

# Efficient matching of heterogeneous roadmap data at Rijkswaterstaat

A. Scholten  
a.scholten@student.utwente.nl

April 7, 2021

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Problem outline</b>	<b>4</b>
<b>3</b>	<b>Research questions</b>	<b>6</b>
<b>4</b>	<b>Database description</b>	<b>8</b>
4.1	NWB . . . . .	8
4.2	TOP10NL . . . . .	9
<b>5</b>	<b>Existing techniques for network matching</b>	<b>11</b>
5.1	Road networks . . . . .	11
5.2	Biology . . . . .	14
5.3	Sociology . . . . .	15
<b>6</b>	<b>Comparison of network matching techniques</b>	<b>16</b>
<b>7</b>	<b>Description of DSO algorithm</b>	<b>18</b>
7.1	Definitions . . . . .	18
7.2	Data preprocessing . . . . .	23
7.3	Indexing connected objects . . . . .	23
7.4	Constructing delimited strokes . . . . .	24
7.5	Matching delimited strokes . . . . .	26
7.6	Similarity score . . . . .	30
7.7	Treating fragmented areas . . . . .	31
<b>8</b>	<b>Implementation</b>	<b>32</b>
8.1	Tools . . . . .	32
8.2	Data preparation . . . . .	32
8.3	Python mapping to SQL . . . . .	33
8.4	Python file structure . . . . .	34
<b>9</b>	<b>Validation</b>	<b>35</b>
9.1	Evaluation metrics . . . . .	35
9.2	Test results . . . . .	35
<b>10</b>	<b>Discussion</b>	<b>41</b>
10.1	Improvements . . . . .	41
10.2	Limitations . . . . .	41
10.3	Performance and scalability . . . . .	42
<b>11</b>	<b>Conclusion</b>	<b>43</b>

# 1 Introduction

The topic of this research is provided by Rijkswaterstaat, part of the Dutch Ministry of Infrastructure and Water Management. It concerns the Nationaal Wegen-Bestand (NWB, Dutch national road database), which is a database containing most of the roads within the Netherlands, totalling about 150.000 km of road. It is used by the government as the default network for managing traffic and transport. For example, it is used to determine the location of traffic accidents, and the control room of the police uses it to dispatch police officers.

The goal of Rijkswaterstaat is to improve the data in the NWB, either by adding new entries of roads that are not in the database, or by updating old data of roads that have been removed or extended. To do this, the data can be compared to other road databases that are more detailed or updated more frequently. Examples are TOP10NL, a database based on aerial photographs, and OpenStreetMap, an open source database that can be updated by anyone online.

When two different databases both have a data entry that describes the same road, those entries are called a match. By making a list of each match between the databases, a set of roads remain that are either new or outdated. Road matching techniques are used to automatically make a list of road that have matching data entries between two databases.

The problems with currently used matching techniques are described in section 2. The main problem is that the currently used matching techniques are not always suitable for the data that has to be matched. This is because each database has a different way to store data about roads. For example, TOP10NL often has multiple data entries to describe one road, where NWB only has one data entry.

Therefore, the goal of this report is to research different matching techniques for road networks and other similar networks. The technique with the best results for matching road data will be implemented specifically to match data in the NWB to TOP10NL, this is the database that has the most useful additional data according to Rijkswaterstaat. See section 4 for a detailed description of these databases. The found matching techniques are described in section 5, and the useful techniques for this implementation are compared in section 6.

The technique that is chosen, with the best balance between accuracy and matching speed, is the Delimited-Stroke Oriented (DSO) algorithm. This technique combines multiple data entries about one continuous road in a so-called delimited stroke. It means that the stroke (i.e. road) is now one data entry, that is delimited (i.e. is limited by) certain conditions.

The result of this research is a detailed description of the DSO algorithm, that is given in section 7. The algorithm is implemented according to this description, this is further explained in section 8. The results of the algorithm are tested, see section 9. The possible improvements on the implementation, as well as things that can be done differently, are discussed in section 10.

## 2 Problem outline

Rijkswaterstaat (RWS) is an organization that manages the road- and waterway-infrastructure in the Netherlands. One of the tools used for managing all the roads in the Netherlands is the database the NWB (National Road Database). The NWB is an important tool used by the government to determine locations of traffic accidents and dispatch police officers. It is also used as a standard database, to which other databases can be linked by third parties.

Because the NWB is used daily by various parts of the government, it is important for the database to have the latest data about all the roads in the Netherlands. To achieve this, RWS wants to add data from other road databases to the NWB. These road databases contain additional information that is not yet stored in the NWB. Examples of such databases are TOP10NL and OpenStreetMap (OSM). TOP10NL contains data of all roads and cycle paths in the Netherlands, including paths without names, such as forest trails, which are not included in the NWB. OSM contains additional information such as the routes of a bus service and cycle paths.

The process of adding missing data from one database to another is called data merging. To determine which data is missing, a match has to be made between two road databases, which results in a list of roads that is not in both databases. There are, however, several problems concerning the matching and merging of road databases.

The first is that the way data is stored is different for each database. In the NWB, each road is described by one or more road section objects. A road section is the part of a road between two intersections. It contains geographic information as a combination of coordinates that form a line, and various properties of the road, such as the name, as text. In TOP10NL, roads are also described by road sections, however each section describes a part of the road that has unchanged attributes, such as the width of the road. This means that for each road section object in NWB, multiple road section objects exist in TOP10NL.

Another difference is the geographic information of a road. In TOP10NL, the stored coordinates do not form a single line, but a plane. This plane consists of four lines that contour the road as seen from above. This means that to match a road in TOP10NL to a road in the NWB, it first has to be converted to a line. Currently, this is managed by creating centre line from the plane, in other words, the line that is in the middle of the road. The resulting line is similar to the geographic data stored in the NWB, however, it is not always accurate.

The second problem is caused by geographic differences of data that is recorded. A road in one database that is close to a road in another database, could either describe the same road or a road that is not present in the other database.

The last problem is that most of the changes in the NWB are made by hand. Thus, adding an entire network of data to the NWB is a time consuming task. One goal of Rijkswaterstaat is to add the network of cycle paths to the NWB, which contains about 300.000 data points. It would take years to add it using the current methods, and would be outdated by the time it is finished. This is a major reason to apply automatic database merging.

A way to make the automatic merging of databases possible is to make a difference analysis. This means that using road network matching, a list of matching roads recorded in both databases is separated from the roads that are only recorded in one database.

Currently used techniques for the analysis are GIS analysis [1] and map-matching [2]. GIS (Geographic Information System) is a framework for gathering and analyzing geographic data. With GIS, a spatial analysis can be made, which is an analysis that uses the geometric features of the input data to make comparisons. Due to certain registration requirements of road networks, this method is not always applicable. Map-matching is done by mapping out routes along roads. The route from A to B in one network should be the same in the other network. A condition to

use this method is that the road direction should be known, which is not the case for all road networks.

In this project, I will research different network matching techniques, that can be applied to different road databases. The matching of road networks will be the first step of automatic merging of databases, as the result can be used to find missing and overlapping roads. The next section describes the goals and research questions in more detail.

### 3 Research questions

The data stored in the NWB can be improved by achieving the two following goals:

- To fill in the gaps of missing road data.
- To update existing road data with the most recent measurements.

The focus of this project will be on the first goal. For both goals, however, it is important to perform a successful road network matching between NWB and other databases. For the implementation of the matching technique, Rijkswaterstaat indicated the following success factors, in order of importance:

- The technique should achieve a higher matching rate than currently used techniques.
- The technique should be applicable on national scale, thus for the entire NWB.
- The technique should be independent of input format, meaning any two sets can be compared.

Regarding the first success factor, the latest data merge that RWS performed was done using routing analysis tools within the program ArcGIS (see section 5.1.2). With this tool, it was possible to add 88% of cycle paths stored in TOP10NL to the NWB [3].

The best technique will be implemented with a proof-of-concept version, and will be used to match the NWB database to TOP10NL. The content and use cases of these databases are explained in more detail in section 4.

With these goals in mind, answering the following research questions will help solve the problems as described in section 2.

1. What is the state of the art for road network matching?

The following sub-questions can be answered for different techniques in order to compare them and conclude which is the best for the purpose of this research.

- What is the accuracy of the matching technique?
- What is the computational speed of the matching technique?
- Can the matching technique be applied on a national scale? (i.e. the Netherlands)

2. What network matching techniques exist in other fields of study concerning networks?

For more context on network matching techniques, the matching techniques in other fields will be researched. This could give some insights on advanced methods that can be applied to road network matching, in order to improve the results.

3. After the best matching technique is chosen, can it be described in detail such that it can be easily implemented?

With the way most research is presented, there is no step-by-step description of how to implement the techniques used. Moreover, since it will be applied to a different database than what the research is performed on, certain pre-processing steps have to be taken as well. A detailed description will help understanding the technique and make it easier to implement.

4. What are the results of the implemented technique?

Once the chosen technique is implemented, it has to be tested on the NWB database. The results of the matching will determine whether it is better than other techniques that are currently used.

5. How can the selected technique be scaled such that it is applicable for large road networks?

Since one of the goals is to make a matching result of the entire NWB, it is necessary to know whether the technique can be scaled up, and what steps are needed for this purpose.

To answer the first two questions, a literature study is done. The results are presented in this report, in section 5 and section 6. The answer to the third question is the description itself, which is given in section 7. The fourth question, which are the test results, is answered in section 9. Finally, the last question is discussed and answered in section 10.

Attribute	Value
ID	58174008
geometry	LINESTRING(29075 387443, 29090 387442)
begin junction ID	58174012
end junction ID	58174011
name street	Bossenburg
name city	Vlissingen

Table 1: Content of a single record from the NWB. Note that only a few attributes are displayed as an example, and attribute names are changed for clarity.

## 4 Database description

Road databases are also called spatial databases. This means that each data entry has both a geographic attribute, which are the coordinates of the road, and several other attributes, which are properties of the road such as the name. An example of a single record in the NWB is given in table 1. Here, the attribute ‘geometry’ describes the location of the road using a set of coordinates.

### 4.1 NWB

The NWB is a spatial database that consists of all named roads in the Netherlands (see fig. 1 for a visual representation). The roads that are included require a street name or number, and are managed by the central government, provinces and municipalities. The data is mostly received from blueprints or measurements done by land surveyors [4].

One use of the NWB is that of a default database, to which multiple databases can be linked. This means that through the NWB, data of third parties can be connected and compared to each other. Examples of data that can be linked are: traffic accidents, traffic jams, parking information, etc. The Dutch government also stimulates its ministries to use the NWB as the default road network. For example police officers are dispatched to the location of traffic accidents using the NWB.

The NWB consists of the geographic objects ‘road section’ and ‘junction’. This follows the same structure as the objects ‘Road Element’ and ‘Junction’ in the European standard, Geographic Data Files (GDF) [5]. Roads are divided into road sections, which lie between two junctions. A junction is thus defined as the beginning or end point of one or more road sections.

Road section objects are described with a unique ID and other attributes of the corresponding road. Junction objects also have a unique ID. The geographic data consists of the simple feature type LineString, according to the OGC standard [6]. This is a set of coordinates that form a line that represents the centre line of a road, with an accuracy of  $\pm 5m$  [4]. The database consists of 1.074.691 road section objects. The coordinate system used for this data is the default Dutch system, RD New (EPSG:28992) [7]. Roads with lanes separated by a median strip, such as highways, are recorded as distinct road sections. The condition is that there should be an obstacle of over 50m long on the median strip.

Road section objects have various administrative attributes that contain semantic information about the corresponding road. Examples are street name, road authority, road type, municipality name and driving direction. However, certain attributes like the driving direction are not known for every road object.





Figure 1: Visualization of multiple records in the NWB. Each line represents a separate record.

## 4.2 TOP10NL

TOP10NL is a spatial database, and the most detailed product in the BRT (Basisregistratie Topografie). It is managed by the Kadaster (see fig. 2 for a visual representation). The recorded data does not only include roads, but also other geographical elements, such as buildings, railroad tracks and water ways. The sources of these data are aerial photographs [8].

Roads in TOP10NL are part of one object type, ‘road section’. Junctions do not have their own object class or unique identifier. A road section is defined as the spatially largest functionally independent part of a road with unchanged, homogeneous attributes.

The geographic data of a road is stored in two ways. The first is a polygon of the surface area of a road section. The second is a centre line of that road section, similar to the NWB. They have the simple feature types Polygon and LineString respectively. The file of just the centre lines consists of 1.277.344 road objects.

The coordinate system is the same as the one used in the NWB, RD New (EPSG:28992). Here, roads with multiple lanes are only recorded as distinct road sections if there is a physical obstacle of more than 500m.

The attributes of road sections in TOP10NL are mostly different from those in the NWB. TOP10NL typically concerns the physical properties or usage of the road, such as type of infrastructure, location w.r.t. other objects (e.g. bridge) and main traffic use.



Figure 2: Visualization of multiple records in TOP10NL, orange = planar data, blue = centre lines, red = narrow roads

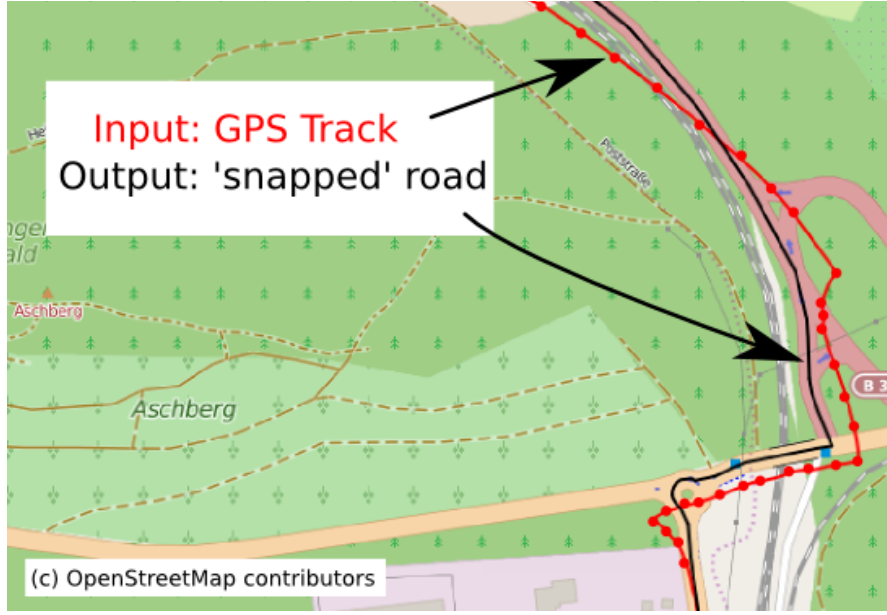


Figure 3: Example of map-matching, GPS data is matched (i.e. snapped) to the nearest road in the database [9].

## 5 Existing techniques for network matching

In this section, an overview is given of network matching techniques used for both road networks and networks in different fields of study, which are biology and sociology. The techniques used in different domains cannot immediately be applied to road networks, however the underlying theory is still relevant.

### 5.1 Road networks

#### 5.1.1 Map-matching

Map-matching algorithms are used to match recorded geographical data to a road network. The geographical data is usually acquired from GPS measurements. An example of input data and a matched road using map-matching is given in fig. 3.

The advantage of map-matching algorithms is that they can be used for real-time applications. For example in GPS systems, to determine the location of the user can be inaccurate. Using map-matching on the history of GPS data points gives a result that can determine their location more accurately. There are also algorithms used in post-processing applications optimized for large databases, to analyze traffic flow information.

The problem solved with map-matching is that of geometric differences in the data. The matched GPS points should follow the road according to a logical, connected route. So even if there are outlying points in the input data, the connection between points ensures an accurate match. This is especially useful for GPS data, since these are often rough estimates. The disadvantage of this is that the driving direction of the road has to be known to make a correct match.

Map-matching algorithms are not the best option for matching complete road databases,

because the data entries consist of accurate road data and not just points like GPS data. This means a more robust technique considering all the available data will have more accurate results. This section provides some background information on the use of map-matching techniques in road network matching, however, they will not be used for the goal of matching road databases.

Map-matching algorithms can be divided in multiple categories [2]: geometric, topological, probabilistic and advanced algorithms.

A geometric algorithm matches data points based on geometry of road segments. A point is typically matched to the closest segment, ignoring the connections of the segment in the network. Geometric algorithms can be further divided into three categories [10]: point-to-point, point-to-curve and curve-to-curve. These algorithms are very dependent on the map data and deliver inaccurate results in areas with a large number of roads.

Unlike geometric algorithms, topological algorithms take the connections and neighbouring roads into account. This means that the input points are connected to each other, and the matched road should follow these points as closely as possible with the same driving direction. An example is the Marschal algorithm [11], which was improved by Pereira [10].

Probabilistic algorithms use an error region around a data point to find a match with the closest edges, which is chosen using direction, proximity and connectivity from the point to the edge [2]. A different probabilistic approach uses Hidden Markov Models (HMM), which are used to find the most probable path from the possible paths that are generated by projecting data points to nearby edges [12] [13].

Advanced algorithms combine multiple simple algorithms or use more attributes and data to improve the accuracy of the matching.

In [14] a Graph Search Based Map Matching (GSMM) is described, which is scalable for large databases. It is based on Dijkstra's shortest path algorithm.

Road networks can be matched using these techniques by taking location data from one network and matching those to the other network. The problem with map-matching techniques is that they assume a directed network is used as input. The NWB is only partially directed, while TOP10NL is undirected. This is why the map-matching techniques will not be further considered for this research.

### 5.1.2 ArcGIS

One of the GIS methods is a map-matching algorithm implemented in ArcGIS, using the integrated route analysis tool [1]. This tool uses a shortest-path algorithm to generate the best route according to GPS input data. It requires stops and barriers as user input. However, ArcGIS is not open source, meaning that the details technology used are not available. This means it is not a viable option for the implementation of this research.

### 5.1.3 DSO

The Delimited-Stroke Oriented algorithm works by combining multiple data objects describing one, continuous road into a single object (i.e. stroke), that is limited (i.e. delimited) by certain conditions. It uses, besides local geometry, the topological relationships within a network to make an accurate matching of delimited strokes [15,16]. The goal of this algorithm is to take two road databases and match road sections of one database (target) to the road sections of the other (reference). The following description serves as a general summary of the DSO algorithm. This algorithm will be used for the implementation of this research, further explained in section 6. Therefore, a more detailed description of DSO is given in section 7.

Road sections are the smallest part of the road recorded in the database. Junctions are the points in the database where two or more road sections connect to each other. Using this, a delimited stroke is more formally defined as a series of connected road sections, limited at junctions under two conditions:

- The junction is either an intersection with four or more connected road sections, or it has only one connected road section (i.e. a dead-end)
- The road sections follow each other in the same direction, meaning the angle between the road sections is approximately  $180^\circ$ .

The DSO algorithm consists of the following five steps:

1. Data pre-processing
2. Indexing connected objects
3. Constructing delimited strokes
4. Matching delimited strokes
5. Treating fragmented areas (i.e. areas with small roads)

For both input databases, the pre-processing and indexing of the data is done in preparation of step 3, the construction of delimited strokes. Junctions are classified to determine what the limits are of each stroke.

In the matching process, the delimited strokes from the reference database are matched to the delimited strokes of the target database. A match is created when the distance between the begin- and endjunctions of both strokes is smaller than a tolerance value. For each match, a score is calculated based on geometrical differences between the strokes in the match. The score is used to filter out bad matches, or to determine the best match when a stroke has multiple matches.

The last step concerns fragmented areas, which are areas consisting of multiple smaller roads. These roads are too small to confidently match using only the distance between junctions and the similarity score, and are checked on other factors, such as connectivity with nearby roads in the network.

#### 5.1.4 PRM

Probabilistic Relaxation Methods have been used for road network matching research in multiple occasions [17–21]. It is usually applied using three steps: initialization, relaxation and selection.

During initialization, the initial probability matrix is created. This matrix describes the matching probability between one edge and its matching candidate edges, based on a similarity score. The candidates are selected within a certain range of the edge. The similarity score depends on the length, angle and shape of the edges. The matching probability is defined as the ratio of the similarity score of one matching to the total score of the matching with all candidates.

The relaxation steps consists of iteratively updating the probability matrix based on a compatibility coefficient. This coefficient depends on the similarity of neighbouring matching pairs. The iteration is terminated once the difference in the new probability is lower than a threshold value.

The selection of the final matches is not straightforward, as the highest probability for a match is not sufficient for M:N matching. The selection is based on five steps, in which the

matching probability is combined with those of neighbouring pairs. Conflicts and 1:1 pairs are subsequently filtered out and the remaining pairs are checked for 1:N matching.

In [20], PRM is used to match hierarchical strokes. A stroke is a combination of road segments which are continuous in the same direction (similar to delimited strokes). A hierarchy is used to match main roads (such as highways) first. This results in a stable and very accurate baseline on which subsequent road levels are matched. The next hierarchical level is determined based on which roads are adjacent to those of the previous level. This method achieves a robust result in which 1:1, 1:N and M:N matching pairs are found.

### 5.1.5 PSO

Particle Swarm Optimization is a global optimization method, which has been applied to road network matching [22]. It is an iterative method, in which multiple particles are flying towards an optimal solution. A particle is a possible solution of the optimization problem. In road network matching, a particle is the collection of all matching pairs of nodes between two data-sets. A measure called the Particle Fitness Value is the fitting function which has to be solved to find the optimal solution. It is defined as the sum of the similarity scores of each pair of nodes in a particle. The calculation of the similarity score follows the same structure as the one used in PRM.

The optimal solution is found by updating the velocity and position of each particle iteratively. The velocity of a particle depends on three measures: the velocity of the particle in the previous step, the best solution of the particle found so far, and the global best solution found by any particle. The weights of each measure is often randomized in each iteration, to prevent finding a local optimum.

Another strategy is used to prevent finding local optima, which is tabu-search. With tabu-search, the particle which corresponds to the best global solution in an iteration is put in a list for  $\sqrt{N}$  time, where  $N$  is the number of particles. The particles in the tabu list are not selected during the search, unless the solution has a better fit than the global solution over all iterations.

Even though finding an optimal solution can be done relatively fast, because this method is an iterative process it can have a large computation time for large databases. To facilitate this, a method has been developed to do the computation in parallel using GPU (Graphics-Processing Unit) threads [23]. The computation speed has been compared with a PRM, resulting in an increase with a factor of over 30 for the largest test sample.

## 5.2 Biology

In biology, networks are often used to describe proteins or genetic data. In an article by Clark and Kalita [24], a comparison is made of several algorithms used to make a pair-wise network alignment, where a solution is searched to make a one-to-one mapping of one network to a bigger network. The algorithms are optimized for and applied to biological networks, specifically protein-protein interaction (PPI) networks. Most of the algorithms follow a generalized two-step solution. The first step is to compute the similarity for each pair of nodes, which is based on the local topology around the nodes and a sequence similarity score. The second step is to create a bipartite graph with weighted edges, determined by the similarities, and to solve the maximum-weight bipartite matching problem (see also [25]).

The article concludes with a recommendation to use one of the following algorithms: SPINAL, NATALIE and PINALOG.

### 5.2.1 SPINAL

SPINAL [26] is one of the algorithms that uses the two-step solution as described above. The first step is performed iteratively. For each pair of nodes, a match confidence score is calculated, based on the match scores of neighbouring nodes from the previous step.

### 5.2.2 NATALIE

The NATALIE 2.0 algorithm [27] uses a Lagrangian relaxation method to solve an integer linear programming formulation. It also does not try to make a one-to-one mapping, as it leaves nodes unaligned if no good match can be found. Solving data association using a Lagrangian relation method is further described by Tauer et al. [28].

### 5.2.3 PINALOG

PINALOG [29] matches two networks based on dense subgraphs which are called communities. First the communities in the different networks are matched to each other. Then the nodes within each community are matched with nodes in the matched community.

### 5.2.4 SNF

Another network fusion technique in biology is called similarity network fusion (SNF) [30]. This is a technique used to combine diverse types of genome-wide data in one network. It takes several measurements for a set of patients, creates a sample similarity network for each type of data, and combines those into one network.

## 5.3 Sociology

In sociology, networks are used to describe relations between people. Farasat et al. [31] describes how a data network can be extracted from fused data, concerning social networks. It is done using a parallel computing-based methodology, and described in a so-called cumulative associated data graph (CDG).

## 6 Comparison of network matching techniques

First, taking a look at the various techniques from domains other than road networks, it seems that the matching is mostly based on node comparisons. The location of the nodes and the nearby structure is also often taken into account. However, for road networks, the geometric data of the edges is an important factor for making a accurate matching. Because the techniques from the other domains do not use this data for the matching, they will not be considered useful for road network matching. The rest of this section describes the results of the road network matching techniques that were found, in order to make a comparison and conclude which technique is the best option to implement for the purpose of this research.

An overview of the results from each technique is given in table 2. Experimental data from the DSO technique are from [16]. The data from PRM and PSO are from the same report [23], where the results of both techniques are compared. A12 and A14 are two different areas on which the matching was performed. A12 has a similar size to the area matched with DSO, whereas A14 is the largest area that was matched with PSO. The PRM method was not able to match this area due to memory issues. Note that the comparison of the matching accuracy is not conclusive, as the areas that are matched are different.

Matching algorithm, area name	DSO	PRM, A12	PSO, A12	PSO, A14
Matching accuracy	99.1%	89.5%	87.5%	74.7%
Reference objects	10947	10820	10820	72130
Target objects	10360	13961	13961	127205
Matching time (s)	22	245.3	67.2	136.1
Matching speed (ref obj/s)	498	44	161	530
Matching speed (target obj/s)	471	57	208	935
Scalable?	yes	no	yes	yes

Table 2: Comparison table of the road network matching techniques, experimental data taken from the articles for DSO [16], PRM [23] and PSO [23]. A12 and A14 refer to two different areas used for matching in [23].

DSO has been successfully implemented as the matching method to perform automatic database merging of pedestrian roadways [32]. The computation speed is 498 reference objects per second, which is approximately equal to the estimated average of matching tests in [16] of 500 obj/s. This estimation shows a linear relationship between the number of objects and the computation time. The reported accuracy is very high, with over 99% matching correctness and recall rate.

PRM, compared to PSO, shows a similar matching accuracy for the same area (A12, 89.5% vs. 87.5%). It does, however, not meet the required standards, since it is too computationally expensive to match large road networks. In PSO article [23] it was not possible to match the largest area with PRM, due to computer memory limitations. As the NWB is even larger than the mentioned area, it is likely not possible to match the NWB using PRM.

The method implementing PSO with a GPU architecture reports an average matching speed of 200 objects per second for the smaller areas. The biggest area however, with 127205 target objects, achieves a speed of 935 obj/s, making it a valid option for large road networks. The reported accuracy however, fluctuates from area to area from 74.65% to 97.73%, with an average of 84.44% [23].

If speed is more important for the calculation, then the PSO method using GPU is the best. However, if accuracy is more important, a method using DSO achieves better results. Since



the computation time of DSO is within an acceptable range, it is the best method to apply for matching road networks. It is, however, not possible to make a decisive conclusion, since the accuracy of the match is highly dependent on the input data and the reference database. This can be seen in figure 7 in article [23], where the accuracy varies a lot. It is possible that the areas matched with DSO had fewer differences and were thus easier to match, resulting in a higher accuracy.

## 7 Description of DSO algorithm

In this section, the Delimited-Stroke Oriented (DSO) algorithm [15, 16] is explained in detail, with all the functionality and terminology that will be used in the implementation. First, the data types and functions used to formally define the algorithm are given. Then, the steps taken to complete the matching process will be explained using these data types and functions.

### 7.1 Definitions

The road network is viewed as a collection of roads that meet at junctions. A *road junction* is a location where two or more roads meet, or where a road ends. Its minimal representation includes an (integer) identifier and a location, which gives us the data type:

```
road_junction = ⟨ id:int, geom:point ⟩
```

Roads are described in terms of *road sections*. Any road section runs between two road junctions.

```
road_section = ⟨ id:int, geom:linestring,  
                bjunction: road_junction,  
                ejunction: road_junction ⟩.
```

The linestring geometry has an implicit direction derivable from its start and end vertex, which should coincide with resp. the begin- and end-junction. This direction, however, is not necessarily identical to the driving direction. For textual convenience, a road section that runs from junction  $A$  to junction  $B$  can be denoted as  $A \rightarrow B$ .

Any of our databases contains the data that describes the road network. The *database* type is:

```
database = ⟨ sections: P road_section,  
            junctions: P road_junction ⟩.
```

The  $P$  type operator in our notation represents a ‘set of’ operator. Likewise below, we will be using a  $L$  type operator that represents a ‘list of’ operator.

It should be understood that meaningful database values are those where sections and junctions fit together, and describe a road network. We can formalize this further through (referential) constraints that ensure that any junction of a road section is in the set of junctions, but will leave this here as ‘understood from context’.

Our aim is to find matches between roads taken from the source and target databases. To this end, a *match* is a pair of road section lists, one list obtained from the source database and one list from the target database. Match objects have their own identifier mechanism. The corresponding data type is

```
match = ⟨ id:int,  
         reference_sections: L road_section,  
         target_sections: L road_section,  
         similarity_score:float ⟩.
```

Clearly, the intention here is that the string of reference sections starts spatially where the string of target sections starts, and vice versa, they both end at the same location. Both may be meandering somewhat differently inside. We leave another constraint as ‘understood from context’, namely that all but the last road section in a list ends where the next section in the list

starts. Match objects may or may not be stored in the underlying databases; for now we assume they are just objects in the runtime environment of the algorithm.

To find matches, the DSO algorithm uses the notion of *delimited stroke*. A delimited stroke is a list of sections that pairwise meet and where at junctions certain conditions hold. To express those conditions, we need a few additional functions. The first of these defines the *degree of a junction* inside a database.

```
degree : (road_junction, database) → int

degree(j, d) =  $\begin{cases} 0 & \text{if not } j \text{ in } d.\text{junctions} \\ \text{count}(\{s \text{ in } d.\text{sections} \mid s.\text{bjunction} = j\}) + \\ \quad \text{count}(\{s \text{ in } d.\text{sections} \mid s.\text{ejunction} = j\}) & \text{if } j \text{ in } d.\text{junctions} \end{cases}$ 
```

In words, the degree of a junction can be described as the number of road sections connected to that junction.

Junctions can be characterized in different ways. A *dead-end* is a junction with a degree of 1. This allows a function definition that gives us the dead-ends in a database:

```
dead_ends : database → P road_junction
dead_ends(d) = {j in d.junctions | degree(j, d) = 1}
```

In a very similar way, we define a *crossing* as a junction with more than two incident road sections.

```
crossings : database → P road_junction
crossings(d) = {j in d.junctions | degree(j, d) > 2}
```

Examples are the junctions *B*, *D* and *E* in fig. 4. Junction *C* is a dead-end.

For the sake of the algorithm, we need the notion of terminating junction also. It is either a dead-end, or a junction with a degree above 3. We use the term *terminator* in the definitions.

```
terminators : database → P road_junction
terminators(d) = {j in d.junctions | degree(j, d) = 1 ∨ degree(j, d) > 3}
```

Terminator examples in fig. 4 are the junctions *A*, *B* and *C*.

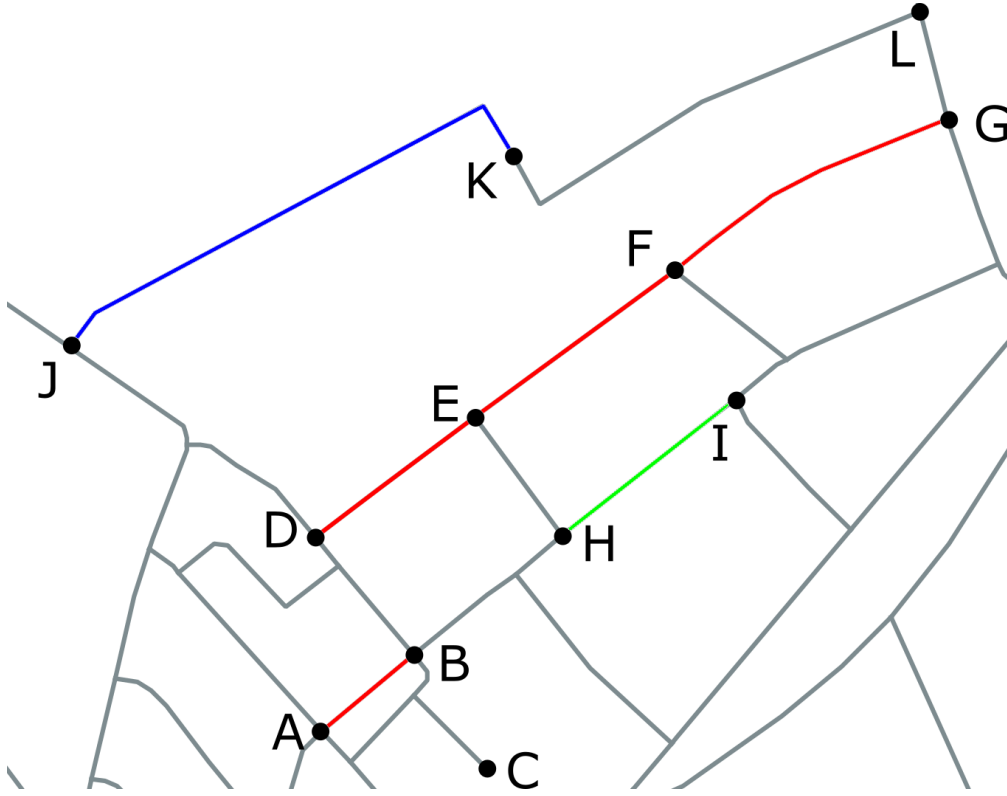


Figure 4: Extract of the NWB, with highlighted junctions used to explain terminology. Colored sections are DSs, where level 1 = red, level 2 = green, level 3 = blue.

A final important notion to string road sections together at a junction is that of *good continuity*. When two road sections follow each other in almost the same direction, they have good continuity. This requires the angle between the two sections to be approximately  $180^\circ$ . For example, in fig. 4, sections  $D \rightarrow E$  and  $E \rightarrow F$  have good continuity, while  $D \rightarrow E$  and  $E \rightarrow H$  do not have good continuity. Obviously, how good the continuity is can be expressed in terms of the derivation from 180 of the angle that the two sections make at the junction.

So let us define how that angle is computed. We assume availability of a number of spatial functions, especially the following, as provided by the PostGIS and GEOS libraries:

- `st_azimuth(p,q)`: returns the angle that the line from point `p` to point `q` makes with respect to north in radians
- `st_startpoint(l)/st_endpoint(l)`: returns the first/last point of the input linestring `l`
- `st_pointn(l, n)`: returns the `n`th point of the input linestring `l`

Given a road section and one of its (internal!) junctions, we may want to determine the angle of continuity at that junction. That is, the angle between the road segment that arrives at the junction and the segment that leaves from the junction.

```

angle_at_junction : (road_section, road_junction) → float
angle_at_junction(s,j) =
{ st_azimuth(st_startpoint(s.geom),st_pointn(s.geom,2))  if s.bjunction = j
{ st_azimuth(st_endpoint(s.geom),st_pointn(s.geom,-2))    if s.ejunction = j

```

The exact angle that determines good continuity is not given in [16]. In the implementation, the variable  $\theta = \pi/9$  is a small deviation angle, that is used to set the limits of good continuity. Other values for  $\theta$  are possible as well. Using this parameter, good continuity can be defined as:

```

good_continuity : (road_section, road_section, road_junction) → Bool
good_continuity(s1, s2, j) =
{ False if not s1.begin_junction_id = j.id and not s1.end_junction_id = j.id
{ False if not s2.begin_junction_id = j.id and not s2.end_junction_id = j.id
{ True  if  $\pi - \theta < \text{angle\_difference}(\text{angle\_at\_junction}(s1,j),$ 
{ True  if  $\text{angle\_at\_junction}(s2,j)) < \pi + \theta$ 

```

Delimited stroke (ds): a delimited stroke is a series of connected road sections, limited by junctions with certain conditions. It is defined with the following type:

```

delimited_stroke = { id:int, geom:linestring,
                    sections: L road_section,
                    level:int,
                    matches: P match }.

```

A delimited stroke is essentially a combination of road segments that form a long road, continuing along the straight part of T-intersections (see junctions E and F in fig. 4). This is also what good continuity entails: the road goes (mostly) straight across these intersections. The goal of creating delimited strokes is that for making matches between two databases, less roads have to be compared to each other.

A delimited stroke is defined at three levels:

- Level 1: a series of connected road sections with good continuity, delimited by terminating nodes. For example in fig. 4,  $A \rightarrow B$  is a delimited stroke, as both junctions  $A$  and  $B$  are terminating nodes. Another example are the combined road sections  $D \rightarrow E$ ,  $E \rightarrow F$  and  $F \rightarrow G$ , because the sections show good continuity at junctions  $E$  and  $F$ , but not at junctions  $D$  and  $G$ . A stroke includes all road sections that have good continuity, it is delimited by the junctions which are either terminating nodes or junctions that do not have additional road sections with good continuity.
- Level 2: a series of connected road sections, delimited by crossings or dead-ends. For example in fig. 4,  $H \rightarrow I$  is a delimited stroke at level 2, because both junctions  $H$  and  $I$  are crossings.
- Level 3: a single road section. In fig. 4,  $J \rightarrow K$  is considered a delimited stroke at level 3, because at junction  $K$  the name of the road changes, resulting in two separate entries in the database between junctions  $J$  and  $L$ .

The different levels of delimited strokes will be used during the matching process. Starting at level 1, the strokes described in different databases are matched to each other. The strokes for which no match can be found are split up in strokes with a higher level of detail, that is level 2. Then, the matching process starts again, but only for the new strokes for which no match was found. This is repeated for strokes with the highest level of detail, level 3.

For a delimited stroke, the following parameters are used during the matching process:

- Geometry: the combined geometry of the road sections in `delimited_stroke.sections`. It is defined as the following constraint:

$$\forall ds:delimited\_stroke \mid ds.geom = st\_union([s.geom \text{ for } s \text{ in } ds.sections])$$

This constraint makes the `geom` attribute of a delimited stroke derivable: it can be calculated at any moment.

- Length: the combined length of each road section in the delimited stroke.

```
length : delimited_stroke → float
length(ds) = st_length(ds.geom)
```

- Area: the area of the polygon which is formed after connecting the first and last points in the delimited stroke with a straight line. In case a valid polygon can not be made (i.e. the delimited stroke is a straight line), the area is returned as 0.

```
area : delimited_stroke → float
area(ds) = st_area(st_makepolygon(st_addpoint(ds.geom, st_startpoint(ds.geom))))
```

The DSO algorithm consists of five main steps:

1. Data pre-processing
2. Indexing connected objects
3. Constructing delimited strokes
4. Matching delimited strokes
5. Treating fragmented areas (i.e. areas with small roads)

A schematic overview is given in fig. 5. The loop in this figure describes the iterative matching approach of the delimited strokes. Delimited strokes that cannot be matched go through the matching process at a higher level of detail, using the three levels described above.

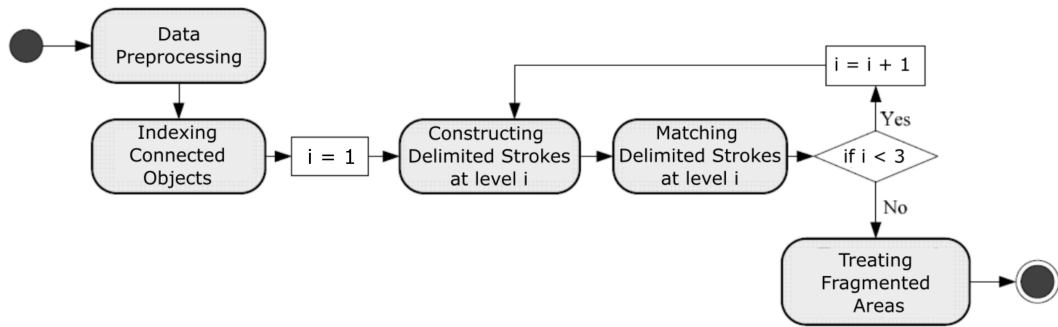


Figure 5: Schematic overview of the DSO algorithm [16]

## 7.2 Data preprocessing

The first step of the DSO algorithm is preprocessing of both input databases. The goal is to prepare the data, such that delimited strokes can be constructed and matched to each other. For this algorithm, it includes the classification of different junction types. The classification variables will be used in the construction of the delimited strokes and during the matching process, to determine which road sections have good continuity and which junctions are similar to each other.

The classification is based on junctions with degree = 3. There are two reasons only these junctions are classified. The first reason is that these junctions are the most common, while junctions with degree  $\geq 4$  are rarer. The second reason is that these junctions are used for the construction of the delimited strokes, as the classification determines whether the roads connected to these junctions have good continuity.

For degree = 3, three different types of junctions can be identified, see table 3. The types are classified with the variable `type_deg3`, with values in the range  $\{0, 1, 2\}$ . The types are based on the largest angle between two road sections, here called  $\alpha$ . The deviation angle  $\theta = \pi/9$  ( $20^\circ$ ) is used to set the condition boundaries for angle  $\alpha$ .

For junctions with `type_deg3` = 1 or 2, the road section marked with angle  $\beta$  can also be used in the comparison of junctions. It is usually a side road of a main road, and its angle is used in the matching process to determine how good two side roads match. The angle is recorded with the variable `angle_deg3`.

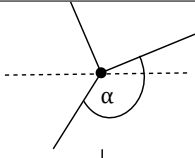
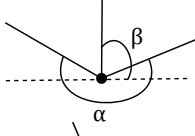
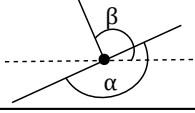
Name	Structure	Condition	<code>type_deg3</code>	<code>angle_deg3</code>
Y-junction		$\alpha < \pi - \theta$	0	
W-junction		$\alpha > \pi + \theta$	1	$\beta$
T-junction		$\pi - \theta \leq \alpha \leq \pi + \theta$	2	$\beta$

Table 3: Classification of junction structures with degree  $d = 3$ ,  $\alpha$  is the largest angle between line segments,  $\beta$  is the angle between the third line segment and the equator (dashed line), and deviation angle  $\theta = \pi/9$ .

## 7.3 Indexing connected objects

The goal of this step is to record connection of road objects with a graph, such that connected components can be easily accessed. The graph is built from road sections as edges and road junctions as nodes. The road section objects are used to create delimited stroke objects and match objects. See the object structure in fig. 6.

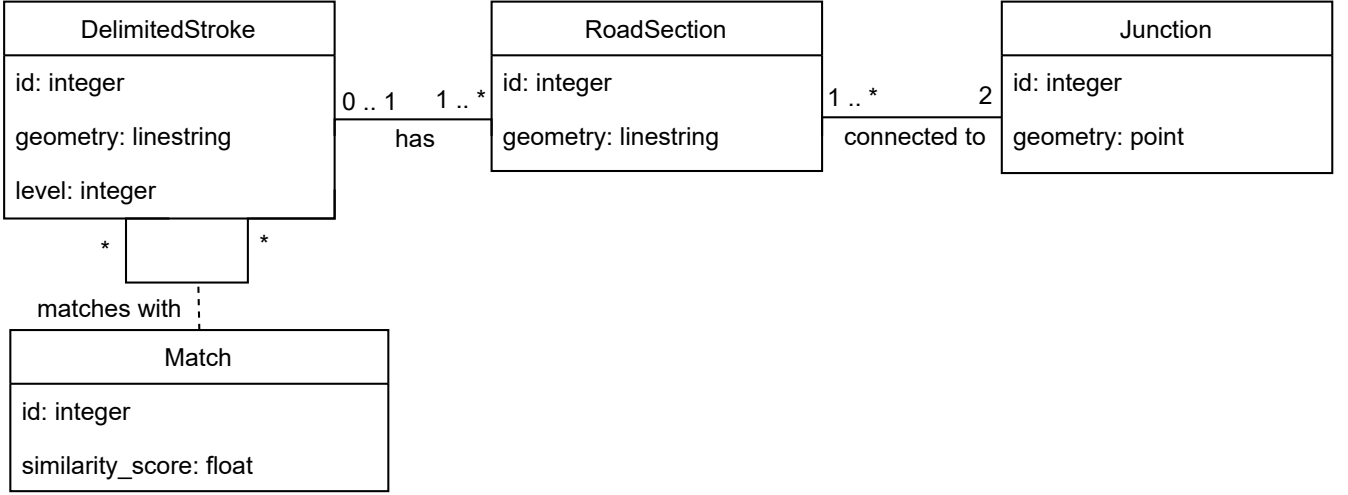


Figure 6: Object structure of a road network, including delimited strokes and match objects.

## 7.4 Constructing delimited strokes

The next step of the DSO algorithm is to construct the delimited strokes for both input databases. The goal is to combine road sections in delimited strokes, such that a matching can be made at a higher level of abstraction. Delimited strokes can be constructed by selecting valid starting road sections, and adding road sections that have good continuity, until a terminating junction is reached. Valid starting road sections are either sections connected to a terminating junction, or the perpendicular section on a T-junction (`type_deg3 = 2` in table 3).

The pseudo-code below displays how this functionality will be implemented. The function `construct_stroke` creates a new instance of `delimited_stroke`, `ds`, from the input `road_section`, `s`. The input `road_junction`, `j`, is a terminating node connected to `s`. The other junction connected to `s` is marked as `next_j`, and is either the begin- or end-junction of `s`, based on what `j` is.

At `next_j`, a new `road_section` will be added to `ds` if it has the right conditions. In case the `degree` of `next_j` is two, it is not a terminating node, and the `road_section` at `next_j` that is not the input `s` will be added to the sections of `ds`. This is done for delimited strokes of level 1 and 2

In case the `degree` of `next_j` is three, the next `road_section` will be added if it has good continuity with `s` at `j`. This is done only for delimited strokes of level 1.

In both cases, `construct_stroke` is called again with the new input, that is the section just added `next_s` at the junction `next_j` and the delimited stroke `ds`. If no `road_section` can be added to the delimited stroke, then it simply returns `ds` at the end.



`construct_stroke`: constructs a delimited stroke from the input road section.

Input:

`road_section`: the first road section of the delimited stroke.

`road_junction`: the begin junction of the delimited stroke.

`int`: the level at which the delimited stroke is constructed.

Output:

`delimited_stroke`: the constructed delimited stroke.

```
construct_stroke : (road_section, road_junction, int) →
                  delimited_stroke
construct_stroke(s, j, lvl) =
  ds ← delimited_stroke(level = lvl, sections = [s])
  s.delimited_stroke = ds
  return construct_stroke(s, j, lvl, ds)
```

`construct_stroke`: extends the input delimited stroke with additional road sections by recursively calling `construct_stroke`, if they conform to the conditions (i.e. have good continuity).

Input:

`road_section`: the last road section added to the delimited stroke.

`road_junction`: the begin junction of the input road section.

`int`: the level at which the delimited stroke is constructed.

`delimited_stroke`: the delimited stroke that is extended. This is the same as the output if no additional road sections can be added.

Output:

`delimited_stroke`: the constructed delimited stroke.

```
construct_stroke : (road_section, road_junction, int, delimited_stroke) →
                  delimited_stroke
construct_stroke(s, j, lvl, ds) =
  if j = s.bjunction
    next_j ← s.ejunction
  else
    next_j ← s.bjunction

  if next_j.degree = 2 and lvl < 3:
    for next_s in next_j.sections where next_s != s
      ds.sections.append(next_s)
      next_s.delimited_stroke = ds
      return construct_stroke(next_s, next_j, lvl, ds)

  if next_j.degree = 3 and lvl = 1:
    for next_s in next_j.sections where next_s != s and
      good_continuity(s, next_s, next_j):
      ds.sections.append(next_s)
      next_s.delimited_stroke = ds
      return construct_stroke(next_s, next_j, lvl, ds)

  return ds
```

The function `construct_strokes` creates a collection of `delimited_stroke` from the input database, by calling `construct_stroke` for each `road_section` of each `terminator junction`.

Then, for each T-junction, that is `type_deg3 = 2`, `construct_stroke` is called for the `road_section` perpendicular to the other two. This is the road section where the angle with the equator is equal to `angle_deg3`, as defined in table 3.

For the remaining road sections without a delimited stroke, `construct_stroke` is called. This ensures that each road section in the database is represented by a delimited stroke.

**construct\_strokes:** constructs delimited strokes from each road section contained in the input database.

Input:

**database:** the database from which the delimited strokes are constructed.

**int:** the level at which the delimited strokes are constructed.

Output:

**P delimited\_stroke:** a collection of delimited strokes.

```
construct_strokes : (database, int) → P delimited_stroke
construct_strokes(db, lvl) =
  result ← []
  for all j in terminators(db):
    for all s in j.sections where s.delimited_stroke = null:
      result.append(construct_stroke(s, j, lvl))

  for all j in d.junctions where j.type_deg3 = 2:
    for all s in j.sections where s.delimited_stroke null and
      angle_at_junction(s, j) = j.angle_deg3:
      result.append(construct_stroke(s, j, lvl))

  for all s in db.sections where s.delimited_stroke = null:
    result.append(construct_stroke(s, s.bjunction, lvl))
  return result
```

## 7.5 Matching delimited strokes

The matching process is iterated over the three levels of delimited strokes. The output of one iteration is a set of matching pairs of strokes from the two input databases. Each stroke that does not have a match is divided in strokes of a higher level of detail, that are used as the input of the next iteration of the matching process.

For the matching process, the two input databases are distinguished as follows:

- Reference database: from this database, each delimited stroke is queried to find a potential match.
- Target database: from this database, the potential matching candidates are selected.

The matching process starts by selecting a delimited stroke from the reference database as input. The next step is to find junction candidates, these are junctions in the target database that are within a certain distance of the begin-junction of the input delimited stroke. This distance is called the tolerance distance. If there are no junction candidates for the begin-junction, a new search is started for the end-junction. If this has no results either, the matching process will end for this delimited stroke.

Since begin- and end-junctions can interchange during the matching process, the following terminology is used regarding junctions:

- **stroke\_ref:** the input delimited stroke from the reference database

- **junction\_ref**: junction connected to **stroke\_ref** for which the junction candidates are found
- **junction\_ref\_other**: junction connected to **stroke\_ref**, that is not **junction\_ref**
- **stroke\_target**: delimited stroke from the target database, connected to **junction\_target**
- **junction\_target**: junction from the target database, that is a candidate for matching to **junction\_ref**
- **junction\_target\_other**: junction connected to **stroke\_target**, that is not **junction\_target**

For each of the junction candidates, the connected delimited strokes are checked whether they match to the input stroke. The tolerance distance is used again, this time to check if **junction\_ref\_other** and **junction\_target\_other** are close to each other. If this is the case, a new instance of **match** is created, since both the begin- and end-junctions of **stroke\_ref** and **stroke\_target** are within the tolerance distance. This is the simplest outcome for a match, called a one-to-one match.

In case the begin- and end-junctions of both strokes are not within the tolerance distance, a many-to-many match has to be created. This is done by the procedure called **extend\_matching\_pair**. This procedure is called when either **junction\_target\_other** is within the tolerance distance of **stroke\_ref**, or when **junction\_ref\_other** is within the tolerance distance of **stroke\_target**.

There is also a special case where the junction candidate is neither the begin- nor end-junction of a delimited stroke, but one in the middle. In this case, the procedure **extend\_matching\_pair** is called when the begin- or end-junction of **stroke\_target** is within the tolerance distance of **junction\_ref\_other**.

**junction\_candidates**: finds a collection of junctions in the input database that are within the tolerance distance of the input junction.

Input:

**database**: the database that contains the possible junction candidates.

**road\_junction**: the junction for which the junction candidates are searched.

**int**: the tolerance distance.

Output:

P **road\_junction**: the collection of road junction candidates.

```
junction_candidates : (database, road_junction, int) → P road_junction
junction_candidates(db, j_ref, dist) =
    result ← []
    for all j_target in db.junctions:
        if get_distance(j_ref, j_target) < dist:
            result.append(j_target)
    return result
```

**matching\_candidates**: finds matches for the input delimited stroke. Starting from the junction candidates, each connected stroke is checked whether they are within the tolerance distance. In case a complete match cannot be made, additional strokes are added using **extend\_matching\_pair**.

Input:

**database**: the database from which the matching candidates are selected.

**delimited\_stroke**: the stroke for which matching candidates are searched.

**int**: the tolerance distance.

Output:

P **match**: a collection of matches.

```

find_matching_candidates : (database, delimited_stroke, int) → P match
find_matching_candidates(db, stroke_ref, dist):
  result ← []
  j_candidates ← junction_candidates(db, stroke_ref.bjunction, dist)
  junction_ref ← stroke_ref.bjunction
  junction_ref_other ← stroke_ref.ejunction
  if j_candidates is []:
    j_candidates ← junction_candidates(db, stroke_ref.ejunction, dist)
    junction_ref ← stroke_ref.ejunction
    junction_ref_other ← stroke_ref.bjunction
    if j_candidates is []:
      return result

  for junction_target in j_candidates:
    for s in junction_target.sections:
      stroke_target ← s.delimited_stroke
      if stroke_target.bjunction is junction_target or
         stroke_target.ejunction is junction_target:
        junction_target_other ← other_junction(junction_target)
        if get_distance(junction_ref_other, junction_target_other) < dist:
          m ← match(stroke_ref, stroke_target)
        else if get_distance(stroke_ref, junction_target_other) < dist or
              get_distance(stroke_target, junction_ref_other) < dist:
          match ← extend_matching_pair([stroke_ref], [stroke_target],
                                       junction_ref_other, junction_target_other, dist)
      else:
        if get_distance(stroke_target.bjunction, junction_ref_other) < dist:
          match ← extend_matching_pair([stroke_ref], [stroke_target],
                                       junction_ref, stroke_target.ejunction, dist)
        else if get_distance(stroke_target.ejunction, junction_ref_other) < dist:
          match ← extend_matching_pair([stroke_ref], [stroke_target],
                                       junction_ref, stroke_target.bjunction, dist)

```

A match can consist of two lists of delimited strokes, one for the reference and one for the target database. Extending a matching pair means that another delimited stroke is added to one of the lists, in order to complete the match.

To determine from which database a delimited stroke should be added, the length of the current delimited strokes in the match is calculated. The shortest stroke is the one that should be extended. Since these can be interchanged between different calls of `extend_matching_pair`, a new set of variables is defined to refer to the strokes and junctions:

- **stroke\_to\_extend**: the short delimited stroke that has to be extended with another stroke to complete the match.
- **stroke\_to\_compare**: the new stroke added to **stroke\_to\_extend** will be compared to this stroke. It is **stroke\_ref** if **stroke\_to\_extend** = **stroke\_target**, or **stroke\_target** if **stroke\_to\_extend** = **stroke\_ref**
- **junction\_to\_extend**: the junction at which the new stroke and **stroke\_to\_extend** are connected.

- `junction_to_compare`: the junction connected to `stroke_to_compare`, used to compare if the new stroke is within the tolerance distance.

The new stroke that is added to `stroke_to_extend` is one connected to `junction_to_extend` with good continuity. If there is no stroke with good continuity, the process `extend_matching_pair` ends and no match will be created.

`new_stroke` is appended to `stroke_to_extend` and `junction_to_extend` is replaced with the other junction that connects to `new_stroke`, as it is now the final junction of the combined strokes.

If the new `junction_to_extend` is within the tolerance distance of `junction_to_compare`, the match is completed. A new instance of `match` is returned, created from the corresponding strokes.

If these junctions are not within the tolerance distance of each other, the matching pair can be extended again. This is done when either `junction_to_extend` is within the tolerance distance of `stroke_to_compare`, or when `junction_to_compare` is within the tolerance distance of `stroke_to_extend`.

`extend_matching_pair`: this function is called when a matching pair is not complete, and a delimited stroke is added to the match where possible. Recursive calls are made to extend the matching even further, if the strokes are within tolerance distance of each other but the end-junctions are not.

Input:

P `delimited_stroke`: a collection of delimited strokes from the reference database.

P `delimited_stroke`: a collection of delimited strokes from the target database.

`road_junction`: the end-junction of the delimited stroke from the reference database.

`road_junction`: the end-junction of the delimited stroke from the target database.

`int`: the tolerance distance.

Output:

`match`: the match created between the collections of delimited strokes.

```
extend_matching_pair : (P delimited_stroke, P delimited_stroke,
                      road_junction, road_junction, int) → match
extend_matching_pair(stroke_ref, stroke_target, junction_ref, junction_target, dist) =
  if get_length(stroke_ref) < get_length(stroke_target):
    stroke_to_extend ← stroke_ref
    stroke_to_compare ← stroke_target
    junction_to_extend ← junction_ref
    junction_to_compare ← junction_target
  else:
    stroke_to_extend ← stroke_target
    stroke_to_compare ← stroke_ref
    junction_to_extend ← junction_target
    junction_to_compare ← junction_ref

  if degree(junction_to_extend) > 1:
    for s in junction_to_extend.sections where
      s.delimited_stroke != stroke_to_extend[-1] and
      good_continuity(s, stroke_to_extend[-1], junction_to_extend):
      new_stroke = s.delimited_stroke

  stroke_to_extend.append(new_stroke)
  junction_to_extend ← other_junction(new_stroke, junction_to_extend)
```

```

if get_distance(junction_to_extend, junction_to_compare) < dist:
    return match(stroke_ref, stroke_target)
else if get_distance(junction_to_extend, stroke_to_compare[-1]) < dist or
    get_distance(junction_to_compare, stroke_to_extend[-1]) < dist:
    if stroke_to_extend == stroke_ref:
        return extend_matching_pair(stroke_ref, stroke_target,
            junction_to_extend, junction_target, dist)
    else if stroke_to_extend == stroke_target:
        return extend_matching_pair(stroke_ref, stroke_target,
            junction_ref, junction_to_extend, dist)
return null

```

When each delimited stroke in the reference database is checked in this way, a number of strokes remain that cannot be matched. The road sections that are part of these strokes are again used as input in the function `construct_strokes`, this time with `level = 2`. After this, a third pass is completed with `level = 3`.

## 7.6 Similarity score

For each delimited stroke, there can be multiple junction candidates, and therefore multiple matches. To determine which match is the best, a similarity score is calculated. This score reflects the similarity of two roads in a match, by comparing their length, distance from each other and area. Each factor in the similarity score is scaled to a tolerance value, this is the maximum allowed value of difference before the factor negatively impacts the score.

**Length:** this is the total length of each road section in the delimited stroke. The length difference is defined as:

$$N(\Delta l) = \frac{|l_1 - l_2|}{\Delta l_{tol}} \quad (1)$$

where  $l_1$  and  $l_2$  are the lengths of two delimited strokes, and  $\Delta l_{tol}$  is the tolerance value for the length difference.

**Distance:** the distance between delimited strokes is determined with the Hausdorff distance. This is defined as the maximum of all distances from a point on one line to the closest point on the other line. The formal definition of the Hausdorff distance between two polylines (PL) is [33]:

$$h(PL_1, PL_2) = \max_{p_n \in PL_1} \min_{p_m \in PL_2} |p_n - p_m| \quad (2)$$

$$H(PL_1, PL_2) = \max(h(PL_1, PL_2), h(PL_2, PL_1)) \quad (3)$$

Here,  $p_n$  and  $p_m$  are points on the respective polylines  $PL_1$  and  $PL_2$ . Normalized to the tolerance value  $H_{tol}$ , this gives:

$$N(H) = \frac{H(PL_1, PL_2)}{H_{tol}} \quad (4)$$

**Area:** To compare the shape of two delimited strokes, a surface area is created by connecting the begin- and end-point of a stroke with a straight line. The area of the created polygon can be calculated, and the difference value is used in the similarity score:

$$N(\Delta S) = \frac{|\frac{S_1}{l_1} - \frac{S_2}{l_2}|}{\Delta S_{tol}} \quad (5)$$

$S_1$  and  $S_2$  are the surface areas, they are scaled to the lengths of the delimited strokes to make a fair comparison, as a longer road is expected to have a larger surface.  $\Delta S_{tol}$  is the tolerance value.

The factors are combined to make a similarity score as a weighted sum:

$$similarity(PL_1, PL_2) = \sum_{i=0}^n w_i(1 - m_i) \quad (6)$$

with  $m_i \in M$ ,  $M = \{N(\Delta l), N(H), N(\Delta S), \}$ ,  $n = |M|$ ,  $w_i \in W$ ;  $W$  is a set of weights where  $\sum_{i=0}^n w_i = 1$ .

The similarity score of a match ranges from 0 to 1, where 1 means the roads in the match are identical. If one or more factor differences exceed the tolerance value, a negative similarity score is expected. Matches with a negative similarity score can be regarded as invalid.

## 7.7 Treating fragmented areas

After the matching process is completed, it is possible to treat fragmented matching areas. These are areas with very small road sections, usually smaller than the tolerance distance set during the matching process. This makes it impossible to find good matching pairs. Treating these areas is done by checking nearby road sections and their matches, which are then combined in a sub-network to match the smaller roads. This process is, however, very complicated and prone to errors, since it relies on previous matches being correct. For these reasons, it will not be considered for the implementation of DSO in this research.

## 8 Implementation

### 8.1 Tools

The data is visualized using QGIS. It is an open source application of the Geographic Information System (GIS), with features that can select road sections based on attribute values.

PostgreSQL is used to store the database. It is an object-relational database system, using SQL queries to add and update data. The PostGIS spatial database extender will be used to store and manipulate geographic data of road sections and junctions. Another extension called pgRouting is used to generate a junction table from the road section data.

pgAdmin 4 is used to manage and administrate the PostGIS database. This tool is open source and often used to develop PostgreSQL projects.

The algorithm is implemented in Python, using the program PyCharm. This is the programming language in which I am the most experienced. Python can integrate with PostGIS databases using the libraries SQLAlchemy and GeoAlchemy 2.

### 8.2 Data preparation

As explained before, the DSO algorithm will be applied to the NWB and TOP10NL databases. A number of steps have to be taken in order to load and process the data.

The format of the databases is called a shapefile. A shapefile is a commonly used format for geographic databases. It consists of three mandatory files, and a number of optional files used for indexing and storing various metadata. The mandatory files contain the following data:

- .shp: shape format; contains the geometric data, in this case the lines describing roads.
- .shx: shape index format; a positional index of the geometric data, that allows forwards and backwards searching.
- .dbf: attribute format; contains all the attributes for each entry in the database, for example road names and cities.

Shapefile data can be visualised in the program QGIS. This program is also used to select and extract small portions of the data, for example only the roads in and around one city or province. This is especially useful during the testing process.

The shapefile data is exported to a local PostGIS database, using the PostGIS shapefile export manager. The PostGIS database is opened in the program pgAdmin. Each road database shapefile initially has one table in the PostGIS database, containing the geometric road data and attributes. A separate junction table has to be generated in order to store additional data about the junctions used in the DSO algorithm.

This is done using the function `pgr_createTopology` from the pgRouting extension. This function creates a network topology from the input road geometry. The points where the roads connect to each other are assigned a unique identifier and saved as `source` and `target` attributes in the road section table. A new junction table is created as well, with the following attributes: `id`, `geometry`, `degree`. If there is a list of existing road junction identifiers, they can be given as input to `pgr_createTopology` as well. This is the case for the NWB database.

Another function called `pgr_analyzeGraph` is used to make sure the roads are properly connected at the junctions, and to check for errors. The error margin is the default of 1mm.

Using the SQL query tool in pgAdmin, some additional changes are made to the data to make it uniform and easier to process. A few new tables are added as well to store new data.



- The function `UpdateGeometrySRID` is used to ensure the correct SRID (spatial reference identifier) value is set, this identifies the coordinate system. For the Netherlands this value is 28992.
- The function `ST_SnapToGrid` is used to snap the geometry of the roads to a grid, to prevent long decimals and make comparisons easier. The snap value is 1mm.
- From the road section tables, the following attributes are selected or added: id, geometry, begin junction id, end junction id, delimited stroke id.
- The following attributes are added to the road junction tables: structure type d3, structure angle d3.
- Two tables for delimited strokes are added, one for the reference and one for the target database. The attributes are: id, geometry, level, begin junction id, end junction id, match id.
- A match table is added, that stores which stroke ids are matched to each other. The attributes are: id, nwb id, top10nl id, match id, similarity score.

This results in a road section table, road junction table and delimited stroke table for each database, plus an additional match table.

### 8.3 Python mapping to SQL

Now that the data is prepared in the PostGIS database, the tables are mapped to Python classes. The library used for this purpose is called SQLAlchemy, and its spatial database extension called GeoAlchemy 2.

The function `declarative_base()` creates a base class, that the mapped classes inherit. The function `Column()` is used to link an attribute from the table to a variable in the mapped class. The function `relationship()` creates a connection from one table to another using foreign keys, in this case these are the road section and road junction ids. This functionality is used in Python to get connected road sections and junctions directly via class variables. For example, `road_junction.sections` returns all the connected road sections to that junction.

In Python, all the functionality of SQL queries can now be used to select necessary data. For example, to select nearby junction candidates from the target database for a junction in the reference database, the PostGIS function `st_dwithin` is called as follows:

```
junction_candidates = session.query(JunctionTarget).filter(
    func.st_dwithin(JunctionTarget.geom, junction_ref.geom, tolerance.distance))
```

This uses the following functionality of SQLAlchemy:

- `session`: the connection to the PostGIS database created at the start of the program.
- `query(TableName)`: performs a query on the input table, in this case `JunctionTarget`. This is equivalent to the `SELECT` statement in SQL.
- `filter()`: filters the content of the query. This is equivalent to the `WHERE` statement in SQL.
- `func`: calls one of the builtin SQL functions.
- `st_dwithin(geomA, geomB, distance)`: returns true if `geomA` and `geomB` are within the input distance of each other.

## 8.4 Python file structure

`__input__.py` : default initialization file, used to initialize the various tolerance values, so they can be used as global variables. This is also where the SQLAlchemy `session` is created to connect to the PostGIS database.

`structure.py` : describes the classes that map to the PostGIS database tables. An additional `Match` class is constructed, that includes the logic to calculate the similarity score.

`core.py` : this is the runnable Python script from which all other functions are called in order. The data from the PostGIS database is loaded with query functions, and used as input for the preprocessing, delimited stroke construction and matching functions. The output of the matching process is written to the matching table in the PostGIS database.

`construction.py` : includes the functionality to classify junctions with the values `type_deg3` and `angle_deg3`. Also includes the functionality to construct delimited strokes from input road sections.

`matching.py` : includes the functionality to find junction candidates, and to search for matches between delimited strokes.

`helpers.py` : includes smaller functions that are used to calculate geometric values of sections and junctions, such as length and angles.

## 9 Validation

In order to validate the algorithm, a test set is matched by hand, and the results are compared to those of the matching of the algorithm. Based on this comparison, multiple matching evaluation metrics can be calculated.

### 9.1 Evaluation metrics

The following definitions will be used to define the evaluation metrics:

- Accurate match or true positive (TP): an object in the reference database is correctly matched to an object in the target database.
- Mismatch (MM): an object in the reference database is correctly identified to have a match, however the object found in the target database is wrong.
- False positive match (FP): an object in the reference database is incorrectly identified to have a match.
- False negative match (FN): an object in the reference database is incorrectly identified to have no match, when in reality there exists one.
- Matching time: the time in seconds to complete the entire matching process

The matching rate, also called matching completeness, is the ratio of objects which have been matched to the total number of objects which should have a match. It is used to see how successful the algorithm is in identifying potential matches.

$$matchRate = \frac{TP + MM}{TP + MM + FN} \quad (7)$$

The matching accuracy is the ratio of accurate matches to the total number of objects which have been matched. It is used to determine the accuracy of the algorithm.

$$matchAcc = \frac{TP}{TP + MM + FP} \quad (8)$$

The matching speed is the total number of objects in the reference database  $N_{ref}$  divided by the matching time. It is used to determine the overall performance of the algorithm.

$$matchSpeed = \frac{N_{ref}}{matchTime} \quad (9)$$

### 9.2 Test results

The selected area for this test is displayed in fig. 7. Here you can see that the road sets are quite similar to each other. The purple lines representing the TOP10NL database are not visible for the most part, because they are covered by the green lines representing the NWB. The area includes a variety of roads and structures, like roundabouts and parallel roads.

The test results are given in table 4, and the parameters used during the testing are given in table 5. The matches made with the DSO algorithm are divided in the categories explained in section 9.1. As you can see in table 4, this results in a matching rate of 0.92 and a matching accuracy of 0.96. One of the reasons for these results is that the area that is matched has a



Figure 7: Area in the centre of the town Nunspeet, used for the hand matched validation. Green = NWB, purple = TOP10NL.

Name database	Reference database NWB	Target database TOP10NL
Number of road sections	278	351
Number of road junctions	218	292
Accurate matches (TP)	217	
Mismatches (MM)	8	
False positives (FP)	1	
False negatives (FN)	20	
matchRate	0.92	
matchAcc	0.96	
matchTime (s)	16.8	
matchSpeed (ref obj/s)	16.6	

Table 4: Evaluation metrics of the matching result of the area in fig. 7.

Parameter	Value
Deviation angle $\theta$	$\pi/9$
Tolerance distance	20
Tolerance for length difference $\Delta l_{tol}$	20
Tolerance for Hausdorff distance $H_{tol}$	20
Tolerance for area difference $\Delta S_{tol}$	1.5
Weight values for similarity score $W$	[0.5 0.35 0.15]

Table 5: Parameters used for testing the algorithm.

similar description in both databases. It still shows that the algorithm is quite accurate for the matches it has determined.

The matching speed, however, is very low compared to other matching results with DSO of 500 obj/s [16]. With only 16.6 obj/s, matching the full databases to each other would take a long time. The reason it is so slow is due to the connection from Python to the SQL database. Each time an object is requested in Python, a new request is sent to the SQL database.

To further analyze the matching rate and matching speed of the algorithm, a number of larger regions have been matched, see figs. 8 to 10. Note that these results are not checked by hand, so the evaluation metrics can not be calculated as explained above. Thus, the matching rate here is taken as the ratio between the number of matched road sections and the total number of road sections in the reference database. This includes road sections for which no match exists in the target database. So while this does say something about the performance of the algorithm in that area, it is not the actual matching rate.

Area	Nunspeet centre	Nunspeet and surroundings	Utrecht	Rotterdam
Road sections NWB	278	1199	10751	15883
Road sections TOP10NL	351	1718	12450	16570
Matching time (s)	16.7	65.7	840.1	1420.8
Matching speed (ref obj/s)	16.6	18.2	12.8	11.2
Matching rate (to total nr of road sections)	0.81	0.79	0.72	0.71

Table 6: Matching results of different matching areas

The results are given in table 6. From this, we can see that the matching speed decreases even further for larger matching areas, with 11.2 obj/s for the largest area. This means that to match the entire NWB of 1.074.691 road section to TOP10NL, it would take at least 26.6 hours. It could be much longer depending on how the matching speed decreases with the number of road sections. The reason the matching speed decreases is likely due to a larger set of data points that have to be analyzed, which causes the search for specific road sections or junctions to take longer.

The matching rate is also lower for the larger areas. There could be two reasons for this. One is that the NWB and TOP10NL have more differences around these areas, resulting in a lower number of matches that can exist. The other is that the road structures in these areas are not correctly handled by the algorithm. For example, a highway intersection is very complex and could cause inaccurate matches.



Figure 8: Matching area of the region around the town Nunspeet. Green = NWB, purple = TOP10NL.

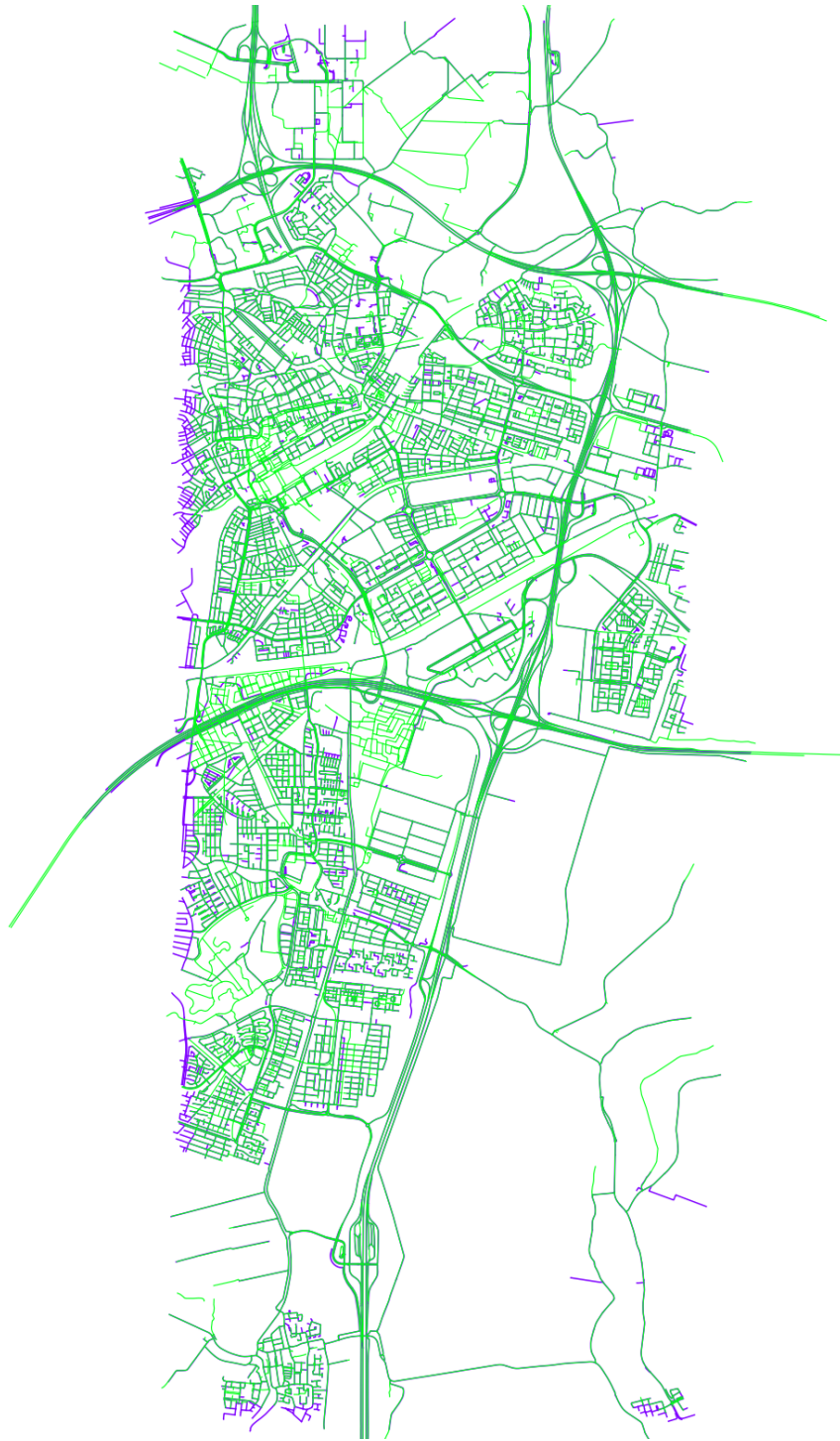


Figure 9: Matching area in the city Utrecht (rotated 90 degrees counterclockwise). Green = NWB, purple = TOP10NL.



Figure 10: Matching area in the city Rotterdam. Green = NWB, purple = TOP10NL.



## 10 Discussion

The results from the validation performed in section 9 show that the matching rate and accuracy are quite high for the selected test area. This is, however, not conclusive, since it could be that this area was easier to match than other areas. Also, the matching speed is far from desirable. In this section, a number of improvements to the implementation and limitations of the algorithm are listed, that can help decide how to achieve better results in the future.

### 10.1 Improvements

On the current implementation, there are a number of features that could be implemented to further increase matching rate and matching accuracy. One is the treatment of fragmented areas, a.k.a. areas with a lot of small roads close together. The issue with these areas is when the length of the roads is smaller than the tolerance distance. This means a road could be matched to a number of similar roads in the area, that are not distinct enough to rank with the similarity score. To solve this, the small roads could be filtered out of the database and treated separately, with smaller values for the tolerance variables.

The similarity score is also something that can be improved. There are two factors that can be calculated to make a better comparison between the roads in a match. The first is the orientation of the road, that is the angle of the straight line connecting the begin- and end-junction of a road. The second is the shape of the road, that describes the number of curves a road has. Other improvements to the similarity score can be made by scaling the tolerance values to the length of the road.

Another feature that can be implemented to increase the effectiveness of the algorithm is a local network comparison. A local network can be created from multiple matches with a high similarity score, that have roads connected to each other. New matches can be created from roads that connect to the network at the same junction, with the same angle. This makes it possible to find partial matches, where only a part of the roads overlap with each other.

### 10.2 Limitations

The first limitation that the DSO algorithm has is that the selection of matching candidates is entirely based on the distance between begin- and end-junctions. This is no problem for the most part, as a lot of roads in both databases are quite close to each other. There are however two cases where this is an issue.

The first case is that of highway exits: there is no clear rule for where a highway exit begins, which results in large difference between the locations of the junctions involved, up to 100m. Simply increasing the tolerance distance in the matching process is not helpful, since this results in a lot of wrong matching candidates for areas with a high road density, such as city centres.

The second case is the matching of certain dead-ends. Just like with highway exits, the location of the junction that marks a dead-end is not always clear, and is different between databases. This causes the same issue, where the difference between two junctions is too large to confidently create a match.

The second limitation involves the creation of delimited strokes at level 1. At certain junctions, it is possible for multiple roads to have good continuity. This issue arises at roads that split into two parallel roads, for example at highway exits. One solution to this would be to add the road that has an angle closest to 180° to the delimited stroke. However, the angles at these junctions are not consistent between the databases, so the issue still remains.

### 10.3 Performance and scalability

From the test results, it is clear that the current implementation has performance issues. Comparing the matching speed from this implementation (11 obj/s) to that from Zhang [16] (500 obj/s), the difference is quite big. In order to scale the implementation to match the entire NWB (over 1 million road sections), the matching speed has to be improved.

As was explained earlier, the low matching speed is due to the high number of repeated connection requests from Python to the SQL databases. Because the Python classes are directly mapped to SQL tables, each time an attribute is requested from an object, a new query is executed to the SQL database. The number of queries can be reduced by saving data locally in Python dictionaries, and only making queries for object searching in the database.

The main functionality of SQL in the current implementation is from the function `st_dwithin`. In combination with the automatically generated spatial index, it is possible to quickly find objects within a certain distance of other objects in the database. Other operations that are done quickly in SQL involve selecting a subset of the database based on certain conditions, for example selecting all delimited strokes that could not be matched.

The other functions that are currently used in SQL involve geometric operations, like computing the length or angle of objects. Also, the selection of connected road objects is done via database relationships, i.e. requesting the begin junction of a road section with its ID value. In order to reduce frequent SQL queries, these functions can be handled by Python. Geometric calculations are possible with certain geometric libraries, for example GeoPandas. Requesting objects via ID values is possible if the data is saved in Python dictionaries. The only problem with this is likely memory limitation, as the dictionaries will be built during runtime.

Memory limitation is, besides matching speed, the other problem involved with scalability. To scale up to databases of at least one million objects each, enough memory should be available to store and analyze the data. One way to circumvent this is to split the data in equal sized chunks, and match each chunk separately. The borders of the chunks should have enough overlap, such that roads at the borders can be matched as well.

Another way to improve the matching speed and scalability is by reimplementing the algorithm in SQL only. Python is known to be slow and has memory issues, especially when dealing with large data sets. The main reason for choosing Python in the first place was for ease of implementing a working version of the algorithm. Using only SQL would remove the overhead that comes with connecting Python to SQL. It would also increase the efficiency of queries, by requesting multiple objects at once.

## 11 Conclusion

This report describes the different techniques used for (road) network matching. In order to find the best technique to implement for matching NWB to TOP10NL, the techniques with the most promising results are compared to each other.

Currently, the road network matching technique used at Rijkswaterstaat is map-matching, using the route analysis tool in the program ArcGIS. With this tool, they were able to match and add 88% of cycle paths from TOP10NL to the NWB. The program ArcGIS is however not open source, meaning it is not possible to improve their results using the same technique.

As for other state of the art techniques for road network matching, they consist of DSO, PRM and PSO. The results of their respective researches are compared in section 6.

According to the corresponding reports, DSO gives the best results in accuracy (99.1%), while PSO is the fastest for matching large areas. Since DSO still has a reasonable computation speed (498 obj/s) for matching large areas, it is chosen as the technique that was implemented to match the NWB to TOP10NL.

The matching techniques in other domains, which are biology and sociology, were not used in the implementation. This is, because they are primarily focused on matching nodes, while for matching road networks the information that describes the location of the edges (i.e. roads) is more important.

As DSO was chosen as the best method for road network matching, a detailed description of the algorithm is given in section 7. The algorithm was implemented in Python according to this description, using PostgreSQL as database storage.

The test results of the implementation are given in section 9. While the matching rate (92%) and accuracy (96%) are very high, they are not indicative for the performance of the algorithm on the entire database, since only a small area could be hand-matched to check the results.

The matching speed of the implementation is very low (11 obj/s) compared to the results of other reports using DSO [16]. The reason for this is the high number of connection requests from Python to SQL. Because of this, the current implementation cannot be reliably scaled to match the entire NWB database to TOP10NL.

Solutions to this problem are to either reduce the number of calls from Python to SQL, by doing simple geometric calculations in Python, or to re-implement the algorithm using only SQL.

## References

- [1] Ron Dalumpines and Darren M Scott. GIS-based map-matching: Development and demonstration of a postprocessing map-matching algorithm for transportation research. In *Advancing geoinformation science for a changing world*, pages 101–120. Springer, 2011.
- [2] Mohammed A Quddus, Washington Y Ochieng, and Robert B Noland. Current map-matching algorithms for transport applications: State-of-the art and future research directions. *Transportation research part c: Emerging technologies*, 15(5):312–328, 2007.
- [3] Egbert Moerland. Het Nationaal Wegenbestand (NWB). <https://arcg.is/0Lfu0v>, 09 2019. Accessed: 2020-02-14.
- [4] RWS. *Handleiding Nationaal Wegenbestand (NWB-Wegen)*, May 2019. Version 0.1.
- [5] ISO/TC 204. *Intelligent transport systems - Geographic Data Files (GDF) - GDF5.0*, July 2011.
- [6] Open Geospatial Consortium et al. OpenGIS Implementation Specification for Geographic information-Simple feature access-Part 2: SQL option. *OpenGIS Implementation Standard*, 2010.
- [7] EPSG:28992, RD New. <https://epsg.io/28992>. Accessed: 2020-01-02.
- [8] Kadaster. *Basisregistratie Topografie: Catalogus en Productspecificaties*, July 2019. Version 1.2.0.1.
- [9] Map matching. [https://en.wikipedia.org/wiki/Map\\_matching](https://en.wikipedia.org/wiki/Map_matching). Accessed: 2020-12-11.
- [10] Francisco Câmara Pereira, Hugo Costa, and Nuno Martinho Pereira. An off-line map-matching algorithm for incomplete map databases. *European Transport Research Review*, 1(3):107–124, 2009.
- [11] Fabrice Marchal, Jeremy Hackney, and Kay W Axhausen. Efficient map matching of large global positioning system data sets: Tests on speed-monitoring experiment in zürich. *Transportation Research Record*, 1935(1):93–100, 2005.
- [12] Paul Newson and John Krumm. Hidden Markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 336–343. ACM, 2009.
- [13] Piotr Szwed and Kamil Pekala. An incremental map-matching algorithm based on hidden Markov model. In *International Conference on Artificial Intelligence and Soft Computing*, pages 579–590. Springer, 2014.
- [14] David Fiedler, Michal Čáp, Jan Nykl, Pavol Žilecký, and Martin Schaefer. Map matching algorithm for large-scale datasets. *arXiv preprint arXiv:1910.05312*, 2019.
- [15] Meng Zhang and Liqiu Meng. Delimited stroke oriented algorithm-working principle and implementation for the matching of road networks. *Geographic Information Sciences*, 14(1):44–53, 2008.
- [16] Meng Zhang. *Methods and implementations of road-network matching*. PhD thesis, Technische Universität München, 2009.

- [17] Wenbo Song, James M Keller, Timothy L Haithcoat, and Curt H Davis. Relaxation-based point feature matching for vector map conflation. *Transactions in GIS*, 15(1):43–60, 2011.
- [18] Yunfei Zhang, Bisheng Yang, and Xuechen Luan. Automated matching crowdsourcing road networks using probabilistic relaxation. *ISPRS annals of photogrammetry, remote sensing and spatial information sciences*, pages 281–286, 2012.
- [19] Bisheng Yang, Yunfei Zhang, and Xuechen Luan. A probabilistic relaxation approach for matching road networks. *International Journal of Geographical Information Science*, 27(2):319–338, 2013.
- [20] Lin Yang, Zejun Zuo, Run Wang, Yaqin Ye, and Maosheng Hu. Matching road network combining hierarchical strokes and probabilistic relaxation method. *Review of. Open Automation and Control Systems Journal*, 6:268–76, 2014.
- [21] Jianchen Zhang, Yanhui Wang, and Wenji Zhao. An improved probabilistic relaxation method for matching multi-scale road networks. *International journal of digital earth*, 11(6):635–655, 2018.
- [22] Lin Yang, Fang Fang, Songling Dai, Bo Wan, and Zejun Zuo. Road network matching method based on particle swarm optimization algorithm. *The Open Cybernetics & Systemics Journal*, 8:1286–1292, 2014.
- [23] Bo Wan, Lin Yang, Shunping Zhou, Run Wang, Dezhi Wang, and Wenjie Zhen. A Parallel-Computing Approach for Vector Road-Network Matching Using GPU Architecture. *ISPRS International Journal of Geo-Information*, 7(12):472, 2018.
- [24] Connor Clark and Jugal Kalita. A comparison of algorithms for the pairwise alignment of biological networks. *Bioinformatics*, 30(16):2351–2359, 05 2014.
- [25] Matching algorithms (graph theory). brilliant.org. <https://brilliant.org/wiki/matching-algorithms/>. Accessed: 2019-11-11.
- [26] Ahmet E Aladağ and Cesim Erten. Spinal: scalable protein interaction network alignment. *Bioinformatics*, 29(7):917–924, 2013.
- [27] Mohammed El-Kebir, Jaap Heringa, and Gunnar W. Klau. Lagrangian relaxation applied to sparse global network alignment. In Marco Loog, Lodewyk Wessels, Marcel J. T. Reinders, and Dick de Ridder, editors, *Pattern Recognition in Bioinformatics*, pages 225–236, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [28] Gregory Tauer, Rakesh Nagi, and Moises Sudit. The graph association problem: mathematical models and a Lagrangian heuristic. *Naval Research Logistics (NRL)*, 60(3):251–268, 2013.
- [29] Hang TT Phan and Michael JE Sternberg. Pinalog: a novel approach to align protein interaction networks—implications for complex detection and function prediction. *Bioinformatics*, 28(9):1239–1245, 2012.
- [30] Bo Wang, Aziz M Mezlini, Feyyaz Demir, Marc Fiume, Zhuowen Tu, Michael Brudno, Benjamin Haibe-Kains, and Anna Goldenberg. Similarity network fusion for aggregating data types on a genomic scale. *Nature methods*, 11(3):333, 2014.
- [31] Alireza Farasat, Geoffrey Gross, Rakesh Nagi, and A.G. Nikolaev. Social network analysis with data fusion. *IEEE Transactions on Computational Social Systems*, PP:1–12, 11 2016.

- [32] Meng Zhang, Wei Yao, and Liqiu Meng. Automatic and accurate conflation of different road-network vector data towards multi-modal navigation. *ISPRS International Journal of Geo-Information*, 5(5):68, 2016.
- [33] Daniel P Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on pattern analysis and machine intelligence*, 15(9):850–863, 1993.