of hours) is discussed per task in the daily stand-up meeting. This is then registered per task on Trello. This way the workload can be evenly spread across the group. If a group member realises that a task significantly differs from the estimated workload. The task can be tweaked or separated in to smaller tasks during the daily stand-up meeting to spread the total workload as equal as possible.

3.5. Tools

See 3.1 for an overview of tools we use during our project.

Table 3.1: Overview of tools.

Tool	Description
Slack	Communication between team members
ShareLatex	Collaboration for writing reports
Git	Version control system
GitHub	Reviewing pull requests and tracking issues
Travis CI	Continuous integration to ensure performance across all systems
Better Code Hub	Automated feedback for software quality by the Software Improvement Group
Trello	Management of tasks, workload and responsibilities
Google Drive	Preparation of meetings, minutes of meetings and preparation of presentations

4

Design

In this chapter we describe the architecture of our application and we define a set of algorithms that can transform the geographic input data to a schematic representation which can be visualised in the web browser of the user.

4.1. Architecture

In this section we briefly explain the architecture of our project. One of the requirements described in Section 2.3 is that the application must be a web application. In Figure 4.1 we outline the architecture of the complete web application.

First the user selects the files containing the geographical network from which a schematic representation should be extracted. These files are then uploaded to the server. The files are read by the transformer which generates abstract input to be used by the algorithms which generate the schematic representation. After the schematic representation is generated it is send to the browser of the client which displays it so the user can interact with it.

On the server side we differentiate between transformer and schema generator, which makes it extensible to implement and use different transformers for other types of geographical networks as described in Section 2.4.5.

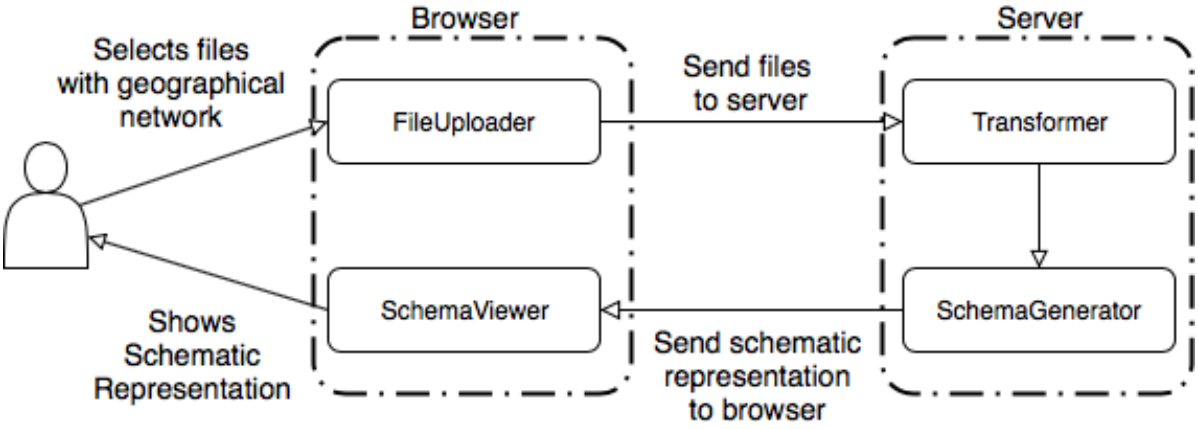


Figure 4.1: General overview of our application

4.1.1. Client and Server Communication

The client communicates with the server through a Remote Procedure Call (RPC). An RPC makes it possible to execute a procedure on a remote computer. This remote computer is our server and the procedure is the transformer and schema generator. The client has access to the API endpoints on the server. We specify one API endpoint, which is the entry to our application. The required parameters for the API endpoint are the files containing the geographical network. The output is the schematic representation.

Our motivation behind choosing RPC is that the clients only desire to obtain a schematic representation of their geographical data. Since the transformation and schema generation procedures are on the server, a remote procedure call is required.

Another common client and server communication architecture is Representational state transfer (REST). REST architectures are generally more suited for manipulating persisted data, by using Create, Read, Update and Delete (CRUD) operations. We do not persist or modify data, thus RPC is better suited for our application than REST.

4.1.2. Separation of transformer and schema generator

We separate the transformer and the schema generator into their own independent modules. A condition for the output of the transformer, is that its data structure is the same as the data structure that the schema generator expects. As a result of this, a schematic representation can be generated by other geographical network data, as long as a user is able to create a valid transformer. This is a functional requirement defined in Section 2.3.1.

We create our own transformer in the application for geographical railway data. We explain this in more detail in Section 4.2.1.

4.2. Algorithms

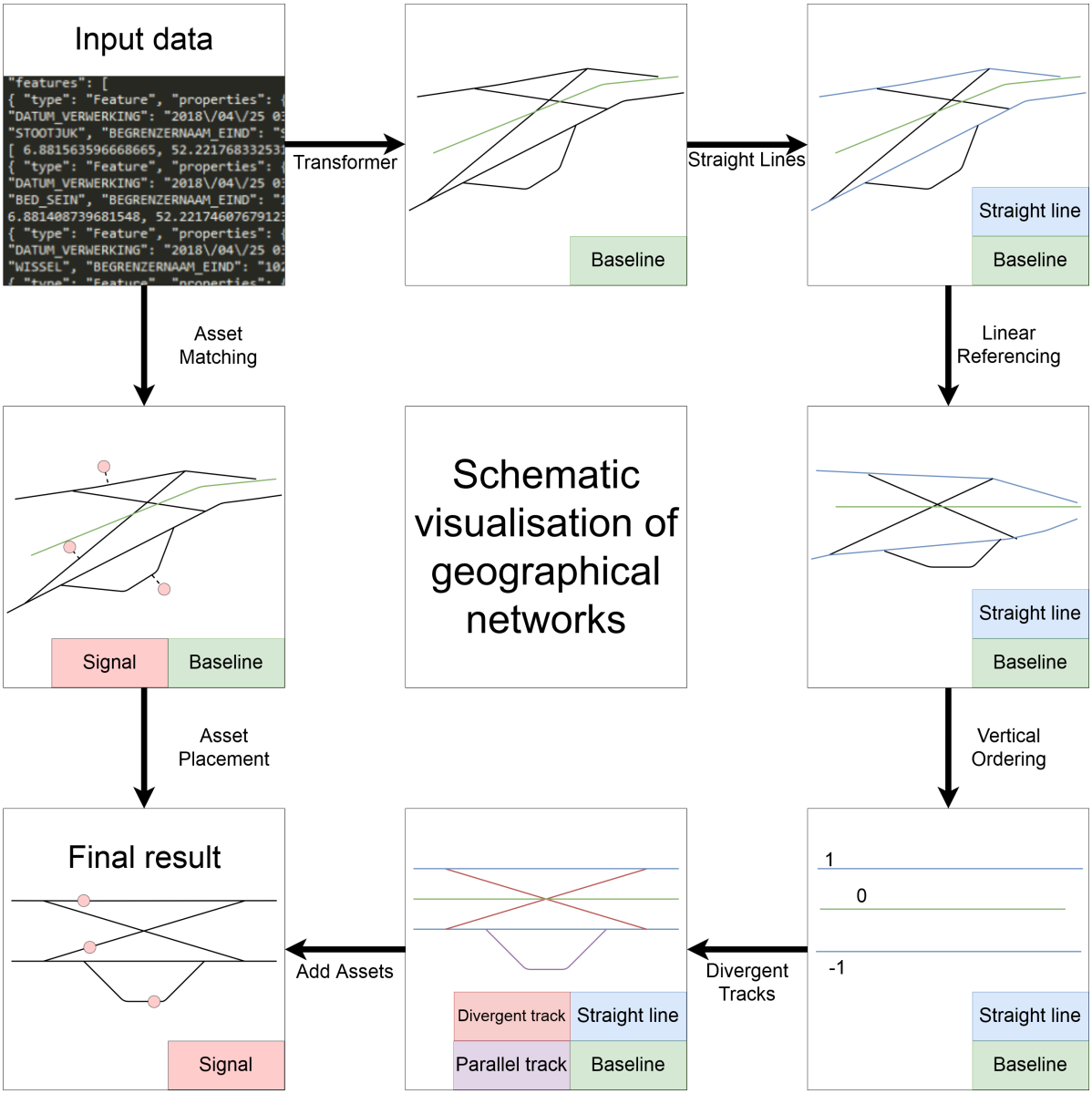


Figure 4.2: General overview of the algorithms

In this section, we present an overview of the algorithms we use. In the subsequent sections, we explain every algorithm in detail.

As described in Section 4.1, the schema generator requires appropriate data structures as input. The task of the transformer is to create these data structures from the data uploaded by the user. Afterwards, the schema generator creates the schematic representation. We describe the algorithms which form the schema generator below.

The first step of the schema generation is the straight line algorithm. This creates a distinction between straight lines and divergent tracks. A definition for a straight line is given in Definition 2.4.3. After this distinction is made, linear referencing is applied to all tracks, which requires a baseline.

This baseline can either be provided by the user, or the longest straight line from the first step

is used. After linear referencing, the direction of tracks is changed, such that the source junction is to the left of the target junction.
Then, the vertical ordering algorithm decides the vertical order of every line.
Next, the horizontal scaling algorithm distributes the junctions evenly on the horizontal axis.
In the last step, the divergent tracks algorithm changes the divergent tracks, such that it meets the aesthetic requirements defined in Section 2.4.1.
See Figure 4.2 for a visual overview of every algorithm step.

4.2.1. Transformer

This component has the purpose of reading geographical data and transforming this into the data structure described in Section 5.2.2.

Location Matching

The input for the transformer is a dataset consisting of objects mapped to real-world locations. The Tracks and Junctions described in Section 5.2.2 have relations that represent them having overlapping locations. These relations are not explicitly given as input data, therefore we create a location matching algorithm. The input format for junctions contain 3 locations: one incoming and two outgoing locations. These are called a, b and c corresponding to the order in which they are given. Tracks have 2 locations, called u and v, namely their source and target locations. Algorithm 1 shows the location matching algorithm design with the previously described data input.

Algorithm 1: Calculate the relations between tracks and junctions

Data: A set T containing track locations and a set J containing junction locations. p a precision constant.

Result: Tracks and Junctions in data format described in Section 5.2.2.

```
for j ← {a, b, c} ∈ J do
    for t ← {u, v} ∈ T do
        if (euclidean distance from u to j_a OR from u to j_b OR from u to j_c) < p then
            t_source ← j
        if (euclidean distance from v to j_a OR from v to j_b OR from v to j_c) < p then
            t_target ← j
    end
end
return T, J
```

Algorithm 1: Algorithm for matching junctions to tracks

Calculating Junction Types

The divergent direction of the junction is also not explicitly stated in the input data. For each junction, three locations are given: the three endpoints which connect to corresponding tracks, where point b is from the incoming track and a and c from the outgoing tracks, and one midpoint m, which is the point where the tracks diverge in the real world. We design an algorithm to use these four locations and extract which one is the divergent track. We do this by calculating whether the m is either on line ab or on line bc, by using the cross product between point m and lines ab and bc. If m is on line ab, the junction is right-divergent and if m is on line bc, the junction is left-divergent.

4.2.2. Track Types

First, all tracks are given three more properties: $type_{src}$, $type_{target}$, and $type$. $type_{src}$ and $type_{target}$ can have one of two values. 0 if it is straight in/out of its target/source junction, and 1 if it is divergent. The type is then calculated by the following formula:

$$type = 2 \cdot type_{src} + type_{target}$$

$type$ can have four values.

0 the track is straight-straight.
1 the track is straight-divergent.
2 the track is divergent-straight.
3 the track is divergent-divergent.

In Algorithm 2 and Algorithm 3 is described how types are extracted from each track.

Algorithm 2: Calculate the source_type of track uv

Data: A set T containing all tracks and a set J containing all junctions. Track $uv = \{u, v\} \in T$
Result: The source_type of track uv

if u *is of type single switch* **then**
  if $uv_s = 0$ *OR* $(uv_s = 1$ *AND* $u_{div} = right)$ *OR* $(uv_s = 2$ *AND* $u_{div} = left)$ **then**
    $uv_{type\_s} \leftarrow 0$
  else
    $uv_{type\_s} \leftarrow 1$
  end
else if u *is of type crossing* **then**
  find track $xu \leftarrow \{x, u\}$ where $xu_t = (uv_s + 2)\%4$
  $uv_{type\_s} \leftarrow Algorithm1(T, J, xu)$
else if u *is of type stop* **then**
  $uv_{type\_s} \leftarrow 0$
end
return uv_{type_s}

Algorithm 3: Calculate the target_type of track uv

Data: A set T containing all tracks and a set J containing all junctions. Track $uv = \{u, v\} \in T$
Result: The target_type of track uv

if v *is of type single switch* **then**
  if $uv_t = 0$ *OR* $(uv_t = 1$ *AND* $v_{div} = right)$ *OR* $(uv_t = 2$ *AND* $v_{div} = left)$ **then**
    $uv_{type\_t} \leftarrow 0$
  else
    $uv_{type\_t} \leftarrow 1$
  end
else if v *is of type crossing* **then**
  find track $vx \leftarrow \{v, x\}$ where $vx_s = (uv_t + 2)\%4$
  $uv_{type\_s} \leftarrow Algorithm2(T, J, vx)$
else if v *is of type stop* **then**
  $uv_{type\_t} \leftarrow 0$
end
$uv_{type} \leftarrow 2 \cdot uv_{type\_s} + uv_{type\_t}$
return uv_{type_t}

4.2.3. Straight Lines

A straight line is a sequence of straight-straight tracks with total length L, such that a track at position $i < L$ in the sequence, has a target junction that is the source junction of the track at position $i + 1$ in the sequence.

We find straight lines by starting from a straight-straight track t. Then we look for another straight-straight track that has a source junction equal to the target junction of t. If it exists, we add it to the sequence and repeat the process. If it does not exist we stop and go in the other direction. We look for a straight-straight track that has a target junction equal to the source junction of t and repeat

that until no more straight-straight tracks can be found. Algorithm 4 contains the pseudocode for obtaining straight lines.

Algorithm 4: Obtain all straight lines from tracks

Data: Set T containing all tracks
Result: Set Q containing all straight lines

```
1  Q = ∅
2  for uv ∈ T do
3      if uv_type ≠ 0 then
4          T = T \ {uv}
5          continue
6      end
7      M = ∅
8      R = {vw|vw ∈ T, vw_type = 0}
9      while |R| > 0 do
10         M = M ∪ R
11         T = T \ R
12         R = {wr|wr ∈ T, wr_type = 0}
13     end
14     L = {tu|tu ∈ T, tu_type = 0}
15     while |L| > 0 do
16         M = M ∪ L
17         T = T \ L
18         L = {st|st ∈ T, st_type = 0}
19     end
20     Q = Q ∪ {M}
21 end
```

Algorithm 4: Obtain all straight lines from tracks

4.2.4. Linear Referencing

Since we want to draw all straight lines horizontally, every straight line needs to have an y-coordinate in the final output. Just using the y-coordinate of the straight lines does not suffice, as this will not accurately represent the main direction the tracks are going. An example of this is shown in Fig. 4.3 where cd is the baseline of the network. If just the y-coordinates would be taken to create a vertical order, then track ef would be above ab. This is the wrong solution as ab is above the baseline and ef below. In order to solve the above mentioned issue, we apply linear referencing. The goal of linear referencing is to find the appropriate positions of every junction with regard to the baseline.

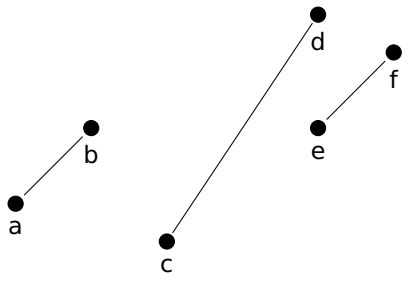


Figure 4.3: Straight lines with baseline cd.

During linear referencing we consider each straight line to be a polyline. We map every point in a polyline to the baseline. For each point we calculate a projection to the closest point of the baseline. A visualisation of this projection is given in Figure 4.4. *ABCD* is the baseline and *EF* is the polyline that we map. The mapped locations are *G* and *H*. The distance between the point on the polyline

and the point on the baseline is stored.

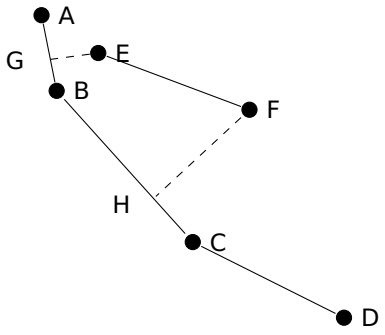


Figure 4.4: Straight line mapped to the baseline.

When we have calculated this point and distance for each polyline, we consider the x-axis to be our new baseline. We place the junctions in a 90 degree angle from the baseline at the projected point and the distance we have previously calculated. Figure 4.5

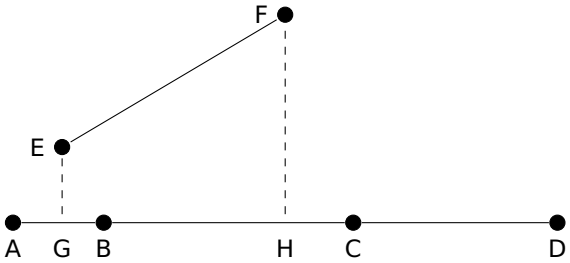


Figure 4.5: The network after linear referencing.

The pseudocode for linear referencing is written in Algorithm 5, Algorithm 6, Algorithm 7 and Algorithm 8.

Algorithm 5: Linear referencing

Data: Polyline $T = \{t_1, t_2, \dots, t_n\}$, baseline $B = \{b_1, b_2, \dots, b_m\}$
Result: Polyline Q, that is linearly referenced from T

```
1 B ← baseline_mapping(T, B)
2 B ← baseline_straightening(B)
3 B ←B.rotate_to_horizontal()
4 Q ← obtain_points(B)
5 return Q
```

Algorithm 5: Linear referencing algorithm

4.2.5. Modify Direction of Tracks

As every track begins in a source junction and ends in a target junction, it is more convenient if all tracks are placed in the same direction. That means that for every track $\{u, v\}$ where $v_x < u_x$, we swap the direction of the track so that it becomes $\{v, u\}$.

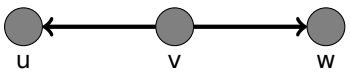


Figure 4.6: Original data where the track goes right to left.

Algorithm 6: Baseline mapping: $B \leftarrow$ *baseline_mapping(T, B)*

Data: Polyline $T = \{t_1, t_2, \dots, t_n\}$, baseline $B = \{b_1, b_2, \dots, b_m\}$
Result: Baseline B, with T mapped

for $t \in T$ **do**
  $d_{min} \leftarrow \infty$
  $L_s \leftarrow \emptyset$
  $k \leftarrow 0$
  for i *is* 1 *to* $(m-1)$ **do**
    $b_s \leftarrow b_i \in B$
    $b_t \leftarrow b_{i+1} \in B$
    $L \leftarrow \{b_s, b_t\}$
    $d \leftarrow$ *distance(t, L)*
    if $d_{min} > d$ **then**
      $d_{min} \leftarrow d$
      $L_s \leftarrow L$
      $k \leftarrow i$
    end
  end
  $W \leftarrow$ line with start point t, length d_{min} and touches L_s
  $p_e \leftarrow$*W.endpoint*
  $p_{e_v} \leftarrow < t_x - p_{e_x}, t_y - p_{e_y} >$
  $B \leftarrow \{b_1, b_2, \dots, b_k\} \cup \{p_e\} \cup \{b_{k+1}, b_{k+2}, \dots, b_m\}$
  $m \leftarrow m + 1$
end
return B

Algorithm 6: Map polyline to baseline algorithm

Algorithm 7: Baseline straightening: $B \leftarrow$ *baseline_straightening(B)*

Data: Baseline $B = \{b_1, b_2, \dots, b_m\}$
Result: Straightened baseline B

$L_B \leftarrow$ length of B
$\vec{v} \leftarrow < b_{m_x} - b_{1_x}, b_{m_y} - b_{1_y} >$
for $i = 2$ *to* m **do**
  $L \leftarrow$ euclidean distance from b_1 to b_i
  $p \leftarrow L / L_B$
  $b_i \leftarrow b_0 + p \cdot \vec{v}$
end
return B

Algorithm 7: Straighten baseline algorithm

Algorithm 8: Obtaining transformed polyline from baseline: $Q \leftarrow$ *obtain_points(B)*

Data: Baseline $B = \{b_1, b_2, \ldots, b_m\}$

Result: Polyline Q, that is linearly referenced

```
1  {v1, v2} ← B.get_orthogonal_vectors()
2  Q ← ∅
3  for t ∈ B do
4      if is_polyline_location(t) then
5          θ1 ← v1.get_angle(tv)
6          θ2 ← v2.get_angle(tv)
7          v ← ∅
8          if θ1 > θ2 then
9              v ← v2
10         end
11         else
12             v ← v1
13         end
14         r ← t + v
15         Q ← Q ∪ r
16     end
17 end
18 return Q
```

Algorithm 8: Obtaining transformed polyline from straightened baseline algorithm.

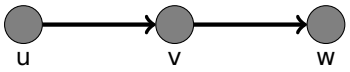


Figure 4.7: Data after modification.

4.2.6. Vertical Ordering

Once the direction of all tracks is uniform, the tracks need to be ordered vertically. As not every junction is located on a straight line, some of the 'lines' consist of a single junction. We consider the baseline as a straight line and give it a y-value of 0. To determine the vertical order we loop through all pairs of straight lines to infer a relation between them.

We decide on this relation based on two rules. The first rule checks if there is a single switch that connects the two straight lines. If that is the case we take the divergent direction of this switch to decide which line is above the other. If line A for example has a right-diverging track that goes to line B, this must mean that line A is above line B. The second rule checks if the start and endpoint of a line are above the start and endpoint of another line. This means that a line is completely above the other. If neither of these cases if applicable we can not infer a relation between the two lines.

We apply an algorithm for topological sorting first described in (Kahn, 1962)[7]. This algorithm works on graphs so we transform our relations to a graph where the straight lines become nodes and the relations become edges. If line A is above line B it means we draw an edge from node A to node B. This topological sorting algorithm generates an order of the nodes such that for every edge uv node u comes before node v. We apply this ordering to our straight lines to determine the final vertical order.

4.2.7. Horizontal scaling

One of the final steps consists of normalising horizontal distances between junctions.

In a schematic representation of the data, real-world distances do not matter. Therefore, to im-

prove readability, we create a visualisation with equal spacing between subsequent junctions.

We take all schema-coordinates from junctions and order them horizontally. Then, we define a standard offset as distance between each subsequent junction. This results in a uniform distribution of junction locations over the horizontal axis.

To give a more aesthetically pleasing output, vertically aligned junctions are placed on the same x-coordinate. This means that junctions that are close to each other by their x-value, get assigned the same x-value. We determine closeness by using standard interval. This interval describes the allowed mismatch in x-value the junctions can have after linear referencing, to still be placed on the same x-value.

4.2.8. Divergent Tracks

Divergent tracks are tracks that are not straight and thus, not drawn as a horizontal line. In Section 2.4.1, we stated that all angles need to be 45 degrees. This means that divergent tracks must have a 45 degree angle from the horizontal axis. In order to achieve this criteria, we define two different groups of divergent tracks. The first group is divergent-divergent tracks. The second group is the non-divergent-divergent tracks. The algorithm can draw the second group of tracks in four different ways, depending on certain conditions. These conditions are summarised in Table 4.1 and the these tracks are visualised in Figure 4.8, where u is the source, v is the target and r is a bending point.

Table 4.1: Condition table of non-divergent-divergent tracks.

	Straight-divergent track type	Divergent-straight track type
Target above source	**Right-up divergent track**	**Up-right divergent track**
Target below source	**Right-down divergent track**	**Down-right divergent track**

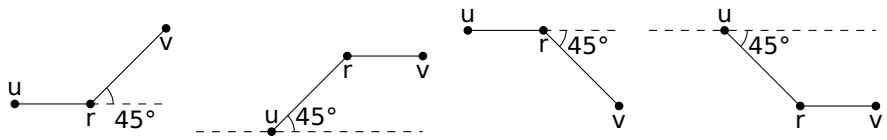


Figure 4.8: Right-up, up-right, right-down and down-right divergent tracks

The group of divergent-divergent tracks consist of parallel tracks and diagonal tracks. A divergent-divergent track is a parallel track if its source and target are on the same level, otherwise its a diagonal track. See Figure 4.9 for a visualisation of parallel track uv, that starts on u, goes to a, b and lastly v. We draw a diagonal track uv directly from u to v, without any intermediate bends.

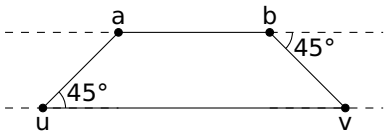


Figure 4.9: Parallel track

A problem that arises when creating a diagonal track uv, is that junction u or v might need to be moved in order to obtain a 45 degree angle. A possible consequence is that the topological order of the network becomes invalid. In order to maintain the correct topology, every junction that is to the right of the previous location of the repositioned junction, is moved to the right the same distance as the repositioned junction.

To reduce the complexity of this problem, we integrate a few rules into the divergent track algorithm:

1. Junctions are only allowed to be moved to the right.
2. It is preferable to move target junctions.

In the first step, the divergent track algorithm focuses on the junctions of diagonal tracks. All divergent tracks are sorted by their source x-value to an ascending order. Then for every divergent track that is classified as a diagonal track, the algorithm check if its target junction can be moved to the right, such that it creates a 45 degree angle with the horizontal axis. If this is the case, then the target junction is set as the selected junction, which is the junction that the algorithm will move. Otherwise the selected junction is the source junction. The algorithm moves the selected junction from its original position x_1 to the new position x_2. Afterwards, the algorithm moves every other junction that is to the right of x_1 with $x_2 - x_1$, excluding the selected junction and its opposing junction.

If the selected junction is a source junction, find a divergent track such that its target is the selected junction. If no such track exists, then continue to the next divergent track.

However, if that track is found, set the selected junction as the source of that track, move the selected junction to the right such that its track has a 45 degree angle with the horizontal axis. Then move every junction, excluding the selected junction and its opposing junction, that is to the right of the old position of the focus junction with the same displacement to the right.

When the algorithm finishes on focusing on the junctions of diagonal tracks, it creates non-divergent-divergent tracks and parallel tracks. If a junction is moved in order to create this track, then every other junction must be displaced as described above.

5 Implementation

In this chapter we will discuss what choices were made in the implementation of the algorithm. For example where we had to deviate from our original pseudocode due to technical limitations.

5.1. Dependency Considerations

In this section we describe different decisions made concerning the implementation of the application.

5.1.1. Back-end

On of the must have requirements is that our application needs to be a web application. There are various back-end frameworks that we can use for such an application. We have chosen to develop in Python, because Python is easy to learn and makes back-end tasks simple with certain web frameworks. Additionally, a lot of different libraries exist that make displaying and reading graphs and geometric data easy.
We decided to use the framework Flask [8]. Flask is very flexible: it is lightweight and easily customisable within your application [9].

5.1.2. Front-end

The front-end is critical to build an application with usability in mind, since the front-end determines how our application interacts with the user. The user needs to be able to use our application with relative ease, therefore it is important that the applications is interactive and has a proper layout. Interactive front-end web development is primarily done in JavaScript, in combination with CSS and HTML, since it is the language native to the web [10]. Furthermore, since JavaScript is so abundantly used on the web, there are many JavaScript packages, frameworks and libraries we can use. For these two reasons we decide to use JavaScript for the interactive front-end of the final application.

It is also beneficial to use some sort of framework, to provide a good structure for the front-end application, thereby improving code quality and to reduce the level of complexity. Frameworks we considered were: Angular[1], Vue[2] and React[3]. The main difference between Angular and the other two is that Angular also features a rich back-end, while Vue and React focus on the display logic [11]. Since we already have a working back-end we do not need the back-end capabilities of Angular for the front-end. We choose Vue as some team members already have experience using it.
Drawing and interacting with graphs is a critical part of the front-end and there exist a number of JavaScript packages providing this functionality, we chose d3.js[4] for this, since it is powerful library

[1] https://angular.io/docs
[2] https://vuejs.org/
[3] https://reactjs.org/
[4] https://d3js.org/

for manipulating and visualising data documents.

5.2. Data Structures

In this section we explain the data structures we use within the entire application.

5.2.1. User Input

The input data structure encompasses the files that are uploaded to the server which are used by the transformer to generate usable data for the algorithm. The input data structure should be multiple GeoJSON files. For the specific railway context there must be one file describing the tracks. another the switches and one describing the stops. Assets should be uploaded as separate GeoJSON files. We use GeoJSON because it is an open format for describing geographical locations using JSON, which is one of the most popular file format for data communication on the internet. Furthermore Moxio fully supports GeoJSON so samples of geographical data are also in this file format.

5.2.2. Schema Generation Input

The schema generation input is the result of the user input after being manipulated by the transformer. With this input we implement the model introduced in Section 2.4.3. The main classes we use for this purpose are shown in Figure 5.1.

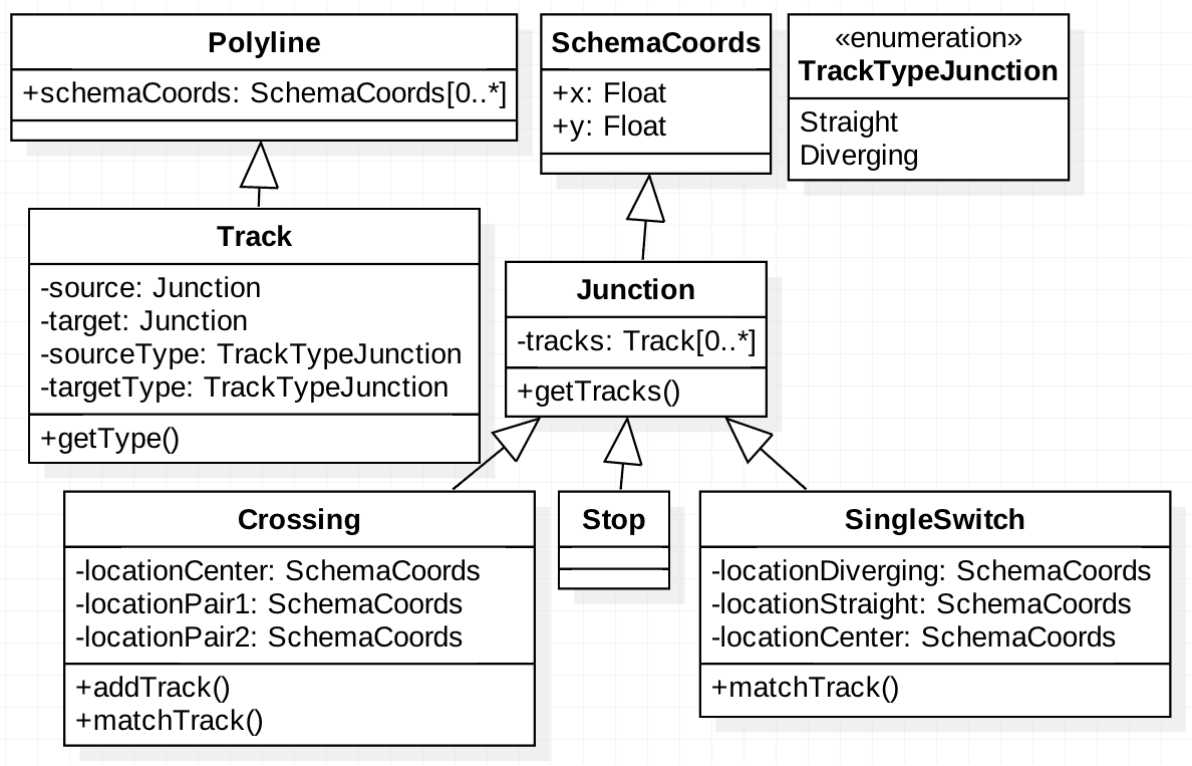


Figure 5.1: Output data structure of tracks and junctions

Figure 5.1 shows an inheritance relation from Track to Polyline. The representation of the Polyline class in this diagram is merely a short overview of what properties it contains. Crossings and SingleSwitches have specific coordinates. The original coordinates are kept in memory, to calculate the direction in which they diverge and to match Tracks.

5.2.3. Output

The server sends a JSON file as output to the browser with all the information needed to draw the schematic representation. After each stage of the algorithm a schema is generated which can be

added to the output so the different stages of the algorithm can be visualised. A schema in the output consists of a list of polylines. A polyline in the output has the lines property which defines the endpoints of the line segments, a points property describing additional assets along the polyline and a tooltip attribute which contents are shown when the user hovers over the polyline. A simple example JSON output is shown below.

```
{"result": {
  "schemaName": [
    {
      "lines": [
        {"id": 0, "x": 0, "y": 0},
        {"id": 1, "x": 1, "y": 2},
      ],
      "points": [
        {"id": 125, "x": 0.33, "y": 0.67},
        {"id": 5, "x": 0.67, "y": 1.33},
      ],
      "tooltip": {
        "ExampleKey": "ExampleValue",
      },
    },
    {
      "lines": [
        {"x": -1, "y": 0},
        {"x": 2, "y": 5},
      ],
      "tooltip": {},
    },
  ],
}
```

5.3. Web Application

The web application is the interface for the user and the part the user interacts with. A screenshot of our web application is shown in Figure 5.2. The features we implemented besides drawing the schematic representation can also clearly be seen. One of these features is highlighting and displaying a tooltip of a polyline when the user hovers over one with their cursor. This gives additional information about the track as described in Section 5.2.3. In this screenshot the junctions on this track and track type are displayed. Labels for assets are also displayed in the application, in this example the id's from the input data for the signals. Furthermore in the right upper corner there is a dropdown where the user can choose which schema needs to be shown.

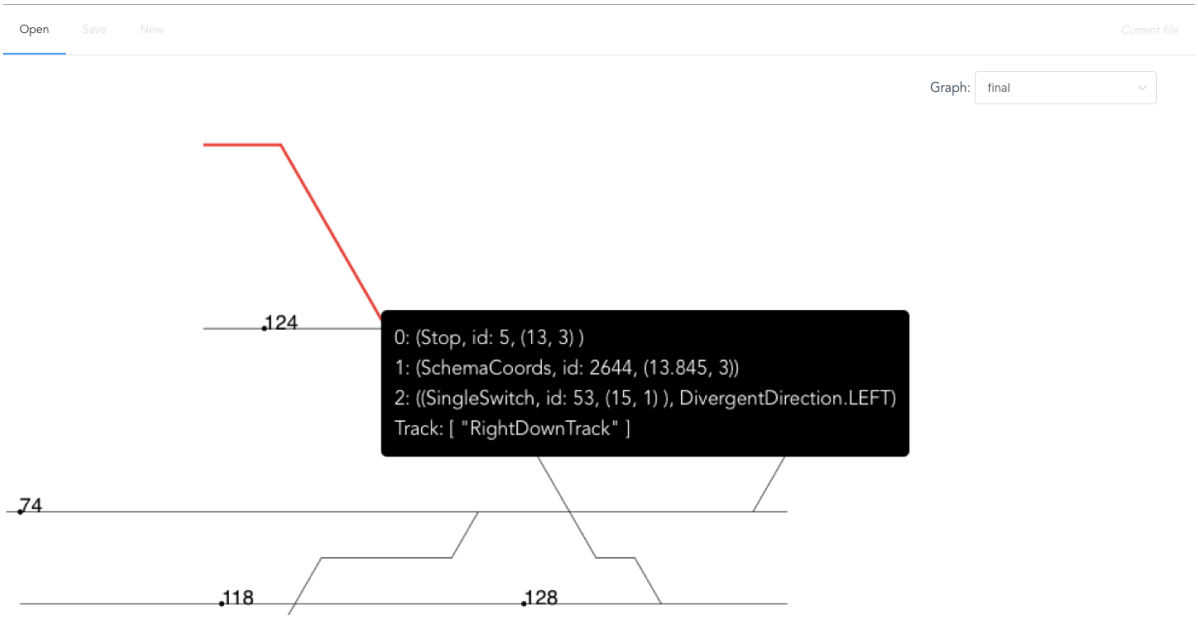


Figure 5.2: Screenshot of the web application

5.4. Algorithms

In this section, some remarks are made that occurred during the implementation of each algorithm previously mentioned in Section 4.2.

5.4.1. Transformer

Location Matching

The input for the transformer is a dataset consisting of objects mapped to real-world locations, in GeoJSON format (Section 5.2.1). We created the first implementation using a hash-map. For every location as given by tracks and junctions, we did the following:

1. Round the longitude and latitude on some precision (to handle input data errors);
2. Calculate a hash of the rounded longitude-latitude combination;
3. Create a hash-map where the key is the output of step 2 and the value is a list of objects that match to this.

The precision factor as described in step 1 was difficult to decide. Because rounding is used, this factor can only be in multitudes of 10. Another issue was that locations that may be very close in the real-world, but not matched through this algorithm as rounding would put the locations on different sides of the boundary. For example: a location with latitude, longitude (0.4999, 1) gets a completely different hash value than the location (0.5, 1) but they are still very close in the real world. This resulted in a lot of mismatches.
Therefore we decided on building another implementation which uses bounding boxes around the locations:

1. Add a small number n to the longitude and latitude of each location to create a bounding box. For example, the location (1,1) would result in a bounding box from $(1-n, 1-n)$ to $(1+n, 1+n)$.
2. Create a key-value store. For each location, try if the location lies in a bounding box. If it does, add it to the store with bounding box as key and a list of locations as value.
3. All locations within the same bounding box are matched.

The downside of this algorithm is that it requires larger time complexity, namely n^2 with n the amount of locations. But the outcome was better and the real running time was feasible.

Crossings

Crossings are a complex type of junction which caused a number of challenges during the implementation. Crossings appear in two different forms in the railroad context, these types of crossings are defined in Section 2.4.3.

The first type of crossing we implement is the full slip switch. Which, in our implementation, behaves the same as a single slip switch. An example of this type of crossing can be seen in Figure 5.3. In this type of crossing four different track segments are linked to the crossing and opposite track segments need to be determined in order to calculate the track type. Since a crossing is defined by five coordinates (the centre and the four endpoints), the two coordinates which form a straight line with the centre coordinate need to be determined. Then when a track is linked to the crossing, the track is added to the correct track pair corresponding with the coordinate pairs determined previously.

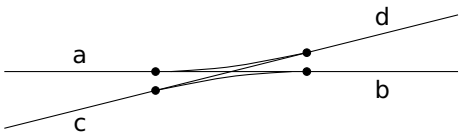


Figure 5.3: Full slip, with track pairs (a,b) and (c,d)

The other type of crossing we implemented, the plain crossing, is only implemented separately for double crossover switches. This followed from the input data we received, where plain crossings only occur in these double cross switches. All other crossings are the full slip variant. In order to correctly visualise the double crossover switches, two tracks are created linking the correct single switches. An example is shown in Figure 5.4 where tracks a and b are created from a crossing. This can be implemented because in double crossover junctions the interconnecting tracks are always diverging, and will therefore not be an issue when determining track types.

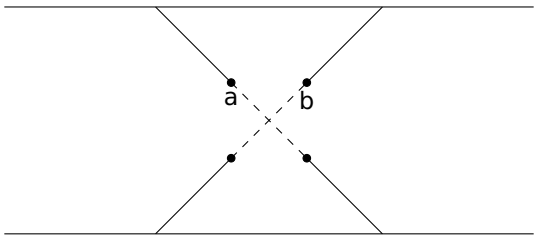


Figure 5.4: Double crossover junction

5.4.2. Linear Referencing

In linear referencing, polylines that consist of junctions are translated and rotated. Whenever a junction is transformed, a new junction would be returned with a new location. A problem that occurred was that the *id* of the old junction was not set to the new junction. This would mean that a new *id* was generated every time a junction is transformed. We kept these *id*s for referencing purposes, and thus, the reference would be lost.

Another problem that occurred was during the transformation of a junction. Its super-type, the *SchemaCoords*, was returned. This had the effect of losing type information, like *SingleSwitch*, *Stop* or *Crossing*. This information was needed later in the algorithm and also for debug purposes.

5.4.3. Straight Lines

In order to correctly draw the schematic representation the application needs to determine which tracks are straight tracks and which tracks diverge from straight tracks. In Section 4.2.3 we defined an algorithm capable of determining these track types. For single switches and stops this was straight forward to implement. However, full slip crossings as shown in Figure 5.3, needed a few additional steps to determine which track pair is the straight line. This was caused by the fact that the type of a track linked to a full slip crossing could only be determined once all four tracks are linked to the full slip crossing.

5.4.4. Order of Execution Changes

In our implementation, we switched the order in which the linear referencing and straight line algorithms are executed. The straight line algorithm splits straight lines from divergent tracks. If linear referencing is done after obtaining straight lines, both straight lines and divergent tracks would have to be linearly referenced separately.

5.4.5. Vertical Ordering

A problem that can occur in vertical ordering is when lines have a circular relation. For example line A is above B and B above C, but C above A. Kahn's algorithm comes to a halt when there are cycles in the graph, so they need to be resolved. During each iteration of Kahn's algorithm, just before it would halt due to cycles, we check for cycles. If they exist we discard conflicting edges until there are no more cycles. This ensures that the algorithm always returns a result and we can always get vertically order of the straight lines.

5.4.6. Divergent Tracks

An issue we came across was a mismatch between a junction location, and a track source or target location. It appeared that while junction locations were updated during previous steps in our application, the track source or target locations did not. We traced this problem back to how location updating is handled. This was due to the data structures of intermediate results of the algorithms not being clearly defined. It regularly occurred that a update location of a junction was stored in the copy of that junction. In the copy process, the tracks property of the new junction gets a reference to the tracks of the old junction. This track list, however, only holds a track that has a reference to the old junction and not the new one. This became a problem, because we were working with tracks instead of a polyline in this step of the algorithm.

6 Testing

Testing is a vital part of software development, since it forms a guarantee that the application is working as expected. It assists in reaching a maintainable codebase by designing for testability. Furthermore testing is a form of self documenting code, since the expected behaviour of each component is defined in the tests.

To make sure the code works in a clean working environment we make use of Docker[1], which exactly describes the environment in which the code should run.

6.1. Test Driven Development

We apply the Test Driven Development (TDD) methodology. The TDD methodology defines that tests should be written before the application logic it tests. By applying this method each components behaviour is exactly defined before the is implemented, which makes sure the implemented component works exactly as expected. It also provides a strong bias towards writing loosely coupled testable code, which improves the overall code quality and maintainability.

6.2. Testing Requirements

We define the following set of requirements that our code should adhere to.

- Code coverage from unit tests must always be above 80%.
- Each new feature should increase overall code coverage.
- Every component should be responsible for a single piece of functionality.
- The code style should be proper and readable.
- The functionality of the code should be well documented.
- All tests must pass, there must be a single point of truth for this purpose.

6.3. Tools

To meet the defined testing requirements we use a set of tools, which verify that the code lives up to the standards.

[1] https://docker.com

6.3.1. Unit Testing

Unit tests form the foundation for testing the code we write by making sure that each individual component works as expected. The unit tests are also the source for the coverage statistics we use. We use PyTest[2] and unittest[3] for writing and running tests written in Python. Jest[4] is used for writing tests in JavaScript.
Unit tests help us with Test Driven Development. They allow for creating a concrete description of the functionalities of components beforehand and their Definition of Done. They create a clear distinction in design and implementation, such that these can be done independently. For example, developer A can describe the required functionalities of class X through a Unit test and developer B can write an implementation to make this test pass.

6.3.2. Static Analysis

Static analysis tools analyse written code and give feedback without compiling or executing the code. We check for the following code smells with static analysis tools.

- Pycodestyle & ESLint are used make sure the code style is uniform and lives up to the language standards, these tools are also utilised to verify that class and method size do not exceed a defined maximum, which encourages the single responsibility requirement.
- pydocstyle is used to make sure each component is properly documented.
- MyPy[5] makes sure type errors are reduced to a minimum.

6.3.3. Pull Requests

We make extensive use of pull request so that all the code is written undergoes manual review by the other team members. This reduces the chance of having errors in the master branch and improves the code quality. Each pull request is also checked against our requirements, making sure the newly implemented code lives up to our standards.

6.3.4. Continuous Integration & Continuous Deployment

Travis CI[6] is the service we use to check each Pull Request against our requirements. Travis CI has the following responsibilities:

- Building and compiling for continuous deployment, so that the product can always be viewed in a web browser, without extra manual work. This is done using a customised Docker[7] image.
- Running the Python and JavaScript tests and check whether they all pass.
- Running static analysis tools and check whether they comply to the requirements as described in Section 6.3.2.
- Generating code coverage reports and sharing the final result with the developers. The coverage report and trend-line can be viewed on Codecov[8], Codecov is also responsible for the check that the code coverage is above 80% and should only increase.

When all checks have passed and Travis is building the master branch a working copy is deployed to Heroku [9]

[2] https://pytest.org/
[3] https://docs.python.org/3/library/unittest.html
[4] https://facebook.github.io/jest/
[5] http://mypy-lang.org/
[6] https://travis-ci.com/
[7] https://www.docker.com/
[8] https://codecov.io/
[9] https://heroku.com/

6.4. End-to-end Testing

End-to-end tests are run by NightwatchJS[10], these are utilised to check if the final product works as defined. We do not gather coverage statistics from the end-to-end tests since these tests are used as a smoke test to verify the application works as expected instead checking whether each individual component is properly implemented.

[10] http://nightwatchjs.org

7

Evaluation

This chapter is an evaluation of the results as output from our algorithms. It contains comparisons of the output schematic diagrams of our application to the current manually drawn schematics, to inspect the differences in terms of aesthetics criteria. We also compare the output to the geographical input data to see whether the output is topologically correct. Since our application is restricted to railway networks as described in Section 2.2, we compare results of a number of railway networks in the Netherlands.

Sneek-IJlst

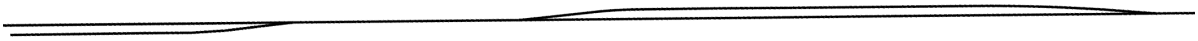


Figure 7.1: Sneek-IJlst: original input

This railway network is the simplest of all networks we ran experiments with. It is a network in the north of the Netherlands. This network contains a parallel track and a diverging track. The output after running the algorithm on this dataset is shown in 7.2.

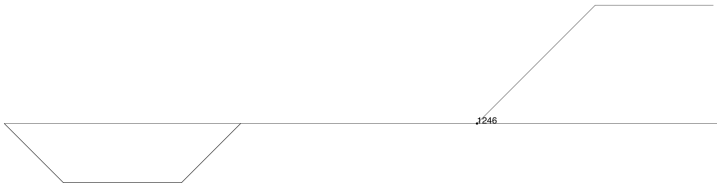


Figure 7.2: Sneek-IJlst: final output

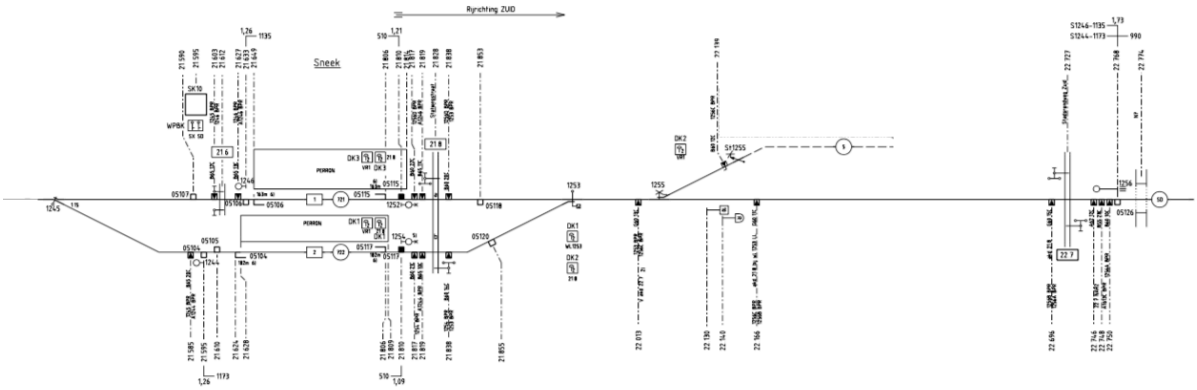


Figure 7.3: Sneek-IJlst: manually drawn schematic diagram

The baseline of this dataset has a direction from right to left, referring to Figure 7.1. For this reason, the output as seen in Fig. 7.2 has the elements in the opposite direction. This output has a correct topology. Both diverging tracks are at the start image on the opposite sides of the main straight track, which is reflected in the final output. The signal is also referenced to the correct track. Fig. 7.3 shows a manually drawn schematic diagram given to us by our client Moxio. The tracks in this drawing shows similarity to the tracks as output from our algorithm.

Kersenboogerd-Enkhuizen

Kersenboogerd-Enkhuizen is a railway network in the north-west of the Netherlands. It is not a very complex dataset, however it is more complex than the Sneek-IJlst dataset.

Figure 7.4: Enkhuizen: geographical input

Figure 7.4 shows the original geographical data for the Enkhuizen area. The signals are not shown here as they would take up too much space to all show them at the same time.

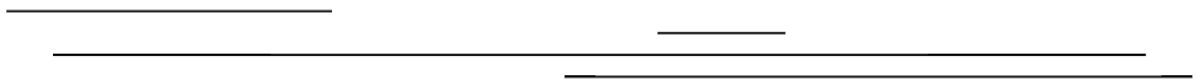


Figure 7.5: Kersenboogerd-Enkhuizen: vertically ordered lines

Figure 7.5 shows the all the straight lines after applying the vertical ordering algorithm.

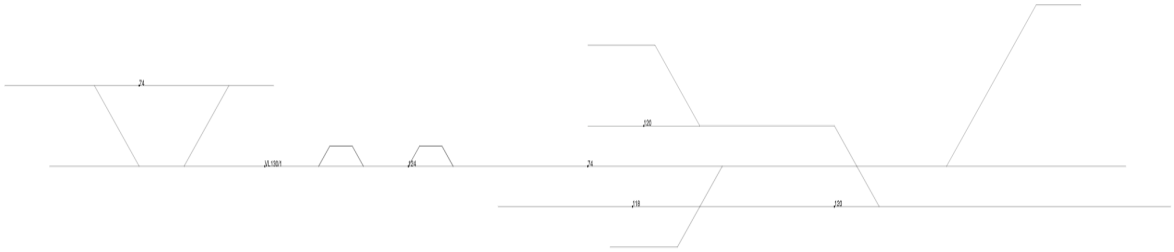


Figure 7.6: Kersenboogerd-Enkhuizen: final output

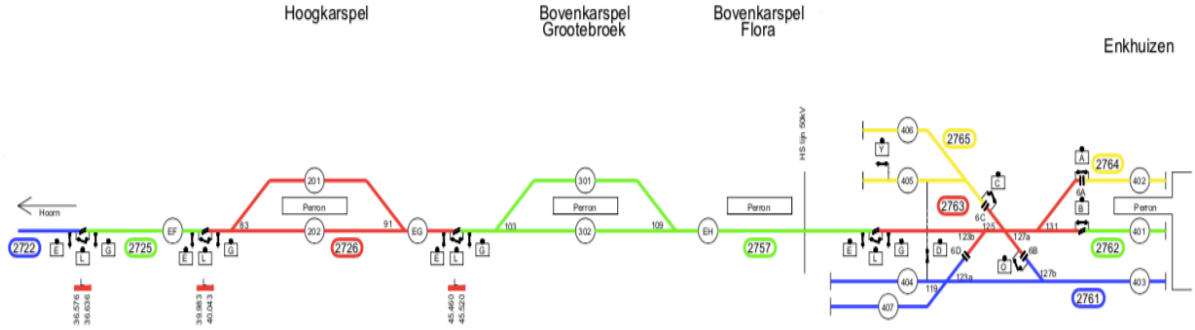


Figure 7.7: Hoogkarspel-Enkhuizen: manually drawn schematic diagram

Figure 7.7 shows a manually drawn schematic diagram given to us by our client Moxio.
The final output of our application has the correct topology and is nearly the same as the manually drawn schematic diagram. The only remark for this output is the placing of signals: Some signals were not matched as they are too far away from tracks. This could be fixed by increasing the tolerance parameter for matching assets to tracks. However this causes false positives to appear, meaning there would be signals shown in the output which are matched to the wrong track compared to the input.

Enschede station

This dataset contains information about a railway network in the Enschede area, which is in the east of the Netherlands. This dataset contains railway signals, in the form of points in latitude/longitude. These are referenced to railway tracks in the right order. As output, the signals are spread uniformly over the straight tracks. The input and output of this is shown in Figure 7.8 and Figure 7.9, correspondingly.

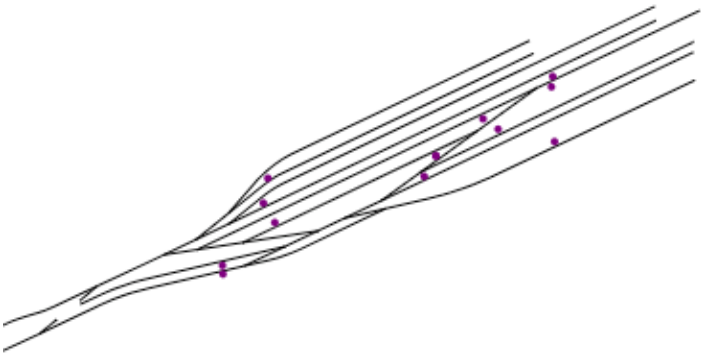


Figure 7.8: Enschede: railway traffic signals and tracks output from Transformer

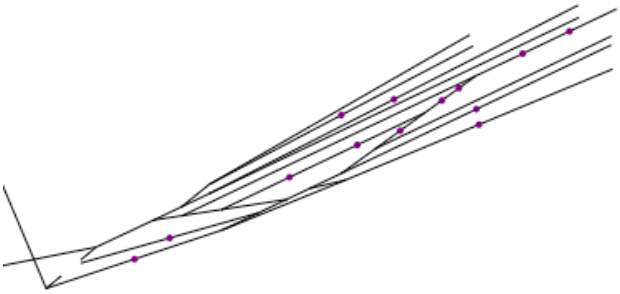


Figure 7.9: Enschede: railway traffic signals after referencing to tracks

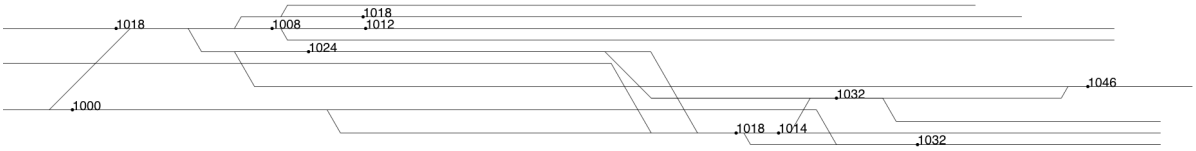


Figure 7.10: Final output of Enschede dataset

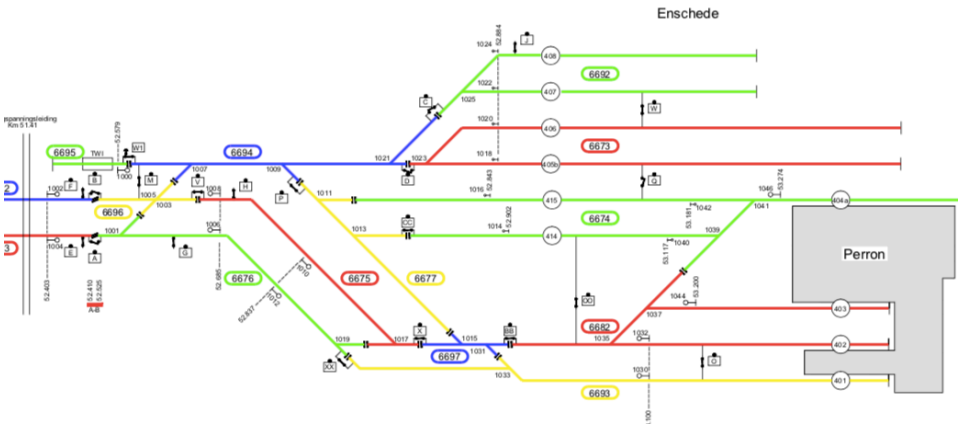


Figure 7.11: Manually drawn schematic visualisation of Enschede provided by Moxio

Figure 7.10 shows the final output for the Enschede area. When compared to Figure 7.11, they show similarities. However, there are differences which can primarily be attributed to two things. The first is that in our application straight tracks are always drawn horizontally. While in the manually drawn schematic this is not necessarily the case. For example the straight track at the switch 1019 in the provided schematic diagram is drawn diagonally. Furthermore, there are some issues with false positives in terms of crossings. Some straight-divergent and divergent-straight tracks are drawn using our own set of heuristics as described in Section 4.2.8 which create bends in tracks. The placement of these bends in these tracks can create unexpected crossings.

Vlissingen-Arnemuiden

The railway network Vlissingen-Arnemuiden is located in the south-western area of the Netherlands. It is simpler than the Enschede dataset in terms of complexity. However it contains many full-slip crossings, of which the track types are difficult to determine.

Figure 7.12: Vlissingen: geographical input

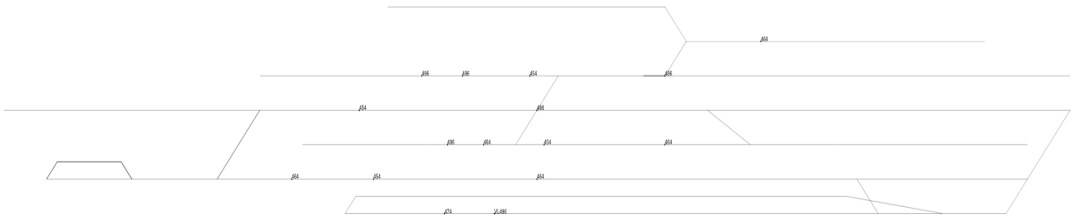


Figure 7.13: Vlissingen-Arnemuiden: final output

The final output as shown in Figure 7.14 shows difficulties with the ordering of tracks. This is due to multiple crossings existing in the dataset. In the vertical ordering algorithm, only the divergent

directions of single-switches are taken into account. This makes the ordering of tracks connected to crossings difficult to determine.

Kijfhoek

Kijfhoek is a complex railway network south of Rotterdam, the Netherlands. This dataset has approximately ten times more tracks and junctions than the Enschede dataset. Due to it being a more complex dataset and our algorithm not being optimised for performance, calculating the output takes infeasible time.
Another issue is that our algorithm does not handle multiple baselines, which this dataset has. Moreover, flyovers and tunnels are supported by our algorithms. For this reason, we can not give a clear output.

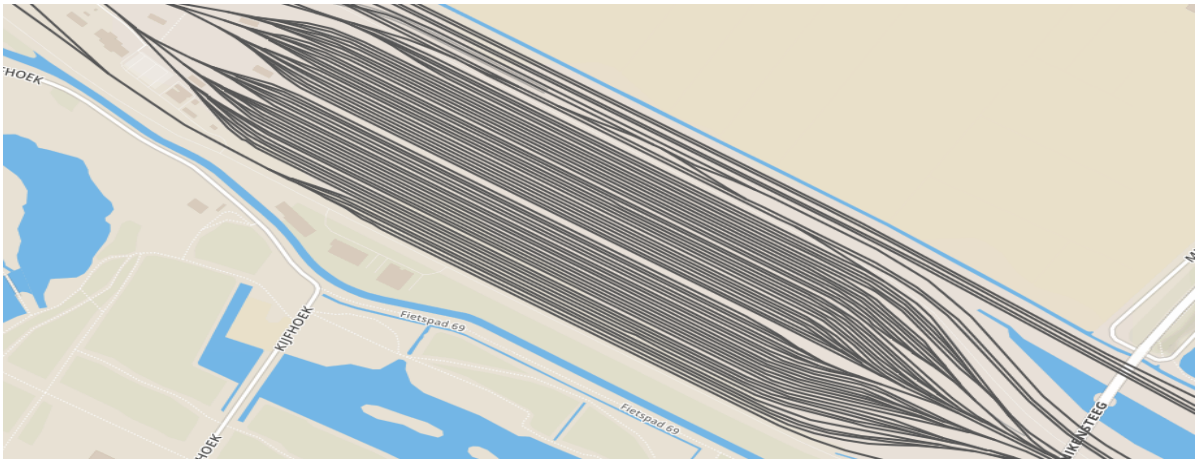


Figure 7.14: Kijfhoek: geographical input

8 Discussion

In this chapter we reflect on the process of our project. We also present the limitations of our application and what future work can be done to improve it. Furthermore, the ethical implications of our work are briefly discussed.

8.1. Process Reflection

In this section we reflect back on the three phases of our project to see if we followed our Project Plan correctly

Research phase

The project started with a research phase where we spent the first two weeks doing literature review. This time was useful to our project as our subject is rooted to theory. It was not easy to decide what to explore while there was no clear idea of the end product. We started with broad exploration about railway networks and narrowed it down to schema generation as the research phase progressed. At the end everyone had a general idea of the algorithm we would formulate.

Implementation phase

At the start we split up the algorithm in parts which we assigned to individual team members. We had one person responsible for the algorithm design, while the others were implementing the algorithm. This led to some confusion about some steps in the algorithm not properly connecting. The output of the linear referencing for example could not be used for vertical ordering. Because of inconsistencies like this we had to refactor code early on. While this lead to higher quality code, it was time-consuming. We would not have had this problem if we sat together at the start of the project and designed the implementation together instead of working on our parts separately. More specifically, we could have defined interfaces for the algorithms mentioned in Section 4.2.

Our weekly sprints worked well in conjunction with our client meetings. As the sprint ended on Monday we had a meeting with our client immediately afterwards to discuss our progress and what we would work on the next week.

Continuous Integration and Deployment

The continuous integration and deployment pipeline proved to be really helpful in maintaining code quality and providing a single source of truth for the status of the project. However we experienced some issues during our project. First of all in the first week of the project when we were setting up the pipeline Travis CI had a day downtime which hindered our progress. Another annoyance we had with Travis CI was that we did not have parallel builds, which sometimes caused the queue to build up and limit our efficiency. Furthermore Heroku changed it's API halfway during our project which broke the deployment process. Therefor the live application was outdated for approximately a week which was unfortunate for the client as they did not see much progression during that time.

Testing
Although we tested our application extensively, some unexpected behaviour appeared when testing with real datasets. This was caused primarily by the fact that our application focuses on visual output, and while we test that our aesthetic criteria as described in Section 2.4.1 are met. It does not guarantee that the visual output is perfect.

Report phase
We documented our progress during the project. Before a feature was considered done it would had to be documented in the report. This made the Report phase go much smoother as we just needed to connect the documentation into a coherent report. The unfortunate situation of the report phase is that we needed to work on the report and the application at the same time. This led us to properly sort out our priorities as we had deadlines for the report and the application in the same week.

8.2. Limitations
The final product has some shortcomings in comparison to the originally planned product. The following three requirements are not satisfied due to time constraints and complexity underestimations. The last requirement was declared as low priority, but should have been a higher priority.

Filters
The "should-have" requirement: "The application should have a set of filters that can be enabled/disabled to obtain a different view of the output." is not satisfied. This is a low priority functionality to our client.

Interactive methods
The "could-have" requirement: "The application could have interactive methods for doing calculations and visualisations of algorithms applied to the output" is not satisfied. This is a low priority functionality as well to our client.

Custom assets
The "could-have" requirement: "It could be possible to submit a custom asset set using the web application, such that custom assets can be connected to nodes" is not satisfied. This was wrongly placed as a low priority functionality to our client, and due to time constraints, we could not finish it.

8.3. Future Work
In this section we present recommendations for improving the application, both functionally and structurally.

Horizontal and Vertical Ordering Improvements
Our implementation of horizontal and vertical ordering is not without flaws. There are situations where tracks and junctions are not placed in the correct location in the output and introduce topology errors. An example of this can be seen in the output of the Enschede dataset Chapter 7. Future research can be conducted towards improving these algorithms.

Multiple Baselines
During the project, the client informed us that many railway networks contain multiple baselines. While our algorithm can only handle a network with a single baseline, we have an idea on how to handle multiple baselines.

It is possible to extend the transformer to assign all objects in the network to a baseline. When there are multiple baselines all objects are assigned to the baseline closest to them by distance. When all objects are assigned, the algorithm can be run multiple times, once for each baseline. This will generate separate schematics which can be overlapped based on the relations between the baselines.

Circular Baselines

It is theoretically possible for a railway network to have a circular baseline which connects to itself. Our algorithm is not prepared to handle this as it is based on the assumption that a network has a main linear direction. This is not the case when the baseline is circular. This can be solved by splitting the circular baselines in multiple linear baselines and applying the solution proposed in Section 8.3.

Flyovers and Tunnels

Flyovers and tunnels are types of crossings we currently do not support. These tracks that cross the same location but at a different height in the real-world. This is currently not something our transformer recognises as it only looks at the endpoint of tracks, and does not check for crossings in the middle of tracks. Because it is not possible to switch tracks at flyovers and tunnels, they do not change the topology of the network. It would however be useful to have an indication in the final schema of where these flyovers or tunnels are located by a special link between the two tracks.

Complex Assets

Our current application supports signals, which are represented as single locations using small circles. The asset matching algorithm can be extended to support assets which are more complex than single points.
An example of a complex asset is a railway platform. Platforms are polygons, and should be shown like such in the output. Supporting platforms would require assets to be referenced to multiple tracks at the same time. This is because platforms can serve one or two tracks. Currently, assets can only be referenced to single tracks.

8.4. Ethics

Our application is build as a proof-of-concept to reduce the amount of work a technical drafter has to perform. It has no risks and can not be used maliciously.

9

Conclusion

During the course of this project, we explored the possibilities for automatically generating a schematic diagram of railway networks. We improved on an existing algorithm presented in the paper *Automatic generation of schematic diagrams of the dutch railway network* [1]. Because no implementation was publicly available, we had to implement the algorithm from scratch. We made a major improvement with regards to (Brands, 2016) in the vertical ordering of lines as described in Section 4.2.6. We developed a web-application that allows for uploading geographical data which gets converted into a schematic diagram that adheres to our aesthetic requirements.

We experienced that many railway networks contain multiple baselines. Our algorithm currently does not support multiple baselines but we outlined a possible solution in Section 8.3. Furthermore due to limited time we were not able to properly display relevant assets (e.g. platforms, stations, flyovers). The vertical ordering algorithm is vulnerable when the number of parallel tracks gets large. A mistake in the vertical ordering can invalidate the aesthetics requirements.

Our application works as a proof-of-concept. It demonstrates that our algorithm is effective and can be implemented efficiently. If Moxio decides to implement our algorithm they can refer to the design chapter of this report.

Based on our results we conclude that it is possible to automatically generate a schematic representation from a geographical network while ensuring that its properties are correctly represented, however our implementation is limited to small datasets and minimal assets.

A

Infosheet

General Information

Title: Schematic visualisation of geographic networks
Client: Moxio
Presentation date: 4th of July, 2018

Description

The analysis of geographical networks in its current form is inefficient. Network analysis becomes much simpler when an analyst uses a schematic representation of a network. In a schematic representation only relevant information is shown. A large drawback of schematic representations is that they are made manually. This is a cumbersome process, and when a change to the network is made, the drawing needs to be updated as well.
The main challenge of our project is to build a web application that automates the process of drawing schematics railway networks. During the research phase, we found a Master's Thesis which explains a way of modelling railway tracks and junctions and attempts to draw schematics. We improve upon the findings of this thesis.
We worked in weekly sprints. By the end of each week, we presented the changes to the client and received feedback. At the start of every sprint, we created a sprint plan. In this sprint plan, we defined the division of tasks and responsibilities, as well as their priority levels. An unexpected challenge was that we were too optimistic in our planning. We overcame it by being discussing the difficulty of tasks with the client to get a more realistic assessment of the workload.
We have created a product that is able to create schematics from geographical railway data. During development, we wrote extensive unit tests and integration tests. Additionally, we created end-to-end tests and presented our schematics to the client at the end of every sprint and received feedback. The application serves as a proof-of-concept to our client and will not be integrated on its own to their existing services.

Members

- **I. Dijcks**: I was responsible for designing the algorithms and in the final stages the report documentation.
- **R. Heddes**: In the first half of the project I was mainly concerned with building an interactive front end and enabling continuous integration and deployment, I later helped with implementing transformer and schema generation algorithms
- **T. Wissel**: My contributions mainly consisted of calculating and displaying assets, and building the data transformer.
- **K. Yilmaz**: My contributions were mainly implementing schema generation algorithms.

Contact Information

Client: H. van der Kolk, Moxio
Coach: S. Roy, PhD student SERG, TU Delft
Project: I. Dijcks. Email: ishadijcks@gmail.com
The final report for this project can be found at: http://repository.tudelft.nl

B

SIG feedback

In this appendix we discuss the feedback gained from the Software Improvement Group and reflect on their suggestions.

B.1. First submission

Our first submission was at the end of sprint 4. Most of our algorithm worked at this time although not all functionality was connected.

B.1.1. Feedback

De code van het systeem scoort 3.5 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Size en Unit Complexity vanwege de lagere deelscores als mogelijke verbeterpunten.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.

In jullie project gaan de twee vaak samen. Zo is TrackFactory.calculate_tracks() een vrij groot algoritme dat in zijn geheel in één methode is geïmplementeerd. Dat maakt het op termijn moeilijk om de methode aan te passen, want door de grote scope is nu moeilijk te overzien wat een aanpassing voor gevolgen gaat hebben. Voor _calculate_target_type in track.py geldt eigenlijk hetzelfde. Probeer meer abstractie in dit soort methodes aan te brengen, zodat de code onderhoudbaar blijft op het moment dat de hoeveelheid functionaliteit gaat toenemen.

~~Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.~~

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

Erratum

We gebruiken onze tool voor automatische herkenning van testcode. Dat gaat blijkbaar niet voor alle frameworks goed, na handmatige controle zie ik nu dat jullie inderdaad een behoorlijk aantal

tests hebben. Beschouw mijn opmerking over testcode daarom als niet verzonden.

B.1.2. Discussion

To reduce our Unit Complexity we refactored the calculate_track() method in TrackFactory and the _calculate_target_type method in Track by splitting them into multiple functions. This makes the methods more readable and easier to test. To make sure we will not add any more smells to our codebase, we have enabled the bettercodehub plugin to our Continuous Integration. If new code contains a bad practice we will refactor it before we add it to our codebase.

C

Project description

The project description as given by our client, Moxio:

“Moxio is an innovative software company in Delft based near the TU Delft campus. We create high-level software for managing, visualizing and validating information for large infrastructural projects. For example on the Spoorzone Delft Tunnel. We’re a team of around 20 people, mostly with a TU Delft background. Clients include [...] (some large companies that work with geometric data).

Infrastructural projects generate a lot of data. This information is generally linked to a geographical representation which you can display on a map. Since infrastructure is usually about networks (rail, road, water, energy, telecommunications) we want to be able to create a schematic representation of these networks so we can convert data into geographical and schematic visualizations. We want to be able to switch between graphic representations such as such as logical and physical display and create flow charts.

In an interactive version of the schematic visualization we want to be able to add information to views and manipulate these views. Manipulating views is about adding additional assets, enlarging certain areas, selecting network-parts and generating a new schematic result as a clear and uncomplicated visualization. Currently available graph software and libraries are not extensible enough and do not fit real world representations.

One of the items is that the schematic representation should link to the real world representation of the layout and should be able to render crossings and multiple levels.
We’re looking for enthusiastic students with skills in graphs, visualization techniques and algorithms who want to tackle this project.
We develop products and our RailNEXT platform, using an advanced framework of JavaScript and PHP with specific parts in C# and java."

Bibliography

[1] A. Brands, *AUTOMATIC GENERATION OF SCHEMATIC DIAGRAMS OF THE DUTCH RAILWAY NETWORK*, Ph.D. thesis, Radboud University (2016).

[2] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Algorithms for drawing graphs: an annotated bibliography,* Computational Geometry **4**, 235 (1994).

[3] ESRI, *ArcGIS Schematics | Overview* (2018 (accessed 25-04-2018)).

[4] S. Seyedi-Shandiz, *Schematic representation of the geographical railway network used by the swedish transport administration,* LUMA-GIS Thesis (2014).

[5] G. Esri, *Mapping software,* ArcGIS: http://www. esri. com/software/arcgis (2006).

[6] K. M. Curtin, G. Nicoara, and R. R. Arifin, *A comprehensive process for linear referencing,* URISA Journal **19**, 41 (2007).

[7] A. B. Kahn, *Topological sorting of large networks,* Commun. ACM **5**, 558 (1962).

[8] *Flask,* http://flask.pocoo.org/ (2018).

[9] G. Dwyer, *Flask vs. django: Why flask might be better,* https://www.codementor.io/garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7mdf8v (2017).

[10] E. Ecmascript, *Language specification,* (2015).

[11] M. Petrosyan, *Angular 5 vs. react vs. vue,* https://itnext.io/angular-5-vs-react-vs-vue-6b976a3f9172 (2017).