UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master thesis

Efficient generation of map graphs

How to navigate the unnavigatable

Tobias Zeitz Floor

Advisor: Pawel Winter

Submitted: May 31, 2022

This thesis has been submitted in association of DIKU, The Faculty of Science, University of Copenhagen

Abstract

This thesis assumes a basic level of knowledge in graph theory and optionally GIS (Geographic Information System) as most of the project will be working with map data.

The project will be working with real world data typically used in GIS software, where the goal is to generate networks based on the paths and their associated data (such as speed limits, capacity, length etc.). QGIS is open-source project, which works with spatial data visualization and processing. I will be drawing comparisons to one of the processing tools they provide to generate the shortest path between two points by creating a network from the paths. The main problem is the conversion of the data to a obtain a reasonable network, that operates as expected in most if not all cases, where QGIS falls short in some data-sets and I will cover these cases in this project and which improvements are to be made.

The methods I will be using to generate such a network, are range searching data-structures as it can significantly reduce the problem complexity to only look at neighbourhoods of paths. The reason for looking at a neighbourhood of paths, is that it is only expected to connect to other proximate paths, that could be considered to intersect.

The solution should also strife towards creating a network, s.t. it can solve general graph problems, such as, but not limited to, shortest path or fastest route (ETA - Estimated Time of Arrival). The solution will also be implemented as a prototype as means to test practical differences between QGIS's solution for creating networks and my own.

The project did succeed at providing a better solution for generating networks than QGIS for the data-sets chosen. It provided better practical run-times in the prototype, but not theoretical as there might be hidden costs in the library used to assist with the porting of QGIS's solution to the prototype. The proposed solution also shows better quality (connections between paths, where it is expected) and provided the functionality for shortest path, fastest route and more.

Contents

Contents 3

1 Introduction 5

2 Problem description 6

3 User scenarios 9

4 Data source 9
- 4.1 Vejhastigheder . . . 10
- 4.2 GeoDanmark60 . . . 12
- 4.3 Observations . . . 13

5 QGIS solution 15
- 5.1 Query structure . . . 16
- 5.2 Analysis . . . 17
  - 5.2.1 QGIS pseudocode breakdown . . . 17
- 5.3 Improvements . . . 19

6 Proposed solution 21
- 6.1 Exploiting locality . . . 21
  - 6.1.1 Desired connection cases . . . 22
  - 6.1.2 Range searching algorithms . . . 23
  - 6.1.3 Choice of range search algorithm . . . 28
  - 6.1.4 Simplifying paths . . . 29
- 6.2 Order of operations . . . 31
- 6.3 Pseudocode . . . 32
- 6.4 Pseudocode breakdown . . . 33

7 Implementation 35
- 7.1 Reference algorithm . . . 35
  - 7.1.1 Non-trivial differences from original . . . 36
- 7.2 Proposed algorithm . . . 36
  - 7.2.1 Range tree . . . 36
  - 7.2.2 Disabling mid-point pairing . . . 38
- 7.3 Prototype . . . 38
  - 7.3.1 Getting started . . . 39
  - 7.3.2 User guide . . . 39

8 Results 42
- 8.1 Data-set statistics . . . 43
  - 8.1.1 Vejhastigheder small - Statistics . . . 43

8.1.2 GeoDanmark60 small - Statistics . . . 44
8.1.3 Vejhastigheder medium - Statistics . . . 45
8.1.4 GeoDanmark60 medium - Statistics . . . 47
8.1.5 Vejhastigheder large - Statistics . . . 48
8.1.6 Geodanmark60 large - Statistics . . . 49
8.2 Quality . . . 49
8.2.1 Vejhastigheder - Quality . . . 49
8.2.2 Geodanmark - Quality . . . 51

9 Evaluation **52**

10 Possible improvements **54**

11 Conclusion **54**

Bibliography **55**

A Geodetic Datum **56**
A.1 WGS84 . . . 56
A.2 UTM . . . 57

B QGIS algorithm **58**
B.1 Implementation details . . . 58
B.1.1 Tie point . . . 58
B.1.2 Hash-tables . . . 59
B.1.3 Path direction . . . 59
B.2 QGIS - overview . . . 60
B.3 QGIS - detailed . . . 61
B.4 QGIS - helper functions . . . 62
B.4.1 GetPaths . . . 63
B.4.2 UpdateTiePoints . . . 63

1. Introduction

This introduction is used to provide formulations that will be used throughout all parts of the project as some terminology from GIS (Geometric Information Systems) differs from academic graph theory terms.

The goal of the thesis is to provide an efficient method to generate networks (see def. 1), that could uses common data-formats associated with GIS maps, such as GML[1](Geometric Markup Language), SQL[2] and shape files[3], which is used in industry software such as ArcGIS, MapInfo, QGIS and Maptitude etc. The format I will be using is GML as it provides easy, human-readable data, but is in essence very slow to parse compared to other formats as it is stored in text format.

Definition 1. ***Network****; A network or graph is denoted as $G(V, E)$, which describes a set of vertices V and in this case a set of connecting* ***weighted directed*** *edges E. The network should represent some underlying data from a map, such as roads, pipes or cables.*

In my research I found the following two solutions that do approximately what I desire to do and can be found here:

QGIS network analysis: https://docs.qgis.org/3.16/en/docs/pyqgis_developer_cookbook/network_analysis.html#building-a-graph

esri ArcGIS generating network: https://developers.arcgis.com/python/guide/part1-introduction-to-network-analysis/

Due to its open-source license and existing network processing tools in the QGIS software[4], then I will be using QGIS as a reference solution and compare its short-comings and strengths to my proposed solution.

Similar to most GIS software and in general map terminology, then I will be referring to geometry throughout the project as defined in def. 2.

Definition 2. ***Geometry****; In map terminology, a geometry refers to either a point, path or polygon. Most commonly these geometries use a coordinate system to define their shapes, typically using either WGS84 or UTM coordinates (if curious, more can be read in appendix A Geodetic Datum).*

Geometry typically has associated data, such as speed limits, capacity and length, which could also be considered equivalent to columns in the context of databases. Each layer (see def. 3) has a unique set of data, that is universal for all geometries in the layer ex. in the case of roads, it might be speed-limits.

[1]Read more about GML format here: https://www.w3.org/Mobile/posdep/GMLIntroduction.html

[2]Read more about SQL geometric datatypes here: https://www.red-gate.com/simple-talk/databases/sql-server/t-sql-programming-sql-server/introduction-sql-server-spatial-data/

[3]Read more about shape (.shp) file format here: https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf

[4]Read more about QGIS at their main website: https://qgis.org

The data needs to also be added as part of the generation of the network to provide meaningful weights for the edges of the network, ex. for ETA (Estimated Time of Arrival, read more details in 3 User scenarios), which would require a combination of distance and speed of edges to deduce a weight.

Definition 3. ***Layer****; A layer is a collection of geometries, such as points, lines or polygons, which all share the same data columns, but can vary in their content.*

The focuses on applying a transformation specifically to paths (see def. 4) and turn them into a navigable network s.t. it can allow the use of graph problems ex. shortest/-fastest path or flow problems. But regardless, the graph algorithms are not the main focus of the thesis, but rather to create a network that can support graph operations.

I will be comparing the quality[5] and measuring the performance of the networks generated by QGIS's solution and my own proposed solution (more on how to do this in 7 Implementation and results at 8 Results).

Definition 4. ***Path****; A path is a graph,* $G(V, E)$*, defined by* $1 \leq |E|$*,* $|V| = |E| + 1$ *and* $\forall v \in V;\ 0 < degree(v) \leq 2$*. In other words, a path will consist of exactly two vertices of degree 1 (end-points) and zero or more vertices of degree 2 (mid-points). In map terminology, paths are typically referred to as poly-lines*[6] *or line-strings*[7]

To further allow for usability, the network generated will allow the use of coordinate queries and find closest vertex or edge to interface directly with the network. More on this can be read in 7.3 Prototype and why it is important for the user experience.

To achieve the goal of generating a network from the paths, then I will be utilizing algorithms from Computational Geometry to provide fast point- or range querying, Algorithms and Data-Structures for path finding algorithms and Programming Language Design to provide weights to be expressed as formulas. More on which data-sets I have picked for this thesis in 4 Data source. The most important part of the thesis is the range-searching algorithms, that can be utilized to find relevant neighbouring paths s.t. it prevents running over all paths when checking for potential connections to other proximate paths. More on the details of how Computational Geometry and evaluation of the range searching data-structures available in 7 Implementation.

2. Problem description

The paths (see def. 4), that I will be working with does not *necessarily*[8] contain data or an associated data-structure, that determines which paths are connected (see def. 5). Therefore it would be very inefficient to attempt to solve graph problems on said data as it would be both difficult and expensive to create connections in run-time.

[5] Referring to the existence of connections where one may expected to see them when visually inspecting the network

[6] Stemming from the meaning of poly (many) lines, so a geometry that consists of a collection of lines connected. This definition is typically used in GIS software ex. MapInfo.

[7] Used most commonly in SQL or GML.

[8] There are data-sets utilizes id-matching and always breaks the paths up when a connections are intended, such that endpoints are a safe assumption to be connected if adjacent. But for the project I will not assume that this type of data is available

Definition 5. ***Connection****; A connection is a vertex in the final network generated, which connects two or more paths by a common vertex shared between them and allows for traversal across the paths.*

To generate a network based on the paths purely based on their spatial adjacency, then certain assumptions has to be introduced based on inspection of the data, such as seen in hypothesis 1.

Hypothesis 1. ***Generating connections****; Proximity of different paths' vertices and edges is an indicator of intend for a connection. An end-point of a path is a strong indication for a connection to be made between paths.*

As stated in the hypothesis 1, connections made using a some distance between paths can at best be an indicator of the intend for a connection to be made. Based on an inspection of multiple data sources (see 4 Data source), then I can conclude that end-points of paths are a stronger indication of intend for a connection than mid-points depending on the structuring and quality of the data source[9].
To clarify what exactly is considered to be an intended connection then all combinations of end-points, mid-points and the line-segments between points has to be considered. This is not a trivial problem to define as it does introduce biases as to how one could expect the input data to behave or how it is constructed. The problems that might occur using biased interpretations of what a connection required will be covered in later in 4.3 Observations, where QGIS also imposes expectations to the data, which causes undesired results. I will be introducing a way to get around the drawbacks of the assumptions made in QGIS's implementation at the end of 5 QGIS solution in 5.3 Improvements.

Definition 6. ***Legality of generating connection****; Connections between paths that show an intend for connection based on hypothesis 1 are considered legal. Additionally to allow for scenarios with bridges or tunnels, then intersections between crossing line-segments will be considered illegal (see example in figure 2.1f).*

In the following figure 2.1 there is provided examples of how definition 6 can generate connections between paths, where individual paths are separated by color. The description of the symbols in the figures can be seen in sub-figure 2.1a, where I distinguish between mid-points and end-points as will later be discussed in why it is of interest to separate the types of vertices in a path.

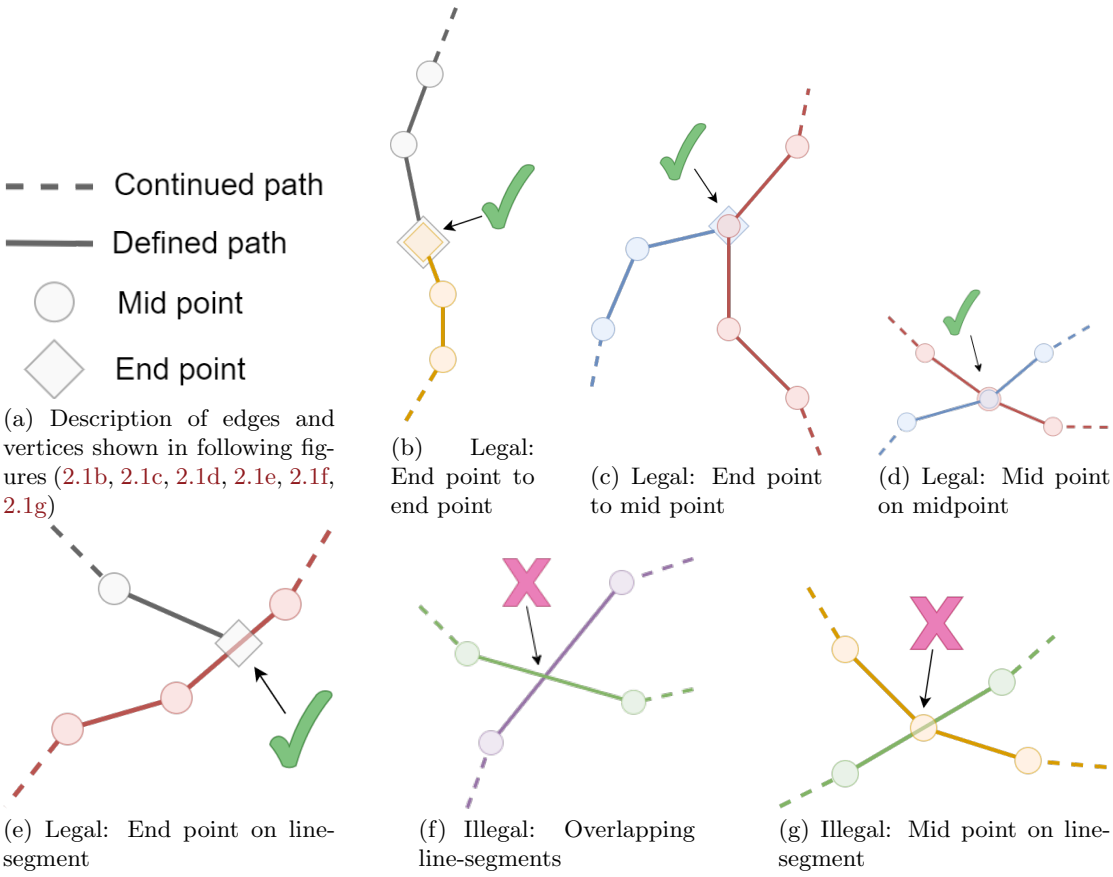
The first example of a legal connection is when two end-points are in close proximity or overlapping as seen in figure 2.1b. It is an obvious candidate to create a connection by hypothesis 1. In the second case in figure 2.1c, where an end-point is intersecting a mid-point, is typically associated with updates to a map where paths are added, ex. a new villa road that branches from an existing paths from the layer. The third legal case in figure 2.1d. It is a connection between two mid-points, which is still fine by hypothesis 1 and will later be discussed in 4.3 Observations, how this type of connection is not as strong as end point connections in certain cases.

The first three cases from figure 2.1b, 2.1c and 2.1d are exactly what QGIS does when generating the network, but by hypothesis 1, then the last legal case seen in figure 2.1e, should also be considered for making a connection. Figure 2.1e assumes the same scenario as figure 2.1c, where side-paths may branch out from existing paths, where the

[9] Quality is here referring to the density of points along the path, that adds detail to the path. In some cases for higher density paths, it would be possible for bridges to connect to the underlying road, which is not a desired behaviour. More on these cases will be covered in 4.3 Observations.

existing path may not be transformed to have a point introduced at the location of the intersection. Figure 2.1e also supports a broader range of data-sets as it does not assume that branches from existing paths have to have a matched vertex on the existing path.

But there is also cases where the intend for a connection is vague at best and it also has to be considered. The first 'illegal' case for a connection will provide the functionality to hopefully support bridges and tunnels, and is seen in figure 2.1f. Otherwise if a connection is allowed at the intersection point of paths' segments, it would allow the user to drive off the side of the bridge or dig underground to the tunnel, where neither scenario is desired. Similarly, an example of where the overlapping segments should not allow for connections can be seen in 4.1 Vejhastigheder in figure 4.2b. Lastly I will separate the importance mid-points and end-points as mentioned in hypothesis 1 and example seen in figure 2.1g. Compared to figure 2.1c, the scenario of a mid-point intersecting a line-segment, then it is not allowed. As mentioned in earlier 2 Problem description, some maps may have a high density of mid-points along a path to add detail, that causes a connection to underlying roads (more on this in 4.3 Observations).



(a) Description of edges and vertices shown in following figures (2.1b, 2.1c, 2.1d, 2.1e, 2.1f, 2.1g)

(b) Legal: End point to end point

(c) Legal: End point to mid point

(d) Legal: Mid point on midpoint

(e) Legal: End point on line-segment

(f) Illegal: Overlapping line-segments

(g) Illegal: Mid point on line-segment

Figure 2.1: Models for all combinations of end-points, mid-points and line-segments of paths. These models are based on the discussed hypothesis 1 and the extensions definition 6 for when a connection should be made.

3. User scenarios

The hypothetical use-cases for the network, that is the product of the algorithm, can also be the following:

GPS navigation Connecting roads as discussed with some knowledge of speed can provide approximations to the ETA (Estimated Time of Arrival). Using the algorithm and assuming the data is present for the paths, then one could also get the fastest-, shortest-, longest routes etc.

Flow control Flow problems are interesting in the case of water supply pipes as they have a certain capacity and making an extension to a city would impose more pressure on whatever supply pipe that might be constructed to the new area of the city.

Marketable area Stores and supermarkets are interested in maximizing the amount of inhabitants in the proximate area to have a larger potential group of customers, hence it is useful to generate a polygon of x distance away in all connected directions from the store via roads.

The algorithm should allow for the use of arbitrary data from the paths to partake in the weight calculations for the edges in the network. Allowing this type of import of the data will support the above scenarios and have default values in cases where a path's data might be not be set, which is common in GIS. An example of relevant data for road navigation can be read in the following section 4 Data source. More on the topic for parsing features to the network will be covered in 7.3 Prototype.

I based most of my assumptions leading to the rules discussed in figure 2.1 of legal connections on the following two 4 Data sources and a combination of the above scenarios. As I have picked a limited number of data-sources, then the algorithm might be biased based on their data and structure, but I will later discuss in 7 Implementation, how the algorithm can be generalized given some parameters for a given data-set. The two data-set 4 Data source are vastly different in the way they are constructed and therefore providing a level of freedom for the algorithm will be desired.

4. Data source

The data I will be utilizing for the project is provided by `https://dataforsyningen.dk/` or `http://www.vejman.dk`. The data is typically separated into layers (see def. 3), where the layer provided from `http://www.vejman.dk` is called `ws112:hastighedsgraenser`, that can be fetched at `https://du-portal-ui.dataudveksler.app.vd.dk/data/112/overview` with a valid nem-id. `ws112:hastighedsgraenser` contains the roads that 'vejdirektoratet'[10] manages. Read more in 4.1 Vejhastigheder as to how the data can be used to solve some of the 3 User scenarios.

The second data-set of interest is provided by dataforsyningen and is found at `https://dataforsyningen.dk/data?filter=129;tag-145`, which contains a layer with all roads, but does not contain information about the speed limits. The data-set is interesting as it provides multiple types of roads/paths, which are mixed together. This

[10] The company that manages `http://www.vejman.dk`, more can be read at `https://www.vejdirektoratet.dk`, but as the domain suggests, is written in danish.

causes the requirement of specifying certain types of roads, as not all types make sense in all contexts, read more in 4.2 GeoDanmark60.

The project will mainly be focusing on the data-set seen in 4.1 Vejhastigheder as it has the most interesting data. In the second data-set, GeoDanmark60, different types of roads are mixed together, leading to walking-, bicycling- and larger vehicle paths are mixed. GeoDanmark60 is a good case for where separating the layer when generating the network by the its road type would make sense, as ex. a car should not traverse bicycle paths.

4.1. Vejhastigheder

The total number of paths in the data-set from `http://www.vejman.dk` is 194.062, hence it is large data-set, where I will sample regions for testing. Read more about the sampled regions in the 8 Results.
A general overview and an example of which municipalities are included can be seen in figure 4.1. While not all regions are available in the layer, all the major roads are still included, allowing for generation of a network that can traverse between regions.

A closer look on the data can be seen in figure 4.2, where an example of an interchange can be seen in 4.2a. In the second sub-figure 4.2b where I have applied styling similar to the way I modelled mid-points and end-points in figure 2.1 from 2 Problem description. As also seen in figure 4.2, then each path consists of two parallel lines. This can cause the complexity of the final network to double if a radius for the connection between the two parallel lines does not overlap. From my inspection in QGIS software, then there is approximately 2 meters between each pair of points along the path, hence a safe bet for creating connections is a tolerance radius of 2.5 meters.

The features of the layer can be seen in figure 4.2e, that can be utilized to create a network and provides the possibility of performing ETA (Expected Time of Arrival) estimations. Due to a change of file format in the 7.3 Prototype, then a lot of the names has been truncated to 10 characters. But more details on how to use the data-set in 7.3.2 User guide.

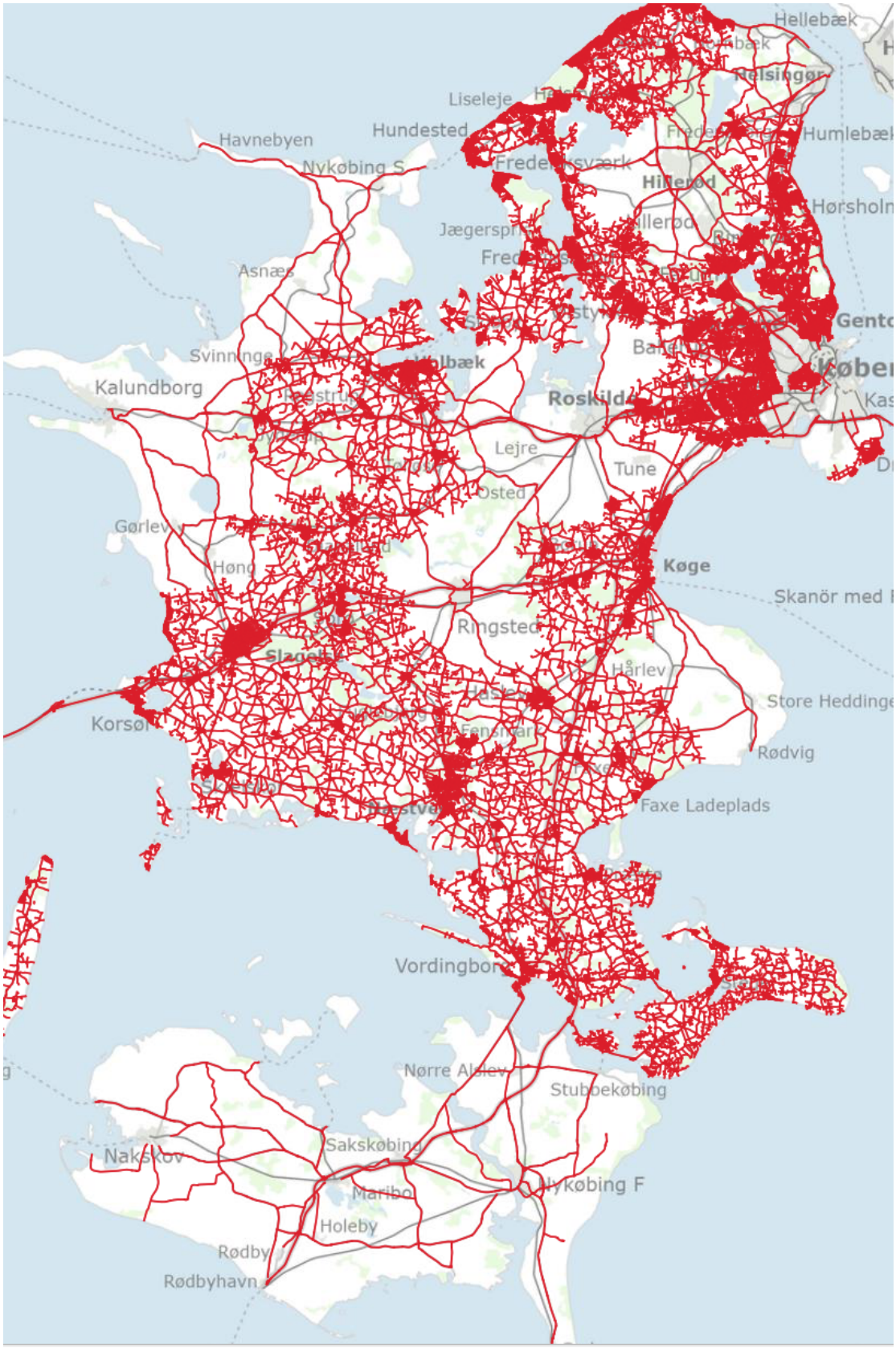
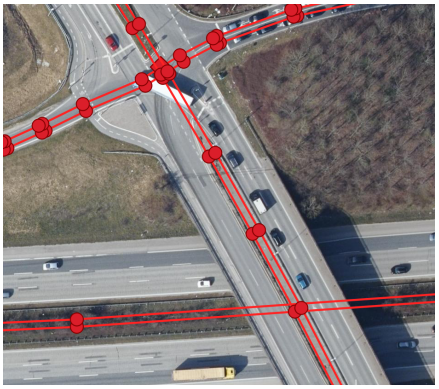
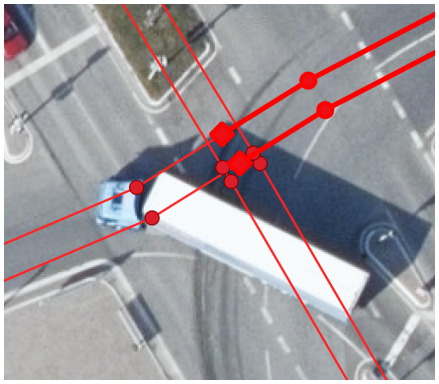


Figure 4.1: Zoomed out image of Zealand, part of Denmark. It is clearly visible that not all municipalities allow Vejdirektoratet to monitor or publicize their roads, such as Ringsted (middle), Roskilde and Lolland (bottom island). But the most important part of this data-set is the mayor roads are included in the more barren parts, allowing for testing long-distance traversal across most regions of Zealand, which is important when I evaluate the quality of the network in 9 Evaluation.

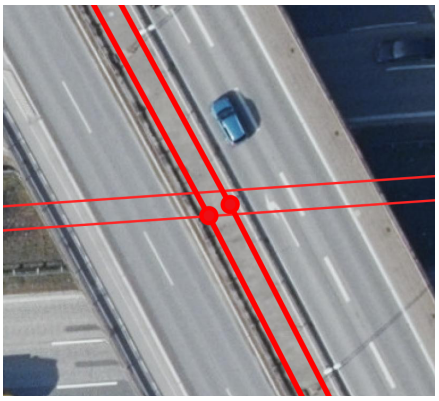
(a) Example of a motorway intersection from the `ws112:hastighedsgraenser` data-set. The paths that will be used for generating the network are shown in red.



(b) Zoomed in view from figure 4.2a with styling similar to the one used in figure 2.1 from 2 Problem description, see figure 2.1a for a description of what the shapes mean.



(c) Further zoomed in view at the crossing seen in figure 4.2b. I highlighted the right ramp to show there is a separation between the two paths leading up/down the ramps from/to the motorway and the path from the bridge is not broken up (has no end points, only mid points at crossing).



(d) Further zoom in view at the bridge seen in figure 4.2b. The highlighted path is the bridge and the underlying path is the road below the bridge. In this instance it is not expected to have a connection with the underlying road

Feature	Value
▾ ws112:hastighedsgraenser	
▾ SIDEOFROAD	B
▸ (Derived)	
▸ (Actions)	
gml_id	hastighedsgraenser.fid--594d71
BESTYRER	0
ADMVEJNR	11
ADMVEJDEL	0
FRAKMT	200698
FRAKM	20
FRAM	698
TILKMT	260000
TILKM	26
TILM	0
CPR_VEJNAVN	Holbækmotorvejen
HAST_VEJSIDE	Begge
KODE_HAST_VEJSIDE	B
SIDEOFROAD	B
ID_HAST_VEJSIDE	0
SIDEOFROAD_ID	0
OFFSET	1
HAST_GENEREL_HAST	130 - Motorvej
KODE_HAST_GENEREL_HAST	130
HAST_LOKAL_HAST	110
HAST_GAELDENDE_HAST	110
HAST_ANBEFALET_HAST	NULL
HAST_VAR_HAST_TAVLER	NULL
KODE_HAST_VAR_HAST_TAVLER	NULL
HAST_BYKODE	NULL
KODE_HAST_BYKODE	NULL
VEJSTIKLASSE	Trafikvej, gennemfart land
KODE_VEJSTIKLASSE	1
VEJTYPESKILTET	Motorvej
KODE_VEJTYPESKILTET	1
HAST_BEMAERKNING	NULL
HAST_SENEST_RETTET	04/08/2006 07:45:54 (Romance
HAST_BRUGER	ssc

(e) Example of the features found in the `ws112:hastighedsgraenser` data-set. Most relevant for expected time of arrival estimations is the column `HAST_GEALDENDE_HAST`, which is the relevant speed limit. The `KODE_HAST_GENERAL_HAST` feature can be used in cases where `HAST_GEALDENDE_HAST` is not provided, otherwise use a default value for speed.

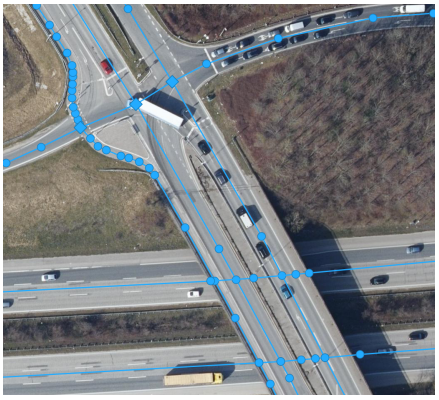
Figure 4.2: Examples of the data-set `http://www.vejman.dk`, with relevant cases for discussion provided in 2 Problem description.

4.2. GeoDanmark60

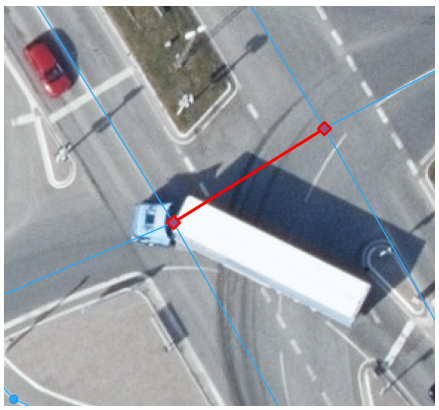
GeoDenmark from `https://dataforsyningen.dk/` is a significantly larger by a factor of ten compared to 4.1 Vejhastigheder with a total of 2.339.320 individual paths. I could not provide a zoomed out view of the data as was shown in 4.1 Vejhastigheder as there is a limit of the amount of data that can be queried and displayed at once. This data-set will be used to test the performance against the 5 QGIS solution as it complies with the expectations that QGIS has (that there is always overlapping vertices at the locations where a connections should be created). Figure 4.3 provides a good visualization of the differences compared to the 4.1 Vejhastigheder. 4.2 GeoDanmark60 provides a lot more detailed paths by increasing the density of mid-points, which in effect better models the roads. The heightened detail for each path also vastly increases the complexity of the problem as the 5 QGIS solution and 6 Proposed solution operate on each path's vertices.

The increase in detail can also introduce unintended connections (see def. 5), as seen in figure 4.3d where the bottom intersection between the road below and above the bridge has visually overlapping mid-points, which according to figure 2.1d is a perfectly legal connection. I will later in 8 Results provide examples of how the 6 Proposed solution goes about handling these scenarios.

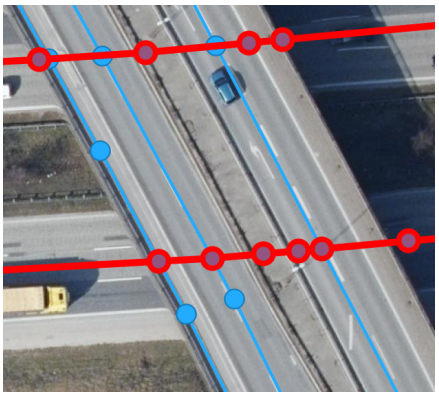
(a) Example of an interchange from the `Vejmidte` data-set. The paths that will be used for generating the network are shown in light blue.



(b) Zoomed in view from figure 4.3a with styling similar to the one used in figure 2.1 from 2 Problem description, see figure 2.1a for a description of what the shapes mean.



(c) Further zoomed in view at the crossing seen in figure 4.3b. The highlighted path shows that each path is separated by end points in contrary to 4.1 Vejhastigheder where it was a mixture of end points and mid points to show an intend for connections.



(d) Further zoom in view at the bridge seen in figure 4.3c. The highlighted paths are the road leading under the bridge, where it is clearly visible that problems may occur with making mid point to mid point connections.

Feature	Value
Vejmidte	
id.namespace	http://data.gov.dk/geodanmark
(Derived)	
(Actions)	
id.namespace	http://data.gov.dk/geodanmark
id.lokalId	1210889358
templD	NULL
status	Anlagt
geometristatus	Endelig
registreringsspecifikation	GeoDanmark Spec 6.0
dataansvar	Ikke tildelt
forretningshaendelse	Nyt objekt
forretningsomraade	1
forretningsproces	Fotogrammetrisk ajourføring
registreringsaktoer	DSM GEODATA (P) LTD
registreringFra	23/07/2020 16:54:14 (Romance S
registreringTil	NULL
virkningsaktoer	SDFE
virkningFra	05/06/2018 02:00:00 (Romance S
virkningTil	NULL
planNoejagtighed	0.10
planStedfaestelsesmetode	Fotogrammetri
vertikalNoejagtighed	0.15
vertikalStedfaestelsesmetode	Fotogrammetri
applikation	GeoDanmark Klient
kommentar	Ajourføring 2018
vejmidtetype	Vej
vejmyndighed	Ikke tildelt
CVFadmnr	0
kommunekode	0169
vejkode	2825
vejkategori	Hovedrute
trafikart	Motorvej
niveau	NULL
overflade	Befæstet
tilogfrakoersel	false
rundkoersel	false

(e) Example of the features found in the `Vejmidte`. The most interesting features for generating a network is `vejkategori` or "road category" and `vejmidtetype` or "road type", as it gives some notion of what the expected speed could be and intended type of traffic on these paths.

Figure 4.3: Examples of the data-set from `https://dataforsyningen.dk/`, with relevant cases for discussion provided in 2 Problem description.

4.3. Observations

The interesting difference to note between the two 4 Data sources, is making an assumption of some constant search-radius to find neighbouring paths does not provide the best result[11] for all both data-sets. As seen in 4.1 Vejhastigheder, the paths consist of two parallel lines, which do not always align their vertices with other paths, they intend to

[11] Best result referring to what is closest to the expected resulting network in terms of connections created

connect to. This results in a gab between the vertices (see figure 4.2c for an example), whereas 4.2 GeoDanmark60 has more accurately placed vertices between the paths. 4.2 GeoDanmark60 also has paths that consists of single lines, hence a smaller radius would be better for this specific data-set. The data-set also seems to introduce overlapping end-points when a connection is intended (example seen in figure 4.3), hence the algorithm should provide parameters to adjust tolerance radius of end-points and mid-points individually, which further to supports hypothesis 1. Providing these parameters should make it possible for the 6 Proposed solution to prevent generation of connections as in figure 4.3d between the roads above and below the bridge. Another example found in 4.2 GeoDanmark60 with a bridge crossing, which has no connecting ramps can be see in figure 4.4.

Figure 4.4: Two more examples where the rule for connecting mid-points (from figure 2.1d) in 4.2 GeoDanmark60 will generate cases where bridges connect to underlying road, which is undesired.

But in the other data-set 4.1 Vejhastigheder, it would seem that there is a need for midpoint connections as seen in figure 4.5. This is also a case for the rule seen in figure 2.1e, where a connection can occur between an endpoint and a segment. This further supports that there is no universal parameters that would work for all data-sets.

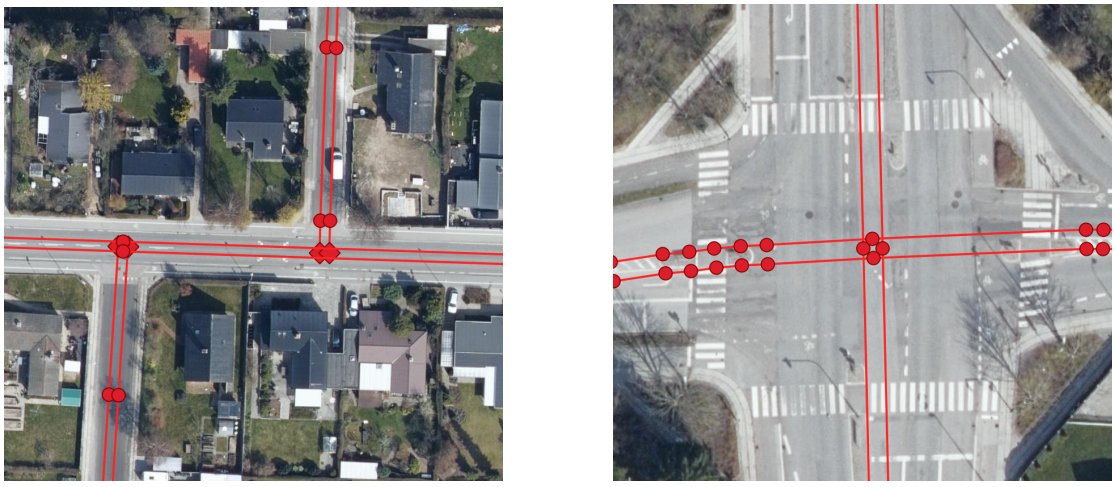


Figure 4.5: Cases from 4.1 Vejhastigheder data-set where the rule for endpoint to line-segment connection (see figure 2.1e) is desired (left). To the right is also an example where a connection between more than two mid-points is desired in the intersection.

To further allow for usability on the data-sets then as discussed in the features provided by 4.2 GeoDanmark60 in figure 4.3e, there is no notion of legal speed limits except for a text in the feature `vejkategori` or "road category" and `vejmidtetype` or "type of traffic". In these feature, the following is what I have deduced:

Observation. ***The text column*** `vejmidtetype` *or "road type" from 4.2 GeoDanmark60 provides clues related to the type of traffic that utilizes the path, where only contains two values, either 'Vej' or road and 'Sti' or path (pedestrian/bicycle).*

Observation. ***The text column*** `vejkategori` *or "road category" from 4.2 GeoDanmark60 can provide approximations to expected speed limits, as it contains information about the type of the road, ex. motorway, large-, medium- or small road.*

Using these observations, then the algorithm should atleast be able to include/exclude paths based on data like `vejmidtetype`, if the user intend to travel only by car. The other column `vejkategori` can provide an approximate speed limit based on the road category by specifying a numerical value for each unique text value in the column.

Vejhastigheder on the other hand provides legally expected and local speed limits for the paths, hence the algorithm should allow parsing of numbers as text when computing the weights. But the data-set has poor precision when it comes to the vertices, which results in a much greater radius required to find possible connections to be made in the final network that can have consequences in terms of unexpected connections, more on this in 8 Results and 9 Evaluation.

Lastly no data column in the two data-sets provides the length of the paths explicitly and cannot be expected to exist already in all data-sets. Therefore the algorithm should be able to perform euclidean distance computations between the vertices, which is quite trivial.

5. QGIS solution

I will be drawing comparison to QGIS, as they have solved a similar problem to the problem this project aims to solve.

QGIS is an open-source project, hence it is ideal to utilize as a reference algorithm as it otherwise would be hard to argue for semantics of the program and the details of the implementation. The code is available at:
`https://github.com/qgis/QGIS/blob/master/src/analysis/network/qgsvectorlayerdirector.cpp`
And is written by Yakushev Sergey. The solution is written in C++ and my own interpretation of the implementation can be found in appendix B.2 QGIS - overview and in more detail in B.3 QGIS - detailed. These pseudo-code interpretations are optional for the remaining project. The most important appendix section for gaining an understanding of the algorithm is B.2 QGIS - overview, while the B.3 QGIS - detailed aims to obfuscate as little as possible of the original 250 lines of code and will be used for the run-time analysis. There will be references to the relevant lines when used during the run-time analysis.

5.1. Query structure

The algorithm utilizes the R-tree query data-structure to find neighbouring points to assist with pairing points when generating the vertices of the network. For the analysis I will be using the following simplified run-times for the operations on the data-structure based on [10].

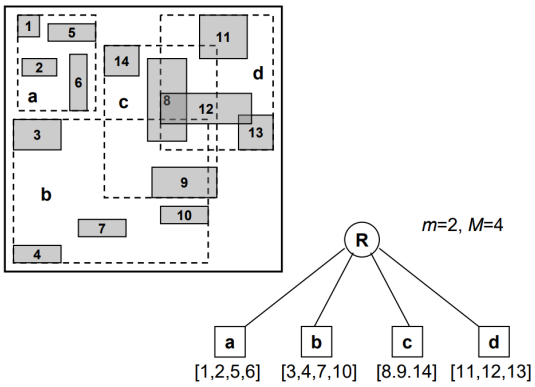
- n is the total number of elements in tree
- m is the threshold for how many elements can be in each per rectangle node, splitting where needed

• Insert/Search; Average: $O(\log_m n)$ Worst:$O(n)$

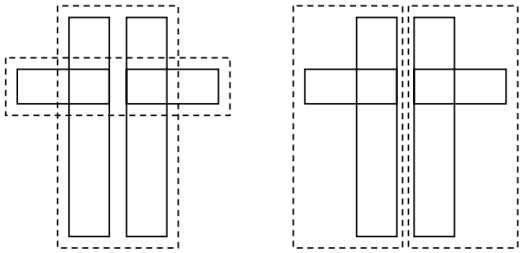
The main function that is used their implementation is:

`Rtree.Query` Provides closest point around the provided point (Point, 1st parameter) within the tolerance radius (Float, 2nd parameter). Returns the index of the closest point of where it can be found in the vertex array.

What is it?: R-trees or rectangle trees attempts to encapsulate data in rectangular nodes. Each node has a capacity M and after that threshold is overflown, then it will split the current node and link the children, which each has a size of atleast $m \in [0, M/2]$ (therefore atleast 2 children) that is contained in the parent node. As the expectation for each node to have atleast m nodes and $m < M$, then it provides us with $O(\log_M n) \leq O(\log_m n)$[10]. The source [10] also provides some great figures for visualizing the R-tree behaviour as seen in 5.1a.



(a) Example of how an R-tree stores its content, note there is no duplicates as any search query overlapping multiple children would have to also get those children's content.



(b) Example of how to query optimization problems with rectangle trees as overlap exists between nodes on the same layer of the tree unlike other query structures like quad-trees as they have well-defined non-overlapping splits at each layer. Either case above is not great as the one to the left provides a chance for $O(n)$ worst case run-time, if we query an area that overlaps throughout every layer. The other one on the right increases the area of the R-tree nodes, making it more likely to include more nodes in the search query.

Figure 5.1: Visualization of an R-tree, as can be seen, it looks very typical for a tree-like structure. Figure source: [10]

5.2. Analysis

The algorithm segregated a of the work into smaller pre-processing steps performed. There is a couple of important symbols for the analysis, which follows:

V_{paths} = #Number of vertices introduced from all paths of the layer
V_{add} = #Input parameter, Points to be added to the graph by closest distance to paths
V = #Final number of vertices that is part of the network

As I focus on the path connections to determine the quality of the network generated, then I will not be working with V_{add} as part of the analysis as I will be generating networks based on just paths and their points. Hence:

$$O(V_{add}) = 1$$
$$O(V_{paths}) = O(n)$$
$$O(V) \leq O(V_{paths})$$

The reason for the relationship $O(V) \leq O(V_{paths})$ is as we connect the points of the original paths, then we reduce the total number of vertices in the final network. Worst case for $O(V) = O(V_{paths})$ is when there is no paths in close proximity of each other resulting in the original layer (see def. 3), where you can only traverse individual paths, but never cross between them. The 5.2.1 QGIS pseudocode breakdown will be using these variables for the run-time analysis.

5.2.1 QGIS pseudocode breakdown

A breakdown of the QGIS's solution from B.2 QGIS - overview goes as follows:

`UpdateTiePoints` Helper function seen in B.4 QGIS - helper functions. This function runs over all additional points and updates the tie-points in constant time if better candidate is found, hence this runs in $O(V_{add})$. As discussed in 5.2 Analysis, then this is equivalent to $O(1)$ as I will not be introducing additional points into the network during generation in my algorithm at 6 Proposed solution when comparing.

`GetPaths` Helper function seen in B.4 QGIS - helper functions. This function is only seen in B.3 QGIS - detailed as part of the first outer loop to extract the relevant type of the input geometry. It is mostly used for casting to the correct type (paths) and the inner loop over path points between line 10-21 in B.3 QGIS - detailed will still run over all path vertices, $O(V_{paths}) = O(n)$, in the end. Hence I consider this as constant time $O(1)$ as it does not change the work done inside the inner loop.

line 3-12 This is the initial pre-processing step, where all the path points are added to the R-tree without adding multiple overlapping points within the tolerance radius (more detail can be read at line 11-17 in B.3 QGIS - detailed), where they reuse existing vertices if it is within the tolerance for what is considered proximate, essentially providing vertex pairing to connect paths.

```
1
2   ...
3   foreach Path path in paths do // Inner path loop
4       pt1 ← NULL, pt2 ← NULL;
5       isFirstPoint ← true;
6       foreach Point p in path do // Inner points of path loop
7           Insert p into R-tree unless other exists within tolerance;
8           pt2 ← Vertex inserted or found by R-tree;
9           if not isFirstPoint then
                // Helper function, Algorithm 5, run-time O(V_add)
10              V_tie ← UpdateTiePoints (path.id, pt1, pt2, V_add, V_tie);
11          pt1 ← pt2;
12          isFirstPoint ← false;
13  ...
```

The cost of the body in the loop is the query and insertion into the R-tree, which as discussed at the start of 5.1 Query structure, resulting in $O(\log_m n)$ for the query and $O(\log_m n)$ for the insertion. Hence we end up with a total cost of $O(2n \log_m n) = O(n \log_m n)$ for this processing step. More detailed version can be seen on line 5-21 in B.3 QGIS - detailed.

line 13-14 This loops inserts all tie points into the R-tree from V_{tie} found from the previous step, which always has the size of $|V_{tie}| = |V_{add}|$. But as discussed at the start of 5.2 Analysis, I will consider V_{add} to be $O(1)$, hence the operation of this loop is $O(1)$ (zero times looped).

```
11
12  ...
13  foreach Point p in V_tie do
14      Insert tie-point p into R-tree unless other exists within tolerance;
15  ...
```

More detailed version can be seen on line 22-28 in B.3 QGIS - detailed.

line 16-35 This is the final processing step for the graph, similarly to line 3-12, loops over all path vertices. Inside the body of the inner loop, a search query to the R-tree is performed costing $O(\log_m n)$ once and then the edge is added between the last two points

```
14
15  ...
16  foreach Path path in paths do // Outer path loop
17      pt1 ← NULL, pt2 ← NULL;
18      isFirstPoint1 ← true;
19      foreach Point p1 in path do // Inner points of path loop
20          pt2 ← Get vertex from R-tree within tolerance of p1;
21          if not isFirstPoint1 then
22              ordered_segment ← Order tie-points in segment (pt1→pt2);
23              v1 ← NULL, v2 ← NULL;
24              isFirstPoint2 ← true;
25              foreach Point p2 in ordered_segment do
26                  v2 ← Get vertex from R-tree within tolerance of p2;
27                  if not isFirstPoint2 then
28                      if path.Direction = forward or both then
29                          Insert edge (v1 → v2) into E;
30                      if path.Direction = backward or both then
31                          Insert edge (v2 → v1) into E;
32                  v1 ← v2;
33                  isFirstPoint2 ← false;
34          pt1 ← pt2;
35          isFirstPoint1 ← false;
36  ...
```

Summary: The conclusion to each processing step we have (reminding that $n = |V_{paths}|$ and $O(V) \leq O(V_{paths})$ from the start of 5.2 Analysis):

line 3-12 : Pre-process paths, insert all points from the paths into R-tree and find tie-point candidates.

$$O(n \log_m n)$$

line 13-14 : Insert found tie-points into R-tree, see 5.2 Analysis for reason.

$$O(1)$$

line 16-35 : Generate edges using the combined R-tree of path points and tie-points.

$$O(n \log_m n)$$

We can then conclude, the path conversion to a network is done using a overall average run-time of $O(n \log_m n)$ with absolute worst-case run-time of $O(n^2)$ when querying a region of the R-tree that intersects all sub-regions as discussed in 5.1 Query structure.

5.3. Improvements

As discussed in 2 Problem description and reinforced by the 4.3 Observations, specifically figure 4.5, then there exists desired connections (see def. 5), that should be created between an *end-point*s and line-segments (as seen in figure 5.2). By hypothesis 1, then this should only apply for end-points (vertex of degree 1) and not mid-points (degree 2) as was shown in figure 4.2d, where mid-points are simply used as added detail to the underlying road of a bridge. Adding a connection between an end-point and a line-segment

is not currently supported in QGIS's implementation as it only considers pairing vertices within a certain tolerance(/radius).

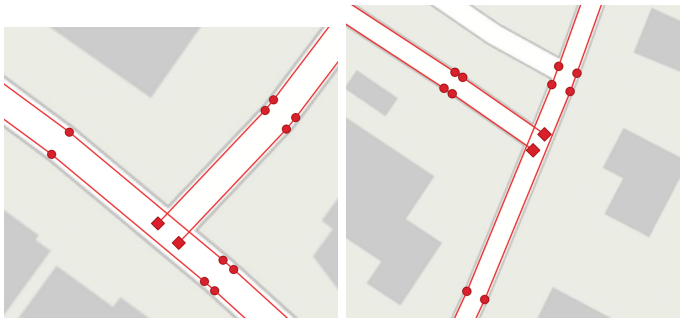


Figure 5.2: Extra examples of situation where endpoints (diamonds) intersect a line-segment from 4.1 Vejhastigheder data-set and a connection is desired, but QGIS's solution fails as it only matches vertices.

Therefore I will be looking into methods of allowing such connections (as seen in figure 2.1e) between an end-point and a line-segment.

6. Proposed solution

This section will cover important considerations in the design of the algorithm that provides the improvements mentioned in 5.3 Improvements, where I will later compare varying sub-sets from the 4 Data sources in 8 Results and evaluating the speed and quality[12] of the produced network in 9 Evaluation.

As the data-sets path structures are not completely arbitrarily constructed, then I will impose the following assumptions based on my observations of the 4 Data sources and biased generalizations of similar data-sets that have paths, which could be used for constructing networks.

Assumption 1. ***Paths can intersect other paths****; Edges of a path may intersect/overlap other paths' edges and vertices.*

Assumption 1 is based on the existence of bridges as discussed in 4.3 Observations and the stark contrast between the two data-sets in 4 Data sources. 4.1 Vejhastigheder has a lot of cases where edges overlap and the inverse seen in 4.2 GeoDanmark60 with end-points being matched and rare cases of overlapping edges (mostly for bridges, as was shown in figure 4.3 or 4.4).

Assumption 2. ***Paths are pseudo-directional****; Starting at an end-point and traversing each mid-point along a path, then the angle between the current and next edge is somewhere between* $90°$ *degrees to the left or right in the direction of travel and when traversing, it generally increases the distance from the starting point.*

Assumption 2 stems from observations of behaviour in the 4 Data sources' paths.

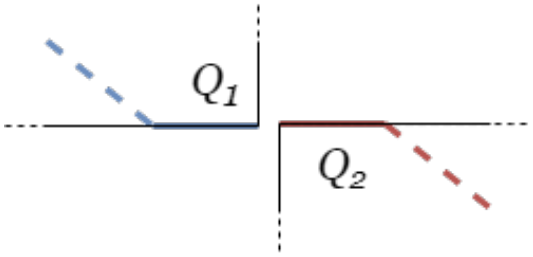
These assumptions of course do not hold in all scenarios or data-sets and should be thought of as a generalization of data-sets, which are good candidates for generating networks (such as roads, cable networks, pipes etc.), but the assumptions will lay the foundation for how I will evaluate pros and cons of certain decisions for my 6 Proposed solution. More on how the assumptions apply in the following 6.1.2 Range searching algorithms and 6.1.4 Simplifying paths.

6.1. Exploiting locality

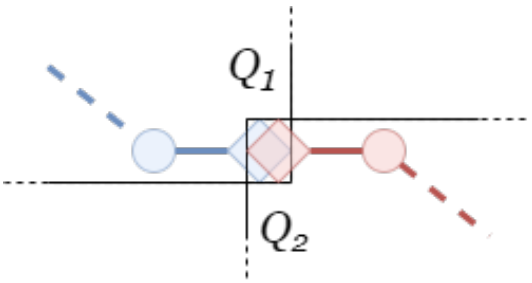
When generating the network, it is expected to only create connections (see def. 5) between paths that are in close proximity. This can be exploited to prevent checking every single other path when adding vertices or edges to the network, as it is much cheaper to only consider the neighbouring paths.

It is possible to query these nearby paths by considering an inflated rectangular query region for each path based on the discussion about how to obtain connectivity in 2 Problem description, where a certain radius around each end-/mid-point is used to determine the amount the rectangle query should be inflated. A visualization for reasoning can be seen in figure 6.1.

[12] Refers to the cases where connections are undesired, see the discussion of the two data sets found in 4.3 Observations



(a) Example of a rectangular query area for a path where there is no radius around the end-/mid-points. It is seen that Q_1 and Q_2 do not intersect and would hence not find each other as a potential match.



(b) Proposed version with the same scenario as (a), where a radius is applied as an inflation to the rectangular query. This allows for an intersection between Q_1 and Q_2, which is similarly to what 5 QGIS solution did by adding a tolerance parameter around each vertex

Figure 6.1: Examples of how inflating query rectangles (Q_1 and Q_2 above) are important when attempting to match paths, as having a tight boundary box (a) may result in missing connections in the network, where inflating the query rectangle (b) by a radius or tolerance (as it is referred to by QGIS), then it can ensure neighbouring paths to be returned when queried.

In the Computational Geometry course, we were introduced to range or interval search data-structures, such as kd-, range-, interval-, priority search- and segment-trees, which will be covered in 6.1.2 Range searching algorithms. These data structures have different use-cases and requirements for the input and is therefore important to consider the problem of generating the network and what cases that should be solved.

The cases that should be covered by the algorithm is discussed in the following section 6.1.1 Desired connection cases and which data-structure can support the cases in 6.1.2 Range searching algorithms after discussing the range searching algorithms.

6.1.1 Desired connection cases

The simplest legal cases described in figure 2.1(b,c,d) from 2 Problem description(also seen in following figure 6.2 as 1, 2 and 3), where connections are made from matching vertices only (end- to end-, mid- to end-, end- to mid- and mid- to mid-point connections), which is exactly what the 5 QGIS solution performs using an R-tree. QGIS creates the connections by adding boundary boxes around each vertex before adding them to the R-tree to check for existing vertices.

But it is not so trivial to introduce the connection seen in 2.1e or case 4 in figure 6.2, where an end-point vertex is matched with a line-segment. This where interval searching algorithms would be interesting, but as will be discussed later, interval-, priority search- and segment trees in 6.1.2 Range searching algorithms all provide certain restrictions as to how the query is performed, that makes rectangular queries difficult by assumption 1 (overlapping edges allowed).

Therefore due to the restrictions and how the data-set is constructed, then I will just be focusing on the range query algorithms and below is how I plan to solve each case for a connection (see def. 5) to be added to the network from 2 Problem description. In figure 6.2 is seen a summery of the types of connections that the algorithm should support.

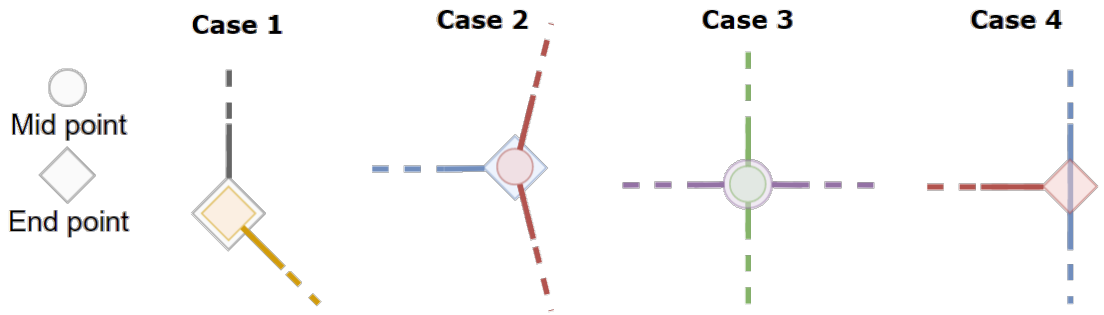


Figure 6.2: A summery of the cases from 2 Problem description, which are considered "legal" when adding connections to the network.

In case 1-3 from figure 6.2 it is easily solved similarly to how the 5 QGIS solution did it by querying an inflated rectangle query around each vertex, whereas case 4 is not so trivial as there is no vertex to be matched with the end-point. The easiest way to find vertices that are close enough to any given path is by querying an inflated rectangle that envelops the entire path, which will return any other vertices that might intersect a line-segment along the path, see figure 6.3 for an example.

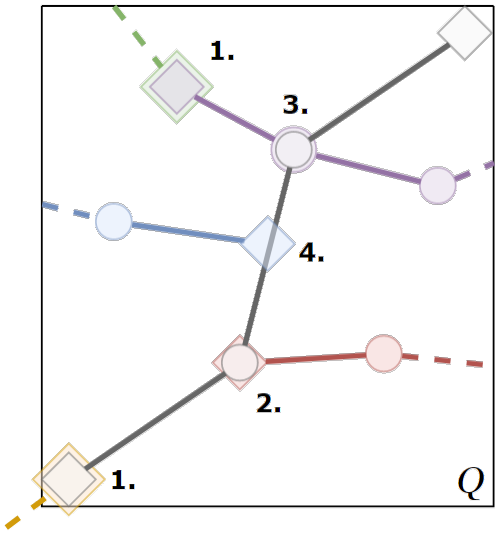


Figure 6.3: An example an inflated rectangular query around the black path, Q (inflated by the radius/tolerance of the end-/mid-points). The query will return the vertices of the other paths, that are contained in the window and the case numbers for each type of condition is shown based on figure 6.2.

Further discussion on how this rectangular query can be optimized to reduce number of vertices that has to be checked for roads will elaborate on in 6.1.4 Simplifying paths based on assumption 2.

6.1.2 Range searching algorithms

To solve which data-structure to use from Computational Geometry[8] course for the cases mentioned in 6.1.1 Desired connection cases, where a rectangular query is required, then each data-structure has to be evaluated if it is applicable. The following descriptions of each data-structure is based on [8] and is meant to loosely give a perspective on the time-complexity, space-complexity and use-case:

kd-tree kd-trees is a range search data-structure that works by partitioning space into (left, right) and (top, bottom) for even and odd layers of the tree. An example of such a tree can be seen in figure 6.4

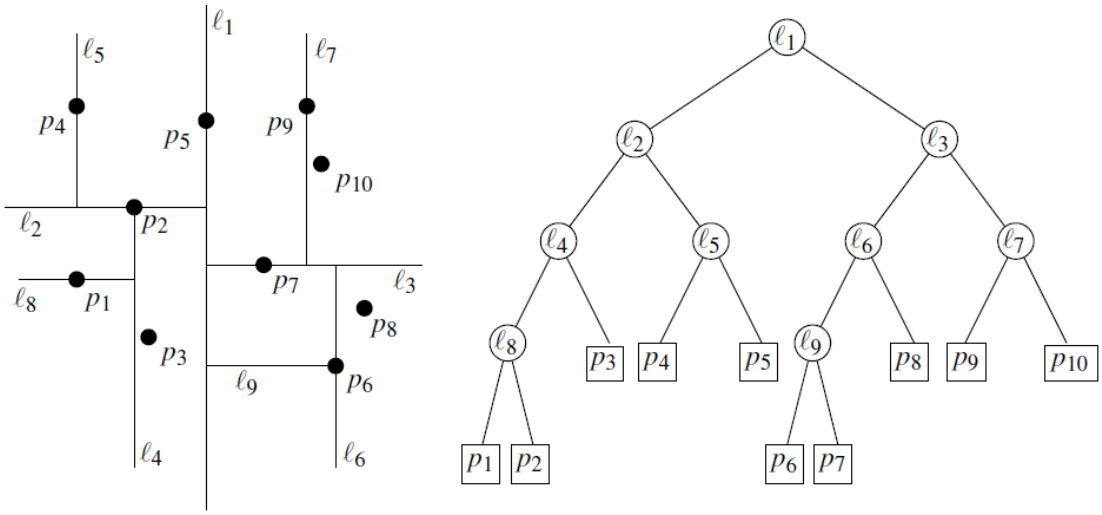
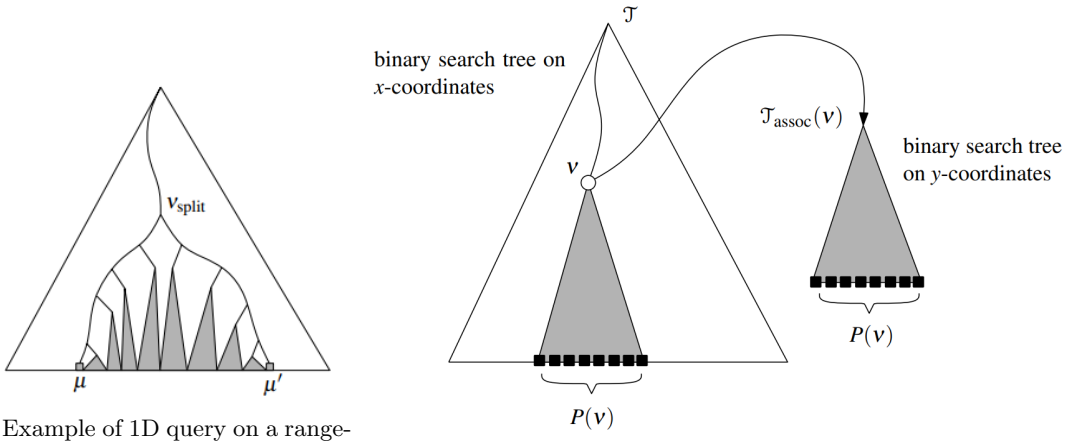


Figure 6.4: Example of kd-tree structure taken from [8] at page 103 figure 5.3, where it is shown how the each partition ℓ_x has a left and right sub-tree and each layer is either (left, right) or (bottom, top) interchanging for even/odd layers. An example is ℓ_2 partitions the space on the left of ℓ_1 and ℓ_3 partitions the right side of ℓ_1 as seen in the tree. This is then repeated, but traversal is rotated, s.t. ℓ_4 is the "left" or bottom of ℓ_2 and ℓ_5 being at the "right" or top of ℓ_2. This process is repeated till a threshold of points per region is obtained.

According to Theorem 5.5 from [8] page 105, the construction of a kd-tree takes $O(n \log n)$ time, can be stored in $O(n)$ space and the query time is $O(\sqrt{n} + k)$, k being the number of elements returned within a rectangle query and $O(\sqrt{n})$ being the expected number of nodes that has to be visited.

Range-tree Range-trees require axis-aligned queries and it consists of ordered binary searches in both x and y coordinate. To query the structure, see figure 6.5, where it traverses the binary search tree and finds all x-coordinate nodes that are contained in the query interval, then it proceeds into an associated binary tree querying the y-coordinate within the query range.



(a) Example of 1D query on a range-tree taken from [8] at page 106, where it reports all leaves in sub-trees that are contained in the interval μ to μ'

(b) Example of range-tree structure for 2D taken from [8] at page 107 figure 5.6, where instead of reporting all leaves, it will run through the y-coordinate BST(binary search tree), that are associated at each node in the x-coordinate BST

Figure 6.5: Examples of range tree querying. The left (a) shows a 1D query, where right (b) shows the associated range-tree at each node that allows for 2D queries.

According to Theorem 5.8 at page 109 [8], then the time for construction of a 2d range tree is $O(n \log n)$ and uses $O(n \log n)$ storage, while the query time is $O(\log^2 n + k)$, k being the number of reported leaves and $\log^2 n$ due to having to search each associated binary search tree structure of the top-level reported x-coordinate nodes (as seen on the right). Compared to the kd-tree, this increases the space requirement to provide a better large-scale[13] run-time improvement from $O(\sqrt{n} + k)$ down to $O(\log^2 n + k)$[14]. The reason for the increased space requirement is the associated y-coordinate BSTs(binary search tree) at each level of the x-coordinate BST. Since the tree is $\log n$ tall and each layer will contain a non-overlapping y-coordinate binary tree, then we need $O(n)$ storage per layer, hence end up with $O(n \log n)$ space requirement.

[13] Comparing growth-rate of $\sqrt{n}$ and $\log^2 n$, then $\sqrt{n}$ is better till $n = 2^{16} = 65536$ elements, where $\sqrt{65536} = \log^2(65536) = 256$, hence it is not a significant improvement on smaller data-sets.

[14] This can be improved using the method described in chapter 5.6 Fractional Cascading in [8] to $O(\log n + k)$ for 2 dimensions by utilizing a more pragmatic approach, that stores pointers to an associated data-structure

Interval trees This structure is useful for systems consisting of axis-aligned line-segments, where it is desired to find all line-segments that intersect a stabbing[15] query point. An example of the structure can be seen in figure 6.6.

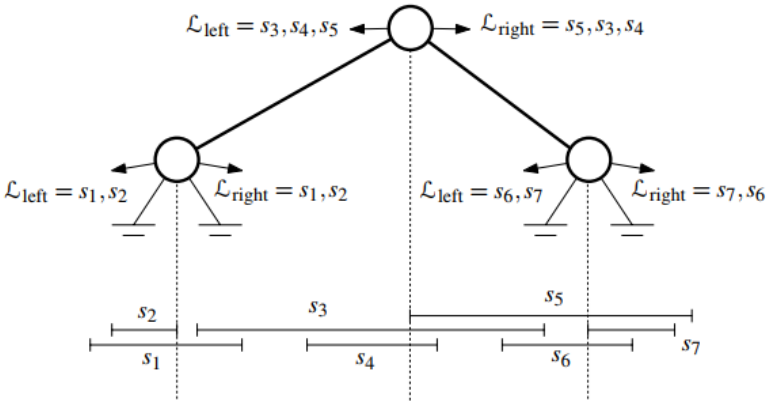


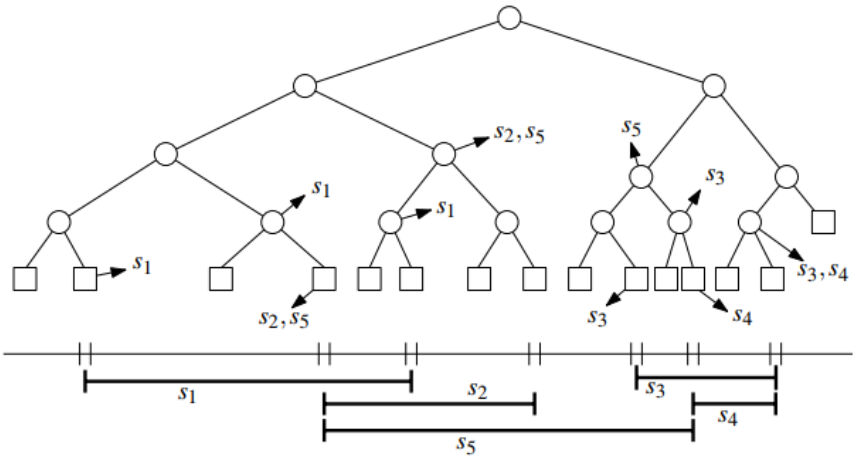
Figure 6.6: Example of interval-tree structure taken from [8] at page 222 figure 10.3. Traversal through the tree is similar to any other binary search tree, where the coordinate of each node is compared to the stabbing query point till a leaf is reached. While traversing, add any segments that are stored in the left or right path chosen at each node. The 1D example above hence uses $O(n)$ storage as it at most stores each segment twice (once for each end-point) [8] (lemma 10.2, page 223)

As it is a stabbing problem, it will return all segments that intersect a perpendicular query, hence a secondary structure of range-trees as previously mentioned can be used as an associated data-structure for intersecting lines returned.
According to Theorem 10.5 at page 226 [8], this 2D rectangle query implementation of interval trees would require $O(n \log n)$ to be constructed, $O(n \log n)$ in storage due to the introduced range-trees and $O(\log^2 n + k)$ query time.

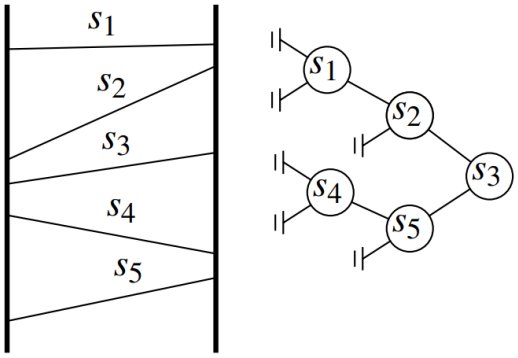
Priority search tree Priority search trees provide a method to report all points within a one-side unbounded rectangle query, ex. $[q_x, +\infty) \times [q_y, q'_y]$ or $(-\infty, q_x] \times [q_y, q'_y]$, hence it is not awfully useful for bounded rectangle queries, which is needed for my algorithm.

Segment trees Segment trees attempts to provide a solution to the arbitrarily oriented line-segments. This is done by simplifying the problem to axis-aligned segments by using an enveloping boundary box and its bounds for each line-segment. An example is seen in figure 6.7, where it is shown how the segment-tree is constructed along one axis of the boundary box.

[15] Referring to a query line that is perpendicular with the axis the intervals are on, hence you "stab" through all the intervals



(a) Example of a segment-tree taken from [8] at page 233, figure 10.9. As seen it works the same as a binary search tree, where every node traversed concatenates their overlapping segments till the query leaf node is reached.



(b) Example taken from [8] at page 236, where the slap returned from the segment tree can be queried using a range tree as the structure assumes no overlapping lines, hence they can be ordered.

Figure 6.7: Examples of segment tree querying and finding the intersecting lines of the vertical query of the returned slab/segment using range-trees

When querying the segment tree that spans the x-axis, then all segments intersecting a vertical query q_x is returned, where the segment (also known as vertical slab) $\textbf{Int}(v) = [x, x']$ can be searched using a range tree of the lines returned *assuming they do not overlap* [8] (page 231) as they can then be ordered. If line-segments overlap, like it can under the assumption 1 and shown in 4.3, then it is not a trivial problem to find the intersecting lines in the returned slab of lines.
According to Theorem 10.13 at page 237 [8], then the segment tree can be constructed in $O(n \log n)$ time and uses $O(n \log n)$ space. The query time to find intersecting lines of a 2D query is $O(\log^2 n + k)$ as it is using a secondary range-tree structure.

Quad-tree A supplementary data-structure from our book in [8] for range searching is the quad-tree, where I will consider the uniform quad-tree. Similarly to the R-tree the 5 QGIS solution used, it splits the plane into multiple layers of smaller and smaller tiles, but does so using non-overlapping uniform tiles, where R-trees does not guarantee non-overlapping children at the same layer. Example of such a split can be seen in figure 6.8.

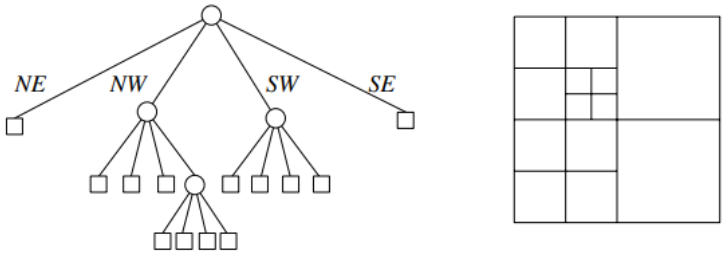


Figure 6.8: Example of uniform quad-tree structure taken from [8] at page 310 figure 14.3. Each time a node splits given a threshold of points is reached, it gets four (hence quad) evenly split children, which represents either NE (North-East), NW (North-West), SE (South-East) or SW (South-West) sector of the parent.

There are multiple implementations for quad-trees that have different splitting conditions and the depth is therefore not guaranteed. But for any quad-tree, one

can provide an upper-bound for the depth as described in [8] in Lemma 14.1:

$$d \leq \log\left(\sqrt{2}\frac{s}{c}\right) + 1 = \log\frac{s}{c} + 3/2$$

Which states the depth, d, is proportional to the side-length of the root boundary box, s, and the smallest distance between any two points, c, which is bounded by the smallest tile diagonal $c \leq \sqrt{2}\frac{s}{2^d}$ contained in the quad-tree. Isolating d and adding 1 for the root node, we get lemma 14.1.

6.1.3 Choice of range search algorithm

As described in the last two sections 6.1.1 Desired connection cases and 6.1.4 Simplifying paths, then a rectangular query of the vertices is desired to cover all cases, where a connection (see def. 5) is desired. As mentioned in 6.1.2 Range searching algorithms, not all range searching algorithms can perform such a query, atleast not without imposing restrictions on the lines. It was discussed that interval-, priority search- and segment trees are not fulfilling of the criteria of bounded rectangular query on data that allows for intersecting lines.

This leaves kd-trees and range-trees, which of the two, range-trees provides the best search time complexity of $O(\log^2 n + k)$ and at the cost of $O(n \log n)$ space against kd-tree's $O(\sqrt{n} + k)$ queries and $O(n)$ space. As this algorithm is intended to be used on large datasets then I will be using range-trees to query vertices using inflated boundary boxes of each line-segment.

I will also be implementing the same solution using the supplementary Quadtree as part of the presentation[16]. As discussed at the end of 6.1.2 Range searching algorithms, quadtrees are quite easy to implement and have some interesting properties for their height time-complexity.

[16] But not as part of this report, as it is not part of the original scope, as I planned to used curriculum of Computational Geometry to provide a solution

6.1.4 Simplifying paths

I will be using the inspiration from the 5 QGIS solution to split each path into its line-segments, as it may by assumption 2 increase the number of queries, but will also greatly reduce the total area queried when performing a boundary box query as visualized in figure 6.9. The ideal is to minimize k from a rectangular search query running in $O(f(n)+k)$ time of a range search data-structure, where $f(n)$ is some function scaling with n ex. $f(n) = \log n$.

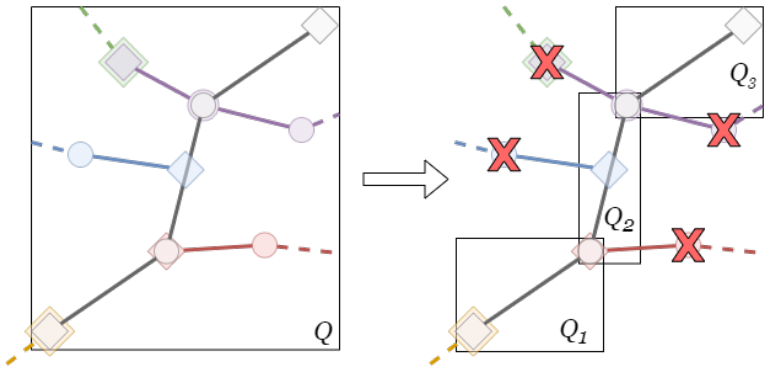


Figure 6.9: Example of an inflated rectangle Q (left), that envelops the path and how splitting the path into line-segments can greatly decrease the total area that has to be queried as seen for queries Q_1, Q_2 and Q_3, making more path-dense areas such as cities cheaper as the total query area is reduced.

But there exists degenerate cases where roads may zigzag diagonally of the axes as seen in figure 6.10. This diagonal zigzagging can cause a larger complexity as the split queries, Q_1, Q_2 and Q_3, have the same area as the boundary box of the full path, but can get repeated results due to overlap.

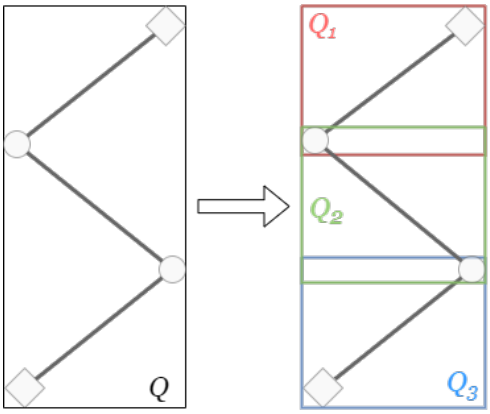


Figure 6.10: Example of how splitting the path into line-segments can increase total area queried as the split queries Q_1, Q_2 and Q_3 fetches the same area as the original path boundary box, but can provide repeated returned results due to the overlapping regions between ($Q_1 \cap Q_2$) and ($Q_2 \cap Q_3$) during generation.

The worst scenario when splitting paths into sub-queries on the line-segments, is when it goes against assumption 2 of pseudo-directional paths as seen in figure 6.11, where paths may have very sharp corners or decrease the distance between start and endpoint while traversing.

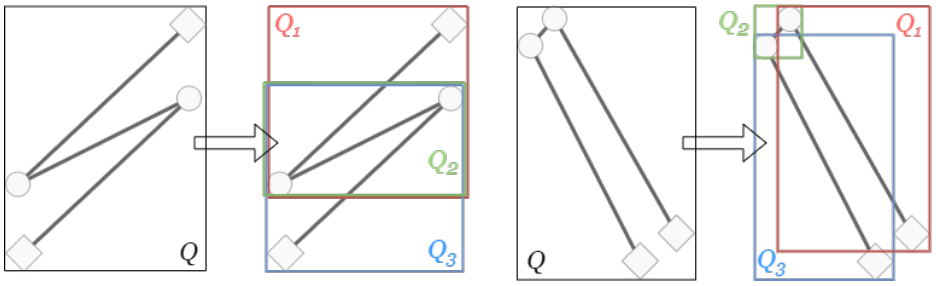


Figure 6.11: Example of how splitting the path can significantly increase the area queried in comparison to the original Q due to the overlapping regions of ($Q_1 \cap Q_3$), ($Q_1 \cap Q_2$) and ($Q_2 \cap Q_3$)

The queries seen in figure 6.11 has a much greater area than a single query of the entire path. But despite this, I will assume the cases seen in figure 6.10 and 6.11 to be negligible in comparison to the "good" cases seen as in figure 6.9 by assumption 2, where a large reduction in the queried area is achieved by splitting into line-segments.

Another consideration when querying the entire path and getting all other points within the boundary box, is that not all points within the bounds are relevant for every line-segment on the path. An example can be seen on the left of figure 6.9, where each vertex on the blue path would have to be matched with the closest line-segment of the path. This problem of finding the relevant segments per found vertex can quickly become a $O(V_{path} \cdot k)$ problem, where V_{path}[17] is the number of vertices of the path the boundary box envelops and k being the other found vertices within the bounds. This is due the having to check which segment in V_{path} is relevant for each k vertices.

But when querying each split line-segment (V_i, V_{i+1}) where $i \in [1, 2, \ldots, |V_{path}| - 1]$, then it will always contain relevant points as they are either:

1. Found vertices are close enough to be considered for a connection to either endpoint V_i or V_{i+1} of the line-segment.

2. Or found vertices are able to be projected onto the line-segment (V_i, V_{i+1}) for check of case 4 from figure 6.2.

Hence by splitting the paths into sub-queries simplifies the problem and provides the guarantees as listed above, where it could also be expected to see a performance boost in practice due to the reduction of k results as seen from figure 6.9[18].

I will be drawing a comparison to the area queried in total when splitting paths into their segments or querying entire paths once in 8 Results to support my assumption 2.

[17] Note: I am not using E_{path} despite case 4 for connections because a path is a chain graph, hence $|E_{path}| = |V_{path}| - 1$, which implies $O(E_{path}) = O(V_{path})$

[18] Purely speculative, but it is also seen in other problems, such as the discussion between merge-sort and quick-sort algorithms to order a list of comparable elements, where both boast a time-complexity of $O(n \log n)$, but quick-sort is generally found to be faster on average due to the a smaller constant footprint per step[1].

6.2. Order of operations

As also seen in 5 QGIS solution it might be beneficial to split the generation of the network into multiple steps, allowing for simpler operations. In QGIS, it inserted all non-overlapping vertices into the R-tree to trivially pair vertices in the main generation of the edges in the network.

As I will be using an inflated rectangular query for each line-segment to solve case 4 from figure 6.2, then it would be beneficial for all vertices to already be in the range-tree prior to attempting to query. Hence the order of operations that leads to being able to generate the network is as follows:

1. Pre-process: For each end-point/mid-point.

   - Query range tree to check for existing vertex
   - If existing vertex, check if it needs to be upgraded (midpoint → endpoint)
   - Else add vertex and their type (endpoint or midpoint) to range-tree

   This ensures non-overlapping vertices in the range-tree for the following operations.

2. Main loop: For each line-segment (pairs of end-point/mid-points).

   - Query relevant vertices within inflated boundary box of the line-segment using range-tree from step 1.
   - Solve any points that is close enough to be considered connected to the segment (if end-point only).
   - Sort the found points added to the line-segment by distance from start vertex of line-segment.
   - Add edges between each vertex of the line-segment.

3. Return the network of the vertices from step 1 and the edges from step 2.

In the following section 6.3 I will be providing a more concrete example of the procedure.

6.3. Pseudocode

Algorithm 1: GenerateGraph(List of Path, Float, Float) → Graph

Input: `paths`(List of Path); All paths contained in a layer.
Input: `endtol`(Float); Tolerance distance for a end-point connections.
Input: `midtol`(Float); Tolerance distance for a mid-point connection.
Output: Graph; Navigable network

```
1  Range_tree ← Instantiate Range-tree for vertex pairing;
2  V ← ∅;
   // List to pair point id to a vertex id
3  V_lookup ← ∅;
4  vertex_id ← 0, point_id ← 0;
5  foreach Path path in paths do // Outer path loop
6      foreach Point p in path do // Inner points of path loop
7          tolerance ← endtol if p is end-point else midtol;
8          ext_p ← Query Range_tree for vertices around p within tolerance;
9          if ext_p > -1 then
10             if p is an end-point then Upgrade V[ext_p] to an end-point; ;
11             V_lookup[point_id] ← ext_p;
12         else
13             Insert p to Range_tree;
14             Push p into V;
15             V_lookup[point_id] ← vertex_id;
16             vertex_id ← vertex_id + 1;
17         point_id ← point_id + 1;
   // Generating the actual graph using Range_tree
18 E ← ∅ // Final edges in the graph;
19 point_id ← 0;
20 foreach Path path in paths do // Outer path loop
21     pt1 ← NULL, pt2 ← NULL;
22     isFirstPoint ← true;
23     foreach Point p in path do // Inner points of path loop
24         pt2 ← V_lookup[point_id];
25         if not isFirstPoint1 then
26             pts ← Get vertices in inflated bounds of (pt1,pt2) from Range_tree;
27             Remove any mid-points, pt1 and pt2 from pts;
28             foreach Vertex v in pts do
29                 if v is between pt1 and pt2 and within tolerance then
30                     Insert v to line-segment (pt1,...,pt2) in order;
31             Push edges based on segment (pt1,...,pt2) to E
32         pt1 ← pt2;
33         isFirstPoint ← false;
34         point_id ← point_id + 1;
35 return Graph(V, E);
```

6.4. Pseudocode breakdown

A breakdown of the proposed algorithm from 6.3 Pseudocode goes as follows:

line 5-17 The pre-processing step adds all non-overlapping vertices to the range-tree based on their corresponding tolerance for their type (mid- or end-point). This is very similar to how B.2 QGIS - overview does it, except for the separation of mid-/end-points, which is heavily used in the main loop between line 20-34. To optimize a little on the QGIS's method, then this algorithms stores a lookup array V_{lookup}, that maps all point ids to its corresponding vertex id found. This spares having to query the tree when matching vertices for cases 1, 2 and 3 discussed in 6.1.1 Desired connection cases, in the main loop (line 20-34).

```
4  ...
5  foreach Path path in paths do // Outer path loop
6      foreach Point p in path do // Inner points of path loop
7          tolerance ← endtol if p is end-point else midtol;
8          ext_p ← Query Range_tree for vertices around p within tolerance;
9          if ext_p > -1 then
10             if p is an end-point then  Upgrade V[ext_p] to an end-point;
11             V_lookup[point_id] ← ext_p;
12         else
13             Insert p to Range_tree
14             Push p into V
15             V_lookup[point_id] ← vertex_id;
16             vertex_id ← vertex_id + 1;
17         point_id ← point_id + 1;
18 ...
```

Assuming the use of a range-tree with fractional cascade behavior, then it is expected to take $O(n \log n)$ to perform a search query ($O(\log n)$) followed by an insert ($O(\log n)$) per iteration of n points.

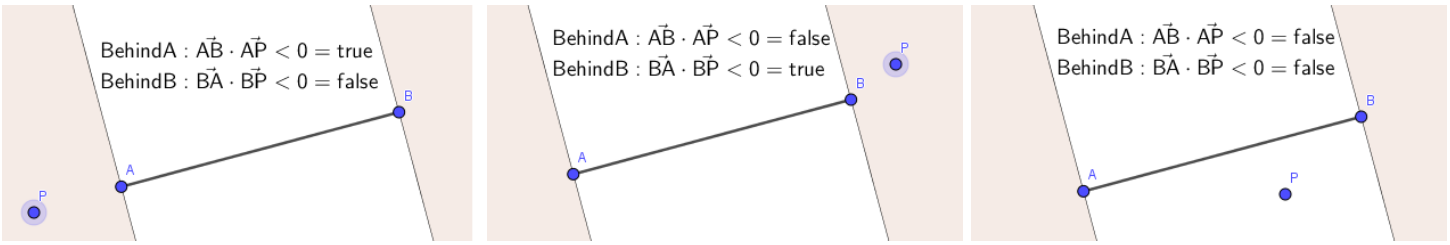
line 20-34 The main loop, where the edges are generated. Each vertex is paired as segment (`pt1`, `pt2`) as discussed in 6.1.4 Simplifying paths, which is used for querying any relevant *end-points*(line 29) that lies between `pt1` and `pt2`. To get the vertices that are relevant, each points' type (`pt1`, `pt2`) is checked and then inflates the bounding box around the segment by their tolerances accordingly (different tolerances for mid- and end-point). A list of other vertices between (`pt1`, `pt2`) is then received in `pts`, that should be projected onto the line-segment, then sort them in order of distance from `pt1`. Use an ordered list of satisfying vertices between (`pt1`, `pt2`) and add edges from `pt1` till `pt2` is reached.

```
21 ...
22 foreach Path path in paths do // Outer path loop
23     pt1 ← NULL, pt2 ← NULL
24     isFirstPoint ← true
25     foreach Point p in path do // Inner points of path loop
26         pt2 ← V_lookup[point_id]
27         if not isFirstPoint1 then
28             pts ← Get vertices in inflated bounds of (pt1,pt2) from Range_tree
29             Remove any mid-points, pt1 and pt2 from pts
30             foreach Vertex v in pts do
31                 if v is between pt1 and pt2 and within tolerance** then
32                     Insert v to line-segment (pt1,...,pt2) in order
33             Push edges based on segment (pt1,...,pt2) to E*
34         pt1 ← pt2
35         isFirstPoint ← false
36         point_id ← point_id + 1
37 ...
```

** This is done similar to how seen in B.2 QGIS - overview where it checks for direction and adds edges accordingly for each, but is not an important detail.*
*** To find out if a point P is between the gap spanned by line-segment A and B, then one can use the dot product of vectors $\vec{AB} \cdot \vec{AP} > 0$ and $\vec{BA} \cdot \vec{BP} > 0$. An example of this behaviour is seen in figure 6.12.* To check the distance between a point P and the line-segment A and B, then a projection onto $\vec{AB}$ is used: $\frac{\vec{AP} \cdot \vec{AB}}{\vec{AB} \cdot \vec{AB}} \vec{AB}$, where it is then added to the line-segment in the order of distance from A. Example of getting the distance to any line-segment, see figure 6.13

The expected run-time of the main loop is based on the querying in line 28, which can get $k \in [0, ..., n]$ return results. If the segment spans the entire map, it would require running over all the vertices, n, to check if they are close enough to the current segment. In reality k can be expected to be very small as line-segments of paths are very small relative to the map. Therefore the expected run-time is $O(n(\log n + \hat{k}))$, where $\hat{k}$ is the mean number of vertices within every line-segment query rectangle. The absolute worst case where $k = O(n)$ results in $O(n^2)$ run-time, which is far from ideal.

In 8 Results there will be provided run-times comparing 5 QGIS solution and 6 Proposed solution, where the relative complexity of the input data is compared to see if it follows an expected logarithmic run-time.



(a) Case 1: P is behind A in respect to $\vec{AB}$ (b) Case 2: P is behind B in respect to $\vec{BA}$ (c) Case 3: P is between $\vec{AB}$

Figure 6.12: Different cases for checking if the point found within the query window is relevant for a distance check. If it is behind either A or B (in red area), then it is not considered valid.

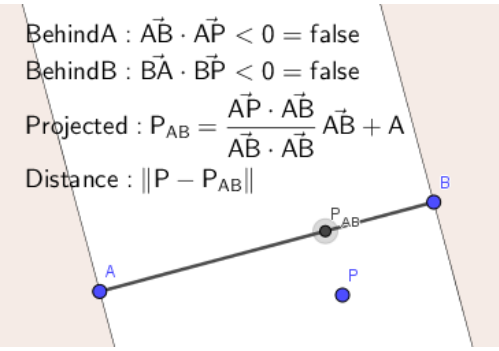


Figure 6.13: Projection of P onto $\vec{AB}$ where the distance between P and P_{AB} is then used to check if the vertex is within the tolerance. The vertex is considered valid when: `BehindA == false & BehindB == false & Distance` $\leq$ `tolerance`

Summary The overall expected run-time is of the algorithm is $O(n(\log n + \hat{k}))$, where $\hat{k}$ is the average number of points returned when querying the bounds of each n line-segments. Worst case is $O(n^2)$ if all line-segments' bounds contain each other, resulting in a check over all vertices.

7. Implementation

The following section will cover the details of the implementations of the two algorithms from 5 QGIS solution and 6 Proposed solution.

The code is available at my git repository:
https://github.com/Djixify/Speciality2022MapGraphs
The prototype operates in a web-service, so there is no need for download and for the duration of the examination period, will be available for test at my website, that is linked to the github:
https://djixify.github.io/Speciality2022MapGraphs/

The implementations are written in `C#` as it is the language I am most familiar with and the algorithms themselves can be found in the `SpecialityWebService/Generation/` folder.

The plan is not to have the fastest algorithms, but rather see if the improvement stated in 5.3 Improvements. More on the performance of each algorithm in 9 Evaluation.

7.1. Reference algorithm

As the implementation from QGIS is written in `C++` and utilizes the QGIS framework found at:
https://github.com/qgis/QGIS/blob/master/src/analysis/network/qgsvectorlayerdirector.cpp
I will be making my own version of the algorithm in `C#` by my interpretation found at B.2 QGIS - overview and B.3 QGIS - detailed (varying detail for the interested reader).

QGIS utilizes an R-tree for its query structure and I will be using the existing library provided at:

https://github.com/viceroypenguin/RBush
And is made by Stuart Turner, which was ported from java-script's RBush library written by Volodymyr Agafonkin. I chose this library as the R-tree was not part of curriculum of Computational Geometry at the time I had the course and provides the necessary functionality to implement QGIS's algorithm.

The most important part of this implementation, is to provide the functionality of pairing vertices of any type, essentially supporting case 1, 2 and 3 of 6.1.1 Desired connection cases, but not case 4 (end-point to line-segment connection).

7.1.1 Non-trivial differences from original

As a disclaimer, the implementation I have written is not a complete replica from its C++ counterpart, as stated in 5.2 Analysis, I will not be including B.1.1 Tie points, which would support adding source points ex. addresses to be added to the network graph. This is outside the scope of this project, where I focus on connections between paths. This only improves the run-time of the algorithm and both algorithms, as discussed in 5 QGIS solution and 6 Proposed solution, should theoretically have similar time-complexities.

7.2. Proposed algorithm

The proposed algorithm should allow for support of all 4 cases from 6.1.1 Desired connection cases. It should also allow for the support of bridges as seen in 4.3 Observations from 4.2 GeoDanmark60, which has mid-points (circles) that should not be connected, despite completely overlapping. 4.2 GeoDanmark60 has end-points at every location a connection is desired, hence the algorithm should support disabling mid-point connections. More on this in 7.2.2 Disabling mid-point pairing.

7.2.1 Range tree

For my implementation, I use a heavily modified range tree, that utilizes a Red-Black BST (Binary Search Tree) data-structure to allow for fast 2D rectangular queries. A Red-Black BST balances tree by performing rotations on branches that contains more nodes than other branches to maintain a height of [3] (lemma 13.1):

"A red-black tree with n internal nodes has height at most $2\log(n+1)$"

This implies $O(\log n)$ height is maintained, giving the guarantee for $O(\log n)$ insert and search queries in one dimension.

I will not be going into the code I used for creating a red-black BST from [3] (chapter 13), as it can be read in better detail from the book, but the modifications to support range searches is not trivial.
To allow for the kind of range searches that are typically used in range-trees, it requires each node to know its interval. To maintain the interval then I will introduce a modified left and right rotation. During the insertion of elements into the range tree, then rotations as seen in figure 7.1.

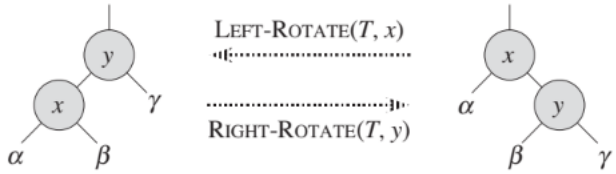


Figure 7.1: Example figure taken from [3], page 313. The figure shows the rotations of the Red-Black tree, the idea is to move a larger child-tree one layer up and propagate the rotations until the Red-Black node rule are satisfied (read more about those details at page 308 of [3])

I assume that the ranges stored at each node is correct from the start before rotation and the way to correctly update the ranges of x and y in figure 7.1, then it must follow:

Left-rotate x has the same left-side range as it keeps its child α, the same goes for y's right-side γ child. Therefore only x's right-side range and y's left-side range should be updated. This can be done by setting $\textbf{Int}(y) = [x.left, y.right]$ and $\textbf{Int}(x) = [x.left, \beta.right]$.

Right-rotate Similar to left-rotate, but just the opposite. It just has to update the new intervals of the nodes to become $\textbf{Int}(y) = [\beta.left, y.right]$ and $\textbf{Int}(x) = [x.left, y.right]$.

The above modification will maintain ranges when Red-black tree performs its height reducing procedure. Otherwise when inserting, it is as simple as updating the interval of x when inserting new key k by $\textbf{Int}(x) = [min(k, x.left), x.right]$ when $k \leq x.key$ and $\textbf{Int}(x) = [x.left, max(k, x.right)]$ when $k > x.key$.

To allow for querying in two dimensions, then each node also contains its y-coordinate, such that when returning from the x-coordinate query leaf nodes, concatenate any nodes with keys that are within the vertical range of the query. This requires $O(\log n + k)$ to solve all satisfying x-coordinate leaf nodes. Then for each k it propagates the result back to the root node, having discarded any nodes along the way that does not satisfy the x and y range query, resulting in $O(\log n + k \log n) = O(k \log n)$. This does not seem ideal as a range tree with an associated y-coordinate query structure takes $O(\log^2 n + k)$. But when the satisfying x-coordinate elements k is small and n is large, then $\log n + k \log n < \log^2 n$. This is visualized in the following table 7.1:

k\n	100	1000	10000	100000	1000000
4	49	104	181	280	402
8	53	108	185	284	406
16	61	116	193	292	414
64	109	164	241	340	462

(a) Scaling for lower values of k vs. larger n for $O(\log^2 n + k)$

k\n	100	1000	10000	100000	1000000
4	27	40	54	67	80
8	54	80	107	133	160
16	107	160	213	283	319
64	426	638	851	1080	1276

(b) Scaling for lower values of k vs. larger n for $O(k \log n)$

Table 7.1: Comparison between scaling for lower values of k vs. large values of n for $O(\log^2 n+k)$ (a) and $O(\log n+k \log n)$. The results are rounded up. As seen in (a) and (b), then (b) favors large values of n and low number of k, whereas (a) is better for larger values of k in most cases for any n.

As also discussed in 6.1.4 Simplifying paths, then breaking the paths up into their line-segments will in most cases cause a very low value of k, hence favoring $O(k \log n)$ over $O(\log^2 n + k)$. This also scales well as adding new areas to the map, increasing n significantly does not increase k unless the new areas are more dense (such that the line-segments has more neighbours).

7.2.2 Disabling mid-point pairing

As discussed previously in 6 Proposed solution, then bridges in 4.2 GeoDanmark60 are a problem, as the typical mid-point to mid-point connection (see def. 5) is not desired (see figure 7.2).

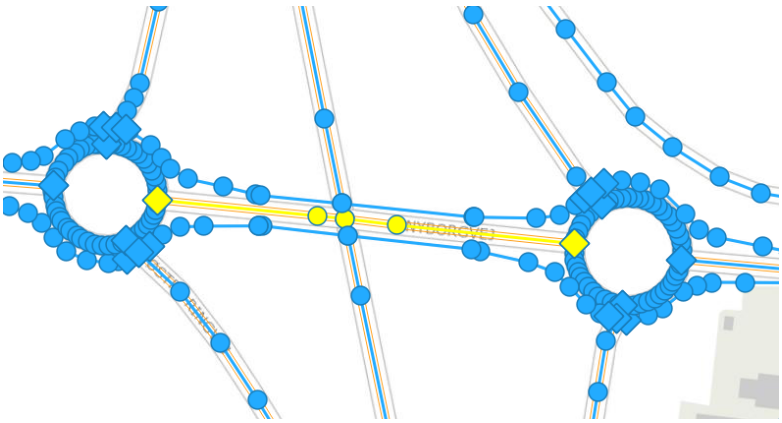


Figure 7.2: Example of a bridge that will be seen in 8 Results, where the three horizontal line crossing the bridge all have mid-points on the underlying road (when inspecting the data in QGIS's software, they are perfectly overlapping, but hard to visualize for that reason).

4.2 GeoDanmark60 has endpoints at every location a connection is expected, hence it would be ideal for the implementation to support parameterization of tolerance radii for end-points and mid-points independently. Setting a negative value would disable the search for connecting with other paths for the type of point except the path it is part of. Examples of a working separation of the bridge and the underlying road can be seen in 8 Results.

7.3. Prototype

As mentioned at the start of 6 Proposed solution, then the implementation can be tested at:
`https://github.com/Djixify/Speciality2022MapGraphs`
For the duration of the examination period.

At the time of delivery, the prototype can:

- Display and zoom/pan around in a map.
- Select data-sets that also will be used in 8 Results and 9 Evaluation.
- Choose between QGIS's algorithm and the proprosed algorithm to generate the network.
- Parameters such as end-point and mid-point tolerances for vertex pairing and direction values.
- A parser for expressing mathematical formulas to compute diffent types of weights, such as:
  - Unit cost edges: `1`
  - Euclidean distance: `distance`
  - ETA in minutes: `distance / 1000 / (HAST_GAELD != "NULL" ? HAST_GAELD : 50) * 60`
  - And much more (syntax shown under the formulas on the website)

- Statistics on the generation of the network, such as time taken, complexity of $G(V, E)$ and some more relevant results that will be discussed in 9 Evaluation.
- Show Dijkstra's algorithm the provided weights before generating the network.

There might still be unintended interactions with the prototype (bugs), but it is operational if following the brief 7.3.1 Getting started and the more detailed 7.3.2 User guide.

7.3.1 Getting started

The least amount of interactions to see a network in action is to provide a name for the network in the "Network settings" box and press "Generate network" at the bottom of the page. When the network is fully generated then update the map view by pressing in the view (to see progress, press "Update" in the "Network statistics" box till it shows the stats of the network).

To then find shortest path between two points, click at those two locations in the map view till two green dots are shown. If a path exists, then it will be drawn with a contrasting color. If there is no path, which happens quite often with 5 QGIS solution, then both green dots will be drawn, but no path between.

7.3.2 User guide

At the top of the website `https://github.com/Djixify/Speciality2022MapGraphs` is seen a map, where one can pan by holding left-click on the mouse and releasing. You can also zoom using the buttons in the bottom left of the map view (ignoring toggle view for now).
An explaination of each section of the website is as follows:

1. **Selecting dataset** Firstly one should select a dataset to work with as seen in figure 7.3. When generating a network, it will be stored in existing networks till the dataset is changed. This allows multiple networks to be generated for the same dataset and freely switch between them (ex. try to use 5 QGIS solution and another network using 6 Proposed solution).

Figure 7.3: Figure showing the dataset setting box, where the user can select which of the 8 provided (except for geodanmark large as it does not cooperate with my machine) to use.

2. **Setting network parameters** The network requires a name in figure 7.4 to be recalled in step 1. Then choose an algorithm to use, either QGIS or the proposed solution. Set the tolerance radii for mid- and/or end-points (optional, default parameters works fine in most cases). As it stands, there is not a lot of single direction roads, so just leave the direction column parameters empty.

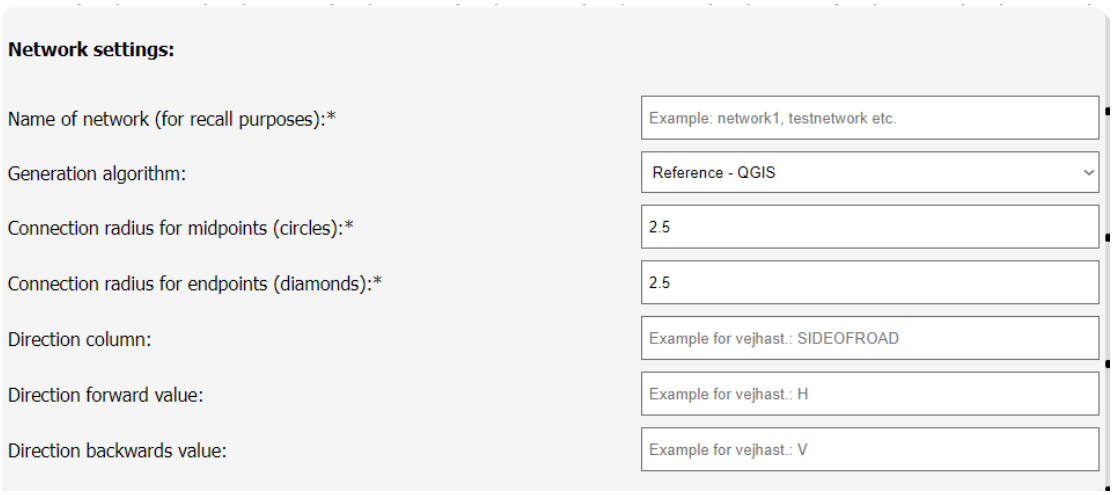


Figure 7.4: Figure showing the network setting box, tune these parameters as see fit. The most interesting case is working on the "Geodanmark60 bridge" dataset and the proposed algorithm, with the mid-point tolerance set to -1 (write manually), as it disables the mid-point to mid-point connections and in effect separates the bridge from the underlying road, which it fails using QGIS's solution.

3. **Network weight settings** The weights used when generating the edges can be set in figure 7.5. The supported syntax can be seen at the bottom of the box. To test if a formula would work as a weight formula, it can be validated by pressing the validate button on the right and the validation message below will be updated. These weights are what is used for Dijkstra's algorithm and will be shown simultaneously if multiple are defined.

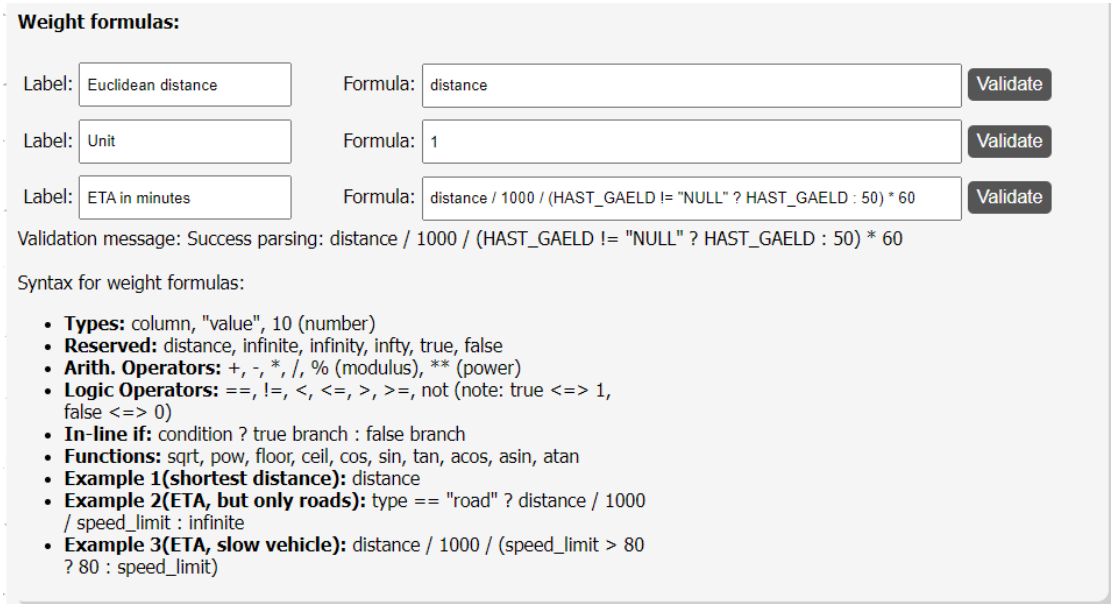


Figure 7.5: Figure showing the network weight settings. The weights are the same as shown at the start of 7.3 Prototype.

4. **Generate button** Lastly the network will be generated when pressing the "Generate network" button at the bottom of the page.

Figure 7.6: Figure showing the last part of the network settings where the network can be generated by pressing "Generate network". If an error in the settings is identified, then the message on the left will be updated.

5. **Statistics** The statistics and state of the generation can be followed in the box on the right of the website as seen in figure 7.7. If the data-set is larger, then it might be an idea to press "Update" till the network stats are shown and then update the map view to see the network.

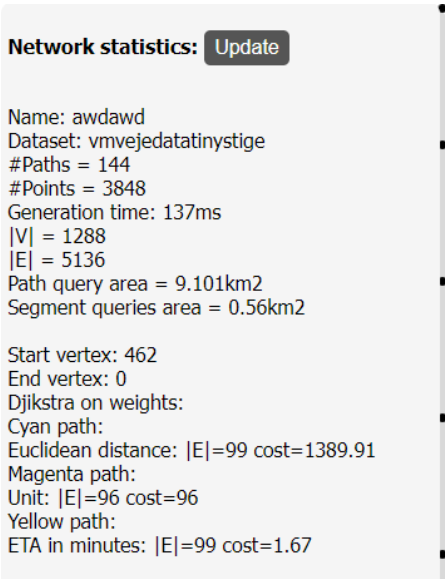


Figure 7.7: Figure showing the network statistics for the small data-set of 4.1 Vejhastigheder.

6. **Using the map** After a network is generated, it will by default be shown on the map as seen in figure 7.8. New points for Dijkstra's algorithm can be selected on the map by left-clicking somewhere on the network and the oldest point from the previous path will be moved to that location. The "Toggle view" button in the bottom left toggle between the initially shown data-set the network is based on and the actively chosen network from figure 7.3. The ability to toggle the view provides an easy comparison for why some behaviour in the network might occur, such as case 4 from 6.1.1 Desired connection cases using QGIS's solution, where there is no connection in multiple t-sections.

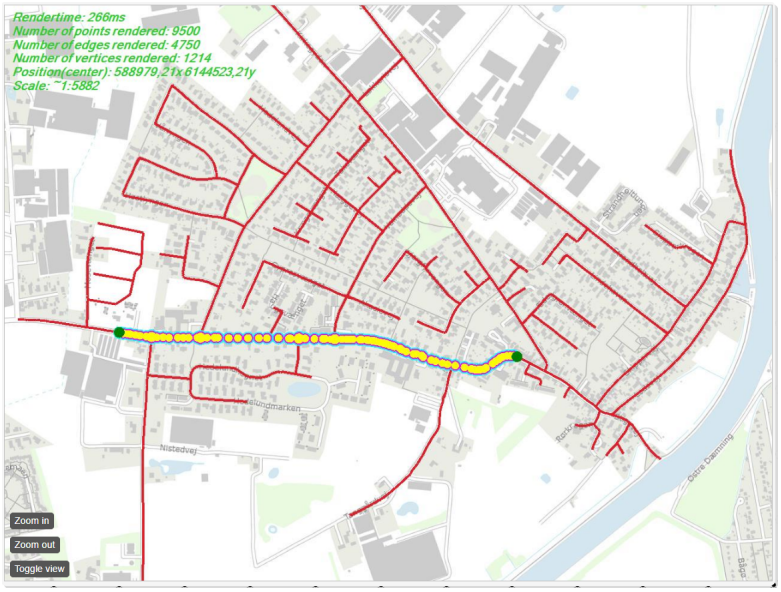


Figure 7.8: Figure showing the map with two points selected for Dijkstra's algorithm. The color of the path shown is the three weighted paths defined in figure 7.5 and the color's meaning is shown in figure 7.7 under "Dijkstra on weights:".

8. Results

I will be showing the results on subsets of the original 4 Data sources as it is restricted how much RAM my machine has during generation or drawing the maps (ex. the large geodanmark60 data-set cannot render properly, despite same setup as the rest of the subsets).

For this project there is two results that are interesting. The generation time should be reasonable and the two algorithms are comparable and the quality of the network as discussed throughout with the improvements that 6 Proposed solution should provide (more on this in evaluation).

I would also expect to see that 4.2 GeoDanmark60 takes longer to generate than 4.1 Vejhastigheder on approximately the same areas as 4.2 GeoDanmark60 has a higher density of points along its paths. From a theoretical standpoint, then it could also be expected to see that QGIS's solution may be faster as concluded in 5.2 Analysis ($O(n \log n)$) vs. 7.2.1 Range tree ($O(n(\log n + \hat{k}))$).

8.1. Data-set statistics

8.1.1 Vejhastigheder small - Statistics

The following is the statistics of running each of the generation algorithms 10 times (extracted from the "Network statistics" box) on "Vejhastigheder small" sub-set. The data-set is shown in figure 8.1.

Figure 8.1: The "Vejhastigheder small" data-set, internally I used the name "vmvejedatatinystige" as read below in the statistics and seen in the prototype.

```
(Global statistics)
Dataset: vmvejedatatinystige
#Paths = 144
#Points = 3848
Path query area = 9.101km2
Segment queries area = 0.56km2

(QGIS algorithm)
|V| = 1298
|E| = 5077
```

```
Generation times (10 runs, no warm-up):
(#1)127ms, (#2)72ms, (#3)33ms, (#4)44ms, (#5)40ms,
(#6)34ms, (#7)34ms, (#8)38ms, (#9)41ms, (#10)43ms

Average generation time: 50.6ms (1x speedup)

(Proposed algorithm)
|V| = 1287
|E| = 5135

Generation times (10 runs, no warm-up):
(#1)68ms, (#2)23ms, (#3)18ms, (#4)22ms, (#5)16ms,
(#6)13ms, (#7)14ms, (#8)14ms, (#9)22ms, (#10)13ms

Average generation time: 22.6ms (2.24x speedup)
```

8.1.2 GeoDanmark60 small - Statistics

The following is the statistics of running the generation algorithms 10 times (extracted from the "Network statistics" box) on "Geodanmark60 small" sub-set. The data-set is shown in figure 8.2.

Figure 8.2: The "GeoDanmark60 small" data-set, internally I used the name "dfvejedatatinystige" as read below in the statistics and seen in the prototype.

```
(Global statistics)
Dataset: dfvejedatatinystige
#Paths = 422
```

```
#Points = 2908
Path query area = 1.168km2
Segment queries area = 0.421km2

(QGIS algorithm)
|V| = 2206
|E| = 4567

Generation times (10 runs, no warm-up):
(#1)78ms, (#2)84ms, (#3)160ms, (#4)81ms, (#5)62ms,
(#6)73ms, (#7)79ms, (#8)85ms, (#9)81ms, (#10)64ms

Average generation time: 84.7ms (1x speedup)

(Proposed algorithm)
|V| = 2164
|E| = 4489

Generation times (10 runs, no warm-up):
(#1)26ms, (#2)14ms, (#3)22ms, (#4)24ms, (#5)11ms,
(#6)13ms, (#7)19ms, (#8)21ms, (#9)23ms, (#10)29ms

Average generation time: 20.2ms (4.19x speedup)
```

8.1.3 Vejhastigheder medium - Statistics

The following is the statistics of running each of the generation algorithms 10 times (extracted from the "Network statistics" box) on "Vejhastigheder medium" sub-set. The data-set is shown in figure 8.3.

Figure 8.3: The "Vejhastigheder medium" data-set, internally I used the name "vmvejedata" as read below in the statistics and seen in the prototype.

```
(Global statistics)
Dataset: vmvejedata
#Paths = 1200
#Points = 29872
Path query area = 81.507km2
Segment queries area = 5.087km2

(QGIS algorithm)
|V| = 10310
|E| = 41005

Generation times (10 runs, no warm-up):
(#1)761ms, (#2)744ms, (#3)794ms, (#4)748ms, (#5)782ms,
(#6)847ms, (#7)746ms, (#8)760ms, (#9)733ms, (#10)782ms

Average generation time: 771ms (1x speedup)

(Proposed algorithm)
|V| = 10177
|E| = 41193

Generation times (10 runs, no warm-up):
(#1)198ms, (#2)220ms, (#3)268ms, (#4)219ms, (#5)227ms,
(#6)206ms, (#7)272ms, (#8)225ms, (#9)230ms, (#10)186ms
```

```
Average generation time: 225ms (3.43x speedup)
```

8.1.4 GeoDanmark60 medium - Statistics

The following is the statistics of running the generation algorithms 10 times (extracted from the "Network statistics" box) on "Geodanmark60 medium" sub-set. The data-set is shown in figure 8.4.

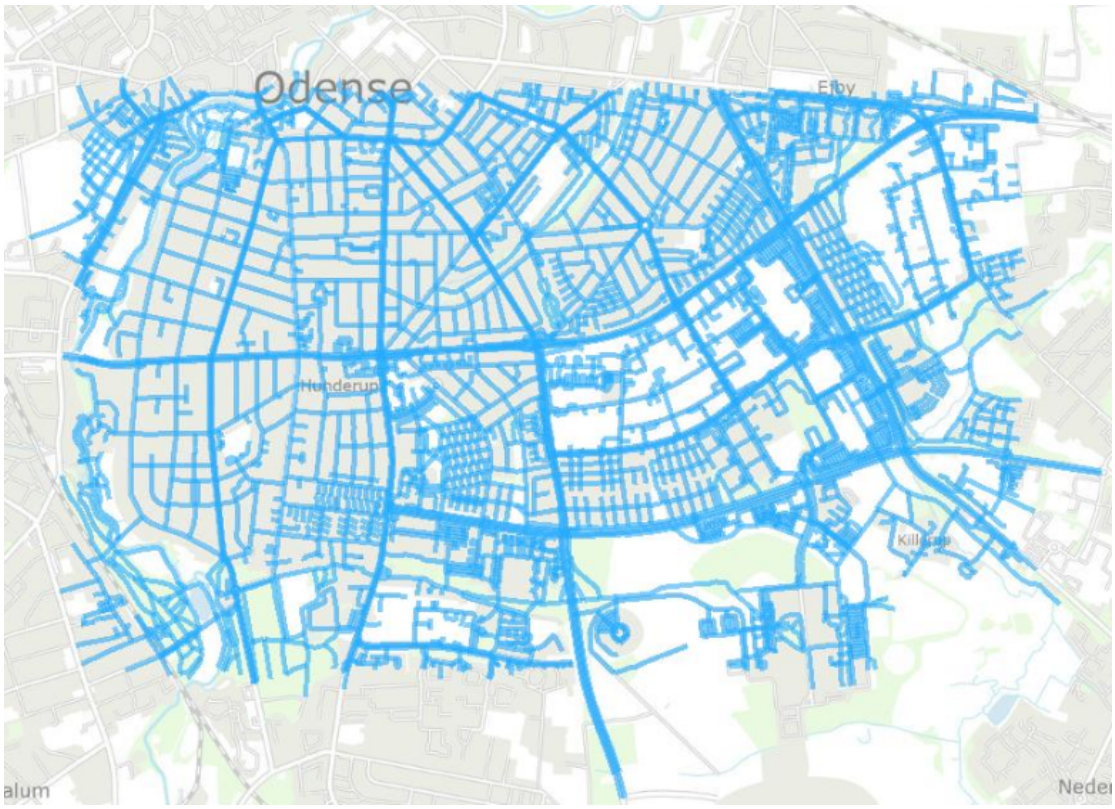


Figure 8.4: The "Geodanmark60 medium" data-set, internally I used the name "dfvejedata" as read below in the statistics and seen in the prototype.

```
(Global statistics)
Dataset: dfvejedata
#Paths = 7427
#Points = 40981
Path query area = 12.476km2
Segment queries area = 5.024km2

(QGIS algorithm)
|V| = 27388
|E| = 59007

Generation times (10 runs, no warm-up):
(#1)3464ms, (#2)3546ms, (#3)3440ms, (#4)3396ms, (#5)3493ms,
(#6)3451ms, (#7)3446ms, (#8)3421ms, (#9)3393ms, (#10)3550ms

Average generation time: 3460ms (1x speedup)

(Proposed algorithm)
|V| = 26883
```

```
|E| = 58109

Generation times (10 runs, no warm-up):
(#1)595ms, (#2)571ms, (#3)471ms, (#4)499ms, (#5)464ms,
(#6)443ms, (#7)598ms, (#8)438ms, (#9)437ms, (#10)485ms

Average generation time: 500ms (6.92x speedup)
```

8.1.5 Vejhastigheder large - Statistics

The following is the statistics of running each of the generation algorithms 10 times (extracted from the "Network statistics" box) on "Vejhastigheder large" sub-set. The data-set is shown in figure 8.5.

Figure 8.5: The "Vejhastigheder large" data-set, internally I used the name "vmvejedatalarge" as read below in the statistics and seen in the prototype.

```
(Global statistics)
Dataset: vmvejedatalarge
#Paths = 14056
#Points = 351029
Path query area = 1474.806km2
Segment queries area = 57.309km2

(QGIS algorithm)
|V| = 118420
|E| = 463394

Generation times (10 runs, no warm-up):
(#1)61026ms, (#2)62857ms, (#3)68466ms, (#4)60097ms, (#5)59862ms,
(#6)62246ms, (#7)60705ms, (#8)59509ms, (#9)60267ms, (#10)59342ms
```

```
Average generation time: 61437ms (1x speedup)

(Proposed algorithm)
|V| = 117177
|E| = 467181

Generation times (10 runs, no warm-up):
(#1)3986ms, (#2)4092ms, (#3)3839ms, (#4)3856ms, (#5)3999ms,
(#6)3967ms, (#7)4018ms, (#8)4035ms, (#9)3988ms, (#10)3825ms

Average generation time: 4060 (15.1x speedup)
```

8.1.6 Geodanmark60 large - Statistics

As mentioned at the start of 8 Results, this data-set causes issues when run mostly on the RAM as my host machine runs out of RAM during generation of the network and drawing the data-set. Therefore I will not be providing statistics for this data-set.

8.2. Quality

As discussed throughout this project, the objective was to provide a method of generating

8.2.1 Vejhastigheder - Quality

The most important quality improvement for 4.1 Vejhastigheder was to add connections between end-points and line-segments as discussed in 6.1.1 Desired connection cases (case 4).

In figure 8.6 is seen a case where QGIS should fail to create a connection in one of the t-sections as there is no other vertices under the end-points of the path coming in from the top.

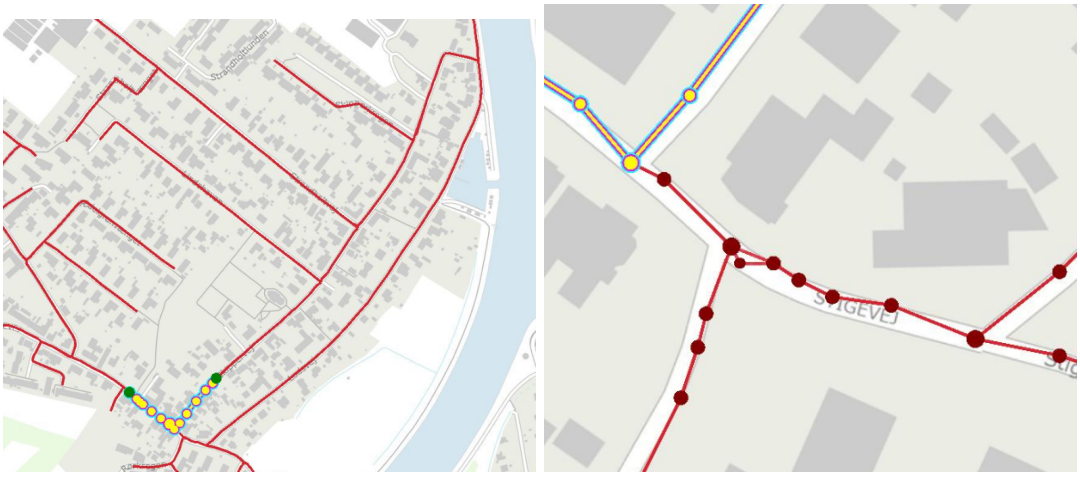
Figure 8.6: Example of a location where end-points from 4.1 Vejhastigheder are "connecting" to a line-segment in the left-most t-section.

In the following figure 8.7 is shown the same intersection and how well the 5 QGIS solution performed when generating the network.

(a) Zoomed out view of a path that has an undesired length as the t-section in the bottom left is not actually connected.

(b) Zoomed in view of the same path from (a), as shown there is no vertices that are

Figure 8.7: Figures show the same area as in figure 8.6, but it fails at creating the connection via case 4 (end-point to line-segment). (a) Shows a clear example of the consequences that such missed connections can have on the quality of the network.



(a) Shows the result of the proposed algorithm opposed to QGIS's shown in figure 8.7a

(b) Zoomed in view of the same path from (a), where the expected connection is added.

Figure 8.8: Figures show the same area as in figure 8.6 and succeeds at creating the connection via case 4 using the proposed algorithm. (a) Shows the expected path compared to the poor result that was given in figure 8.7a.

8.2.2 Geodanmark - Quality

QGIS performs well with creating connections in 4.2 GeoDanmark60 data-set, but cannot support cases where a bridge might have intersecting mid-points with the underlying road. An example of such a case can be seen in figure 8.9.

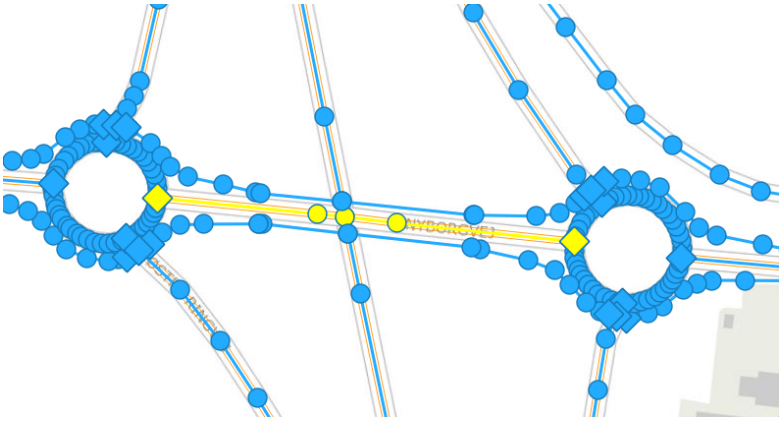
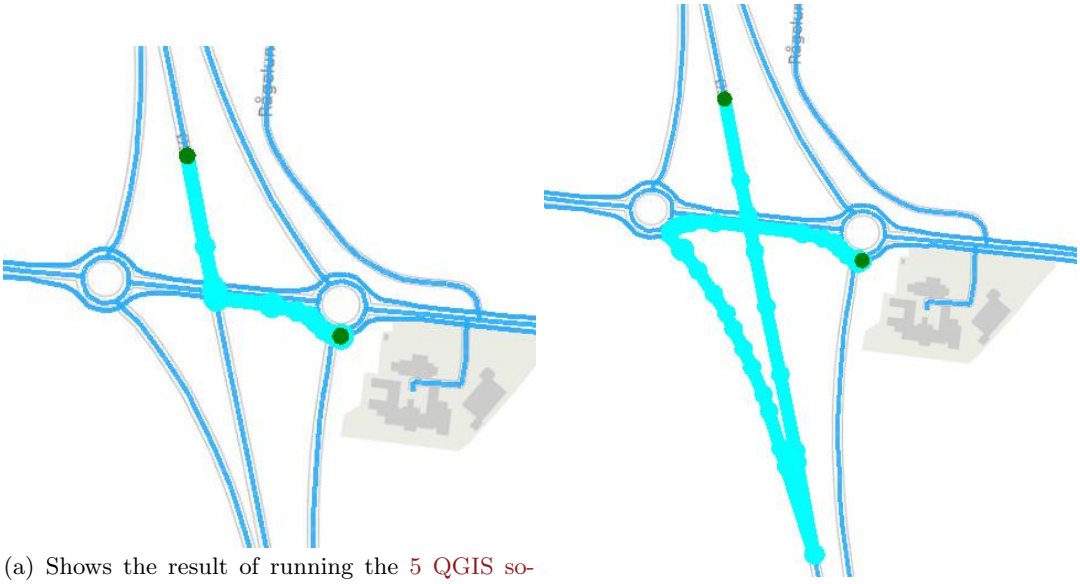


Figure 8.9: A bridge that has overlapping mid-points, which can create undesired connections to the underlying road.

As 4.2 GeoDanmark60 has end-points at every location a connection is expected, then by using the 6 Proposed solution and setting the mid-point tolerance to -1 (do not pair mid-points), we get the result in figure 8.10.



(a) Shows the result of running the 5 QGIS solution on the bridge data-set. As seen, there is connections between the underlying road and the bridge, resulting in it suggesting that the potential user should jump off the bridge.

(b) This is generated using 6 Proposed solution and is the shortest ramp (this example disregards road direction, it is ill-advised to do a u-turn on a high-way)

Figure 8.10: Figures show the example from figure 8.9

9. Evaluation

The comparison between 5 QGIS solution and 6 Proposed solution in terms of generation time might not be entirely fair as I used a library to get the R-tree used in for the QGIS solution. During testing in 8.1.5 Vejhastigheder large - Statistics, I observed a large overhead during the post-processing of the result, where I insert all vertices and edges into an R-trees to query when clicking on the screen to perform Dijsktra's algorithm (not part of the time displayed to generate the network, as it is only for user interface purposes). When generating the network using the proposed algorithm in the large data-set, I could observe up to 5-10 seconds after the network was done in 4 seconds (see the proposed algorithm's performance in 8.1.5 Vejhastigheder large - Statistics).

Regardless my 6 Proposed solution performed the expected improvements from 5 QGIS solution, where it adds the missing edges observed in 8.2.1 Vejhastigheder - Quality and allows for scenarios as bridges seen in 8.2.2 Geodanmark - Quality. The proposed solution, despite the pessimistic comparison between 5 QGIS solutions' $O(n \log n)$ generation time vs. my 6 Proposed solution generation time of $O(+\hat{k})$) (in the actual implementation from 7.2.1 Range tree), outperforms my implementation of the algorithm used in QGIS. The speedups on smaller data-sets was seen to be about 2.24x up to 15.1x speedup. This progression according to the data-set size can be seen in figure 9.1, where a large discrepancy can be seen between the two algorithms.

Returning to the effectiveness of assumption 2 applied in 6.1.4 Simplifying paths to reduce the total area queried by splitting the paths into segments, then it can be seen summarized below in figure **??** from 8 Results:

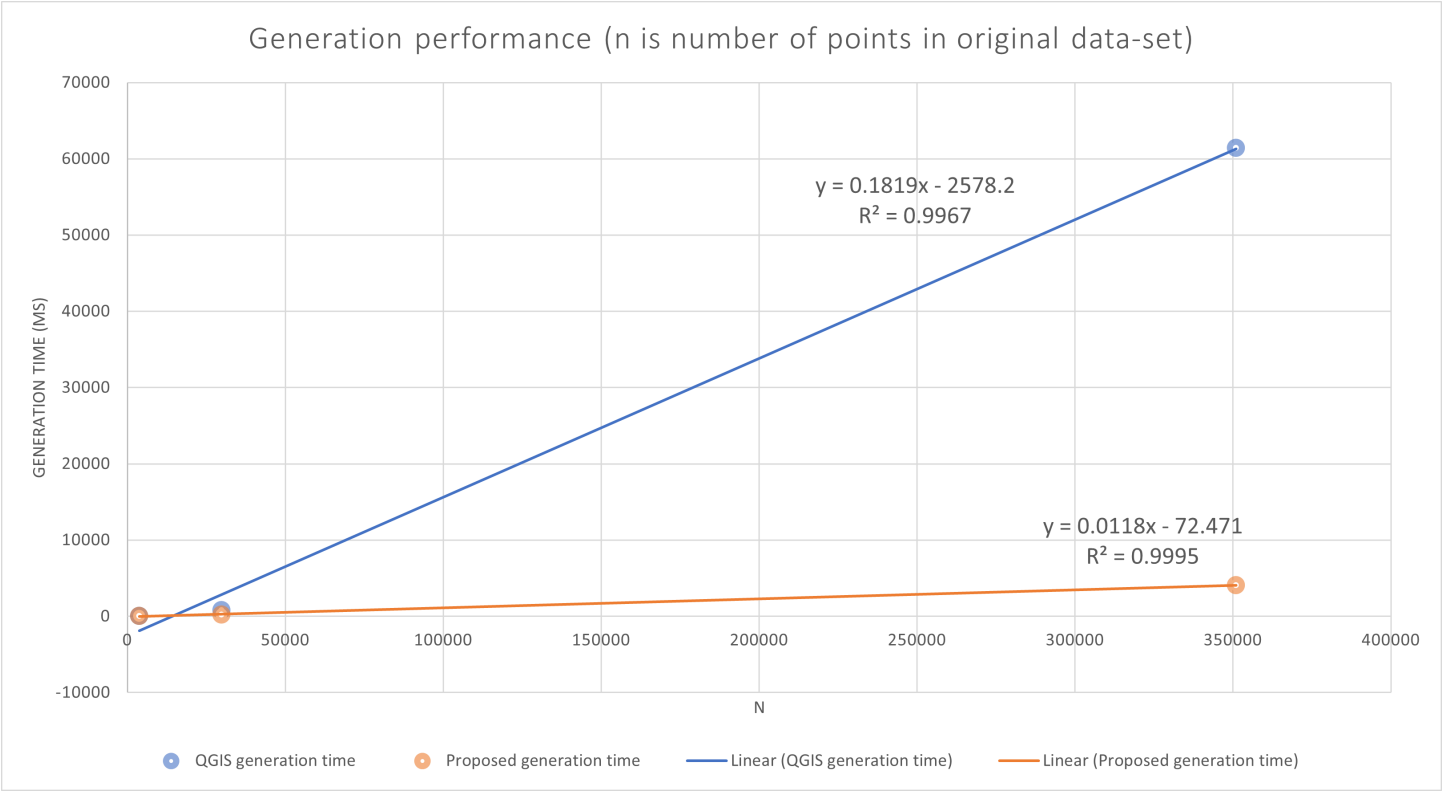


Figure 9.1: Generation times from 8 Results based on the data-points from the small, medium and large subsets of 4.1 Vejhastigheder. As seen with the trend-lines there is a large gab between the two algorithms' performances.

Dataset	Single path query	Split path queries	Improvement
8.1.1 Vejhastigheder small - Statistics	9.101km2	0.56km2	16.3x less area
8.1.2 GeoDanmark60 small - Statistics	1.168km2	0.421km2	2.8x less area
8.1.3 Vejhastigheder medium - Statistics	81.507km2	5.087km2	16x less area
8.1.4 GeoDanmark60 medium - Statistics	12.476km2	5.024km2	2.5x less area
8.1.5 Vejhastigheder large - Statistics	1474.806km2	57.309km2	25.7x less area

Table 9.1: The figure shows a summary of the results, where it compares querying for neighbouring paths with a single query, that envelops the entire path vs. many smaller queries of each line-segment of the path. From this data, it can be concluded that 4.1 Vejhastigheder saw a massive decrease in the total area queried, despite the possible overlapping queries when splitting the path into line-segments as discussed in 6.1.4 Simplifying paths. But on the contrary 4.2 GeoDanmark60 only benefited about 2.5-2.8x times less area queried, which is still an improvement, but that is not account for the cost of the additional queries in the range searching algorithm.

The quality of the networks generated was significantly improved as discussed in 8.2.1 Vejhastigheder - Quality and 8.2.2 Geodanmark - Quality. The 6 Proposed solution also work for more complex connections, such as that between a cross section as seen in figure 9.2.

(a) Result of 5 QGIS solution on an intersection, where there is two end-points pairing the east and west direction, but no connection to the north and south directions.

(b) Result of 6 Proposed solution on an intersection. As can be seen in the figure, there is connections between all four directions now.)

Figure 9.2: Figures shows a more complex situation than the t-sections shown in 8.2.1 Vejhastigheder - Quality. I recommend trying the prototype website and generating a network using QGIS and see how many cases can be found, that are similar to the one shown in this figure

Using prototype, it is also possible to compare different weights for the edges as mentioned in 7.3 Prototype. This was the ideal to support the 3 User scenarios as the parser can use features of each path in its weight computation. If the data is present in the layer, then flow, ETA (Estimated Time of Arrival) and service area queries are all possible.

10. Possible improvements

Possible improvements to the algorithm:

- Improve generation time by using a data-structure that can fetch a two dimensional range query in $O(\log n + k)$ time instead of $O(\hat{k} \log n))$ from 7.2.1 Range tree.
- Post-processing the network to reduce the complexity of $|V|$ and $|E|$. An example of such improvement is to collapse a vertex if it has only two connected vertices, as the edge weights added together would be equivalent to traversing over the vertex.

11. Conclusion

The project succeeded at providing the improvements to the reference solution and had much better generation time in comparison as seen in 8 Results and 9 Evaluation. The theory should favor the reference algorithm, but in practicality with the port I made from `C++` to `C#` might have suffered from the library used for getting an R-tree.

The prototype, that was used for a lot of the visualizations throughout this report, was also a success and has provided a lot of insight into applying the chosen data-structures from Computational Geometry, Dijkstra's algorithm from Algorithms and Data-structures and Programming Language Design to implement a lexer and parser for the formulas expressing the edge weights as instructed in 7.3.2 User guide. Using the formula expressions, then all 3 User scenarios should be satisfied if the paths contains the data foundation for the computation.

11. Bibliography

[1] Irfan Ali, Haque Nawaz Lashari, Imran Keerio, Abdullah Maitlo, M. Chhajro, and Muhammad Malook. Performance comparison between merge and quick sort algorithms in data structure. *International Journal of Advanced Computer Science and Applications*, 9:192–195, 01 2018.

[2] Mugnier J. Clifford. Kingdom of norway, available at: https://www.asprs.org/a/resources/grids/10-99-norway.pdf. *Photogrammetric engineering & remote sensing*, pages 1130–1131, 1999.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009. ISBN: 978-0-262-53305-8, Chapter 11, 13.

[4] Geokov. Utm - universal transverse mercator. http://geokov.com/education/utm.aspx. Last accessed: 2022-05-05.

[5] GISGeography. How universal transverse mercator (utm) works. https://gisgeography.com/utm-universal-transverse-mercator-projection/, 2021. Last accessed: 2022-05-05.

[6] GISGeography. World geodetic system (wgs84). https://gisgeography.com/wgs84-world-geodetic-system/, 2021. Last accessed: 2022-05-05.

[7] LatLong. Degrees minutes seconds to decimal degrees. https://www.latlong.net/degrees-minutes-seconds-to-decimal-degrees. Last accessed: 2022-05-05.

[8] Marc Kreveld Mark Overmars Mark Berg, Otfried Cheong. *Computational Geometry - Algorithms and Applications*. Springer Berlin, Heidelberg, 3 edition, 2008. Chapter 5, 10, 14; ISBN: 978-3-540-77974-2.

[9] Alan Morton. Utm grid zones of the world. https://www.dmap.co.uk/utmworld.htm, 2021. Last accessed: 2022-05-05.

[10] Jian Pei. R-tree. https://www2.cs.sfu.ca/CourseCentral/454/jpei/slides/R-Tree.pdf, 2022. Lecture at Simon Fraser University, Last accessed: 2022-05-04.

A. Geodetic Datum

A geodetic datum is a way to provide precise coordinates of a point on the planet. As the planet is not perfectly spherical and consists of oceans and landmass, then there are multiple approaches that has drawbacks or benefits when it comes to precisely representing a point anywhere on earth. The most two most commonly used in GIS software in the following subsections, where I will be using WGS84 as it is the given geodetic measurement the data was provided in (read more about the data I will be using in 4 Data source). It is otherwise very important to know of the existence of these geodetic measuring systems, as it otherwise can be difficult to display geometries correctly in GIS software.

A.1. WGS84

WGS84 (World Geodetic System 1984) is probably the most commonly used coordinate system and the standard that comes with most gps devices[6], where coordinates are given in latitude and longitude. As seen in figure A.2, longitude is the coordinate lines that are parallel with equator and latitude is the coordinate lines parallel with the Prime meridian also known as Greenwich meridian. Both longitude and latitude is described using degrees, typically denoted as 12.3456° N/S 12.3456° W/E to a certain precision, where N or S and W or E is commonly used instead of negative degrees as seen in figure A.2, but the coordinates can alternatively be written using hours, minutes and seconds, example: 12° 34′ 56″ interpreted as 12 hours, 34 minutes and 56 seconds. Conversion between the two follows [7]:

$$
\begin{aligned}
degree &= hour + (minute/60) + (second/3600) \\
hour &= \lfloor degree \rfloor \\
minute &= \lfloor (degree - hour) \cdot 60 \rfloor \\
second &= \lfloor (degree - hour - minute) \cdot 3600 \rfloor
\end{aligned}
$$

A practical example; the coordinates of Denmark according to https://latitude.to/map/dk/denmark is 56.2639° N 9.5018° E or 56° 15′ 50″ N 9° 30′ 6″ E, which makes sense as Denmark is located quite far from equator and to the north (N) and close to the Prime meridian and to the east (E). The precision of WGS84 is considered to be down to 2 centimeters [6], which is acceptable in most applications. Read more at https://gisgeography.com/wgs84-world-geodetic-system/

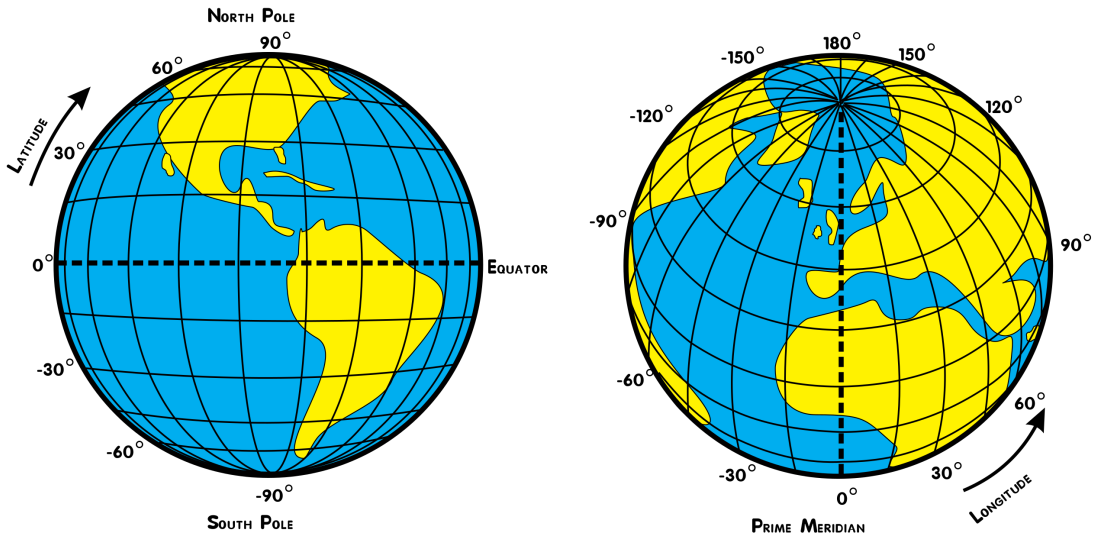


Figure A.1: Model of the WGS84 geodetic datum. Left side shows a side-view at equator where latitude is the vertical lines and provide the coordinate of the distance above or below the equator. The right side can be a little misleading, but it shows the horizontal lines being the longitude, providing the distance away from the Prime meridian or the Greenwich meridian. Image source: https://commons.wikimedia.org/wiki/File:Latitude_and_Longitude_of_the_Earth.svg

A.2. UTM

UTM (Universal Transverse Mercator) is another geodetic system, which utilizes grid coordinates. In comparison to WGS84, latitudes are separated by alphabet characters and longtitude by numbers. A precise point in a tile on the grid is given by decimal coordinates provided in meters from the central meridian of the vertical slice with 500.000m added to prevent negative numbers and the distance to equator[5]. In the case of Denmark, one would use the UTM32 vertical slice and spans the tiles V32 (Northern Jytland), U32 (Southern Jytland, Fyn, most of Zealand) and U33(Bornholm). The same coordinates mentioned in A.1 WGS84 (56.2639° N 9.5018° E) would be 32V (531082.424 eastings, 6235564.766 northings), where the point is located in tile 32V, is 31km to the east of the central meridian of UTM32 and 6.235km north of equator. The benefit of this system is that each grid can have slightly different datums allowing more accurate measurements and provides down to 0.1% at the edges of the widest zones at the equator, given an accuracy of ±1mm[4]. of the irregularities in the earths curvature. Read more about UTM: https://gisgeography.com/utm-universal-transverse-mercator-projection/

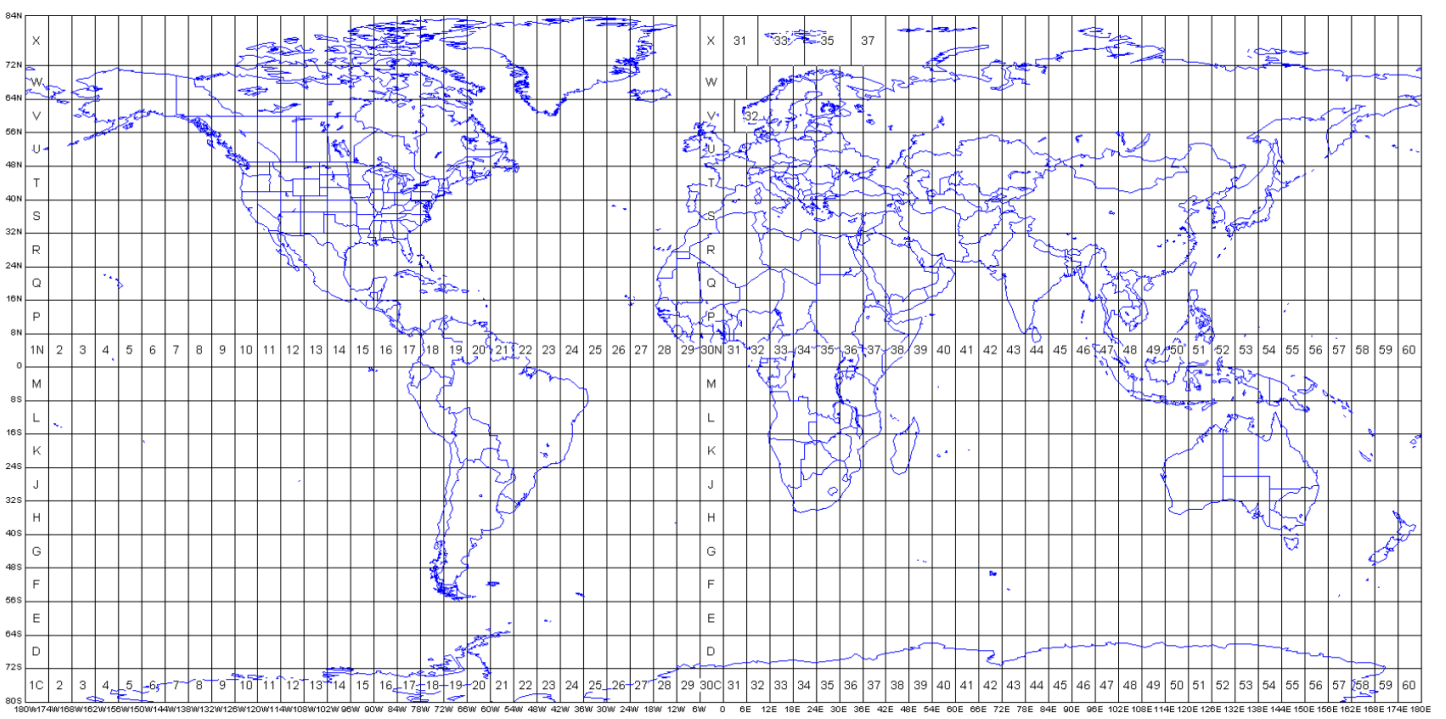


Figure A.2: Model of the UTM geodetic datum. As seen, it splits the world into a grid, where each grid position can be denoted by a alphabet character (latitude) alongside with a number (longtitude). As seen in the figure, the coordinate system is split into 60 slices at 6° each perpendicular with equator and 20 slices at 8° each in parallel with equator (with exception of X which is 12°) [9]. There is a couple of exceptions in the grid seen in Norway and Svalbard, which was historically done to prevent odd land and ocean cutoffs[2]. Image source: https://www.dmap.co.uk/utmworld.htm

B. QGIS algorithm

B.1. Implementation details

I will be using definition 7 to describe points without a path that wish to add to the network

Definition 7. ***Additional point****; An additional point is a point that can be placed anywhere in a layer (see def. 3). These points are meant to be added to the network by the shortest euclidean distance to the closest path. An example for where these points are relevant is addresses for a road navigation network.*

I will also be denoting V_{paths} as the set of all vertices (*interchangeably called points in the algorithm*) from the paths used for the generation. This will simplify the analysis of the reference algorithm in 5.2 Analysis. The following subsections will cover other details that are relevant to understand the intricacies B.3 QGIS - detailed, while only B.1.1 Tie point and 5.1 Query structure are important to understand B.2 QGIS - overview.

B.1.1 Tie point

In the algorithm they define a point on a path segment, which the additional point was closest[19] to, as a "tie point", essentially tying the additional point to the closest point on a path of the network.

[19] Euclidean distance

The algorithm also utilizes objects to bundle data together, the most important object used in the algorithm is a *TiePoint*. A tie-point is used when adding ex. an address to the euclidean closest path segment. These tie points are meant to be additional vertices added to the existing vertices in the path when the network is generated. A tie-point has the following data:

`TiePoint.Path_id` The id of the path the tie-point was added to.
`TiePoint.FirstPoint` The start point of the segment in the path the tie-point was added to.
`TiePoint.LastPoint` The end point of the segment in the path the tie-point was added to.
`TiePoint.Point` The point between `TiePoint.Pt1` and `TiePoint.Pt2` which was closest to the free point.
`TiePoint.Distance` The distance from the additional point to the segment it was added to.

B.1.2 Hash-tables

The algorithm utilizes hash tables to provide constant inserts and look ups according to [3]:

- Insert/Lookup: Average: $O(1)$

B.1.3 Path direction

Paths also has some information that is necessary as part of the algorithm:

`Path.Id` The unique ID of the path, which is used to be able to recall the path at later points in the program at constant time (used like an index).
`Path.Direction` Provides the intended direction of the path, in the order of the first to last vertex defined in the data. This can either be forwards, backwards or both.

B.2. QGIS - overview

Algorithm 2: GenerateGraph(Float, List of Path, List of Point) → Graph
Original line 169-419

Input: `tol`(Float); Tolerance distance for a connection (minimum $1.^{-10\circ} \approx 0,01mm$).
Input: `paths`(List of Path); All paths contained in a layer.
Input: V_{add}(List of Point); Additional points of interest to be added to the network, ex. addresses for roads.
Output: Graph; Navigable network based on paths and additional points

```
1  begin
       /* List of tie-points to network from V_add, this is instantiated
          with the size of V_add                                          */
2      V_tie ← [|V_add|];
       /* Initialization, pre-process paths and add to R-Tree, also
          find best candidate tie points from V_add to the paths          */
3      foreach Path path in paths do // Inner path loop
4          pt1 ← NULL, pt2 ← NULL;
5          isFirstPoint ← true;
6          foreach Point p in path do // Inner points of path loop
7              Insert p into R-tree unless other exists within tolerance;
8              pt2 ← Vertex inserted or found by R-tree;
9              if not isFirstPoint then
                   // Helper function, Algorithm 5, run-time O(V_add)
10                 V_tie ← UpdateTiePoints (path.id, pt1, pt2, V_add, V_tie);
11             pt1 ← pt2;
12             isFirstPoint ← false;
13     foreach Point p in V_tie do
14         Insert tie-point p into R-tree unless other exists within tolerance;
       // Generating the actual graph using R_tree and tiepoints
15     E ← ∅ // Final edges in the graph;
16     foreach Path path in paths do // Outer path loop
17         pt1 ← NULL, pt2 ← NULL;
18         isFirstPoint1 ← true;
19         foreach Point p1 in path do // Inner points of path loop
20             pt2 ← Get vertex from R-tree within tolerance of p1;
21             if not isFirstPoint1 then
22                 ordered_segment ← Order tie-points in segment (pt1→pt2);
23                 v1 ← NULL, v2 ← NULL;
24                 isFirstPoint2 ← true;
25                 foreach Point p2 in ordered_segment do
26                     v2 ← Get vertex from R-tree within tolerance of p2;
27                     if not isFirstPoint2 then
28                         if path.Direction = forward or both then
29                             Insert edge (v1 → v2) into E;
30                         if path.Direction = backward or both then
31                             Insert edge (v2 → v1) into E;
32                     v1 ← v2;
33                     isFirstPoint2 ← false;
34             pt1 ← pt2;
35             isFirstPoint1 ← false;
36     V ← Extract all vertices of R-tree;
37     return Graph(V, E);
```

B.3. QGIS - detailed

Algorithm 3: GenerateGraph(Float, List of Geometry, List of Point) → Graph
Original line 169-419

Input: `tol`(Float); Tolerance distance for a connection (minimum $1.^{-10\circ} \approx 0,01mm$).
Input: `geoms`(List of Geometry); Geometries (see def. 2) contained in a layer.
Input: V_{add}(List of Point); Additional points of interest to be added to the network, ex. addresses for roads.
Output: Graph; Navigable network based on paths and additional points

```
1  begin
2      R_tree ← Instantiate R-Tree for vertex pairing;
       // List of vertices added from paths and additional points
3      V ← ∅;
       /* List of tie-points to network from V_add, this is instantiated
          with the size of V_add                                        */
4      V_tie ← [V_add.Length];
       /* Initialization, pre-process paths and add to R-Tree, also
          find best candidate tie points from V_add to the paths        */
5      foreach Geometry geom in geoms do // Outer loop
6          paths ← GetPaths (geom);
7          foreach Path path in paths do // Inner path loop
8              pt1 ← NULL, pt2 ← NULL;
9              isFirstPoint ← true;
10             foreach Point p in path do // Inner points of path loop
11                 pt2 ← p;
                   // Find existing vertex within tolerance of point pt2
12                 ext_p ← R_tree.Query(pt2, tol);
13                 if ext_p = -1 then // If existing vertex is not found
14                     Push pt2 into V;
15                     Insert pt2 to R-tree;
16                 else
17                     pt2 ← V[ext_p];
18                 if not isFirstPoint then
                       // Algorithm 5, run-time O(V_add)
19                     V_tie ← UpdateTiePoints (path.Id, pt1, pt2, V_add, V_tie);
20                 pt1 ← pt2;
21                 isFirstPoint ← false;
       /* Add tie points to the R-Tree and update accordingly if
          better candidate already exists                               */
22     foreach TiePoint tp in V_tie do
           // Find existing vertex within tolerance of tp.Point
23         ext_p ← R_tree.Query(tp.Point, tol);
24         if ext_p = -1 then // If existing vertex is not found
25             Push tp.Point into V;
26             Insert tp.Point to R_tree;
27         else
28             tp.Point ← V[ext_p];
29     ...
```

Algorithm 3: GenerateGraph(Float, List of Geometry, List of Point) → Graph
Original line 169-419

```
30
31  PathTieHash ← new hashtable between path id and tie points;
    // Relation ship is 1 path to zero or more tie points
32  foreach TiePoint tp in V_tie do
33      Push tp into PathTieHash[tp.Path_id];
    // Generating the actual graph using R_tree and tiepoints
34  E ← ∅ // Final edges in the graph;
35  foreach Geometry geom in geoms do // Outer loop
36      paths ← GetPaths(geom);
37      foreach Path path in paths do // Outer path loop
38          pt1 ← NULL, pt2 ← NULL;
39          isFirstPoint1 ← true;
40          foreach Point p1 in path do // Inner points of path loop
41              ext_p ← R_tree.Query(p1, tol);
42              pt2 ← V[ext_p];
43              if not isFirstPoint1 then
                    /* An ordered list of all vertices along this
                       segment (pt1→pt2) of the path                  */
44                  ordered_segment ← [(0,pt1), (Distance (pt1,pt2),pt2)];
45                  foreach TiePoint tp in PathTieHash[path.Id] do
                        // Find out if tie point is on segment (pt1→pt2)
46                      if tp.FirstPoint = pt1 & tp.LastPoint = pt2 then
                            // O(log n) for sorted insert based on distance
47                          Insert (Distance (pt1,tp.Point), tp.Point) into
                             ordered_segment
48                  v1 ← NULL, v2 ← NULL;
49                  isFirstPoint2 ← true;
50                  foreach Point p2 in ordered_segment do
                        // At this point existence in R_tree is expected
51                      ext_p ← R_tree.Query(p2, tol);
52                      v2 = V[ext_p];
53                      if not isFirstPoint2 then
                            // Add edges depending on direction
54                          if path.Direction = (forward or both) then
55                              Push (v1, v2, Distance (v1, v2)) into E;
56                          if path.Direction = (backward or both) then
57                              Push (v2, v1, Distance (v2, v1)) into E;
58                      v1 ← v2;
59                      isFirstPoint2 ← false;
60              pt1 ← pt2;
61              isFirstPoint1 ← false;
62  return Graph(V, E);
```

B.4. QGIS - helper functions

Helper functions are referenced from B.2 QGIS - overview and B.3 QGIS - detailed, and are not so important for the overall understanding of the algorithm, but rather consider this section as supplementary to the algorithm pseudo-code if you are curious about the

inner workings.

B.4.1 GetPaths

`GetPaths` below essentially try to parse the correct type of geometry (see def. 2), such that we ignore points and polygons, and returns whatever paths it finds in the current geometry given.

Algorithm 4: GetPaths(Geometry) → List of Path
Original line 213-217

Input: `geom`(Geometry); Geometry to decipher if it is relevant
Output: List of Path; Paths extracted from `geom`

```
1 paths ← ∅;
  /* If geometry contrains multiple paths (MultiLineString in QGIS),
     extract each path (LineString)                                  */
2 if geom is MultiPath as multipath then
3 |   foreach Path path in multipath do
4 |   |   Push path onto paths
  // If single path geometry, then just add it to the paths list
5 else if geom is Path as path then
6 |   Push path onto paths;
7 return paths
```

B.4.2 UpdateTiePoints

`UpdateTiePoints` is a function that looks at all the additional points and checks if the segment created by `pt1` and `pt2` is a better candidate for a tie point for all additional points, hence it can be an expensive operation.

Algorithm 5: UpdateTiePoints(Integer, Point, Point, List of Point, List of TiePoint) → List of TiePoint
Original line 242-271

Input: `id`(Integer); Id of the path the segment comes from
Input: `pt1`(Point); First point of segment
Input: `pt2`(Point); Second point of segment
Input: V_{add}(List of Point); Additional points of interest to be added to the network, ex. addresses for roads.
Input: V_{tie}(List of TiePoint); Tie points to be added to the final graph based on V_{add}.
Output: Updated V_{tie} list with updated tie points if found

```
1  Point snapped_point ← NULL;
2  Float closest_dist ← +∞;
3  Integer i ← 0;
4  foreach Point ap in V_add do
5      if pt1 = pt2 then
6          closest_dist ← Distance (ap, pt1);
7          snapped_point ← pt1;
8      else
9          (closest_dist, snapped_point) ← DistanceToSegment (pt1, pt2, ap)
       /* Check to see if new closest point is closer than existing
          closest tie point, old distance denoted as .Distance from
          TiePoint                                                  */
10     if closest_dist < V_add[i].Distance then
11         V_tie[i] ← new TiePoint(id, pt1, pt2, snapped_point, closest_dist);
12     i ← i + 1
13 return V_tie
```