DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Road traffic congestion detection and tracking with Spark Streaming analytics

THORSTEINN THORRI SIGURDSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Road traffic congestion detection and tracking with Spark Streaming analytics

THORSTEINN THORRI SIGURDSSON

Master's Thesis
Supervisor: Zainab Abbas
Industrial Supervisor: Ahmad Al-Shishtawy, RISE-SICS
Examiner: Vladimir Vlassov

TRITA-EECS-EX-2018:652

Abstract

Road traffic congestion causes several problems. For instance, slow moving traffic in congested regions poses a safety hazard to vehicles approaching the congested region and increased commuting times lead to higher transportation costs and increased pollution.

The work carried out in this thesis aims to detect and track road traffic congestion in real time. Real-time road congestion detection is important to allow for mechanisms to e.g. improve traffic safety by sending advanced warnings to drivers approaching a congested region and to mitigate congestion by controlling adaptive speed limits. In addition, the tracking of the evolution of congestion in time and space can be a valuable input to the development of the road network.

Traffic sensors in Stockholm's road network are represented as a directed weighted graph and the congestion detection problem is formulated as a streaming graph processing problem. The connected components algorithm and existing graph processing algorithms originally used for community detection in social network graphs are adapted for the task of road congestion detection. The results indicate that a congestion detection method based on the streaming connected components algorithm and the incremental Dengraph community detection algorithm can detect congestion with accuracy at best up to $\approx 94\%$ for connected components and up to $\approx 88\%$ for Dengraph. A method based on hierarchical clustering is able to detect congestion while missing details such as shockwaves, and the Louvain modularity algorithm for community detection fails to detect congested regions in the traffic sensor graph.

Finally, the performance of the implemented streaming algorithms is evaluated with respect to the real-time requirements of the system, their throughput and memory footprint.

Keywords: streaming, graph processing, congestion, community detection, connected components

Referat

Vägtrafikstockningar orsakar flera problem. Till exempel utgör långsam trafik i överbelastade områden en säkerhetsrisk för fordon som närmar sig den överbelastade regionen och ökade pendeltider leder till ökade transportkostnader och ökad förorening.

Arbetet i denna avhandling syftar till att upptäcka och spåra trafikstockningar i realtid. Detektering av vägtrafiken i realtid är viktigt för att möjliggöra mekanismer för att t.ex. förbättra trafiksäkerheten genom att skicka avancerade varningar till förare som närmar sig en överbelastad region och för att mildra trängsel genom att kontrollera adaptiva hastighetsgränser. Dessutom kan spårningen av trängselutveckling i tid och rum vara en värdefull inverkan på utvecklingen av vägnätet.

Trafikavkännare i Stockholms vägnät representeras som en riktad vägd graf och problemet med överbelastningsdetektering är formulerat som ett problem med behandling av flödesgrafer. Den anslutna komponentalgoritmen och befintliga grafbehandlingsalgoritmer som ursprungligen användes för communitydetektering i sociala nätgravar är anpassade för uppgiften att detektera vägtäthet. Resultaten indikerar att en överbelastningsdetekteringsmetod baserad på den strömmande anslutna komponentalgoritmen och den inkrementella Dengraph communitydetekteringsalgoritmen kan upptäcka överbelastning med noggrannhet i bästa fall upp till ≈94% för anslutna komponenter och upp till ≈88% för Dengraph. En metod baserad på hierarkisk klustring kan detektera överbelastning men saknar detaljer som shockwaves, och Louvain modularitetsalgoritmen för communitydetektering misslyckas med att detektera överbelastade områden i trafiksensorns graf.

Slutligen utvärderas prestandan hos de implementerade strömmalgoritmerna med hänsyn till systemets realtidskrav, deras genomströmning och minnesfotavtryck.

Contents

1 Introduction **1**
1.1 Background . . . 2
1.1.1 The data set . . . 2
1.1.2 Congestion detection . . . 3
1.1.3 Intelligent Transport Systems . . . 4
1.2 The problem . . . 5
1.3 Benefits . . . 6
1.4 Contributions . . . 6
1.5 Methodology . . . 6
1.5.1 Research method . . . 7
1.5.2 Ethical considerations . . . 7
1.6 Limitations . . . 8
1.7 Structure of the thesis . . . 8

2 Traffic flow theory **9**
2.1 Main metrics . . . 9
2.2 Traffic congestion . . . 10
2.2.1 Relationship with capacity . . . 10
2.3 The fundamental diagram of traffic flow . . . 11
2.4 Metrics for congestion detection . . . 12
2.5 Traffic queues . . . 13
2.6 Shockwaves . . . 14

3 Theoretical background and related work **15**
3.1 Community detection . . . 15
3.1.1 Community detection in weighted graphs . . . 16
3.1.2 Girvan-Newman algorithm and edge-betweenness . . . 16
3.1.3 Modularity . . . 16
3.1.4 Louvain modularity . . . 18
3.1.5 Hierarchical clustering for community detection . . . 19
3.1.6 Density based methods . . . 20
3.1.7 Dengraph . . . 20
3.2 Apache Spark . . . 22

3.2.1 Streaming in Spark . . . 23
3.2.2 Spark Structured Streaming . . . 24
3.3 Apache Kafka . . . 26
3.4 Related work . . . 28
3.4.1 Congestion detection . . . 28
3.4.2 End-of-queue detection . . . 30

4 Implementation **33**
4.1 The road network graph . . . 33
4.1.1 Graph types . . . 33
4.1.2 Time-span graphs . . . 34
4.1.3 How the graph is used . . . 34
4.1.4 Graph construction . . . 35
4.2 Identifying congested sensors . . . 38
4.2.1 Congestion classes . . . 39
4.3 Batch approaches . . . 40
4.3.1 Congested components . . . 41
4.3.2 Louvain modularity . . . 41
4.3.3 Hierarchical (data) clustering . . . 42
4.3.4 Dengraph . . . 43
4.4 Streaming approaches . . . 44
4.4.1 Data set . . . 45
4.4.2 Stream source . . . 45
4.4.3 System architecture . . . 45
4.4.4 Spark Structured Streaming programming abstractions . . . 46
4.4.5 Watermarking . . . 47
4.4.6 Congested components . . . 48
4.4.7 Incremental Dengraph . . . 50
4.4.8 Queue tracker . . . 53
4.4.9 File sink programs . . . 56

5 Evaluation **59**
5.1 Ground truth . . . 59
5.2 Batch approaches . . . 60
5.2.1 Louvain modularity . . . 60
5.2.2 Hierarchical (data) clustering . . . 62
5.3 Chosen test congestion patterns . . . 62
5.4 Experimental setup . . . 64
5.5 Accuracy evaluation . . . 66
5.5.1 Congested components . . . 69
5.5.2 Dengraph . . . 70
5.5.3 Dengraph noise resistance . . . 72
5.6 Detecting individual queues . . . 73
5.7 Queue tracker . . . 75

5.8 Performance evaluation . . . 76
5.8.1 Experimental setup . . . 76
5.8.2 Performance evaluation results . . . 79
5.8.3 Effect of parameter selection on performance . . . 81
5.8.4 Comparison of congestion detection algorithms . . . 83
5.8.5 Queue tracking algorithm . . . 84

6 Conclusions and future work **89**
6.1 Future work . . . 91
6.1.1 Scalability considerations . . . 91
6.1.2 Ground truth and accuracy evaluation strategy . . . 91
6.1.3 End-of-queue warning system . . . 92
6.1.4 End-of-queue detection system . . . 92
6.1.5 Integrate with streaming predictions . . . 92
6.1.6 Evolutionary clustering . . . 93
6.1.7 Continuously update minimum free flow speed per sensor . . . 93

Appendices **94**

A Test congestion pattern heat maps **95**

B Congested components results **103**

C Dengraph results **111**

D Accuracy evaluation result tables **121**

Bibliography **127**

Chapter 1

Introduction

Congestion in road traffic systems poses several problems. Traffic congestion leads to increased pollution and fuel consumption [1], detrimental effects on both psychological and physical health [2][3], increased commuting times for drivers with higher transportation costs [4], and finally reduced traffic safety due to the speed differential between free flowing traffic and congested traffic [5], to name a few.

Congestion mitigation strategies are therefore an important part of the operation of a traffic system. The focus of this thesis project is on real-time congestion detection and tracking with the goal to improve traffic safety by enabling mechanisms to improve drivers' situational awareness. With real time congestion detection in place, systems to send advanced warnings to drivers approaching the end of a traffic queue can be implemented. The information can also be used to control variable speed limits in an effort to improve safety or mitigate congestion.

The real-time aspect is essential in the context of traffic safety in order to communicate relevant information to drivers about the current traffic conditions. However, the results and methods presented in this thesis could also be used in an offline setting, providing information on the congestion behaviour of a road system to the traffic authorities which could be used as guidance in the development of the road system.

The project involves processing of sensor measurements from traffic sensors placed around Stockholm's road system. The sensors are represented as a weighted directed graph and the problem of congestion detection is formulated as a graph processing problem. Streaming graph processing algorithms are implemented to perform real-time congestion detection and tracking, with methods based on streaming connected components as well as a community detection algorithm originally used for the analysis of social network graphs.

1.1 Background

The thesis project is part of an ongoing research project at RISE SICS called BADA (Big Data Analytics for Automation)[1]. The research project is a collaboration between RISE SICS, Volvo cars & trucks, Scania trucks and the Swedish road traffic authority, Traffikverket.

1.1.1 The data set

The data set used in the project consists of radar sensor measurements provided by Traffikverket. The data was collected from 2005 to 2016, totalling 391 GB.

A total of 2059 sensors have been placed at various locations in Stockholm's road network, as well as on a single road in Göteborg. The sensors are concentrated on major traffic arteries covering 67 distinct roads. The distribution of the sensors over Stockholm's road network can be seen in figure 1.1.

Figure 1.1: The distribution of traffic sensors over Stockholm's road network.

The sensors are organized in groups spanning all lanes of a road at a given location, with a single sensor responsible for each lane. Each group of sensors is identified by a road ID, as well as a kilometer reference number, relative to the starting point of each road. Each sensor within the group is further identified by a lane ID, counted from the rightmost lane starting at 1. An example of a sensor array spanning 4 lanes can be seen in figure 1.2. The sensor arrays are spaced about 150-400 meters apart.

[1] http://bada.sics.se/

Figure 1.2: A sensor array. This one is located on E4N at kilometer reference 55650, as can be seen on the yellow signs. The sensor furthest to the right has lane ID 1, the sensor next to it lane ID 2, and so on. Picture taken from Google Maps' street view.

Each sensor gives a reading every minute, reporting measured values averaged over the past minute. The sensor measurement data fields of interest for this project are listed in table 1.1.

In addition to the sensor readings, the data set contains meta-data on the sensors. It includes, among other things, the GPS coordinates of each sensor along with valid-from and valid-to dates, signifying the dates that the sensor was installed and removed (if applicable).

1.1.2 Congestion detection

Traffic congestion can be detected using a number of different methods. At a high level, the methods can be classified as taking either a microscopic or macroscopic view of the traffic flow.

Microscopic methods are concerned with observing traffic flow on an individual vehicle basis. The use of probe vehicles falls in this category, where vehicles send GPS traces and/or speed measurements to a centralized location for analysis [6][7]. The location traces and speed measurements can then be used to identify congestion. Congestion can also be detected by examining link journey times of vehicles. The link journey time of a vehicle is the time it takes the vehicle to get from point A to some point B downstream. Abnormally high link journey times are then

Field	Example	Explanation
Timestamp	2016-11-03 03:47:00	The timestamp of the sensor measurement.
Ds_Reference	E4Z 53,115	Location of the sensor. Road and kilometer reference. The kilometer reference is given in kilometers from the start of the road.
Detector_Number	49	The lane number of the sensor, in ASCII. This one would be lane number 49-48 = 1.
Flow_In	6	The number of cars that passed the sensor in the past minute, in cars/min.
Average_Speed	104	Average speed of cars passing the sensor in the past minute, in km/h.
Status	3	A status code for the sensor reading. Status 3 represents OK.

Table 1.1: Relevant data fields of each sensor measurement.

taken to indicate congestion. The calculation of link journey times requires the identification of a vehicle at point A, and subsequent re-identification at point B. This can be achieved through the use of number plate recognition cameras [8], and the analysis of traffic sensor data giving measurements on a per vehicle basis, e.g. measured vehicle length [9]. Vehicle to vehicle communication systems are also able to detect congestion. A vehicle receives e.g. speed information from other vehicles downstream and compares their speed to its own current speed, allowing the vehicle to detect if it is approaching congestion [10].

Macroscopic methods on the other hand view traffic flow in aggregate instead of observing the movement of individual vehicles. Camera and video surveillance of traffic [11], satellite imagery [12] and acoustic sensors [13] can be used to detect congestion on a macroscopic level, as well as traffic sensors giving aggregated traffic measurements such as average speed and traffic flow (number of cars per time unit).

This thesis project will take a macroscopic view of the traffic flow in Stockholm's road network, detecting and tracking traffic congestion using aggregated traffic measurements from existing radar-based infrastructure traffic sensors.

1.1.3 Intelligent Transport Systems

An Intelligent Transport System is a transport system that tries to provide innovative services with respect to traffic management and enable the users of the traffic system to be better informed and make smarter use of the transport system. This is

achieved through the use of e.g. information and communication technologies [14].

The work performed in this thesis can be considered as part of the Intelligent Transport Systems field. The methods proposed and implemented could be integrated as a function in an Intelligent Transport System, providing the ability to detect and track congestion in real-time based on infrastructure sensor data.

1.2 The problem

The goal of this thesis project is to use measurements from the traffic sensors placed around Stockholm's road system to detect and track the evolution of traffic congestion in real time. Instead of performing congestion detection at each sensor individually this thesis project aims to detect congestion over a number of adjacent sensors, detecting congestion not just at discrete sensor locations but extending the detection into the spatial dimension. A sequence of connected congested sensors can then be thought of as representing a traffic queue[2]. In addition to detecting the individual traffic queues in the road system, the evolution of the queues through time should also be tracked, allowing for monitoring of a queue as it may grow, shrink, split, travel through the road network, and finally dissipate. This should be done at as fine a resolution as the underlying data sources allow, i.e. down to the individual road lane level.

In order to achieve this, a graph will be constructed to represent the road system. This graph should model the spatial relationship between different traffic sensors as well as the road segments between them. Using this graph, the real-time congestion detection and tracking problem can be formulated as a graph streaming problem. The real-time requirements are that the processing of each minute's worth of sensor data should be done in a streaming fashion and complete in well under a minute, to ensure that the processing can keep up with the rate of data received from the sensors.

The questions this project aims to answer are the following:

- Can we represent the sensors in the traffic network as a graph? What kind of graph suits our problems the best?
- If we construct a graph to represent the traffic network, what methods can be used to detect and track congestion through the network?
- What graph streaming algorithms can be used to detect and track congestion in real-time?

[2] The term queue is used to refer to a series of adjacent sensors experiencing congestion, not an actual row of cars waiting to be served.

1.3 Benefits

The real time detection of traffic congestion allows for improved traffic safety by enabling the implementation of systems to send warnings to drivers approaching the end of a queue, and control adaptive speed limits to reduce the speed differential between free flowing traffic and congested traffic. Adaptive speed limits as well as traffic rerouting strategies can then also be employed in an effort to mitigate congestion.

The tracking of traffic congestion allows for the congestion behaviour of a road system to be mapped. The identification of the formation, growth, movement and dissipation of congestion can guide future development of the road system to improve its efficiency. For instance, by identifying locations where congestion on a certain road builds up to extend across an intersection and disrupt traffic on an otherwise uncongested road (queue spillback).

1.4 Contributions

The main contributions of this thesis project are as follows:

- The road infrastructure sensors are represented as a directed weighted graph, allowing for the detection of congested regions in the road network.
- Streaming graph processing algorithms were adapted to be used for congestion detection by analyzing the weighted sensor graph, namely connected components and community detection algorithms. Community detection algorithms originally intended for social network graph analysis were re-purposed for traffic congestion detection. The connected components and community detection algorithms allow for the detection of traffic sensor groups connected by edges with similar readings of average speed or traffic density. The detected sensor groups reflect different traffic states, e.g. free flow or congestion.
- The adapted connected components and community detection algorithms were evaluated with respect to the application at hand, namely congestion detection.
- An open source implementation[3] of streaming graph processing algorithms considered in this thesis is provided, implemented using Apache Spark Structured Streaming [15].

1.5 Methodology

Initial data exploration was done on the Hops Apache Spark cluster [16], exploring large amounts of historical data. As the goal of the project is to implement a streaming system, large amounts of historical data are not needed for the implementation

[3] https://github.com/thorsteinnth/road-congestion-detection

and evaluation of the system. A subset of the data, containing sensor measurements gathered over three consecutive days, was therefore extracted for processing on a local machine. This was done to facilitate a more reliable development environment.

In the first phase of the project the road system graph was constructed. Four different congestion detection approaches were then implemented on batch data and evaluated. The two best performing approaches were then selected for implementation as streaming systems, using Spark Structured Streaming.

As no ground truth is available for the existence and propagation of traffic congestion, the accuracy of the results of the implemented congestion detection and tracking methods were evaluated visually through the use of spatio-temporal heatmaps comparing observed congestion patterns (discernible by a human expert) to the detected congestion patterns. The accuracy of the congestion detection methods was evaluated with respect to different values of the parameters used by the methods.

Finally, the performance of the implemented systems with regards to execution time and memory requirements was evaluated quantitatively, comparing both the differences between the different systems, as well as the effects of different values for the parameters used by the methods.

1.5.1 Research method

The project follows the applied research method. The applied research method involves solving practical problems using existing research and real-world data [17]. The work performed in the project uses real world traffic sensor data and builds on existing research on graph processing to solve a practical problem, namely real-time congestion detection and tracking.

The experimental research method is also employed. It tries to establish relationships and causalities between different variables through experimentation [17]. The different congestion detection methods implemented in the thesis project are compared, and the effects of different parameters for the methods are compared quantitatively.

1.5.2 Ethical considerations

The main ethical consideration of the project is whether the data set used contains any sensitive information, and whether the proposed methods allow for the identification and tracking of individual vehicles.

The data set consists of sensor measurements aggregated over one minute long timespans from fixed locations in the road network. It contains no information that can be used to identify individual vehicles within the data set, and thus there is no way to track the movement of individual vehicles.

During the implementation of the project images from Google Maps' Street View were used as a reference to get a clear picture of the actual set up of different road segments and intersections in the road network. Some of these images have

been included in this thesis. Google takes care to blur any sensitive or identifiable information in their Street View product, such as license plates. All images used in this thesis have been confirmed to show no sensitive information.

1.6 Limitations

Two limitations are imposed on the project due to the nature of the data source:

- The accuracy of the proposed congestion detection methods is limited by the spatial resolution of the traffic sensors. The sensors measure traffic variables at fixed locations, and the measurements are assumed to apply to the road segments between the sensor locations. This assumption may not always be accurate, especially if the distance between two consecutive sensor locations is great. The distance between sensor locations in the road network graph under study is about 150-400 meters.
- Due to the sensors only giving measurements at one minute intervals, the implemented congestion tracking methods will not be truly real-time.

1.7 Structure of the thesis

The remainder of the thesis is organized as follows. Chapter 2 gives the theoretical background in traffic theory that this thesis is based on. Chapter 3 gives the theoretical background with respect to computer science, as well as an overview of related work. Chapter 4 describes the steps taken during the implementation of the work, followed by an evaluation of the work in chapter 5. Finally, chapter 6 gives a conclusion and directions for future work.

Chapter 2

Traffic flow theory

This chapter will give an overview of the traffic flow theory that this thesis is based on.

2.1 Main metrics

Traffic flow is characterized by three main metrics; flow, speed and density [18]. These measures take a macroscopic view of traffic, describing the movement of the traffic stream as it flows through the road system instead of looking at each vehicle individually.

Flow (F) is defined as the number of vehicles passing a certain point or road segment in a given unit of time. It is typically expressed as number of vehicles per hour.

Speed (v) is measured in units of distance over units of time, typically kilometers per hour. In the case of macroscopic analysis of traffic streams, the speed of each vehicle passing a particular point in the road system can be measured, and the average speed of all vehicles that have passed the point in a particular time interval calculated (temporal average speed). Another way to calculate average speed is to measure the travel time of each vehicle between two fixed locations, and calculate the average speed as the distance between the two fixed locations divided by the average travel time of all the vehicles driving between the locations (spatial average speed).

Density (D) is defined as the number of vehicles per unit of distance. It is typically expressed as the number of vehicles per kilometer.

These three metrics are related by the following equation:

$$F = v \times D \tag{2.1}$$

Using equation 2.1, one can use values for two of the parameters to estimate the value of the third. Since density is very hard to measure directly with physical sensors it is often estimated from the relationship with the two other parameters, flow and speed [18].

Estimating density directly in this manner is a simplification though. The speed of vehicles is measured at the fixed location of the sensor and averaged over a given time interval. Density however is a measure of the number of vehicles on a unit length of road. Taking a speed measurement at a fixed location to estimate a spatially defined metric can lead to discrepancies from the real density values. If vehicle speed and flow are spatially inhomogeneous, the temporal averaging of speed measurements taken at a fixed location can give a very different result from spatial averaging of vehicle speeds made over a road segment of a given length. One should be vary of this, especially at higher vehicle densities [19].

The data set used in this project however contains only flow and (temporal) average speed measurements. Density is then calculated from the flow and average speed using equation 2.1.

2.2 Traffic congestion

There are many possible causes for congestion. Features of the road system itself may cause congestion, such as on- and off-ramps, lane merges, road curves and road gradients, etc. It can also be caused by external and intermittent factors such as accidents, road works and bad weather. Furthermore, traffic jams can form without any obvious bottleneck impeding the flow of traffic. If the density of vehicles on the road exceeds a certain critical value, small fluctuations in the movement of vehicles (caused by drivers adjusting their speed to react to the movement of surrounding vehicles) will grow until eventually the free flow of vehicles breaks down and a traffic jam forms [20].

Roughly, traffic flow can be either "free" or "congested" [19]. It is said that traffic is in "free flow" when it is possible for vehicles to drive, change lanes, overtake, and in general perform any maneuver the driver wishes [21]. Congested traffic flow can then be defined as the complement to free flow, i.e. when the conditions on the road do not allow for the free movement of vehicles [19].

2.2.1 Relationship with capacity

The capacity of a traffic facility refers to how much traffic the facility can carry in a sustainable way, defined in terms of hourly flow rate [22].

When the flow rate of the traffic facility approaches or exceeds the facility's capacity the traffic flow through the facility can become congested. This transition from free flow to congestion is known as breakdown [22].

Breakdown of free flow is characterized by a drop in speed along with the formation of queues [22]. The identification of breakdowns is usually done by observing a speed drop greater than some threshold with the added constraint that the speed drop must persist longer than some minimum duration, indicating that the flow of traffic has entered a congested state [18].

The Highway Capacity Manual from 2010 (HCM2010) defines the maximum capacity for a freeway segment as 45 vehicles per mile per lane [22]. This translates

to a density of just under 28 vehicles per kilometer per lane. If the density exceeds this threshold, a breakdown into congestion can be expected.

2.3 The fundamental diagram of traffic flow

The fundamental diagram of traffic flow shows the relationship between two of the main traffic flow metrics; vehicle density and flow rate.

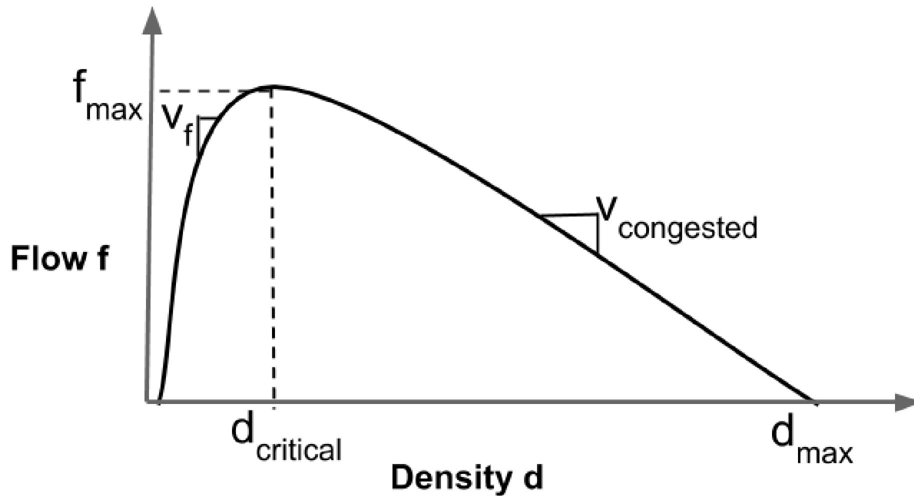


Figure 2.1: The fundamental diagram of traffic flow.

By analyzing the fundamental diagram it is possible to identify the transition from free flow to congested flow (breakdown). Empirical data points from free flow roughly line up on a curve with a positive slope, until a certain empirical maximum point of free flow is reached. This point is denoted by the dotted lines in figure 2.1. The maximum flow rate is marked as f_{max} and the corresponding density value $d_{critical}$ is the critical density value where free flow breaks down and congestion takes over. Congested data points can then be identified in the fundamental diagram as the points roughly following a curve with negative slope, originating at the maximum point of free flow [19].

In addition to looking at the critical density as the threshold where congestion takes over from free flow, one can also identify the free flow-congestion boundary with respect to average speed. This can be done by identifying the empirical maximum point of free flow ($d_{critical}$, f_{max}) and drawing a line from the origin of the fundamental diagram to that point. The slope of this line gives the minimum empirical (average) free flow speed, according to equation 2.1. Average speed measurements can then be classified as belonging to free flow if they fall on the left side of the line (speed higher than minimum free flow speed) or the right side of the line

(speed lower than minimum free flow speed) [19]. Figure 2.2 shows an example of a fundamental diagram plotted with empirical data, and the division of data points into free flow and congested regions.

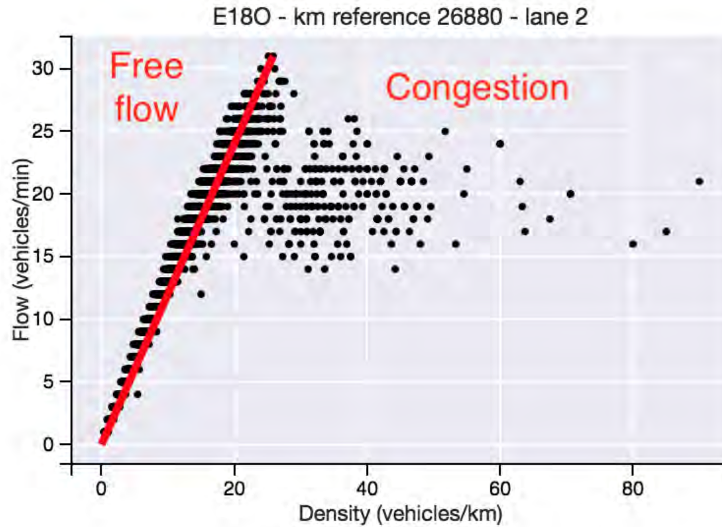


Figure 2.2: Empirical fundamental diagram with minimum free flow speed line FC. Points to the left of the red FC line are classified as belonging to free flow, while the points on the right side of the line are classified as congestion.

Fundamental diagrams from any highway show similar shapes, indicating that they capture the underlying physical relationship between the vehicle density and flow rate. Furthermore, the critical density has almost the same value for different highways [20].

2.4 Metrics for congestion detection

As discussed in section 2.3 both vehicle density and average speed can be used as congestion indicators. Congestion can occur when the density of vehicles on the road exceeds a certain critical threshold. Sensor measurements can then be classified as belonging to congestion if the measured density is higher than this threshold. Alternatively, the minimum free flow speed at the sensor location can be found by calculating the average speed at the maximum free flow point. Sensor measurements can then similarly be classified as congestion if the measured average speed is lower than this minimum free flow speed. As previously discussed, while the minimum free flow speed can be different between different locations in the road network, the critical density is less variable, at 28 vehicles per kilometer per lane.

When classifying sensor measurements as belonging to either free flow or congestion, both metrics have their drawbacks. In the case of average speed, one can imagine a single vehicle driving in the middle of the night at low speeds (perhaps a

truck transporting a heavy load such as a house during the night). The sensor measurements from this vehicle would incorrectly be classified as congestion. In the case of density, a dense group of vehicles might still be travelling at high speeds (think of a NASCAR race). Sensor measurements from this group would also incorrectly be classified as congestion.

It can be argued that the NASCAR scenario is less likely in the real world than the single vehicle at low speeds scenario, and density would therefore be the better congestion indicator if deciding between density and average speed. However, in this thesis the assumption is made that a car will travel at the maximum safe (and legal) speed, and that such anomalies like a single car driving at very low speed are unlikely. Furthermore, one of the goals of the thesis is to detect congestion to increase traffic safety. If a single vehicle is driving slowly it may not be correctly classified as congestion, however it still poses a safety risk as it is driving well under the historical speed measured at the road segment. Therefore it is still useful to detect it so that cars approaching from downstream may be notified.

The methods for congestion detection implemented in this thesis will use both average speed measurements as well as density measurements to define thresholds for congestion.

2.5 Traffic queues

A traffic queue can be defined as a row of vehicles waiting to be served. The queue length is usually defined as the number of vehicles waiting to be served [18].

Queue formation is an important issue in traffic systems. They pose a safety hazard as cars driving in free flow at high speeds come up on a queue travelling at lower speeds. They can also disrupt the flow of traffic that would otherwise not need to be served by whatever traffic system feature the queue is formed around. For instance, if a queue that forms at the exit ramp of a freeway grows long enough, the tail of the queue can extend into the mainline freeway, blocking traffic that is not headed for the exit ramp. This phenomenon is called queue spillback [18].

It is tricky to define the exact boundaries of a queue. One has to determine which vehicles to consider as part of the queue. It could be only vehicles that are at a standstill, or vehicles travelling at a speed lower than some threshold. This is even more complex when analyzing queues that form on freeways (as opposed to e.g. intersections with traffic lights) since freeway queues move slowly and traffic may flow in a "stop-and-go" fashion over long distances [18].

In this thesis we will use the term queue for a section of road with higher density or lower average speed than the road sections up- and downstream of the queue section. The end of the queue (EOQ) is then the upstream front of the queue (i.e. the tail of the queue).

2.6 Shockwaves

A shockwave is defined as a change or discontinuity in traffic conditions [22]. More precisely, shockwaves are a propagation of change in flow and density [18].

Shockwaves travel through the road network (either upstream or downstream) with a certain speed. The speed with which they travel is a function of the flow and density differences in the regions on either side of the discontinuity [18].

For example, shockwaves can form around intersections with traffic lights. When the light turns red a queue forms at the traffic lights. This queue then grows upstream as more vehicles arrive at the red light. The movement of the end of the queue upstream in the road network is a shockwave. When the light turns green, another shockwave is formed as vehicles start to drive through the intersection. This shockwave however moves downstream through the road network [22].

Chapter 3

Theoretical background and related work

This chapter will go into the theoretical background of the project, as well as related work.

3.1 Community detection

Community detection in graphs refers to the problem of finding community structure within graphs. Communities can be defined as subsets of vertices such that the connections between vertices within each subset are more dense than connections between different subsets [23].

Community detection has been used to analyze a wide variety of graphs, e.g. social networks, protein-protein interaction networks and the World Wide Web. The meaning of a community depends on the domain from which the graph originates and the goals of each application. For instance, a community in a protein-protein interaction network might reveal proteins that have the same function within a cell while a community in the World Wide Web might reveal web pages that cover the same or similar topics. In general, the goal of community detection is to find groups of vertices that share common properties and/or have similar roles in the graph [24].

In this thesis community detection is used to identify road traffic sensors that share similar properties, i.e. similar measured values. The end goal being to identify a "community" of sensors that represents a congested area of the road system, i.e. a queue.

The community detection problem is sometimes called graph clustering, and the communities the methods are attempting to identify are then similarly called clusters instead of communities.

3.1.1 Community detection in weighted graphs

According to the definition of a community given in section 3.1, a community is a subset of vertices that has more *dense* connections between the vertices within the subset than to vertices not in the subset. Interpreting the *density* mentioned in the definition to refer only to the number of connections works well for complex graphs such as social networks and the World Wide Web. However, the road system graph is a simple, planar graph, where the sensors along a lane of road form a simple chain, so that the degree of each vertex in the graph is rarely higher than 2 (only in the case of lane additions/merges and road intersections).

Therefore, instead of only relying on the number of connections between vertices in the graph to identify communities, we need to assign a weight to each connection. The *density* of connections mentioned in the definition of a community then arises from the weights of the connections instead of just the number of connections.

Furthermore, we are not interested in finding communities based solely on the structure of the graph (i.e. the set of vertices and the existence/non-existence of an edge between them). The road system graph is static, but the sensor measurements we assign to it change every minute. Therefore it is natural to treat the sensor measurements as weights to the graph and look towards weighted community detection algorithms to identify communities based on the weights along with the graph structure.

3.1.2 Girvan-Newman algorithm and edge-betweenness

Girvan and Newman introduced a community detection method based on the idea of edge betweenness [23]. The edge betweenness of an edge is defined as the number of shortest paths between pairs of vertices that run along the edge. The idea is then that the edges that connect different communities (edges between communities) will have a high edge betweenness as many shortest paths between pairs of vertices, where each vertex falls into a different community, will pass through them. By removing the edges with high edge betweenness it is then possible to separate the communities in the graph [23].

This method is of limited usefulness for this project, since the road system graph is simple and planar. As mentioned in section 3.1.1, the basegraph in its simplest form is just a chain of vertices. In that case, the edge betweenness is not a suitable measure for community detection. Edge betweenness can however be adapted to weighted graphs by simply dividing the edge betweenness of an edge with the edge's weight [24]. However, the intuition behind edge betweenness still does not apply to a simple chain-like graph.

3.1.3 Modularity

Modularity is a quality measure for the division of a graph. It was introduced in 2004 by Girvan and Newman as a quality measure for their edge betweenness community detection algorithm [25], discussed in section 3.1.2 .

Modularity gives a quantifiable measure of the quality of the communities discovered by a community detection algorithm, as a single number in the range of -1 to 1.

In most practical cases the community structure of the graph under investigation is not known beforehand and must, as the name suggests, be discovered by the applied community detection algorithm. Modularity thus gives a way to evaluate how "good" the result of the community detection algorithm is. The communities discovered in a graph with no real underlying community structure are then expected to have a low modularity value of close to 0, while the communities discovered in a graph with a clear community structure will (hopefully) have a high modularity value approaching 1.

Hierarchical community detection algorithms such as the Girvan-Newman algorithm will produce a dendrogram of the entire hierarchy of possible community divisions for the graph. It is then possible to choose the best division by selecting the one which maximizes modularity. Modularity can in this way be used to decide where to cut the dendrogram. It can also be interpreted as a stopping criterion for the algorithm, i.e. by halting the algorithm when a certain satisfactory modularity value has been reached, eliminating the need to compute the entire community division hierarchy (dendrogram).

Modularity can be used to assess the quality of community divisions generated by community detection algorithms, but also as an objective function which algorithms seek to optimize [26]. After its introduction, modularity optimization became the most popular method for community detection[24]. Modularity optimization is an NP-complete problem however, so the community detection algorithms based on modularity maximization try to find good approximations in a reasonable time [24]. An example of one of these algorithms is the Louvain modularity algorithm discussed in section 3.1.4.

Modularity definition

Modularity is defined as [25]:

$$Q = \sum_i (e_{ii} - a_i^2) \tag{3.1}$$

where $a_i = \sum_j e_{ij}$.

If a graph is split into k communities, $\mathbf{e}$ is a $k \times k$ symmetric matrix whose element e_{ij} is the fraction of all edges in the graph that connect vertices in community i to vertices in community j. The sum of the elements on the main diagonal of $\mathbf{e}$ (i.e. the trace of $\mathbf{e}$), $\sum_i e_{ii}$, gives the fraction of edges in the graph that connect vertices within the same community, while a_i gives the fraction of edges that connect to vertices in community i.

Modularity measures the fraction of edges in the graph that connect vertices within the same community, minus the expected fraction in a graph with the same community division but with edges connecting vertices randomly [25].

As previously stated, the possible values for the modularity Q lie in the range of [-1, 1]. If Q is greater than 0, the fraction of edges connecting vertices within the same community is higher than what would be expected in a graph with random connections between vertices. If $Q = 0$, the community structure of the graph under consideration is no better than what would be expected in a random graph. As the community structure of the graph gets more defined, the modularity values will approach $Q = 1$ [27].

3.1.4 Louvain modularity

Louvain modularity [28] refers to a community detection method based on modularity optimization. The method focuses on scalability, finding the high modularity partitions of large graphs in a short time. It also reveals a hierarchical community structure for the graph, allowing inspection of the detected communities at different resolutions

The Louvain modularity algorithm works on weighted graphs. Modularity for weighted graphs is defined as [28]:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta\Big(c_i, c_j\Big), \tag{3.2}$$

where A_{ij} is the weight of the edge connecting vertices i and j, $k_i = \sum_j A_{ij}$ is the sum of the weights of all the edges connecting to vertex i, c_i is the community that vertex i is assigned to, $\delta(u, v)$ is 1 if $u = v$ otherwise 0, and $m = \frac{1}{2} \sum_{ij} A_{ij}$.

The algorithm consists of two phases that are repeated iteratively. Initially, each vertex of the graph is assigned to its own community.

The first phase of the algorithm is as follows: For each node i, look at all neighbors of i and evaluate the change in modularity if i is moved from its current community and placed into the community of the neighbor. After examining all neighbors, move node i into the neighbor community which results in the greatest gain in modularity (and only if the gain is positive). In the case of a tie, a tie breaking rule is used. If there is no positive gain found then i stays in its current community. This is repeated sequentially for all nodes until no further improvement to modularity can be achieved, at which point this first phase of the algorithm is complete.

In the second phase of the algorithm a *new graph* is constructed. The vertices in this graph are the communities found in the first phase of the algorithm. The weights of the edges between the community-vertices in this new graph are the sum of the weights of edges between vertices in the two communities that the community-vertices we are linking represent. Links between nodes of the same community lead to self loops in this new graph.

Once the second phase is complete the algorithm moves on to the next iteration, performing the steps of phase 1 on the new graph constructed in phase 2 of the

previous iteration. The iteration continues until there are no more changes to be done in phase 1 of the algorithm and maximum modularity has been reached.

The algorithm constructs a hierarchy of community divisions from the bottom up (agglomerative), with the result of each iteration representing a level in the hierarchy.

3.1.5 Hierarchical clustering for community detection

Hierarchical clustering is one of the "traditional" methods for community detection [24]. To begin with, a weight W_{ij} is computed pairwise between all vertices in the graph. Starting with the set of vertices in the graph (and no edges), an edge is added between pairs of vertices in the order of the computed weights. First, an edge is added between the pair of vertices with the highest weight, then the next highest, and so on [23].

As more edges are added, connected components of vertices appear, with each connected component representing a community. A hierarchical structure to the communities also materializes. The communities can be represented by a dendrogram with the edge weights representing the height of the tree, decreasing by height. A cut of the dendrogram at a given height X thus gives all the communities connected by edges with a weight greater than X.

There are various alternatives for how to define the weights between two vertices. For instance, the weight can be defined as the number of node-independent paths between vertices. Node-independent paths are paths that share no vertices (nodes) other than the start and end vertices. Other possible weight definitions are for instance the number of edge-independent paths, or the total number of paths between two vertices (i.e. not only node- or edge-independent) [23]. Note that these weights are derived from the structure of the graph, and not some application specific weight definition.

Relationship with connected components approach

This traditional community detection method is interesting in the context of this thesis since it resembles the connected components approach implemented in the project (see section 4.3.1). However, the weights in the connected components approach are application specific weights (sensor measurements) and not derived from the graph structure. The connected components method can be thought of as starting with the set of all vertices in the graph (and no edges), and adding edges between pairs of vertices in the order of decreasing weights, until a certain weight threshold is reached. However, an edge is only added between two vertices if the vertices have an edge connecting them in the underlying graph (equivalent to setting the weight as 0 for all pairs of vertices that are not directly connected by an edge). The weight treshold is equivalent to cutting the dendrogram at the height of the threshold value. The resulting connected components then represent communities.

Note that the weight threshold represents the dendrogram cutoff point. The hierarchical structure can be explored in the resulting connected components by choosing a new, higher, weight threshold, in effect moving the dendrogram cut off point lower. It would then be possible to remove all edges with a weight lower than this new threshold from the connected components result, and extract the communities at a lower level in the hierarchy.

3.1.6 Density based methods

Density based clustering approaches view clusters as a set of data objects lying in a contiguous region with a high density of objects in the data space, separated from other clusters by contiguous regions with a low density of objects. They do not require the number of clusters k to be defined beforehand, and can handle clusters of arbitrary shapes [29].

3.1.7 Dengraph

Dengraph [30] is a density based graph clustering algorithm inspired by the data clustering algorithm DBSCAN. Dengraph identifies dense regions in the data space by defining a neighborhood of a given radius ϵ around each data point and examining if this neighborhood contains some minimum number of data points η. If it does, the neighborhood around the given data point is considered dense.

To identify which data points fall within the neighborhood, a distance function is required to determine the distance between data points. The original Dengraph paper is concerned with finding community structure in a social network and defines the distance between two actors p, q in the network as [30]:

$$dist(p,q) = \begin{cases} 0 & p = q \\ min(I_{p,q}, I_{q,p})^{-1} & (I_{p,q} > 1) \wedge (I_{q,p} > 1) \\ 1 & otherwise, \end{cases} \tag{3.3}$$

where $I_{p,q}$ is the number of interactions between actors p and q that were initiated by p.

As previously discussed, the algorithm works by looking at a neighborhood of a given distance radius around each data point, and examining the number of data points that fall in this neighborhood. More formally, the ϵ-neighborhood of vertex p in graph $G(V,E)$ is defined as $N_\epsilon(p) = \left\{q \in V \mid \exists(p,q) \in E \wedge dist(p,q) \leq \epsilon\right\}$.

Vertices in the graph are classified into one of the three following types: *Core*-, *noise*- or *border*-vertex. A neighborhood is considered dense if it contains at least some set minimum number of data points. A vertex p is classified as a *core* vertex if and only if the neighborhood around it contains at least η other data points, i.e. $\|N_\epsilon(p)\| \geq \eta$. If vertex p is not a core vertex, it is classified as a *noise* vertex, unless p is itself part of the neighborhood of some other core vertex q, i.e. $p \in N_\epsilon(q)$, in which case p is classified as a *border* vertex.

The algorithm has three classifications for the relationships between vertices; direct density-reachability, density-reachability and density-connectivity. It uses these relationship classifications to define what constitutes a cluster. For graph $G(V, E)$ and vertices $p, q \in V$, they are defined as [31]:

- p is *directly density-reachable* from q within V w.r.t. ϵ, η if and only if q is a core vertex and p is in its neighborhood, i.e. $p \in N_\epsilon(q)$
- p is *density-reachable* from q within V w.r.t. ϵ, η if there is a chain of vertices $p_1, ..., p_n$ such that $p_1 = q$ and $p_n = p$ and for each $i = 2, ..., n$ it holds that p_i is directly density-reachable from p_{i-1} within V w.r.t. ϵ, η. Density-reachability is denoted as $p >_V q$.
- p is *density-connected* to q within V w.r.t. ϵ, η if and only if there is a vertex $o \in V$ such that $p >_V o$ and $q >_V o$.

To summarize, the vertices in the neighborhood of a core vertex are said to be directly density-reachable from the core vertex. A vertex p is said to be density-reachable from vertex q if there is a chain of directly density-reachable vertices linking them. Note that this means that q must be a core vertex. Finally, two vertices p, q are said to be density connected if they are both density reachable from some third vertex o. In this case, neither p nor q needs to be a core vertex. These relationships can be seen graphically in figure 3.1.

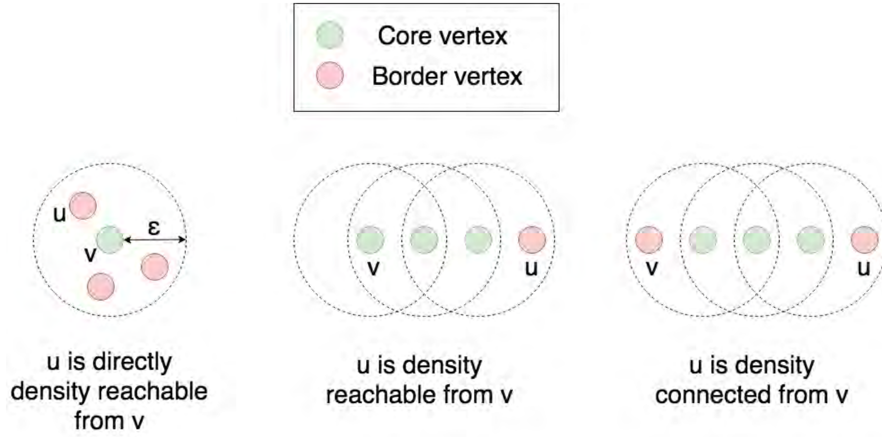


Figure 3.1: Concepts of relationships between vertices in Dengraph. Adapted from [32].

Finally, a cluster in a graph $G(V, E)$ is defined to be a "dense subgroup" $DS \subseteq V$ of all vertices that are density-connected within V w.r.t. ϵ, η. Formally, a dense subgroup in $G(V, E)$ is defined as [31]:

- A non-empty set $DS \subseteq V$ is a "dense subgroup" w.r.t. ϵ, η if and only if:

- For all $p, q \in V$ it holds that if $p \in DS$ and $q >_V p$, then $q \in DS$ (maximality condition).
- For all $p, q \in DS$ it holds that p is density-connected to q within V w.r.t. ϵ, η (connectivity condition).

Dengraph can deal with noisy data effectively. Vertices that fall into a cluster are classified as either core or border vertices, while data points that do not fall into a cluster are classified appropriately as noise vertices. The parameter η controls the minimum number of data points needed within the neighborhood of a vertex for the neighborhood to form a cluster. Increasing η and thus requiring the neighborhood of a vertex to be more dense results in more data points being classified as noise. This mechanism allows the algorithm to efficiently remove outliers, explicitly classifying them as noise [31].

An incremental version of the algorithm is also available. It is able to receive a stream of edges and update the computed clustering incrementally, by adding new edges, updating current edges, and removing old ones [31][32].

Further details of the algorithm (both static and incremental) will be given in the implementation chapter of this thesis, sections 4.3.4 and 4.4.7.

3.2 Apache Spark

Apache Spark [15] is a general purpose cluster computing system. Originally, it was built to allow efficient iterative computation on large volumes of batch data in a cluster setting [33]. Since then, features such as stream processing, machine learning libraries and graph processing libraries have been added.

The main abstraction in Spark is the *resilient distributed dataset (RDD)*. An RDD represents a read-only collection of elements which is partitioned across multiple machines in the cluster for scalability, and can be re-built automatically in case of node failures. An RDD can be persisted in memory, allowing for efficient iterative computation [33].

Since the RDD is split into a number of partitions each partition can be operated on in parallel, achieving scalability. There are two types of RDD operations: *Transformations* and *actions*. Transformations take an existing RDD, apply some transformation to it, and return a new RDD. Actions on the other hand perform some computation on the RDD and return the result to the driver program. Transformations are evaluated lazily, only being computed when an action which requires the result of the transformations is run [34]. Fault tolerance of RDDs is achieved by remembering the sequence of transformations (lineage) that resulted in the creation of an RDD. If a partition of an RDD is lost, it is possible to rebuild the RDD from some base dataset (e.g. a file) by re-applying the transformations of the lineage [33].

Support for structured data processing with SparkSQL is also available. It allows users to leverage the extra structure information while programming, for instance by

running SQL queries on the data, and SparkSQL to perform extra optimizations in its execution engine. The main programming abstractions in SparkSQL are *Datasets* and *DataFrames*, both of which are based on RDDs. A Dataset is a distributed collection of data, same as RDDs, which takes advantage of the optimized execution engine of SparkSQL. A DataFrame is a Dataset which is organized into named columns and can be thought of as a table in a relational database [35].

The cluster architecture of Spark consists of a *driver* program, *executors* and a cluster manager. The driver and executors together form a Spark application. The driver program is the main program, housing a `SparkContext` object. The `SparkContext` is used to coordinate the execution of tasks on worker nodes in the cluster. These tasks are run within processes called executors. Outside of the main Spark application a cluster manager is used for resource allocation on the cluster. Possible cluster managers are Spark's own standalone cluster manager, Mesos[1], YARN[2] and Kubernetes[3] [36]. A diagram of Spark's cluster architecture can be seen in figure 3.2.

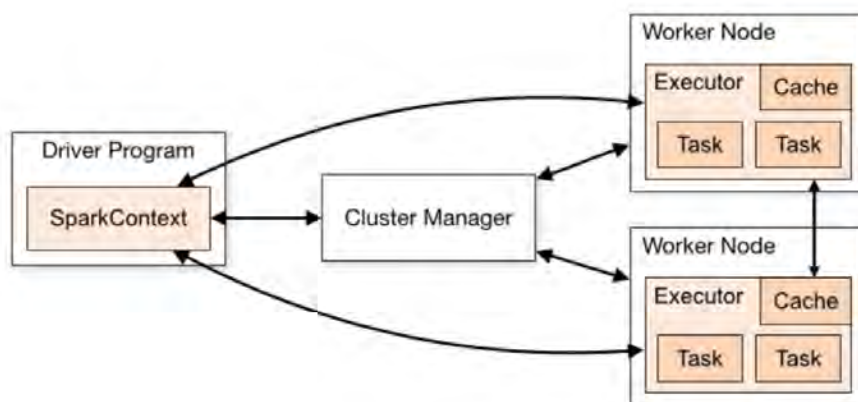


Figure 3.2: Spark cluster architecture [36].

Apache Spark was chosen for this project since it can be used for both batch processing and stream processing. Furthermore, the graph processing libraries GraphX[4] and GraphFrames[5] are available in Spark, which is of interest for the graph related problems of the project.

3.2.1 Streaming in Spark

Spark offers two streaming APIs: Spark Streaming and Spark Structured Streaming.

[1]https://mesos.apache.org/
[2]https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[3]https://kubernetes.io/
[4]https://spark.apache.org/graphx/
[5]https://graphframes.github.io

The main abstraction in **Spark Streaming** is a *discretized stream* or *DStream*. Under the hood, a DStream is a continuous sequence of RDDs with each RDD containing a small batch of data from a certain interval (micro-batching) [37]. Transformations can be performed on a DStream to produce another DStream, and output operations can be used to write the result of the transformations to an output sink (analogous to actions). An example of a transformation can be seen in figure 3.3.

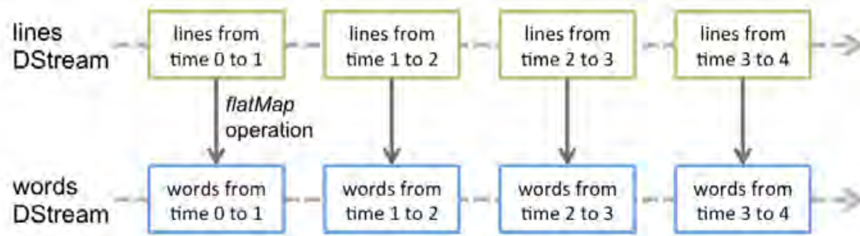


Figure 3.3: Spark Streaming DStream transformations. Each RDD in the DStream contains data from a certain time interval [37].

Spark Structured Streaming is built on top of the SparkSQL engine. It uses the same Dataset and DataFrame API as when working with batch data, with the SparkSQL engine adapting it for streaming computation under the hood. This allows the user to program the same way as would be done in a batch setting, removing the need to reason about streaming [38].

This project was implemented using Spark Structured Streaming. Since the batch data processing in the project was done using the DataFrame API of Spark, it is more convenient to opt for Spark's Structured Streaming as the concepts and code are similar.

3.2.2 Spark Structured Streaming

As previously stated, the main abstraction in Spark Structured Streaming is a DataFrame. However, in the streaming setting the table that the DataFrame represents is unbounded instead of static. As new records arrive in the stream they are appended to this table, with the table growing indefinitely (conceptually). An illustration of the unbounded table can be seen in figure 3.4.

The programming model in Spark Structured Streaming is very similar to batch processing using SparkSQL. New data arriving from a stream source gets appended to an unbounded *input table*. A sequence of transformations or a query can then be applied to the input table, with the end result being written to a *result table*. The contents of the result table can then be written to an external sink, such as a file or a topic in Kafka.

The queries are however run incrementally. Every *trigger interval*, new data is read from the stream source, appended to the input table, run through the query and the result table updated. This trigger interval delimits the micro-batches that

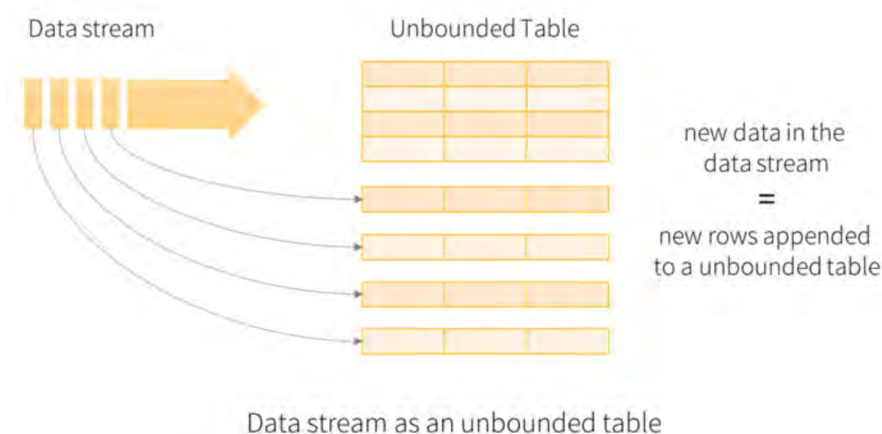


Figure 3.4: Unbounded table in Spark Structured Streaming [38].

run through the streaming system. The trigger can be manually set to some time interval (e.g. 1 second), or left undefined, in which case a new micro-batch will be generated as soon as the previous micro-batch finishes processing, given that there is new data available for processing [38]. An illustration of the data flow from source to sink can be seen in figure 3.5.

By default Spark Structured Streaming uses micro-batch processing, similar to Spark Streaming. In micro-batch processing mode Spark can give exactly-once fault tolerance guarantees, meaning that each record will be processed exactly once even in the event of a failure, and achieve end-to-end latencies as low as 100 milliseconds. A new processing mode called *Continuous Processing* was introduced in Spark 2.3 which can get end-to-end latencies down to 1 millisecond while giving at-least-once fault tolerance guarantees [38]. In this case a continuous trigger is used. However, the API is still experimental and aggregations are not yet supported (as of Spark 2.3.1), so this was not explored in the project.

Fault tolerance is achieved using checkpointing and write ahead logs. The offsets of the data being read from a source during a given trigger is saved so that if a failure occurs, the data which was being processed at the time of failure can be re-processed (replayed). The output sinks are then designed to be idempotent, so any replayed records will not result in duplicates. Any intermediate state of the running queries (e.g. aggregates) is also checkpointed [38].

In addition to the built in aggregations, Structured Streaming supports arbitrary stateful processing through the operations `mapGroupsWithState` and `flatMapGroupsWithState`. Using these operations, developers can define their own arbitrary state and run any user defined code to update the state. The operations operate on grouped DataSets, maintaining the user defined state for each group. The streaming algorithms implemented in this project were implemented using these operations. Finally, Spark Structured Streaming supports watermarking to handle late data.

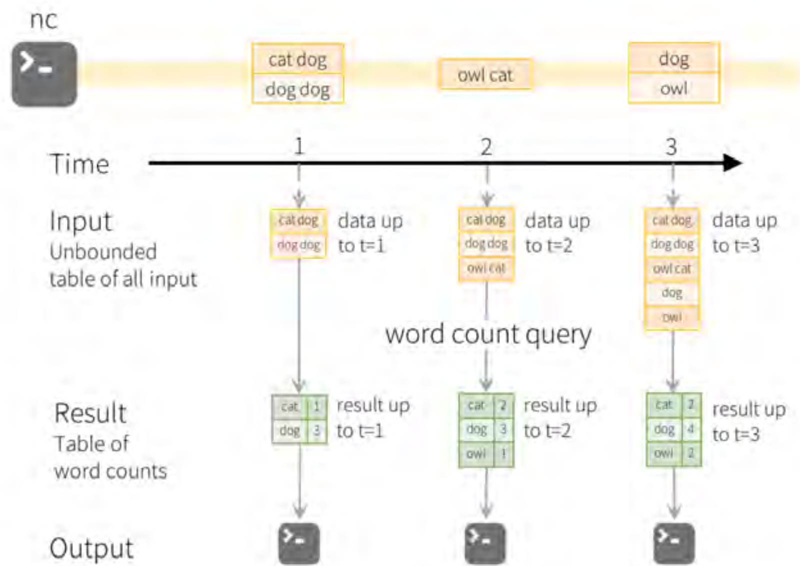


Figure 3.5: Spark Structured Streaming input table and result table. At each trigger interval (1 second in this example) new data is read from the stream source. The data is appended to the input table, a query is run on it, and the result table is updated. The result table is then written to an output sink; either in its entirety, only new rows, or only changed rows, depending on the selected output mode. Adapted from [38].

3.3 Apache Kafka

Apache Kafka [39] is a distributed streaming platform. It can be used as part of a streaming data pipeline, connecting other applications or systems that process the streaming data. It can also be used as a stream processing system, performing application specific operations on the data stream such as aggregations and stateful computations [40]. In this project, the stream processing is done using Spark Structured Streaming while Kafka is used to reliably get data between different Spark stream processing applications.

The basic data unit that Kafka uses is a *record*. A record consists of a key, a value and a timestamp. Records are published to one or more *topics*. A topic is an abstraction used to group together records that flow through the system. A *producer* publishes a stream of records to one or more topics, and a *consumer* subscribes to one or more topics and receives the records published to the topics.

Each topic is split up into a number of *partitions* under the hood, with each partition being "an ordered, immutable sequence of records that is continually appended to, i.e. a structured commit log" [40]. Each record appended to a partition gets a unique, sequentially increasing *offset* ID number. The structure of a topic

can be seen in figure 3.6. The partitions of topics are distributed among the servers in the Kafka cluster, allowing for scalability, with the data replicated for fault tolerance. The data is kept for a configurable retention period before being deleted to clear up space. In this sense, Kafka provides a fault tolerant distributed data store allowing for effective decoupling of producers and consumers.

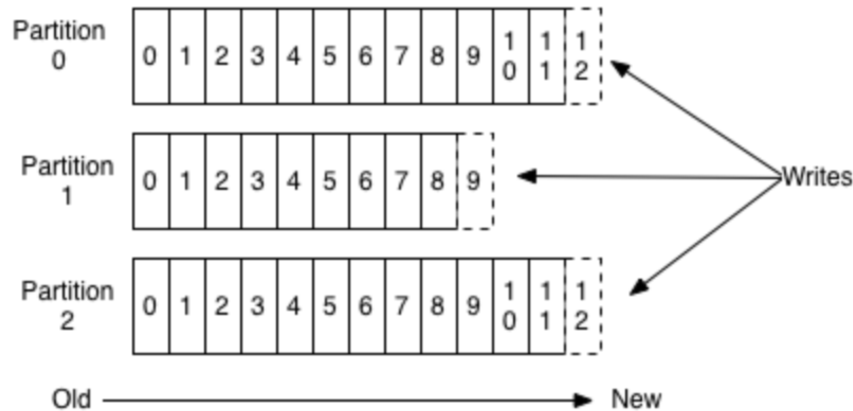


Figure 3.6: The internal structure of a Kafka topic. The numbers represent the unique, sequential offset IDs each record within a partition gets. Adapted from [40].

The producer can choose to which partition within a topic a particular record should go. They can distribute them evenly to the partitions or apply some group-by logic so that some key within the record determines which partition a particular record should go to.

The offset IDs allow consumers to keep track of where to read from the partition. They can increment the offset linearly and thus read records from a partition in the order that they arrived in the partition, or they can select an arbitrary offset to e.g. process old data. Each consumer has its own read offset, so different consumers may consume different parts of the commit log at the same time.

Consumer instances can be grouped into *consumer groups*, forming a single "logical consumer". Each consumer group then subscribes to a topic, and records from that topic will be delivered to *one* consumer instance within the consumer group. This allows for scalability and fault tolerance on the consumer side, spreading the processing of a topic among many consumer instances. Under the hood, each partition of a topic is mapped to a specific consumer instance within the consumer group, distributing the partitions evenly among the consumer instances for load balancing. Since a record will only go into one of the partitions, it will only arrive at one of the consumer instances. This mapping of partitions to consumer instances within a consumer group can be seen in figure 3.7.

The order of incoming records is guaranteed per producer per partition. That

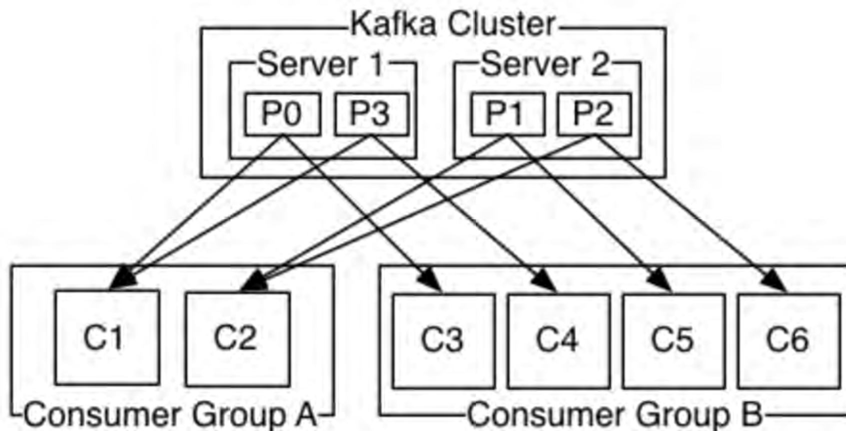


Figure 3.7: Kafka consumer groups. Each partition (P0-P3) is assigned to a single consumer instance (C1-C6) within each consumer group [40].

is, records originating from the same producer, going to the same partition, are guaranteed to appear in the commit log in the order they were sent by the producer. Consumer instances are then guaranteed to see records in the order they are stored in the commit log (partition) [40]. If the system is set up so that a given partition will only receive records from a single producer, the ability of the producer to assign records to partitions by some application specific key makes it is possible to achieve total order per key.

Kafka guarantees at-least-once delivery by default [41]. This means that published messages from a producer will never be lost, but may be duplicated in the commit log.

3.4 Related work

This section gives a review of related work. First, related work within the field of congestion detection is reviewed followed by related work with a focus on end of queue detection.

3.4.1 Congestion detection

Perhaps the most widely used queue detection system is the one provided in Google Maps[6]. Google Maps shows its users the congestion state on road segments via color codes; green for no traffic delays, orange for medium amount of traffic, and red if there are traffic delays, with darker shades of red signifying slower traffic speeds [42]. To achieve this, Google uses live speed measurements collected from other Google Maps users currently driving in the area under consideration. This way

[6]https://www.google.com/maps

they are able to crowdsource speed measurements, turning every vehicle containing a smartphone running Google Maps into a probe vehicle. It is worth noting that the system is built only on speed measurements, other traffic variables such as flow and density are not used [6]. However, Google does not disclose the thresholds and methods used to determine the color of a given road segment.

Coifman [9] used dual induction loop infrastructure detector measurements to identify the onset of traffic congestion. The congestion detection method works by collecting measurements from individual vehicles on a particular lane at an upstream sensor, and then attempting to identify the same vehicles (re-identification) on the same lane at a downstream sensor within a time window of reasonable free flow travel times. If a certain vehicle detected at the downstream sensor does not appear at the upstream sensor within the time it should take to travel the distance between the sensors under free flow conditions, the traffic on the link between the sensors is assumed to be congested. Note that the free flow speed of the road segment is not known; a "reasonable" time window is calculated based on the speed measured at the upstream sensor. The vehicle length is the feature used to identify individual vehicles. Dual induction loop detectors are able to determine the length of vehicles passing overhead from the measured speed of the vehicle and the time it takes the vehicle to traverse the sensor. The algorithm focuses on longer vehicles since they are less common than shorter vehicles and therefore distinct enough for them to be re-identified. The approach looks at detectors on the same lane based on the observation that lane changes are infrequent during free flow conditions. A vehicle may change lanes as it is travelling between the downstream and upstream sensors, in which case the algorithm will consider it a sign of congestion. To smooth out any mis-identifications and remove noise, a moving average of the 10 most recent outcomes is used. To verify the results of the algorithm, video recordings of traffic flow were used as ground truth.

Anbaroglu et al. [8] focused on detecting non-recurrent traffic congestion in urban road networks. Non-recurrent congestion, as opposed to recurrent congestion, is caused by intermittent events that affect the flow of traffic, such as accidents, special events or bad weather. They also focused on urban networks and not freeways. Their approach is based on Link Journey Times (LJT) where an LJT for a vehicle is the time it takes to drive between two measurement stations. The measurement stations used are automatic number plate recognition cameras which are able to identify the same vehicle at an upstream camera and a downstream camera (re-identification), measuring the time it took for the vehicle to travel the distance between cameras. Congestion is then considered to be a cluster of substantially high LJTs. To determine what constitutes a high LJT a threshold is used. Historical LJT measurements are used to find an expected LJT for a given link in the road network, which is then multipled with a congestion factor $c \geq 1$ to give a threshold LJT. The value of c is determined by domain experts. Note that the same congestion factor value c is used for all links and all times. They represent the road network as a directed graph, where the number plate recognition cameras are the nodes and the links between cameras are the edges. The paper defines an *episode* of

congestion as a maximal time interval during which all the observed LJTs on a link are high. This allows for the tracking of congestion on a link through time. The road network graph is then used to cluster spatio-temporally overlapping episodes. Two episodes are clustered together if they occur on adjacent links in the graph, and there is at least one time interval that is common in both of the episodes. Two evaluation methods are proposed in the paper. The first method consists of identifying *high confidence episodes*, which are episodes that persist for a set minimum time duration. The intuition behind this is that these high confidence episodes are severe and demand attention from the traffic operators, so these episodes should therefore be identified by the congestion detection method. Ground truth is then established by historical analysis by domain experts, looking for these severe congestion events, and the results of the algorithm is compared to the domain expert analysis. The second evaluation method consists of relating the identified congestion episodes with reported incidents. Since the method focuses on non-recurring congestion (caused by e.g. traffic incidents), the method should be able to identify the congestion resulting from incidents on the road.

Li et al. [43] proposed the density based clustering algorithm *FlowScan* to identify "hot routes" in road networks. The road network is represented as a directed graph, where the vertices are either a street intersection or important landmark, and the edges represent the smallest unit of road segment between vertices. The algorithm is concerned with trajectories of probe vehicles passing through the road network, where each trajectory is a sequence of edges that the vehicle passed through. A hot route is then a sequence of edges in the road graph that share a high amount of traffic between them. However, the edges in the hot route are not necessarily adjacent (connected), it is considered sufficient that they are close to each other. FlowScan uses the traffic density on edges to discover hot routes. It is based on density-based clustering, defining a ϵ-neighborhood around each edge in the graph, where the distance measure is the number of hops needed to reach other edges. The algorithm then considers both the number-of-hops distance between edges, along with the number of unique vehicle trajectories that the edges have in common when performing density based clustering to discover hot routes. The algorithm was evaluated on synthetically generated vehicle trajectory data over the real-world road network of San Francisco. The results were evaluated on the basis of if the detected hot routes were realistic with respect to the underlying road network. FlowScan is not directly concerned with detecting congestion. Its goal is to find popular routes within the road network. In Stockholm, this might for instance be the route from Solna to Kista, under the assumption that a lot of people that live in Solna work in Kista and therefore commute between the two neighborhoods daily.

3.4.2 End-of-queue detection

A possible method for detecting end-of-queues is by examining vehicle trajectory maps where the end of the queue is represented by the backward forming shockwave (see section 2.6). However, this method requires the tracking of the trajectory

of each individual vehicle (microscopic view) which is difficult and is thus mostly applicable in a simulation environment. Chou and Nichols [44] showed that the EOQ can be detected using aggregated traffic measurements from traffic detectors (macroscopic view), removing the need to collect the trajectories of individual vehicles. This can be done by using contour maps (heatmaps) of aggregated traffic statistics, specifically average speed.

The contour maps show the magnitude of the measured average speed with each cell representing the measurement from a certain detector at a certain time. The different detectors are arranged in spatial order on the road on the vertical axis, with increasing time on the horizontal axis. By examining the magnitude of change in measured average speed values with respect to space and time it is possible to identify the backward forming shockwave, and thus the end of the queue.

The work is concerned with detecting the end of queues that form around an incident on the highway, such as an accident. To identify which detectors represent the end of a queue the following condition was used [44]:

$$EOQ_{i,j} = \begin{cases} 1, & (v_{i,j} > 0) \cap \left(\overline{S_{i,j}} < x\% \times S\right) \cap \left(\left|\overline{S_{i,j}} - \overline{S_{i-1,j}}\right| > \Delta S\right) \\ 0, & otherwise, \end{cases} \tag{3.4}$$

where i is the time index in the grid of the contour map starting from the origin, j is the space index starting from the origin, $v_{i,j}$ is the number of vehicles within cell (i, j), S is the average speed before the incident occurred (representing normal speed of traffic in the system), $\overline{S_{i,j}}$ is the average speed within cell (i, j), $x\%$ is the speed reduction percentage, and ΔS is the speed differential. The authors found the thresholds $x\% = 50\%$ and $\Delta S = 10$ mph ($\simeq$ 16,1 km/h) were appropriate to detect the EOQ successfully.

The method was evaluated using traffic simulation software, to allow for systematic analysis of simulated incident safety impacts. The accuracy of the contour map EOQ detection method depends on the distance between detectors on the road and their measurement frequency, with higher detector density and higher measurement frequency giving better results. The authors assume that the distance between the detectors is 500 feet (152,4 meters) and that they give measurements of average speed and flow every 30 seconds.

Khan [45] noted the fact that traffic detectors must be placed at very short intervals to achieve a resolution high enough to accurately detect the location of the EOQ. The focus of the work was to design an end-of-queue warning system to be used around highway road work zones. A feed-forward neural network was trained to predict the length of the queue between two sensor stations placed on either side of a road work zone, thereby predicting the location of the EOQ. Synthetically generated data based on traffic simulation was used to train and validate the model, as real world data was not available.

Liu et al. [10] took the microscopic view to EOQ detection, building an end-of-queue warning system using Dedicated Short Range Communications (DSRC)

between vehicles on the road. In a DSRC system the vehicles on the road exchange data such as speed measurements and location information. The proposed EOQ warning system uses only the speed difference between a subject vehicle `X` and the downstream vehicles as a trigger for when to display an EOQ warning to vehicle `X`. The authors found that in the worst case scenario (icy road conditions, downstream traffic at a stand still and the subject vehicle travelling at 65 mph) the minimum distance between the subject vehicle and the EOQ to avoid collision is over 300 meters. With better road conditions and/or the EOQ travelling at low speeds (i.e. not stopped), the distance required is reduced. The required speed difference threshold was determined with the help of historical traffic sensor measurements. The authors evaluated the probability of encountering a velocity difference greater than a certain threshold, based on the historical data. The goal of this approach is to trade-off the probability of successful collision avoidance with the occurrance of false positives, i.e. sending drivers an EOQ warning too frequently. The selected velocity difference threshold was 24 km/h, with the corresponding historical velocity difference probability being 3%. However, this applies to only dry pavement conditions. The velocity difference threshold should be lowered in worse driving conditions, such as wet or icy roads.

Chapter 4

Implementation

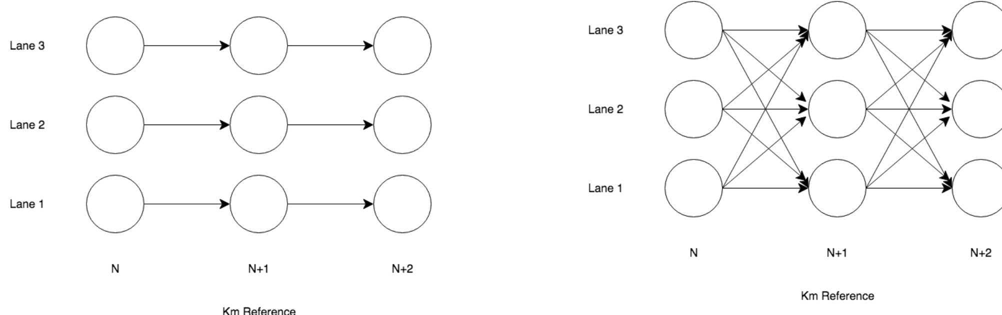
The first step of the implementation was to construct the road system graph. With the graph in place, various methods for congestion detection were tested on batch data. Finally, select batch methods were implemented as a streaming system. The structure of this chapter follows this outline.

4.1 The road network graph

The first step of the project is to construct a road system graph out of the traffic sensors placed around Stockholm. The data set contains information about the road each sensor is placed on, a kilometer reference giving the relative location of the sensor on the road in relation to some starting point, and which lane each detector is monitoring. In addition to this, the meta-data contains the GPS coordinates of each sensor. The constructed road system graph is a disconnected directed graph where the vertices represent the traffic sensors in the road system, and the edges connecting the vertices represent the road segments between sensor locations. As such, a path in the graph is a possible path in the road system that vehicles can drive.

4.1.1 Graph types

Two types of graphs were constructed, a *base graph* and a *reachability graph*. In the base graph, sensors on the same lane are connected with edges. A path in the base graph represents a path that a vehicle might take if it drives without changing lanes. The reachability graph is based on the base graph, but the sensors on a given kilometer reference N are connected to all sensors (on all lanes) at kilometer reference $N + 1$. The paths in the reachability graph represent the possible paths a vehicle might take, allowing for lane changes. An illustration of the two graph types can be seen in figure 4.1.



(a) Base graph. Sensors on the same lane are connected with edges.

(b) Reachability graph. Each sensor connected to all sensors on the next kilometer reference.

Figure 4.1: The two road network graph types.

4.1.2 Time-span graphs

Traffic sensors have been added gradually to the road system throughout the years, with some sensors removed along the way. The meta-data gives information on the valid-from and valid-to dates for each sensor. Using this information, six different road network base graphs were created to accurately reflect the sensor network at any given time. These different graphs will be referred to as "time-span graphs" from here on out. The valid dates and number of sensors for each time-span graph can be seen in table 4.1. The latest time-span graph contains 2037 sensors and 2077 edges.

Graph no.	**Valid from**	**Valid to**	**No. of sensors**
1	2001-1-1	2015-2-25	1843
2	2015-2-26	2015-11-10	1845
3	2015-11-11	2016-3-26	1917
4	2016-3-27	2016-3-28	1994
5	2016-3-29	2016-5-8	1976
6	2016-5-9	onwards	2037

Table 4.1: Time-span graphs. Valid dates and number of sensors.

4.1.3 How the graph is used

The base graph is used for the congestion detection methods, allowing tracking of congestion per lane through the road system. The reachability graph can then be used to send "congestion ahead" warnings to the sensors upstream from a queue.

Each vertex in the base graph is a traffic sensor and the edge between vertices represents the road segment between the sensors in the road system. The graph is directed, so each edge has a source vertex and a destination vertex. Edges are

weighted with the measurement values from the sensor corresponding to the destination vertex of the edge, as is illustrated in figure 4.2. The sensors give measurements every minute, and the assumption is that the measurements from the destination vertex represent the traffic conditions on the edge in the past minute.

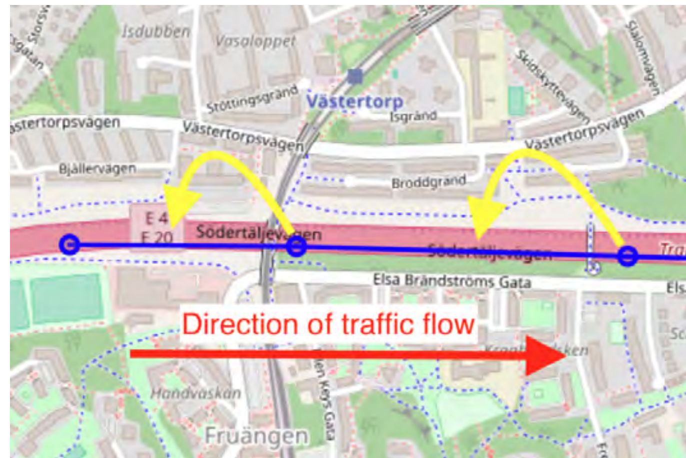


Figure 4.2: Edges are weighted with the sensor measurements from their destination vertex. Traffic sensors are represented as blue circles. The direction of the edges follows the direction of traffic flow.

Another possibility would be to weight the edges with the measurement values from the source sensor of each edge. In this case the assumption would be that the measurement values represent the traffic conditions on the edge in the current minute, since the measured vehicles would currently be travelling down the edge.

In this project, it was decided to weight the edges with the measurement values from the destination vertex. However, the source vertex could just as well have been chosen since the implemented congestion detection methods would work in the same way.

Since the sensors give measurements every minute, the weights on the edges of the graph are updated every minute.

4.1.4 Graph construction

The road system graph was constructed from the sensor meta-data in a couple of steps. First, the sensors were grouped by road and sorted by their kilometer reference with the sort order depending on the direction of the flow of traffic on the given road. On some roads the flow of traffic goes in the order of decreasing kilometer reference; specifically, roads with Z, W, C, D, F, H, S and U in their name as discovered by manual inspection using Google Maps' Streetview feature.

After sorting, edges were added connecting sensors on adjacent kilometer references, with the edge direction representing the direction of the flow of traffic. In the case of the base graph, a sensor on lane X at kilometer reference N was connected

to the sensor on lane X on kilometer reference $N+1$, if it exists. In the case of the reachability graph, the sensor on lane X at kilometer reference N was connected to all sensors (on all lanes) on kilometer reference $N+1$.

The rightmost lane (in the direction of traffic flow) at a given kilometer reference is designated with lane ID 1. The lane ID is incremented for the other lanes at the kilometer reference, counting from right to left (see figure 4.1). This means that whenever a road gains an extra lane (e.g. an on-ramp) or loses a lane (e.g. after an exit ramp) the lane IDs will be shifted with respect to the previous kilometer reference.

In the latest time-span graph (with sensors valid from 2016-5-9 onwards) there are a total of 177 kilometer references in the road system where a road gains or loses a lane. The location of the lane additions and removals can be seen in figure 4.3.

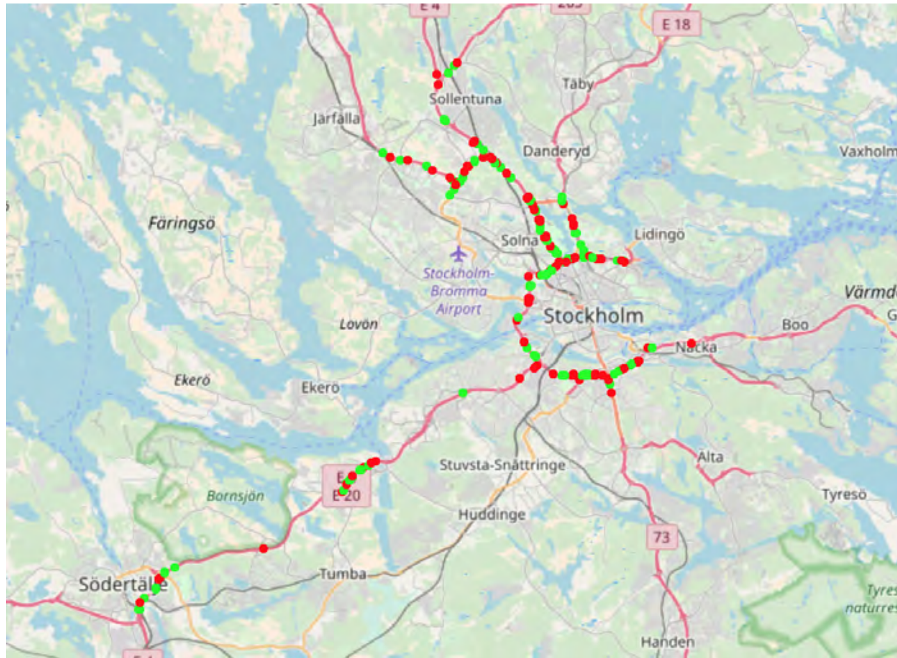


Figure 4.3: The location of lane additions and removals in the Stockholm area, with lane additions shown in green and lane removals in red. Data from the latest time-span graph (valid 2016-5-9 onwards). Note that Göteborg is not shown.

The roads containing congestion patterns selected as test cases for the congestion detection methods implemented in this project were manually corrected in the base graph to account for the shift in lane IDs. This was done using Google Maps' Streetview feature, "driving" along the roads making sure that each sensor was connected to the correct lane at the downstream kilometer reference. The rest of the roads were automatically corrected in the following way: If N lanes are added (resp. removed) at kilometer reference M, connect lane X at kilometer reference

$M-1$ to lane $X+N$ (resp. $X-N$) at kilometer reference M. Edges connecting sensors on different roads were then added manually, after inspection using Google Maps' Streetview. Figure 4.4 shows an example of lane additions and removals, and the corresponding shift in lane IDs.

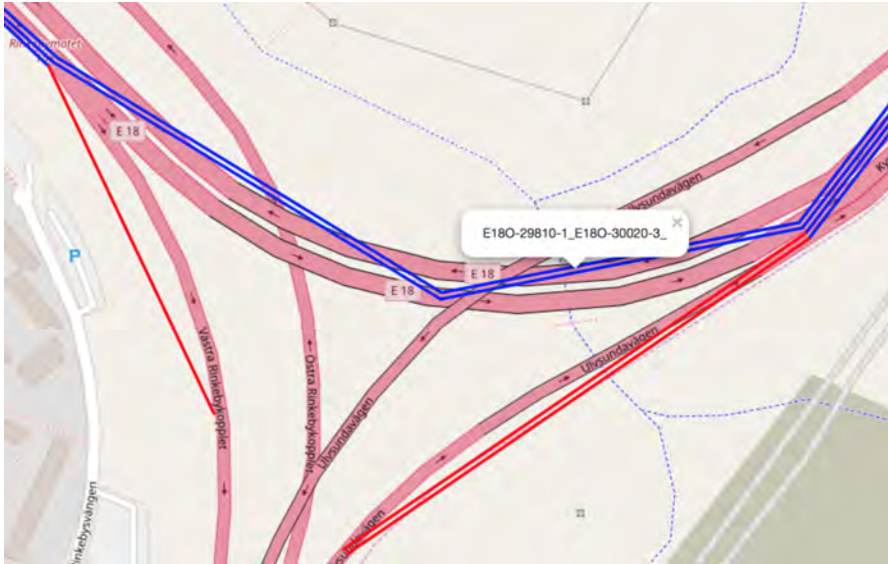


Figure 4.4: An example of lane additions and removals. The image shows the interchange east of Rinkeby in Stockholm. The blue lines represent edges on road E18O. The red line on the left represents the exit ramp from road E18O to road E18_E, reducing the number of lanes on E18O by one. The red lines on the right show where road E279N merges with road E18O, increasing the number of lanes on E18O by two. The tooltip window visible in the image shows how lane 1 of E18O is connected to lane 3 at the point where the two roads merge, as what was previously lane 1 becomes lane 3 after the merge. The flow of traffic for the edges displayed in the image is from left to right.

In some cases, the distance between two adjacent kilometer references on a road is too great to be considered as a valid road segment for congestion analysis. If two kilometer references are too far apart, the sensor measurements from the downstream sensor cannot be considered as representative of the actual traffic conditions on the link between the sensor locations. For instance, a just under 7 km long segment on E4N between Fittja and Fruängen in Stockholm does not have any traffic sensors. Furthermore, the roads (as delimited by the road IDs) are not necessarily connected in the underlying road system. As an example, road E18O is not connected, running between Rinkeby and Kista where it merges with road E4Z, and then reappears in Bergshamra. To avoid connecting sensors too far apart a distance threshold was set at 2000 meters. This is an arbitrary threshold which could be lowered if needed.

Some vertices in the graph are not destination vertices for any edge. This in-

cludes for instance the first sensors in every connected component in the graph. As is described in section 4.1.3, the edges of the graph are weighted with the measurement values from the sensor corresponding to the destination vertex of each edge. A dummy vertex (and dummy edge) is added in front of each vertex that is not a destination for any edge, allowing the sensor's measurements to be represented as weights on the edge from the dummy node to the vertex. An example of dummy vertices can be seen in figure 4.5.

(a) Dummy vertex at the start of a new lane. E4N just north of Södertälje.

(b) Dummy vertices on E4N outside Fruängen. This is the first sensor array after the ~7 km long segment on E4N without any traffic sensors.

Figure 4.5: Dummy vertices.

A number of other edge cases would need to be corrected for the constructed graph to represent the real world road network completely. For instance, in the case of a road gaining an additional lane via an on-ramp from another road, it is impossible for the vehicles coming from upstream of the merge to change lanes on to the on-ramp. This should be reflected in the reachability graph but was ignored. The roads containing queues selected as test cases for the implemented congestion detection methods were manually verified in the base graph, which is sufficient for the purposes of this project.

4.2 Identifying congested sensors

As described in chapter 2.3, it is possible to find the boundary between the free flow and congestion traffic states by inspecting empirical fundamental diagrams of traffic measurements. The empirical fundamental diagram is a scatter plot of sensor measurements from a single sensor, with flow on the vertical axis and density on the horizontal axis. An example of an empirical fundamental diagram from one of the sensors in Stockholm's road system can be seen in figure 4.6.

The sensor measurements belonging to free flow line up on a roughly straight line with a positive slope, until the empirical maximum point of free flow ($density_{max}^{(free,emp)}$, $flow_{max}^{(free,emp)}$) is reached. The slope of a line drawn from the origin to the maxi-

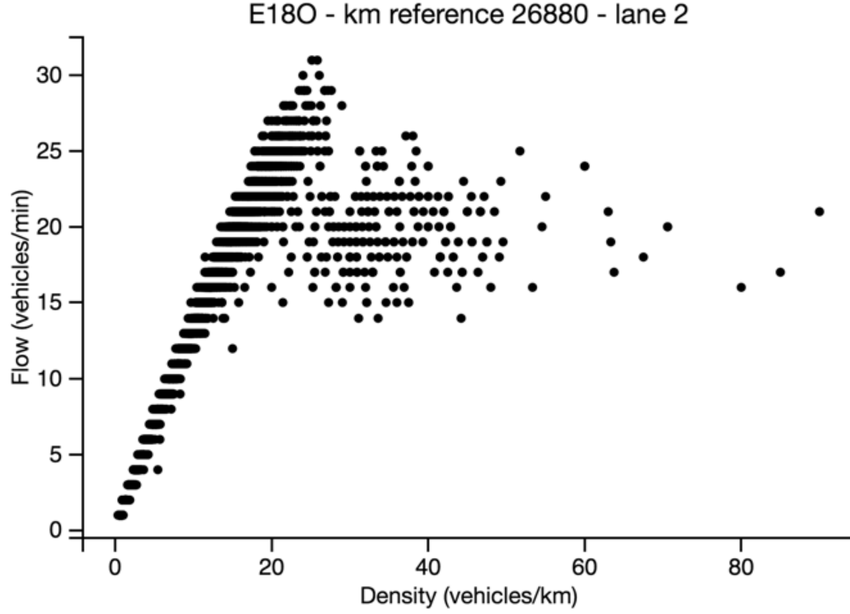


Figure 4.6: Fundamental diagram for the sensor on road E18O, kilometer reference 26880, lane 2. Each of the points on the diagram is a sensor measurement. Data from 2016-11-1 to 2016-11-3.

mum point of free flow then gives the minimum empirical free flow speed $v_{min}^{(free,emp)}$ for the traffic sensor in question. The minimum free flow speed was calculated for each sensor in the road system from historical data using the following equation:

$$v_{min}^{(free,emp)} = \frac{flow_{max}^{(free,emp)} \times 60}{density_{max}^{(free,emp)}}, \tag{4.1}$$

where $flow_{max}^{(free,emp)}$ is given in vehicles/minute, and $density_{max}^{(free,emp)}$ is given in vehicles/km.

A fundamental diagram with the minimum free flow speed line can be seen in figure 4.7.

4.2.1 Congestion classes

The minimum free flow speed $v_{min}^{(free,emp)}$ serves as a congestion threshold, defined per sensor from historical data. This allows sensor measurements from a given sensor to be classified as either free flow or congestion, depending on if the measured average speed is above or below the threshold. A binary classification into free flow or congestion does not suffice however, a measure of the level of congestion is also needed. To achieve this, each congested sensor measurement is assigned a congestion class, reflecting the severity of congestion. The congestion class assigned to a

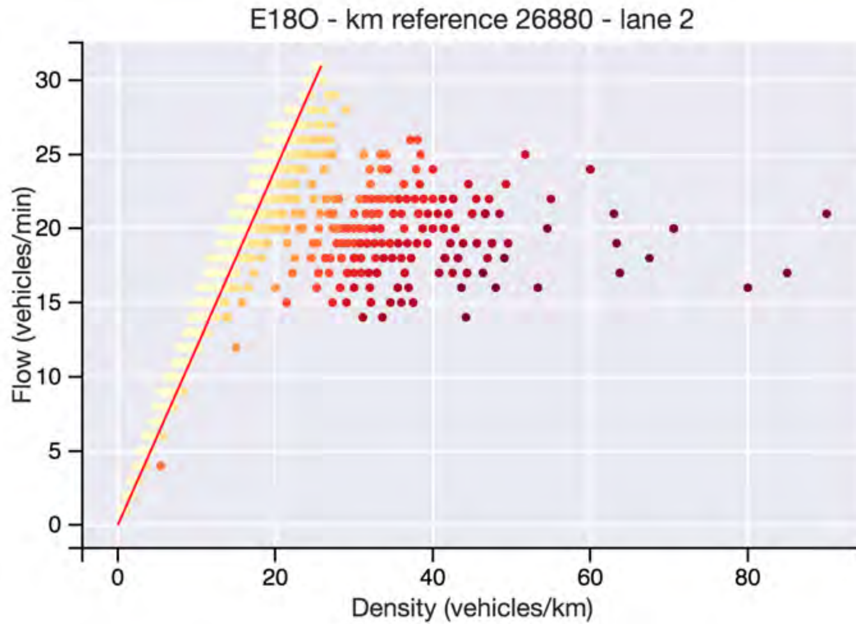


Figure 4.7: Fundamental diagram for the sensor on road E18O, kilometer reference 26880, lane 2. The slope of the red line going from the origin to the maximum point of free flow gives the minimum free flow speed. Points to the left of the red line belong to free flow. Points to the right of the red line are classified as congestion. The shade of red shows the level of congestion (congestion class). Note that this graph shows flow in vehicles/minute, while the minimum free flow speed was calculated from vehicles/hour. Data from 2016-11-1 to 2016-11-3.

given sensor measurement is determined by how far below $v_{min}^{(free,emp)}$ the measured average speed is, in increments of 5 km/hour. The rules for division into congestion classes can be seen in table 4.2. Note that an average speed measurement equal to $v_{min}^{(free,emp)}$ is not considered congestion.

The congestion class of each point in figure 4.7 is represented by the shade of red. Free flow points on the left of the minimum free flow speed line are yellow (congestion class 0), while points to the right of the line get a gradually darker shade of red with increasing congestion classes.

4.3 Batch approaches

A number of approaches to congestion detection were implemented on batch data, with the aim to find promising methods to implement as a streaming system. The following sections outline the batch approaches implemented in the project.

Congestion class	**km/h below $\mathbf{v}_{\mathbf{min}}^{(\mathbf{free,emp})}$**
1	$1 - 4$
2	$5 - 9$
3	$10 - 14$
4	$15 - 19$
5	$20 - 24$
6	$25 - 29$
7	$30 - 34$
8	$35 - 39$
9	$40 - 44$
10	$45 - 49$
11	≥ 50

Table 4.2: Congestion class classification rules.

4.3.1 Congested components

The congested components approach involves identifying connected components of congested sensors in the road system graph. The idea is to weight the edges of the base graph with the sensor measurement from the sensor corresponding to the destination vertex of each edge, remove all edges from the graph that are not considered congested, and running connected components on the remaining (congested) edges. The resulting connected components then reveal a chain of congested sensors, representing the queue. As described in section 3.1.5, this method resembles the traditional community detection method of hierarchical clustering.

The threshold for which edges should be removed from the graph is defined in terms of the congestion class that each sensor measurement falls into (see section 4.2.1). Edges with a congestion class below the set threshold are then removed from the graph. Since the traffic sensors give measurements every minute, a new graph is generated for every minute of data (starting with the full graph, and then removing non-congested edges).

The method was implemented on batch data using Spark for data preparation and the GraphFrames[1] library for working with the graph. GraphFrames' built in connected components method was then used to compute the congested components.

4.3.2 Louvain modularity

The next congestion detection approach evaluated was the Louvain modularity community detection algorithm. As in the congested components approach, each edge in the base graph was weighted with the measurement value of the destination vertex, in this case the measured density.

[1] https://graphframes.github.io/

Note that this method does not differentiate between congested and free flow sensor communities, as each vertex in the graph gets assigned to a community. The idea is that the community divisions should delimit the queues present in the road system graph, with adjacent congested sensors in the base graph being grouped into communities. The congested communities of sensors (queues) could then be identified by comparing adjacent communities to find drops in the average speed within the communities and/or a jump in the average density within the communities. Another approach would be to identify the end-of-queue sensors and taking their respective communities to be queues, and finally, communities with an average density higher than some threshold or average speed lower than some threshold could be labeled as queues.

Spark was used for data preparation and Sotera's GraphX implementation [46] of Louvain modularity on Spark was used to compute the community division.

This method was tested in the exploratory phase of the project, to see if the weighted modularity optimization approach to community detection is feasible for the road system graph.

4.3.3 Hierarchical (data) clustering

In light of the base graph being so simple (mostly just a chain of vertices), another approach evaluated was to use data clustering to cluster the sensors based on their measurement values, irrespective of the graph structure. The idea is that the graph could then be used to split the resulting clusters into connected components of vertices which fall into the same cluster. The split clusters could then delimit the queues in the road system graph, with adjacent congested sensors in the base graph being clustered together.

Hierarchical clustering was chosen as the clustering algorithm since the number of clusters present in the data set is not known beforehand and it allows for the exploration of the clustering result at various resolutions (levels in the hierarchy). The features used for the clustering were normalized average speed and density, normalized over each minute's worth of data. It would have been possible to cluster the vertices depending on their distance from each other in the graph as well. However, since the goal is to find a *connected* chain of congested vertices (i.e. a queue), the connected components of vertices falling in the same cluster would have had to be found regardless. Using the distance between vertices as a feature to cluster by does not guarantee that the resulting clusters consist of connected vertices.

A number of different linkage criteria were evaluated, namely single, complete, average and Ward's variance minimization algorithm.

Like the Louvain modularity method, this method does not differentiate between congested or free flow sensor clusters. The same strategies as mentioned in section 4.3.2 could then be used to identify which split clusters represent the queues.

Spark was used for data preparation, SciPy's hierarchical agglomerative clustering implementation[2] was used for clustering and SciPy's connected components

[2] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.`

implementation[3] was used to split the clusters using the base graph.

4.3.4 Dengraph

The original Dengraph algorithm (non-incremental) was implemented in Java to verify its feasibility as a congestion detection method.

Note that the distance function used in the original Dengraph paper (eq. 3.3) is defined to suit the goals of the application under consideration in the paper, i.e. the clustering of a social network. It only considers relationships between actors where both actors have initiated interactions with one another, and takes the minimum of the number of interactions initiated by each actor to the other as the distance between them. This can be generalized to an undirected weighted graph $G(V, E)$ such that the distance between vertices p and q is defined as:

$$dist(p,q) = \begin{cases} 0 & p = q \\ \frac{1}{W_{p,q}} & \exists (p,q) \in E \\ undefined & otherwise, \end{cases} \tag{4.2}$$

where $W_{p,q}$ is the weight of the edge between vertices p and q. Note that there is at most one edge between each pair of vertices in the base graph. The distance between pairs of vertices that are not connected by an edge in the graph is undefined.

The edges of the graph were weighted with the measured density (vehicles/km). The distance between two adjacent vertices given by eq. 4.2 is thus simply $1/\rho$ where ρ is the measured density of the destination sensor of the edge. Table 4.3 lists the values used for the neighborhood radius parameter ϵ and their corresponding density values. For a given value of ϵ, the density on the link between two adjacent vertices must be greater or equal to the corresponding density value for the vertices to fall into each others neighborhoods.

Neighborhood radius ϵ	**Corresponding density (vehicles/km)**
0,025	40
0,03	33,3
0,035	28,6
0,04	25
0,045	22,2
0,05	20

Table 4.3: The different values for Dengraph's ϵ parameter, and their corresponding density values.

linkage.html

[3]https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.connected_components.html

As this is the non-incremental version of the algorithm, it is run on the entire set of weighted edges from the graph (weighted with the sensor measurements from a single minute). The algorithm essentially does a depth first search from all core vertices in the graph, finding density-connected vertices and assigning them the same cluster ID as the core vertex at the start of the density-connected chain. The pseudo-code for the algorithm can be seen in algorithm 1.

```
Input: Graph G(V, E), ϵ, η
Output: Set V with each vertex labeled with its cluster ID
1  begin
2    while Not all vertices in V labeled do
3      Get p ∈ V that is not yet labeled
4      Compute ϵ-neighborhood of p, N_ϵ(p)
5      if ||N_ϵ(p)|| ≥ η then
         /* p is a core vertex                                   */
6        Generate a new cluster ID cID and assign it to each
           q ∈ {p} ∪ N_ϵ(p)
7        Push q onto stack for p
8        while stack for p is not empty do
9          Pop q from stack
10         Compute ϵ-neighborhood of q, N_ϵ(q)
11         if ||N_ϵ(q)|| ≥ η then
12           Label the non-labeled members of N_ϵ(q) with cID and push
               them on to the stack for p
13         end
14       end
15     end
16     if p is not labeled then
17       Label p as noise vertex
18     end
19   end
20 end
```

Algorithm 1: Pseudo-code for Dengraph. Adapted from [31].

Spark was used when generating edge lists, weighted with the sensor measurements from a single minute. The algorithm was executed as a Java program on a local machine.

4.4 Streaming approaches

Out of the batch approaches implemented, the congested components method and Dengraph were most promising. These methods were chosen for implementation as

a streaming system, using Spark Structured Streaming's Java API. Development was done on a local machine, with a local installation of Spark v2.3.1, Kafka v1.1 and Zookeeper v3.4.12.

The following sections outline the streaming implementation of these congestion detection methods, along with a separate streaming program to track the evolution of the detected congestion through time.

4.4.1 Data set

A subset of the traffic sensor measurement data was selected for the development of the streaming system. Data collected over the course of 3 days, 2016-11-1 to 2016-11-3, was chosen. This date range falls into the latest time-span graph with the greatest number of active sensors (2037). Furthermore, a number of promising queues had been noticed within the time range during the data exploration phase of the project, and finally, three days worth of data was considered enough to calculate the minimum free flow speed of each sensor (see section 4.2). After filtering out errors, the 3 day data set contains 6.593.769 records, totalling 103,7 MB in parquet format.

4.4.2 Stream source

As a stream source for real time sensor measurement data was not available during the implementation of the project, a stream source was created over the batch data. A local python program was implemented to read the sensor data and write its contents to a new CSV file in a designated folder. Spark monitors the folder for any new CSV files created there, reading their contents into a streaming DataFrame.

During development, the sensor measurement data was grouped by minute, and written in order to a new CSV file in the monitored folder. This simulates the real time generation of data from the traffic sensors. However, note that data arriving in order is not a requirement for the implemented streaming algorithms.

4.4.3 System architecture

Spark reads the data from the designated folder, simulating a real time streaming source as discussed in section 4.4.2. The data stream is then processed by either the congested components algorithm, or the Dengraph algorithm (or both, if they are both active and polling the data source). The results from the algorithms are then written to a Kafka topic. There are two subscribers to the congestion detection result topic, one of which is a Spark file sink program, that just collects the results and saves them to file. The other is the queue tracker streaming algorithm, that tracks the same queue through different time steps.

The file sink program can represent any consumer of the congestion detection results, for instance, a system that sends out "slow traffic ahead" warnings to drivers as they approach the end-of-queue. The file sink program was kept separate from the queue tracker in order to be able to act on the congestion detection results as

soon as possible. The queue tracker incurs overhead and its results are not necessary for the real time detection of queues (e.g. to send warnings and control variable speed limits). Figure 4.8 shows the architecture of the system.

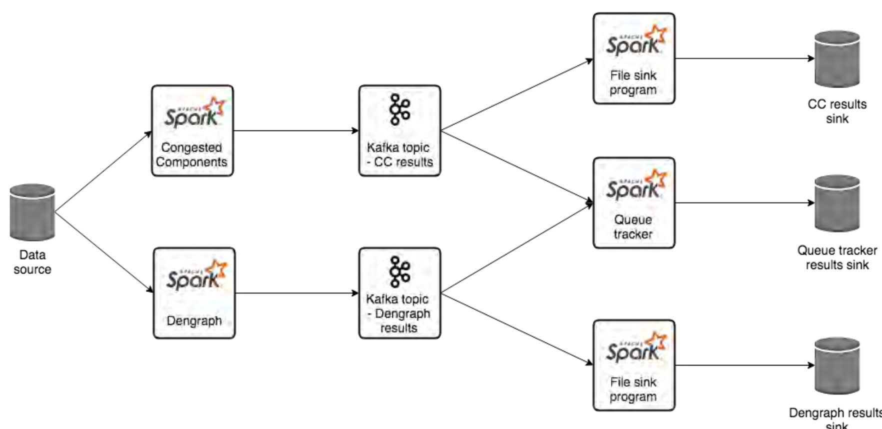


Figure 4.8: The streaming system architecture. Data is read from source, and processed by either the congested components algorithm or the Dengraph algorithm (or both). The results of the algorithms are written to a Kafka topic, and a file sink program subscribes to the results and writes them to disk. A queue tracker program also subscribes to the results, performs computation on the stream and writes its results to file. Note that there is a separate instance of the queue tracker for the results of each algorithm. The queue tracker results are then also written to a file sink.

4.4.4 Spark Structured Streaming programming abstractions

The streaming algorithms were implemented using Spark Structured Streaming's `mapGroupsWithState` and `flatMapGroupsWithState` operations. As discussed in chapter 3.2.2, they allow the developer to perform arbitrary stateful computation on the data stream. They operate on grouped DataSets, maintaining user defined state for each group.

The congested components and Dengraph streaming algorithms implemented in the project operate on weighted edge streams, with the edges weighted with the sensor measurement from the destination vertex of the edge. The edges carry the timestamp of the sensor measurement used as the weight for the edge.

The edge stream is grouped by timestamp, resulting in the edges weighted with sensor measurements from the same minute falling into the same group. The streaming algorithms operate on these groups, performing computations and maintaining state for each minute of data. This is essentially a tumbling window over event time, with one minute long windows.

Note that out of order data is handled seamlessly in this programming model. As the incoming edges are grouped by minute, they will be processed with other

edges from the same minute, irrespective of the order they arrive in.

State, state updates and output modes

In Spark's Java API the state used by the operations `mapGroupsWithState` and `flatMapGroupsWithState` is defined as a Java bean. A Java bean is a Java class which follows a certain standard; all properties in the class are private with getters and setters used to access them, the class contains a public no-argument constructor, and the class must implement the `Serializable` interface. An `encoder`[4] is then used to convert the JVM object to and from the internal `GroupState`[5] Spark SQL representation.

When the `mapGroupsWithState` or `flatMapGroupsWithState` operations are called on a micro-batch of records all the data is shuffled and the state is converted from the internal Spark SQL representation to the Java bean representation. Custom logic can then be implemented to process the records and update the state accordingly.

The output of `mapGroupsWithState` and `flatMapGroupsWithState` is a state update row. The state update row is represented by a Java bean which can be defined freely by the developer. In the case of `mapGroupsWithState`, a single row is output for each group processed in a trigger, while `flatMapGroupsWithState` can output no rows, a single row or multiple rows per group.

The semantics of the output rows is determined by the set output mode of the streaming query. There are three possibilities; append mode, complete mode and update mode. In append mode, only new rows added to the result table since the last trigger will be output. This assumes that the rows will never change. In complete mode the whole result table will be output after every trigger, and in update mode only the rows that were updated since the last trigger will be output [38].

The streaming algorithms implemented in the project are run in update mode. The state update rows are marked with a boolean flag signifying if the output row is part of the final state update for the given minute or not. Consumers of the algorithms' results are then able to act on the intermediate results, without having to wait for the final update after the given minute has been fully processed.

4.4.5 Watermarking

Watermarking is used to allow Spark to track the current event time as the data stream is processed. The `mapGroupsWithState` and `flatMapGroupsWithState` operations use the watermark to determine when all edges from a given minute are assumed to have already arrived, by setting a timeout relative to the watermark. Once the watermark advances beyond the timeout timestamp for a given group

[4]https://spark.apache.org/docs/2.3.1/api/java/org/apache/spark/sql/Encoder.html
[5]https://spark.apache.org/docs/2.3.1/api/java/org/apache/spark/sql/streaming/GroupState.html

state, the final update for the state is generated and the timed out state is cleaned up.

The watermark was set up to follow the maximum event time seen at any given moment. The group state timeout for each minute was set as the watermark + an additional 30 seconds. This means that when data from the next minute starts arriving, the group state for the previous minute will be considered final, resulting in a final state update being sent downstream and the state being cleared.

It is possible that in a real world implementation of the streaming system, with a real time source of sensor measurements, records from minute $t + 1$ might be received before having received all records from minute t. In this case the late records from minute t will not be considered in the processing of that minute. To combat this, the group state timeout might be increased to for instance not timeout until the system starts receiving data from minute $t+2$. Alternatively, a processing time timeout might be used instead of the event time timeout, timing out the group state after e.g. 30 seconds have passed without receiving any records for the group.

4.4.6 Congested components

The connected components algorithm was implemented in Spark Structured Streaming to identify the congested components in the road system.

The graph is loaded in the form of an edge list in a static DataFrame while the sensor measurement stream is read into a streaming DataFrame. Each vertex in the edge list DataFrame is assigned a unique numeric ID, to be used in the connected components algorithm. The congestion class of each sensor measurement is calculated based on the given sensor's minimum free flow speed, with the minimum free flow speed having been calculated beforehand from the 3 day data set. The sensor measurement stream is then joined with the edge list, via a stream-static join, resulting in a weighted edge stream (weighted with the calculated congestion class). Uncongested edges are then filtered out of the edge stream, using a specified congestion class threshold, and the resulting stream of congested edges run through the single-pass connected components algorithm.

Connected components was implemented as a single-pass algorithm [47], processing each edge only once. The algorithm finds *weakly* connected components, as the edge direction is not taken into account. The pseudo-code for the algorithm can be seen in algorithm 2.

The state for each minute was represented by two maps; a component ID to vertex IDs map, and a vertex ID to component ID map. While the vertex ID to component ID map is not necessary, it was included to achieve efficient bi-directional lookup.

```
Map<Integer, ArrayList<Integer>> componentVerticesMap;
Map<Integer, Integer> vertexComponentMap;
```

Listing 4.1: Streaming connected components state

```
   Input: Stream of congested edges E grouped by minute, state for the
          minute in the form of a component ID to vertex IDs map M
   Output: Updated M, with each vertex in E assigned to a component
 1 begin
 2   foreach edge (s, d) in E do
 3     if M contains neither s nor d then
         /* Neither vertex has been seen before                    */
 4       Create new component with ID min(s.ID, d.ID) and assign s
           and d to it
 5     end
 6     else if M contains both s and d then
         /* Both vertices have been seen before                    */
 7       if s and d are in different components then
 8         Merge the two components c1, c2 and set the ID of the new
             component to min(c1.ID, c2.ID)
 9       end
10     end
11     else
         /* Only one of the vertices has been seen before          */
12       If s is in a component, add d to the same component, and vice
           versa
13     end
14   end
15 end
```

Algorithm 2: Pseudo-code for streaming connected components

The end result of the connected components streaming program is a streaming DataFrame with each row containing the following fields:

```
Timestamp eventTime;
String vertexId;
int componentId;
boolean isFinal;
```

Listing 4.2: Streaming connected components state update row schema

The `isFinal` flag denotes if the row is part of the final state update for the given minute. The rows are then written in JSON format to a Kafka topic, with a unique key being generated by concatenating the event time, vertex ID and final flag, to conform to Kafka's key-value record schema.

4.4.7 Incremental Dengraph

The incremental version of Dengraph [31] was implemented in Spark Structured Streaming. Instead of having the entire set of edges available before running the algorithm, the incremental version processes an edge stream, building up and updating the clustering result to incorporate each new edge it sees. The distance function and edge weights are the same as in the batch implementation, as described in section 4.3.4.

The effect an incoming edge can have on the clustering is either *positive* or *negative*. Positive changes refer to new edges being added, or the distance between two previously seen vertices being reduced so that they enter the ϵ-neighborhood of each other. Formally, for two adjacent time points t and $t+1$ and two vertices p, q, a positive change is realized if [31]:

$$((dist_t(p, q) = undefined) \vee (dist_t(p, q) > \epsilon)) \wedge (dist_{t+1}(p, q) \leq \epsilon)$$

In response to a positive change the following updates to the clustering may happen [31]:

- **Cluster creation.** A new cluster is created if vertices previously not part of a cluster become core vertices after the positive change.
- **Absorption.** Former noise vertices become part of an existing cluster, if they become density reachable from a core vertex.
- **Cluster merge.** A new neighborhood is formed which contains core vertices that belong to different clusters. These clusters are merged to form a new cluster.

A negative change occurs if the distance between two vertices that were previously within each others neighborhoods increases so that they no longer fall in each others neighborhoods, or the edge between them is removed entirely. Formally, for two adjacent time points t and $t+1$ and two vertices p, q, a negative change is realized if [31]:

$$(dist_t(p, q) \leq \epsilon) \wedge ((dist_{t+1}(p, q) > \epsilon) \vee (dist_{t+1}(p, q) = undefined))$$

In response to a negative change a cluster may be removed, shrunk or split. In addition, vertices of a cluster that were core vertices before the negative change may become border vertices for a core vertex in another cluster, resulting in them being moved to the other cluster.

Like the streaming connected components algorithm, the incremental Dengraph algorithm was implemented using a one minute long tumbling window over event time, processing and building up state for each minute. As the state was not transferred from one minute to the next, only positive changes occur in the form of

edge additions. While it would have been possible to maintain the same state between minutes, taking both positive and negative changes into account, this was not done as the one minute tumbling window strategy lends itself well to the `mapGroupsWithState` programming model. As previously discussed, the incoming rows are grouped by key (in our case by minute) and timeouts (based on event time watermarks or processing time) are used to mark the group state as final and clear it. It would have been possible to add *the same* "fake" key to all rows, routing all incoming rows to the same group, and disabling the group state timeout. However, getting a clear view of the clustering at the end of each minute would be trickier. It would be possible to mark a state update row for minute t as final once an input row for minute $t+1$ is seen, however, this approach does not allow handling of late data. The main benefits of maintaining the same group state across minutes would be twofold. First, less state might have to be maintained as we do not maintain separate group states for each minute currently being processed (i.e. not timed out). As mentioned, this removes the ability to handle late data. Secondly, in the case that a measurement from a certain sensor s is missing in minute t, the measurement from minute $t-1$ is still accounted for in clustering state. An aging function might then be employed to reduce the weight for the edge corresponding to sensor s gradually, eventually removing the edge from the clustering state. It can be argued that this approach would mitigate the effects of a sensor going offline, under the assumption that the measurement received before the sensor goes offline should remain valid for some time.

Implementation

Like in the connected components implementation, the graph is loaded into a static DataFrame in the form of an edge list, while the sensor measurement stream is read into a streaming DataFrame. The sensor measurement stream is then joined with the edge list, via a stream-static join, resulting in a weighted edge stream where the weight of an edge is the measured density at the sensor corresponding to the destination vertex of the edge. Note that the average speed could have been chosen as the weight instead of density.

The distance between the vertices of each edge is then calculated using the distance function defined in equation 4.2, and the resulting edge stream is run through the Dengraph algorithm. The algorithm was modified from the version presented in [32] to disallow overlapping clusters, and only handle positive changes to the clustering state. The pseudo-code for the incremental Dengraph implementation can be seen in algorithms 3 - 5.

The state for each minute consists of the edges seen so far during the minute, along with the state of each vertex seen. Both are wrapped in maps for efficient lookup.

```
Map<SrcDstPair, DenGraphEdge> edgeMap;
```

```
Input: Stream of edges E grouped by minute, state for the minute M, ϵ, η
Output: Updated M, with each vertex in E assigned to a cluster or marked
        as noise
begin
    foreach edge e (s, d, dist) in E do
        Get old edge e_old between s, d from M, if any
        Add e to M, replacing e_old if it exists
        if (e_old is undefined or e_old.dist > ϵ) and e.dist ≤ ϵ then
            /* This is a positive change                              */
            updateClustering(s, ϵ, η)
            updateClustering(d, ϵ, η)
        end
    end
end
```

Algorithm 3: Pseudo-code for incremental Dengraph. Adapted from [32].

```
Map<String, VertexState> vertexStates;
```

Listing 4.3: Dengraph state

The `DenGraphEdge` class consists of the event time of the measurement the edge is weighted with, the ID of the source vertex, the ID of the destination vertex, and the calculated distance between the two vertices of the edge. The `SrcDstPair` class is a tuple of source and destination vertex IDs used as a key in the `edgeMap` for lookup. The `VertexState` class contains the ID of the vertex, the cluster ID assigned to the vertex, and the vertex type; core, border or noise. The vertex IDs are used as keys in the `vertexStates` map for lookup.

The eventual output of the algorithm is a streaming DataFrame with the following schema:

```
Timestamp eventTime;
String vertexId;
int clusterId;
String vertexType; // Core, border or noise
boolean isFinal;
```

Listing 4.4: Dengraph state update row schema

As noise vertices are not part of any cluster, they get assigned a cluster ID of -1. Like in the streaming connected components implementation, the `isFinal` flag denotes if the row is part of the final state update for the given minute. The output rows are written in JSON format to a Kafka topic, with a unique key being

generated by concatenating the event time, vertex ID and final flag, to conform to Kafka's key-value record schema.

Time complexity

The computation time of the algorithm is dependent on the number of calls to the `UpdateClustering`(v, ϵ, η) method, which is called for both vertices of an incoming edge that causes a positive change. The computation time of the `UpdateClustering` method depends on the size of the $N_\epsilon(v)$ neighborhood. It only performs work if v is a core vertex, in which case it updates the clustering of the vertices within $N_\epsilon(v)$. In the case of a cluster merge, the `MergeClusters` method is called. The complexity of the `MergeClusters` method depends on the number of vertices in the clusters that are merged, in the worst case $O(|V|)$ where V is the set of all vertices in the graph, if all vertices are merged to a single cluster. The worst case time complexity of the algorithm is therefore $O(|V|)$ [32].

This worst case will be realized if the entire road system is congested, and therefore falling in a single cluster. This is however quite unlikely.

4.4.8 Queue tracker

In addition to the congestion detection algorithms a separate streaming algorithm was implemented to allow tracking the evolution of clusters through time. The algorithm operates on the output stream of the congestion detection algorithms, relating the clusters found in minute t to the clusters found in the previous minute, $t-1$. Note that the congestion detection algorithms work on a minute by minute basis and do not give any information on the evolution of the clusters; a cluster found in minute t may have a different ID in minute $t+1$, and it may have grown, shrunk, merged with another cluster, split or disappeared entirely. The goal is to find the "parent" cluster (or clusters in the case of a cluster merge) in the previous minute, if it exists.

The algorithm utilizes Jaccard distance to compare clusterings between minutes. The Jaccard similarity coefficient is defined as the size of the intersection of two sets A, B divided by the size of the union of the sets,

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|},$$

and the Jaccard distance is defined as $d_J(A, B) = 1 - J(A, B)$. The Jaccard distance measures the dissimilarity between two sets at the element level, with the possible distance values falling in the range [0,1]. A Jaccard distance of 0 means that the two sets are identical (contain exactly the same elements) while a Jaccard distance of 1 means that there is no overlap of elements between the two sets. In our case, the elements are traffic sensors and the sets are the clusters found by the congestion detection algorithms.

The algorithm calculates the Jaccard distance between all clusters found in minutes $t-1$ and t, and uses a threshold on the calculated distance to determine which cluster in minute $t-1$ is a parent cluster of a cluster found in minute t. In the final implementation of the algorithm, the threshold was set at 1. If the distance between cluster A in minute $t-1$ and cluster B in minute t is less than 1 (i.e. if there is some overlap of traffic sensors between the clusters), cluster A is marked as the parent of cluster B.

The algorithm reads the clustering result stream from the congestion detection algorithms from a Kafka topic. It then filters all non-final result rows from the stream, so only the final clustering results of each minute are compared. It is also possible to analyze the intermediate results (i.e. not filter to only final rows), resulting in the intermediate queue tracking results being generated earlier. It can however be argued that this is not necessary, as the queue tracking results are not as time sensitive as the congestion detection results, which might be used to e.g. send warnings or adjust speed limits as soon as a given link in the road system is detected as congested.

A time-window column is added to the incoming rows. The time window is two minute long, with a slide of one minute, defined around the event time of the incoming rows. It was necessary to add the window as a column to the rows, since the `mapGroupsWithState` and `flatMapGroupsWithState` operations only operate on DataSets grouped by key, and do not allow windowing directly. It is then possible to use this column as a key to group the incoming rows by, processing and maintaining state for every pair of adjacent minutes together.

Watermarking is used to determine when all incoming rows for a given pair of minutes are assumed to have already arrived. The watermark is set to trail the maximum event time seen so far in the stream of incoming results from the congestion detection algorithms by 5 minutes. The group state for a given pair of minutes $(t-1,\, t)$ is then timed out once the watermark advances beyond t. This 5 minute long window for late data is implemented since the algorithm only works on the final congestion detection updates for a given minute, and it is likely that the queue tracking algorithm will begin to see intermediate congestion detection results from minute $t+1$ before receiving all final update rows for minute t. As discussed previously, the real time requirements of queue tracking are more relaxed than for congestion detection, so this delay is deemed justifiable.

As the intermediate results are not considered necessary, the queue tracking computation is only run for minutes $t-1$ and t once all final rows from the pair of minutes are assumed to have arrived. As such, no intermediate results are output and the computation is performed only once for every pair of minutes. This is beneficial as the computation is quite inefficient, as will be discussed later. The algorithm only collects all incoming *final* congestion detection state updates for minutes $t-1$ and t, until the group state timeout for the pair is triggered by the watermark advancing beyond t, prompting the computation of the parent clusters and emitting the final (and only) result for the pair of minutes.

The state for each pair of minutes consists of the timestamps of the previous and

current minutes, along with a list of `EventTimeVertexIdClusterIdTuple` instances, storing the cluster each vertex is assigned to during the two minutes in question.

```
Timestamp pastMinute;
Timestamp currentMinute;
List<EventTimeVertexIdClusterIdTuple>
    eventTimeVertexIdClusterIdTuples;
```

Listing 4.5: Queue tracker state

This state is built up until all incoming rows belonging to the two minutes are assumed to have arrived, at which point the queue tracking computation is performed. The output of the algorithm is a streaming DataFrame with the following schema:

```
Timestamp eventTime;
String vertexId;
Integer clusterId;
List<Integer> parentClusterIds;
Boolean isFinal;
```

Listing 4.6: Queue tracker state update row schema

Each row contains the event time (minute), vertex ID and cluster ID assigned to the vertex by the congestion detection algorithms during that minute. In addition to that the result contains a list of the IDs of the parent clusters from the previous minute, as determined by the queue tracking algorithm. Finally, an `isFinal` flag denotes if this state update row represents the final state update from the queue tracking algorithm. As no intermediate results are emitted, this flag is always set to true. The pseudo-code for the queue tracking computation can be seen in algorithm 6.

Time complexity

The algorithm does a pairwise comparison of each cluster found in the pair of minutes $(t-1,\ t)$, making the algorithm run in $O(n^2)$ time (or more precisely, $O(n \times m)$ where n is the number of clusters found in minute $t-1$ and m is the number of clusters found in minute t). This worst case is realized if every sensor in the road system falls in its own cluster. With at most 2037 sensors in the road network (time-span graph 6, valid from 2016-5-9 onwards), this results in

$$2037 \times (2037 - 1)/2 = 2.073.666$$

cluster comparisons. However, in the case of congested components each cluster will have at least 2 sensors, bringing the worst case number of cluster comparisons

down to 518.671. In the case of Dengraph with $\eta = 2$, each cluster will contain at least 3 sensors, bringing the number of cluster comparisons down to 230.181. While not efficient, the small size of the road system graph makes the algorithm feasible for this application.

4.4.9 File sink programs

Separate streaming programs were implemented to process the results of the streaming algorithms. These programs read the data from Kafka, representing the consumer of the congestion detection results. They could be used to e.g. drive real-time visualizations, or to control adaptive speed limits on the highway. In our case, they just write the results to file. Separate streaming programs were implemented to decouple the implementation of the streaming algorithms from the processing of their results, and because Spark Structured Streaming's file sink requires the *append* output mode while the `mapGroupsWithState` operation can only output in *update* mode. This means that the output of a `mapGroupsWithState` operation cannot be written to a file sink within the same streaming program.

```
Function updateClustering(Vertex v, ε, η)
    if N_ε(v) ≥ η then
        /* This is a core vertex                                        */
1       foreach n ∈ N_ε(v) do
2           if n.state == core then
3               setOfDistinctClusterIds.add(n.clusterId)
4           end
5       end
6       if v.state == core then
7           setOfDistinctClusterIds.add(v.clusterId)
8       end
9       if setOfDistinctClusterIds.size == 0 then
            /* Create new cluster                                       */
10          newClusterId = maxClusterId + 1
11          v.clusterId = newClusterId
12          foreach n ∈ N_ε(v) do
13              n.state = border
14              n.clusterId = newClusterID
15          end
16      end
17      if SetOfDistinctClusterIds.size == 1 and v.state != core then
            /* v and its neighborhood is absorbed to an existing
               cluster                                                  */
18          existingClusterId = setOfDistinctClusterIds.getOnlyElement()
19          v.clusterId = existingClusterId
20          foreach n ∈ N_ε(v) do
21              if n.state != core then
22                  n.state = border
23                  n.clusterId = existingClusterId
24              end
25          end
26      end
27      if setOfDistinctClusterIds.size > 1 then
            /* Merging of clusters                                      */
28          mergeClusters(v, setOfDistinctClusterIds, ε)
29      end
30      v.state = core
    end
end
```

Algorithm 4: Pseudo-code for incremental Dengraph's UpdateClustering method. Adapted from [32].

```
Function mergeClusters(Vertex v, Set setOfDistinctClusterIds, ϵ)
    newClusterId = maxClusterId + 1
    v.clusterId = newClusterId
    foreach r ∈ V do
        if setOfDistinctClusterIds.contains(r.clusterId) then
            r.clusterId = newClusterId
        end
    end
    foreach n ∈ N_ϵ(v) do
        if n.state != core then
            n.state = border
            n.clusterId = newClusterId
        end
    end
end
```

Algorithm 5: Pseudo-code for incremental Dengraph's MergeClusters method. Adapted from [32].

```
Input: List of clusters from minute t, list of clusters from minute t + 1,
       Jaccard distance threshold t
Output: All parent clusters from minute t have been identified for each
        cluster from minute t + 1
begin
    foreach cluster c1 from minute t do
        foreach cluster c2 from minute t + 1 do
            intersection = c1.vertices ∩ c2.vertices
            union = c1.vertices ∪ c2.vertices
            dist = 1 − intersection.size/union.size
            if dist < t then
                /* Parent cluster found                                  */
                Mark c1 as parent cluster of c2
            end
        end
    end
end
```

Algorithm 6: Pseudo-code for the queue tracking algorithm

Chapter 5

Evaluation

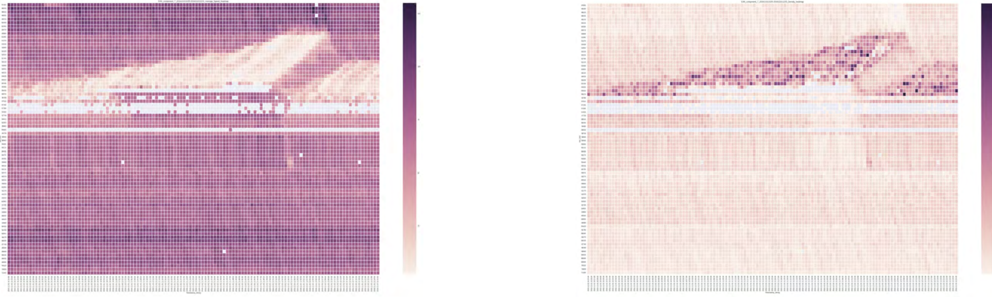
This chapter will detail the evaluation of the implemented congestion detection methods. It follows the same structure as the implementation chapter, beginning with batch approaches before moving on to streaming approaches.

5.1 Ground truth

All of the congestion detection methods implemented in this project are unsupervised. Furthermore, there is no ground truth available to evaluate the results of the methods. Historical data on the formation and evolution of traffic queues and a precise truth of what data points constitute congestion does not exist for the data set under consideration.

Ground truth for traffic analysis is hard to come by. Related work in the field has utilized a number of ways to establish ground truth, for instance video recordings of traffic [9]; domain experts to identify and label ground truth [8]; fusion with external data sets such as incidents, to verify that the congestion formed around an incident is found by the detection method [8]; or using traffic simulators instead of real data [43], allowing for complete observation of the (synthetic) traffic flow in the road system.

The evaluation of the work done in this project relies on the use of heat maps to compare the results of the implemented congestion detection methods to the underlying data visually. The heat maps show the spatio-temporal pattern of a certain measured traffic feature, in our case average speed and density. A set of 16 traffic queues and 15 shockwaves visible in the data set were manually labeled, and the accuracy of the congestion detection methods evaluated with respect to the number of queues and shockwaves the algorithms are able to detect. An example of spatio-temporal heat maps can be seen in figure 5.1.



(a) Average speed heat map. Lighter color is lower average speed.

(b) Density heat map. Darker color is higher density.

Figure 5.1: Heat maps showing the spatio-temporal patterns of measured average speed and density. Sensors, ordered by kilometer reference, on the vertical axis and time on the horizontal axis. The direction of traffic flow is from top to bottom down the vertical axis. Examining the heat maps from left to right a clear congestion build up can be seen, followed by dissipation, and the formation of a new queue. Blank cells represent missing sensor measurements. Data from E4N, base graph component 7, 2016-11-01, 10:25 to 12:25.

5.2 Batch approaches

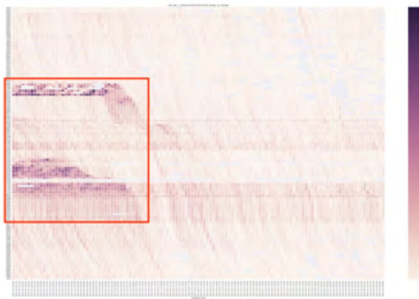
Out of the implemented batch approaches, congested components and Dengraph showed the best results and were chosen for implementation as a streaming system. The results of their streaming implementations are presented in sections 5.5.1 and 5.5.2. As the streaming implementation results are identical to the batch implementation results, the batch results will not be presented separately. The following sections will present results of the two other batch approaches taken, Louvain modularity and hierarchical (data) clustering.

5.2.1 Louvain modularity

The Louvain modularity algorithm was run, minute-by-minute, on data from E4N, lane 1, 2016-11-1 16:20-18:20. The algorithm terminates after only one iteration for every minute, achieving an average modularity of `q=0.728`, averaged over the minutes.

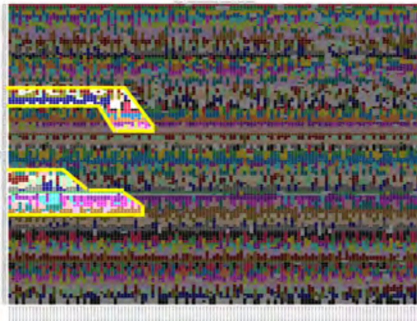
Note that the Louvain algorithm assigns every vertex in the graph to a community, irrespective of if it is congested or not. To verify that the community division makes sense with respect to the observed congestion pattern the communities were plotted on a heat map, which can be seen in figure 5.2b. Adjacent cells within a column, belonging to the same community, are shown with the same color. Comparing the observed congestion pattern to the pattern realized by the community division it is clear that despite the relatively high modularity value achieved, the community division is not representative of the observed congestion pattern. One would expect

sensors falling within the observed congestion pattern to fall in the same communities. This is not the case, as can be seen more clearly in figure 5.2c. The cells within the highlighted congestion area should have the same color, column-wise. However, more communities are detected within the congested region than should be expected.



(a) Measured density values

(b) Detected community division



(c) Detected community division with the observed congestion region highlighted.

Figure 5.2: Louvain modularity results. Figure 5.2a shows a heat map of the measured density values, while figure 5.2b shows the detected community division. Looking at the figure column-wise, adjacent cells of the same color are part of the same community. Note that comparing the colors of cells in different columns is meaningless. Figure 5.2c shows the community division with the observed congested region highlighted. Results for E4N lane 1, 2016-11-01 16:20-18:20.

These results indicate that the (weighted) modularity approach to community division is not suitable for the road system graph. This is perhaps not surprising, as modularity is a measure of the quality of community division in a graph as compared to what would be expected in a graph with the same community division and degree distribution, but with random connections between vertices. However, the vertices in the base graph form a chain, representing a lane in the road system. This chain structure will be lost with the randomization of edges between the vertices, leading to poor congestion detection results, despite the relatively high modularity value

achieved.

5.2.2 Hierarchical (data) clustering

The hierarchical data clustering approach was also run, minute-by-minute, on the data from E4N, lane 1, 2016-11-1 16:20-18:20. The clustering was run with a number of different linkage criteria (single, complete, average, Ward's variance minimization), and the clustering hierarchy examined by cutting the dendrogram at different levels. The results for hierarchical clustering using Ward's variance minimization algorithm and the dendrogram cut to generate two clusters can be seen in figure 5.3.

The Ward's variance minimization linkage criteria proved to give the best results. The motivation for cutting the dendrogram to generate two clusters was to see if they could be clustered as essentially congested and free flow groups. Running connected components on the results of the clustering method should thus give connected components of congested sensors ("split" clusters), which are taken to represent queues.

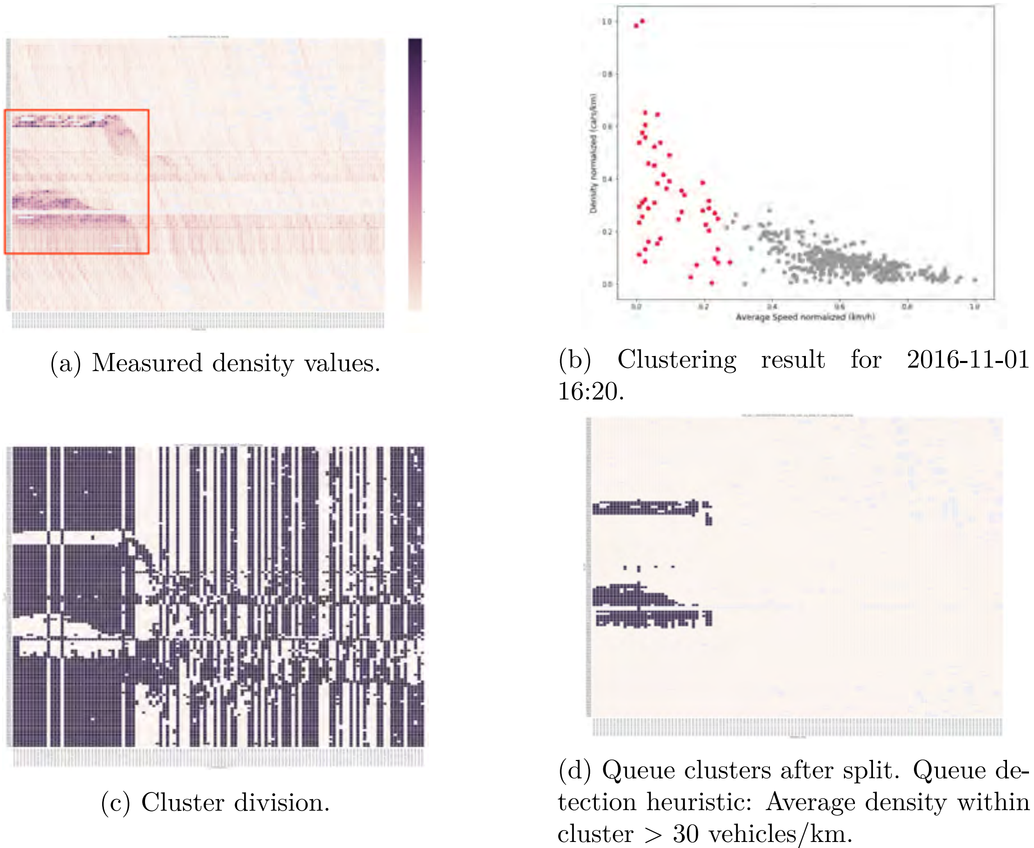
Similar to the Louvain modularity method, every vertex in the graph is assigned to a cluster, irrespective of if it is congested or not. To identify congested clusters (i.e. queues) a heuristic was employed, where (split) clusters with an average density over 30 vehicles/km are considered to be queues.

As can be seen in figure 5.3d, the hierarchical clustering approach is able to detect the congested regions while missing details such as the shockwave pattern realized by the dissipation of the topmost queue. The algorithm requires that the difference between the free flow and congested state, in terms of both density and average speed, to be large in order to precisely group the sensors. Finer details in the congestion pattern, such as the dissipation of queues, are not detected without introducing considerable noise. Furthermore, the method requires two steps; a clustering step followed by the computation of connected components. Therefore, the congested components and Dengraph congestion detection methods are more suitable.

5.3 Chosen test congestion patterns

To evaluate the streaming congestion detection methods, a set of 8 congestion patterns were selected. The congestion patterns are spatio-temporal patterns in the measured average speed or density, i.e. a pattern of low average speed, or high density (see figure 5.1).

All of the selected congestion patterns are from the same day, 2016-11-01, allowing the implemented streaming algorithms to be run over a single day of data for convenience. The criteria used to guide the selection of the test patterns was that they show interesting and varied spatio-temporal patterns and should cover most of the road system.



(a) Measured density values.

(b) Clustering result for 2016-11-01 16:20.

(c) Cluster division.

(d) Queue clusters after split. Queue detection heuristic: Average density within cluster > 30 vehicles/km.

Figure 5.3: Hierarchical data clustering results. Figure 5.3a shows a heat map of the measured density values. Figure 5.3b shows a scatter plot of the clustering result for the first minute, 16:20. Figure 5.3c shows the clustering division. Looking at the figure column-wise, adjacent cells of the same color are part of the same cluster. Figure 5.3d shows the congested clusters, where the average density within each cluster is greater than 30 vehicles/km. Results for E4N lane 1, 2016-11-01 16:20-18:20.

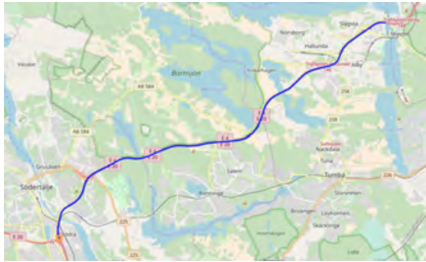
The goal is to detect and track congestion on the lane level. As discussed in section 4.1.4, one cannot rely on the lane IDs given in the sensors' meta data as they are counted from right to left at each kilometer reference. A lane addition or removal at a given kilometer reference will thus result in a shift in the lane IDs of the sensors compared to the previous kilometer reference. Therefore it is not possible to use the lane IDs of sensors given in the meta data to identify the sensors on the same lane. To allow tracking of congestion over *actual* lanes (the path a vehicle would take if it drove through the road system without changing lanes), connected components in the base graph were identified. Each connected component representing an *actual* lane in the road system.

The chosen congestion patterns can be seen in table 5.1. Figures 5.4 - 5.9

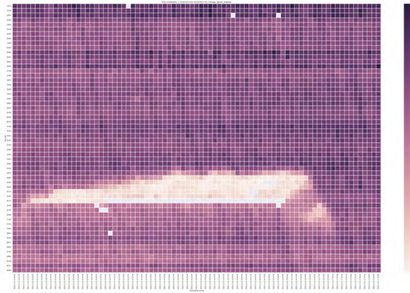
show the test congestion patterns' base graph components on a map, along with the average speed heat map for each chosen congestion pattern on the respective component. The average speed and density heat maps of each test congestion pattern can also be seen in appendix A.

Pattern no.	Road	Component	Time span
1	E4N	0	15:50 - 17:10
2	E4N	7	06:10 - 08:10
3	E4N	7	10:25 - 12:25
4	E4N	7	13:15 - 16:15
5	E6N	0	13:40 - 15:40
6	E18O	0	05:00 - 07:40
7	E20W	1	13:05 - 16:05
8	E75W	0	11:45 - 15:40

Table 5.1: Chosen test congestion patterns. All patterns are from 2016-11-01.



(a) Map view



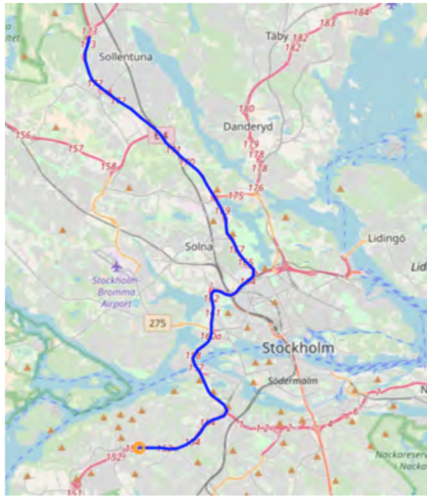
(b) Test congestion pattern 1. 2016-11-01 15:50-17:10.

Figure 5.4: E4N component 0. From Södertälje to the south of Stockholm.

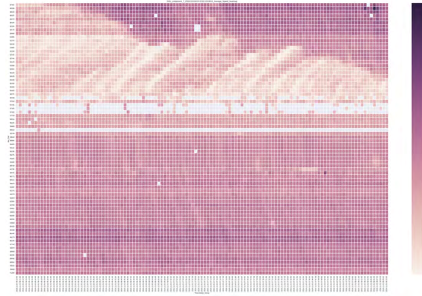
5.4 Experimental setup

The streaming system experiments were performed on a local machine with a 2.5 GHz quad-core processor and 16 GB of RAM. Spark v2.3.1 was run in local mode on 8 threads (using the `local[*]` configuration option, using as many threads as the number of available virtual cores) with 1 GiB memory allocated to the driver. Note that in local mode there are no executors, the driver performs the execution. Spark's Java API was used, with Java v1.8.

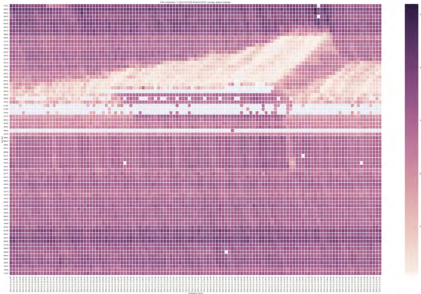
When streaming (as opposed to batch processing of larger amounts of historical data) the amount of data is quite small, at most 2037 sensor measurements per minute, so a local setup is sufficient to run the algorithms and evaluate their



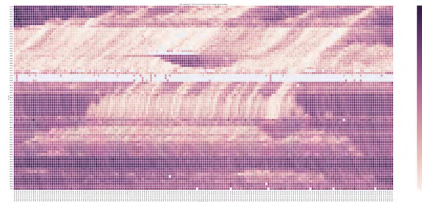
(a) Map view.



(b) Test congestion pattern 2. 2016-11-01 06:10-08:10.



(c) Test congestion pattern 3. 2016-11-01 10:25-12:25.

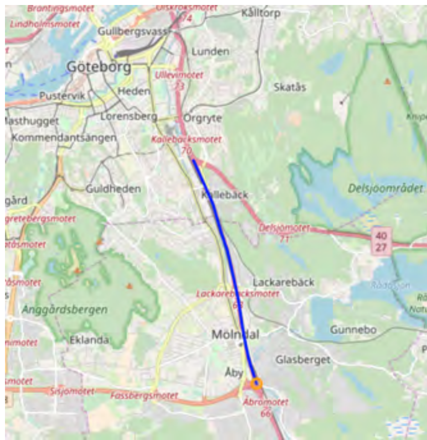


(d) Test congestion pattern 4. 2016-11-01 13:15-16:15.

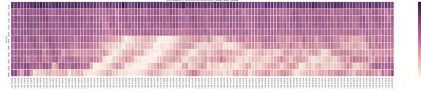
Figure 5.5: E4N component 7. From Fruängen to Sollentuna.

accuracy. The setup is also appropriate for the *relative* performance comparison of the implemented algorithms. Kafka v1.1 was also run on the local machine, with 1 partition per topic and replication factor 1. Data preparation and analysis of results was performed with PySpark, Spark's Python API, using Python v3.6.3.

The streaming algorithms were run over the data from 2016-11-01, as all test congestion patterns are from that day, and the results saved to file. The results for each of the 8 chosen test congestion patterns were then analyzed. The congestion detection algorithms were run with a number of different parameters each, with 8 result heat maps generated for each parameter combination. It is not feasible to include all of them in the main text of the thesis so the following sections will focus on a few example cases, with more result heat maps available in the appendix (see appendices B and C).

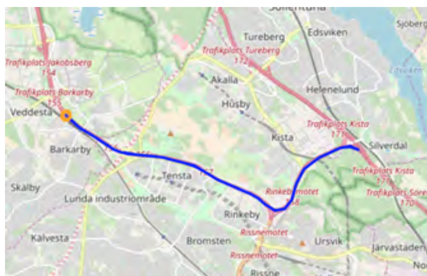


(a) Map view.

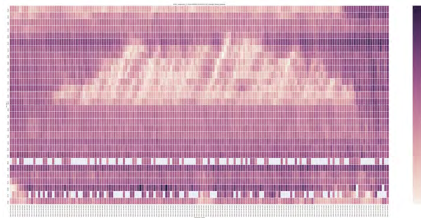


(b) Test congestion pattern 5. 2016-11-01 13:40-15:40.

Figure 5.6: E6N component 0. In Göteborg.



(a) Map view.



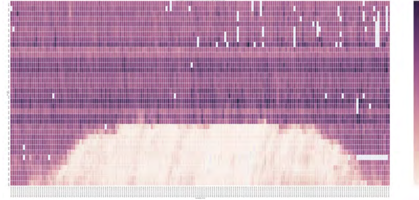
(b) Test congestion pattern 6. 2016-11-01 05:00-07:40.

Figure 5.7: E18O component 0. Between Kista and Rinkeby in north Stockholm.

5.5 Accuracy evaluation

To evaluate the accuracy of the results of the congestion detection algorithms 16 queues and 15 shockwaves in the 8 selected test congestion patterns were manually identified and labeled. The term "queue" refers here to a spatio-temporal region of low average speed or high density (taking a low resolution view of the congestion pattern). Shockwave patterns show a shockwave of congestion travelling backwards through the road system, against the direction of traffic flow. While queue buildup is in itself a shockwave, for the purposes of the accuracy evaluation the shockwave term is reserved for shockwaves occurring within congested regions or extending noticeably out of congested regions. The shockwaves according to this interpretation appear on the heat maps roughly as lines of particularly low speed or high density, with a positive slope (i.e. the congestion front moves backwards in space with time). The shockwaves can appear within queues, and thus represent a higher resolution view of the congestion pattern. Each test congestion pattern contains at least one queue, and 0 or more shockwave patterns. In the heat maps presented in the results,

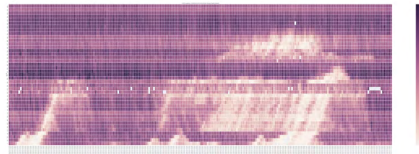
(a) Map view.



(b) Test congestion pattern 7. 2016-11-01 13:05-16:05.

Figure 5.8: E20W component 1. From the Lidingö bridge, north of down town Stockholm.

(a) Map view.



(b) Test congestion pattern 8. 2016-11-01 11:45-15:40.

Figure 5.9: E75W component 0. From Sickla to Liljeholmen in south Stockholm.

the queues are delimited by green borders, and the shockwaves by yellow borders, as can be seen in figure 5.11a.

The algorithms were evaluated with respect to the number of clear queues and shockwaves visible in their results. The clarity of the queues/shockwaves is determined subjectively by visual inspection, using the following critera. A queue or shockwave is considered to be clear if:

- The shape of the detected queue/shockwave conforms to the shape visible in the raw sensor measurement heatmap, with emphasis on the upstream congestion front as it poses the greatest safety risk.
- The boundaries of the detected queue/shockwave are clear, i.e. two different queues/shockwaves are not connected, and noise at the boundary is minimal.

Both the congested components and Dengraph algorithms were run over the data

from 2016-11-01 with different parameter values and their results evaluated. Table 5.2 shows the number of detected queues and shockwaves for different parameters of the algorithms over all test congestion patterns, along with the percentage of detected queues/shockwaves out of all labeled queues/shockwaves (recall). The accuracy evaluation results for each individual test congestion pattern can be seen in appendix D.

To determine the best parameters for the algorithms and allowing for comparison between the algorithms, a weighted sum over the recall fractions of detected queues and shockwaves is calculated, with a weight of 75% for detected queues and 25% for detected shockwaves. It is deemed more important to accurately detect the boundaries of the queues (congested regions) than the shockwave patterns, as the queue boundaries represent a safety risk for vehicles approaching the congested region (end-of-queue). The shockwave patterns are often only visible at a higher resolution (with a higher congestion threshold revealing more detail in the congestion patterns), resulting in the boundaries of the congested regions becoming unclear. There is therefore a trade-off between detection of queues and detection of shockwaves. The weights are chosen to give a reasonable balance between the detection of queues and shockwaves.

	Found queues	**Queue recall**	**Found s.waves**	**S.wave recall**	**WSM**
CC (c. class 1)	1	6.3%	0	0%	0.05
CC (c. class 3)	5	31.3%	1	6.7%	0.25
CC (c. class 5)	**15**	**93.8%**	10	66.7%	**0.87**
CC (c. class 7)	13	81.3%	12	80%	0.81
CC (c. class 9)	9	56.3%	**14**	**93.3%**	0.66
DG ($\epsilon = 0,025$)	3	18.8%	10	66.7%	0.31
DG ($\epsilon = 0,03$)	4	25%	11	73.3%	0.37
DG ($\epsilon = 0,035$)	13	81.3%	11	73.3%	0.79
DG ($\epsilon = 0,04$)	14	87.5%	6	40%	0.76
DG ($\epsilon = 0,045$)	12	75%	2	13.3%	0.60
DG ($\epsilon = 0,05$)	5	31.3%	0	0%	0.23

Table 5.2: Accuracy evaluation. Number of found queues and shockwaves over all test congestion patterns, along with recall out of a total of 16 queues and 15 shockwaves. Weighted sum model score with weight 0.75% for queues and 0.25% for shockwaves, calculated from recall fractions. CC stands for congested components, and DG for Dengraph. The parameters of the algorithms are shown in parenthesis; congestion class for congested components, and neighborhood radius ϵ for Dengraph. The minimum number of nodes required in a neighborhood for it to be considered a cluster in Dengraph was kept constant at $\eta = 2$.

In the case of congested components, a congestion class threshold of 5 gives the best results. 93.8% of the queues are detected and 66.7% of the shockwaves, giving a

weighted score of 0.87. In the case of Dengraph, a neighborhood radius of $\epsilon = 0.035$ (with minimum number of nodes in neighborhood fixed at $\eta = 2$) gives the best results. 81.3% of the queues are detected and 73.3% of the shockwaves, resulting in a weighted score of 0.79.

For queue detection, congested components with congestion class 5 gives the best results overall detecting 93.8% of all 16 labeled queues. For shockwave detection, connected components with congestion class 9 gives the best results, detecting 93.3% of all 15 labeled shockwaves. Overall, the best case accuracy of congested components is greater than that of Dengraph.

5.5.1 Congested components

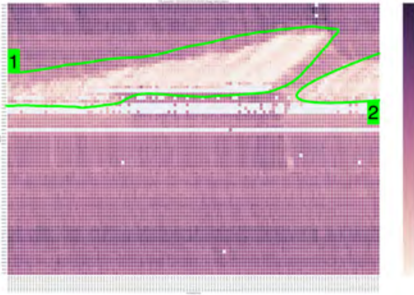
A lower congestion class threshold results in more edges being considered as part of a congested component, resulting in fuzzier congestion patterns being detected. As the threshold is increased, more detailed congestion patterns are revealed, at the expense of perhaps disregarding sensors that are in fact experiencing congestion but not at the magnitude required by the set congestion class threshold. Shockwaves start to become apparent as the congestion class threshold is increased.

Figure 5.10 shows the measured average speed heat map for test congestion pattern 3, along with the detected congestion pattern for congestion class thresholds 1, 3 and 5. For congestion class threshold 1, the outlines of the spatio-temporal congestion patterns are roughly visible, albeit with a considerable amount of noise. As the congestion class threshold is increased, the congestion pattern becomes clearer, revealing the build up of congestion and subsequent dissipation. Furthermore it becomes clear that the congested area is actually two separate queues, the second one starting to build up a bit further down the road as the first one dissipates.

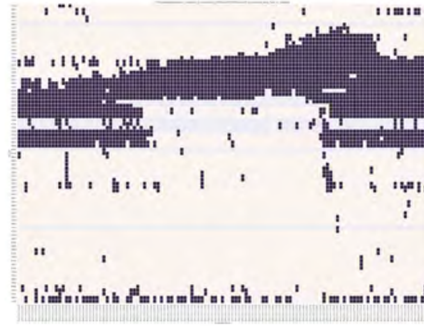
For congestion class 1, no queue is determined to be clear (detected) due to the considerable noise. For congestion class threshold 3, queue 1 is determined to have been detected, as the outlines of the congested area are clear. The two queues are however not clearly separated. For congestion class threshold 5, both queues 1 and 2 are determined to have been detected. Full accuracy evaluation results for the test congestion pattern can be seen in appendix D, table D.3.

Figure 5.11 shows the results for test congestion pattern 4, with congestion class thresholds 5, 7 and 9. This is a more complex congestion pattern, with three different queues present and visible shockwave patterns, four of which were labeled for detection.

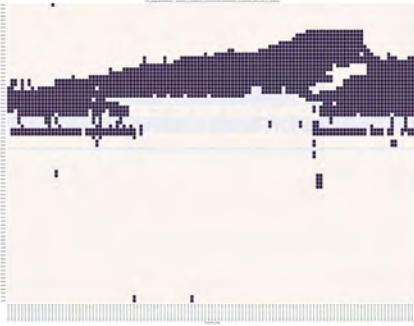
At congestion class threshold 5 all three queues are considered clear and thus successfully detected by the algorithm. They are well separated, and the shape of the detected congested regions conforms well to the shape of the congested regions apparent in the raw measurement values. Shockwaves 2, 3 and 4 are also considered clear. Shockwave 1 is however not clearly detected. At congestion class threshold 7 queues 1 and 2 are determined to have been detected. Queue 3 is not sufficiently clear to be detected, as the outlines of the congested area at the rightmost part of it have been lost. All shockwaves have however become clear. At congestion class



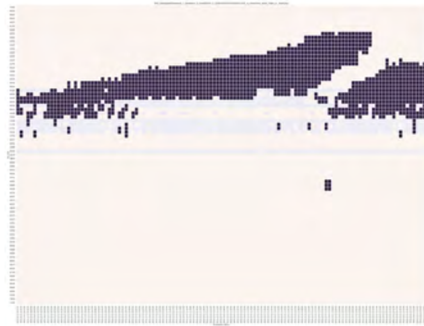
(a) Measured average speed values. Two queues delimited by green borders.



(b) Congestion class threshold 1



(c) Congestion class threshold 3



(d) Congestion class threshold 5

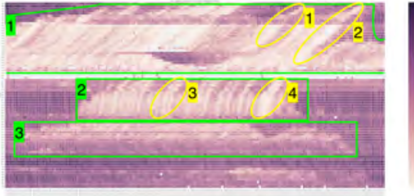
Figure 5.10: Congested components results. Figure 5.10a shows a heat map of the measured average speed values, with the two visible queue patterns circled in green. The other figures show the detected congestion pattern at different congestion class thresholds. Results for test congestion pattern 3, E4N component 7, 2016-11-01 10:25-12:25.

threshold 9, only queue 1 is considered clear. The upstream boundary of the queue remains clear, which is of the greatest importance with regards to traffic safety, while the boundaries of the other two queues have become less clear. Shockwaves 1, 2 and 4 are also considered clear, while shockwave 3 has become less clear. Full accuracy evaluation results for the test congestion pattern can be seen in appendix D, table D.4.

5.5.2 Dengraph

The Dengraph algorithm was run over the data from 2016-11-01 with different values for the neighborhood radius, ϵ. The minimum number of nodes required in a neighborhood for it to be considered a cluster was kept constant at $\eta = 2$.

Figure 5.12 shows the detected congestion pattern for test queue 6, with $\epsilon = 0,05$, $\epsilon = 0,04$ and $\epsilon = 0,03$. With decreasing ϵ, the density weight on an edge required for two adjacent vertices to fall in each others neighborhoods increases. This results in the detected congestion pattern becoming less fuzzy as less dense



(a) Measured average speed values. 3 queues delimited in green, 4 shockwaves in yellow.

(b) Congestion class threshold 5

(c) Congestion class threshold 7

(d) Congestion class threshold 9

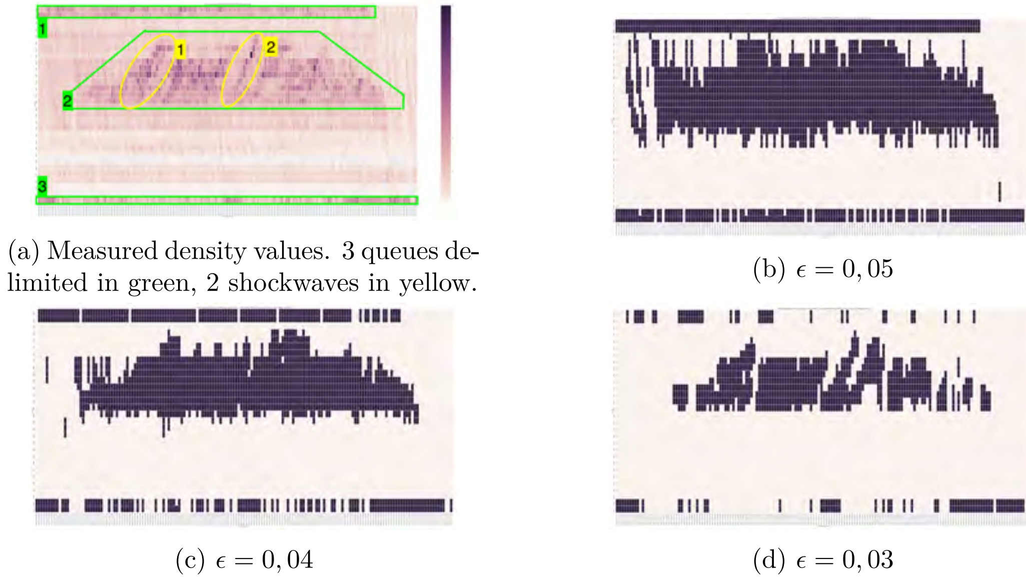
Figure 5.11: Congested components results. Figure 5.11a shows a heat map of the measured average speed values, with 3 queue patterns circled in green, and 4 shockwave patterns in yellow. The other figures show the detected congestion pattern at different congestion class thresholds. Results for test congestion pattern 4, E4N component 7, 2016-11-01 13:15-16:15.

links are filtered out, revealing more detail in the congestion pattern.

At $\epsilon = 0,05$ queues 1 and 3 are determined to have been successfully detected. The boundaries of queue 2 are not clear, with the left side of the queue pattern being considered too fuzzy. Furthermore, none of the shockwaves are detected. At $\epsilon = 0,04$ all queues are considered to have been successfully detected, along with both shockwaves. The top parts of the shockwaves can be clearly seen extending out of the congested region. At $\epsilon = 0,03$ only queue 2 is considered to have been successfully detected. The other two queue regions have become less clear. Both shockwaves are detected clearly. Full accuracy evaluation results for the test congestion pattern can be seen in appendix D, table D.6.

Figure 5.13 shows the detected congestion patterns for test congestion pattern 8, with different values for ϵ. For $\epsilon = 0,03$ the congestion pattern has become clearly visible. For $\epsilon = 0,025$, the congestion pattern also clearly visible, however some sensors within the congested area on the lower right are starting to be dropped from the result (note the blank cells within the dark congested area).

For $\epsilon = 0,04$ all queues are deterimined to have been clearly detected. The shape of the queue patterns are sufficiently clear, and importantly, the upstream front of the queue patterns is relatively noise free. However, only shockwave 1 is considered to be clearly detected. For $\epsilon = 0,03$, only queues 1 and 4 are considered to have been detected, with the leftmost part of the other queues becoming less



(a) Measured density values. 3 queues delimited in green, 2 shockwaves in yellow.

(b) $\epsilon = 0,05$

(c) $\epsilon = 0,04$

(d) $\epsilon = 0,03$

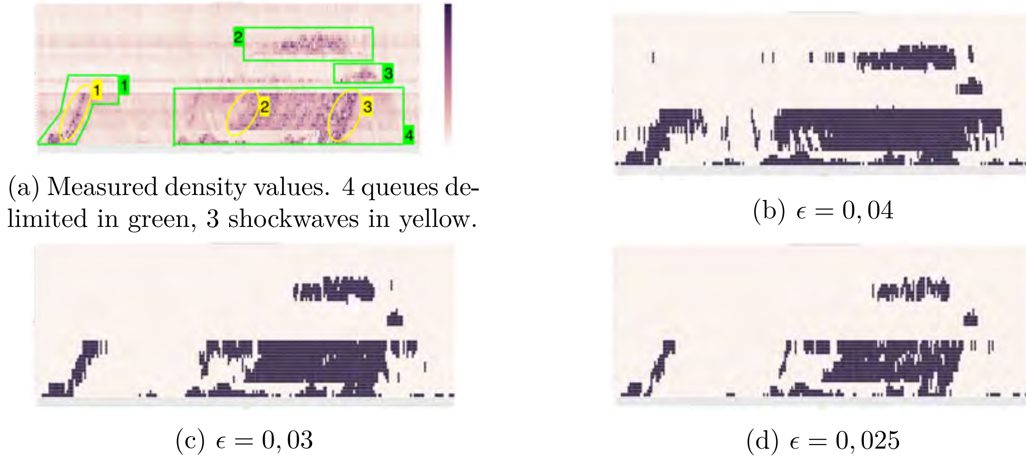
Figure 5.12: Dengraph results. Figure 5.12a shows a heat map of the measured density values, with 3 queue patterns circled in green and 2 shockwave patterns in yellow. The other figures show the detected congestion pattern for different values of neighborhood radius, ϵ. Results for test congestion pattern 6, E18O component 0, 2016-11-01 05:00-07:40.

clear. Shockwaves 1 and 2 are considered clear. For $\epsilon = 0,025$ queues 1 and 4 are considered clear, along with shockwaves 1, 2 and 3. Full accuracy evaluation results for the test congestion pattern can be seen in appendix D, table D.8.

5.5.3 Dengraph noise resistance

The Dengraph algorithm is resistant to noise in the data set. This is achieved through the η parameter, determining the minimum number of neighbors required for a given ϵ-neighborhood around a vertex to be considered a cluster. The Dengraph algorithm was run with $\eta = 2$, meaning that at least two adjacent congested traffic sensors are needed to form a cluster (queue). The cluster will include the two congested sensors along with a third border vertex, which can be either congested or uncongested (recall that the queue is a sequence of congested *edges*, not vertices). An illustration of this can be seen in figure 5.14. The congested components algorithm however will create a congested component (queue) out of a single congested sensor. The congested component will consist of two vertices, the congested vertex, along with its incident vertex.

The effect of this can be seen in the result heatmaps. In the case of the congested components algorithm there will be at least 2 adjacent congested cells within a column, while in the case of Dengraph there will be at least three adjacent congested cells within a column. The only exceptions to this is if a congested connection from



(a) Measured density values. 4 queues delimited in green, 3 shockwaves in yellow.

(b) $\epsilon = 0,04$

(c) $\epsilon = 0,03$

(d) $\epsilon = 0,025$

Figure 5.13: Dengraph results. Figure 5.13a shows a heat map of the measured density values, with 4 queue patterns circled in green and 3 shockwave patterns in yellow. The other figures show the detected congestion pattern for different values of neighborhood radius, ϵ. Results for test congestion pattern 8, E75W component 0, 2016-11-01 11:45-15:40.

another road is visible in the heatmap (see figure 5.15).

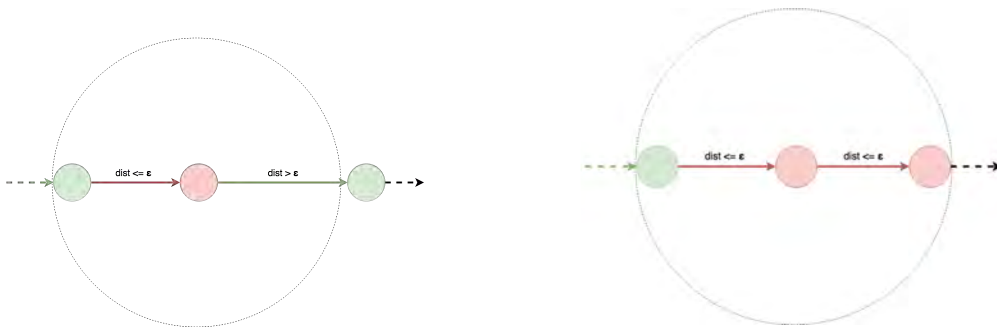
The value of η could be reduced to 1, in which case the Dengraph algorithm would behave like the congested components algorithm, forming a cluster (queue) with just one congested sensor. However, increasing η beyond 2 does not make sense in the case of the base graph. The base graph is mostly a chain of vertices, so each vertex will most often only have two neighbors.

5.6 Detecting individual queues

In addition to detecting the congestion patterns as a whole, the algorithms find individual congested components or clusters, which are taken to represent queues. A queue in this sense is a connected chain of congested vertices. The result heat maps presented thus far have only shown the shape of the detected congestion patterns as a whole. Each individual connected component/cluster of congested sensors within the congestion patterns is however assigned its own queue ID.

Recall that the heat maps show spatio-temporal patterns, with the sensors on the *actual* road lane (path taken if driving without changing lanes) ordered by kilometer reference on the vertical axis and time on the horizontal axis. Each column of the heat map therefore represents the congestion state on the road lane on a given minute.

Each connected chain of congested sensors on the road lane represents an individual queue, with its own ID. Figure 5.16 shows the point where two adjacent queues meet, as detected by the congested components algorithm. Note that even



(a) Single congested sensor. Only one vertex in the neighborhood so no cluster is formed.

(b) Two adjacent congested sensors. The middle vertex is a core vertex and will form a cluster consisting of itself and its neighbors.

Figure 5.14: Dengraph noise resistance. Uncongested sensors and edges in green, congested sensors and edges in red, neighborhood radius represented with dashed circles. For a vertex to be considered a core vertex and form a cluster there must be at least $\eta = 2$ other vertices within its neighborhood. For a cluster to form there must therefore be at least two adjacent congested vertices, the first of which will be a core vertex and form a cluster.

Figure 5.15: Dengraph noise resistance. In the Dengraph result heatmaps there are at least three adjacent congested cells within a column (lower magnifying glass). Exceptions to this are congested connections to other roads (upper magnifying glass). In this case it is a congested connection from E4N to E4_M, which can be seen in the image on the right. Dengraph results from test congestion pattern 2, E4N component 7, 2016-11-01 06:10-08:10, $\epsilon = 0,03$.

though the cells of the two different queues are adjacent in the heat map does not mean that the sensors they represent are part of the same queue (recall that the queues are tracked in terms of edges and not vertices). The road segment between the last sensor in component 906 and the first sensor in component 942 is not considered congested, as the average speed measurement from the first sensor in component 942 is not below the set congestion class threshold. Figure 5.17 shows

the two queues represented on a map.

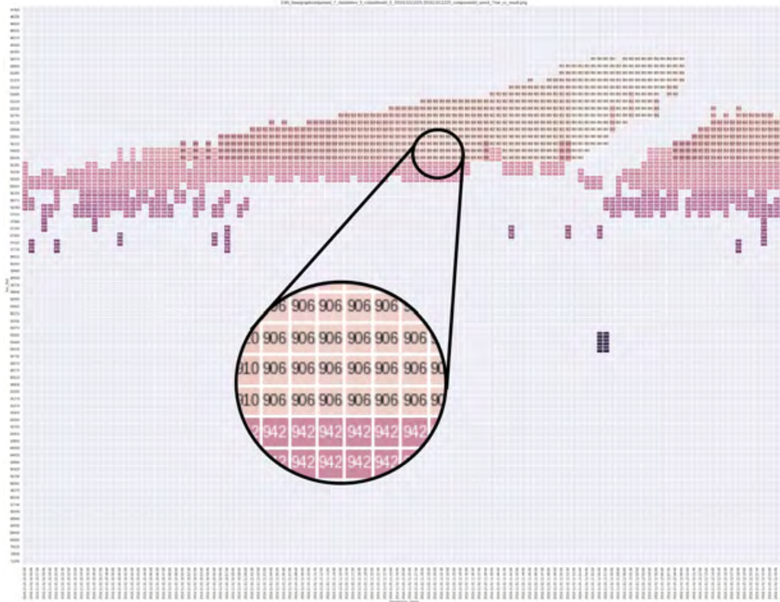


Figure 5.16: Individual queue IDs. Each cell in the heat map is annotated with the ID of the connected component (queue) it belongs to. The zoomed part of the image shows adjacent components 906 and 942. The cell colors represent the different component IDs. Results from congested components, for test congestion pattern 3, E4N component 7, 2016-11-1 10:25-12:25, with congestion class threshold 5.

5.7 Queue tracker

The results of the queue tracking algorithm were verified manually. For a detected queue in minute t, the algorithm is able to find its parent queue or queues in minute $t-1$. Figure 5.18 shows an example of detected queues in two adjacent minutes. In the first minute $t-1$, the queue is split in two, with IDs 890 and 902. In the second minute t, the two queues have merged into a single queue with ID 890. The queue tracking algorithm correctly identifies the parent queues in minute $t-1$, assigning IDs `[890, 902]` as `parentClusterIds` to queue 890 in minute t.

The results were verified manually, examining cases when there is no parent queue in minute $t-1$ for a given queue in minute t, when there is a single parent queue, and when there are multiple parent queues (as in figure 5.18).

Figure 5.17: Adjacent queues. Two different queues are visible in the image, the one towards the bottom of image with ID 906 and the one in the top half with ID 942. The queues are separated by a non-congested road segment. Triangles represent the traffic sensors and the red lines represent congested links. Results from congested components, for test congestion pattern 3, E4N component 7, 2016-11-1 11:30, with congestion class threshold 5.

5.8 Performance evaluation

The performance of the implemented algorithms was also evaluated with regards to execution time and memory footprint. The algorithms were run on the data behind test congestion pattern 1, i.e. data from 2016-11-01 15:50 to 17:10. Note that although the test congestion pattern is on E4N component 0, the entire road network graph is streamed and processed by the algorithms, not just the sensor measurements belonging to the selected queue.

5.8.1 Experimental setup

The stream source simulation program was set up so that every 30 seconds (real world time), the data from the next minute would be ingested by the streaming system. This is to make sure that the system finishes processing minute t before receiving data from minute $t + 1$, the goal being to measure how long it takes

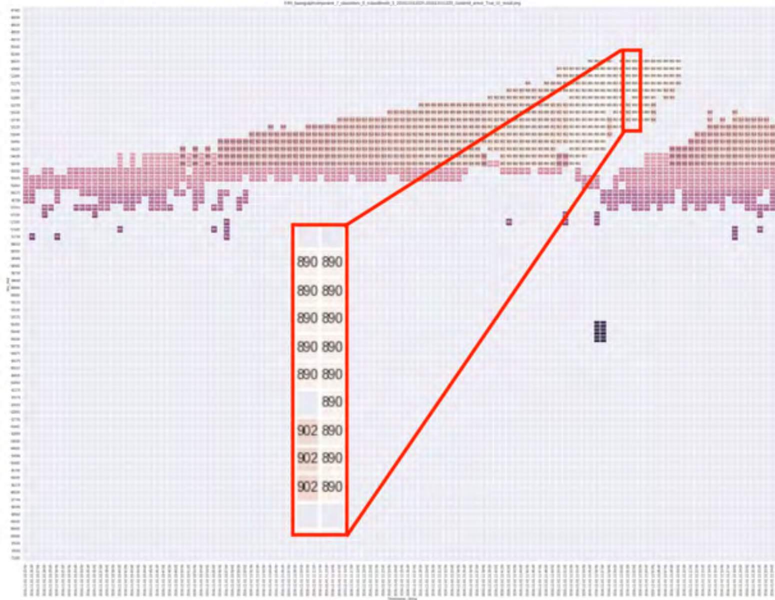


Figure 5.18: Queue tracker results. Queues 890 and 902 in minute $t-1$ are identified as parent queues of queue 890 in minute t.

to process each minute's worth of data, and the size of the state accumulated for each minute. By setting the experiment up this way, the entire minute's worth of data is processed within one trigger (micro-batch), allowing for analysis on a trigger by trigger basis. Focusing on minute-by-minute analysis makes sense for two reasons. First, based on the requirements of the application, the implemented congestion detection methods should be able to process each minute's worth of data well within a minute, to allow for (near) real-time congestion tracking. Second, analyzing minute-by-minute allows for the direct comparison of the implemented algorithms, since it is guaranteed that they are processing the same data during a given trigger.

A `StreamingQueryListener` was set up to monitor the streaming queries. It allows asynchronous monitoring of streaming queries, receiving callbacks when a streaming query is started, terminated, and when it makes progress. A progress callback is generated for every trigger, containing a JSON object with various metrics, such as the number of input rows within the trigger, input rows per second, processed rows per second and the time it took to process the trigger. In addition, it shows the total number of rows present in the state of the query, the number of rows in the state that were updated during the trigger, along with the memory taken up by the state. An example of a streaming query callback can be seen in listing 5.1.

```
{
```

```
    ...
    "batchId" : 47,
    "numInputRows" : 1842,
    "inputRowsPerSecond" : 115125.0,
    "processedRowsPerSecond" : 435.6669820245979,
    "durationMs" : {
      "addBatch" : 4137,
      "getBatch" : 2,
      "getOffset" : 29,
      "queryPlanning" : 34,
      "triggerExecution" : 4228,
      "walCommit" : 25
    },
    "eventTime" : {
      "avg" : "2016-11-01T15:37:00.000Z",
      "max" : "2016-11-01T15:37:00.000Z",
      "min" : "2016-11-01T15:37:00.000Z",
      "watermark" : "2016-11-01T15:36:00.000Z"
    },
    "stateOperators" : [ {
      "numRowsTotal" : 2,
      "numRowsUpdated" : 2,
      "memoryUsedBytes" : 975111
    } ],
    ...
}
```


Listing 5.1: Streaming query progress callback (abbreviated). This callback is from a run of the Dengraph algorithm, when processing data from 2016-11-01 16:37 (the event time in the callback, 15:37, is shown in UTC).

Since the experimental set up is such that each trigger contains only data from a single minute (and all of the data from that minute), the progress callback gives measurements of the time it takes to process the data of each minute, and the amount of space required for the state representing that minute.

Note that in the case of the congestion detection algorithms, with watermarks following the max event time seen so far and the group state timeout set to occur when data from the next minute starts arriving, at any given point the state contains the group state of two adjacent minutes (note that `numRowsTotal` is 2 in the query progress callback example in listing 5.1). This is due to the fact that during a given trigger processing data from minute t, the watermark will be set to t starting from *the next* trigger. Recall that the group state times out and is cleared once the watermark *exceeds* the set group state timeout time stamp. The state accumulated

for minute $t - 1$ in the previous trigger is therefore still present as the state for minute t is being built up in the current trigger (as the watermark at that point is set to $t-1$), and will not be cleared until the next trigger when the watermark has been set to t. During the trigger shown in listing 5.1, the state for minute 15:36 (built up in the previous trigger), and the state for minute 15:37 (built up in the current trigger) are present.

5.8.2 Performance evaluation results

Table 5.3 shows the average trigger execution times, number of processed rows per second and memory taken up by each congestion detection algorithm during a run over the data from 2016-11-01 15:50-17:10. Figures 5.19, 5.20 and 5.21 show the data in a graphical manner.

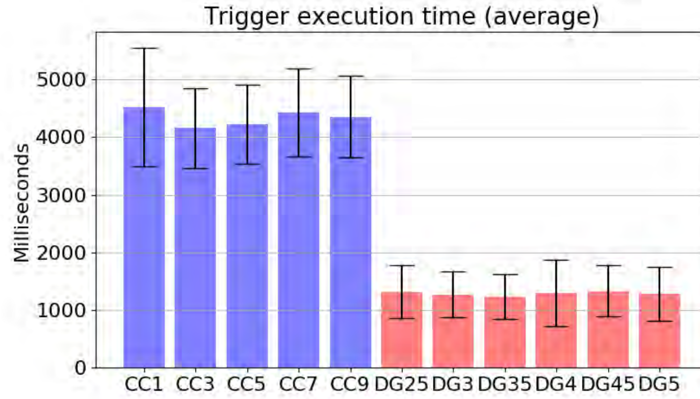


Figure 5.19: Trigger execution times. Average values over a run of the data from 2016-11-01 15:50-17:10, averaged over the values from each minute of the time span (80 minutes). Congested components in blue, Dengraph in red. CC stands for congested components, followed by a number representing the set congestion class threshold. DG stands for Dengraph, followed by a number representing the set neighborhood radius ϵ. All runs of Dengraph have a fixed $\eta = 2$ (min number of nodes in neighborhood).

The time-span under consideration falls under time-span graph 6. According to the meta-data, there are 2037 sensors present in the road network at that time. However, there are at most 1862 valid sensor measurements (rows) per minute in the data set during the time-span (after filtering errors) and a minimum of 1818 measurements, with an average of 1837 measurements per minute.

Note that the measurements taken include the entire streaming program set up around each algorithm, not just the algorithm itself. In the case of congested components, this includes:

	Trigger execution time (ms) (average)	**Processed rows per second (average)**	**Memory per trigger (KB) (average)**
CC (c. class 1)	4514 ($\sigma = 1028$)	421.86 ($\sigma = 68.57$)	50.41 ($\sigma = 8.26$)
CC (c. class 3)	4153 ($\sigma = 692$)	449.31 ($\sigma = 44.44$)	33.66 ($\sigma = 5.64$)
CC (c. class 5)	4224 ($\sigma = 688$)	442.90 ($\sigma = 51.02$)	29.73 ($\sigma = 4.32$)
CC (c. class 7)	4424 ($\sigma = 762$)	424.89 ($\sigma = 57.15$)	27.78 ($\sigma = 3.57$)
CC (c. class 9)	4353 ($\sigma = 704$)	429.49 ($\sigma = 48.34$)	25.75 ($\sigma = 2.70$)
DG ($\epsilon = 0,025$)	1318 ($\sigma = 453$)	1464.82 ($\sigma = 237.63$)	965.71 ($\sigma = 53.23$)
DG ($\epsilon = 0,03$)	1270 ($\sigma = 397$)	1507.88 ($\sigma = 222.93$)	965.71 ($\sigma = 53.23$)
DG ($\epsilon = 0,035$)	1230 ($\sigma = 387$)	1551.77 ($\sigma = 204.06$)	965.71 ($\sigma = 53.23$)
DG ($\epsilon = 0,04$)	1298 ($\sigma = 573$)	1519.47 ($\sigma = 268.25$)	965.71 ($\sigma = 53.23$)
DG ($\epsilon = 0,045$)	1334 ($\sigma = 439$)	1445.01 ($\sigma = 241.93$)	965.71 ($\sigma = 53.23$)
DG ($\epsilon = 0,05$)	1281 ($\sigma = 474$)	1504.72 ($\sigma = 220.47$)	965.71 ($\sigma = 53.23$)

Table 5.3: Performance measurements. Average values over a run of the data from 2016-11-01 15:50-17:10, averaged over the values from each minute of the time span (80 minutes). Standard deviation σ shown in parenthesis. CC stands for congested components, with c. class representing the set congestion class threshold. DG stands for Dengraph, with ϵ representing the set neighborhood radius. All runs of Dengraph have a fixed $\eta = 2$ (min number of nodes in neighborhood).

1. Joining the sensor measurement stream to the static edge list DataFrame to generate a weighted edge stream.

2. Calculating the congestion class

3. Filtering out un-congested edges.

4. Records are run through the connected components algorithm.

5. Exploding the result from the congested components algorithm so that the result for each vertex is in a separate row.

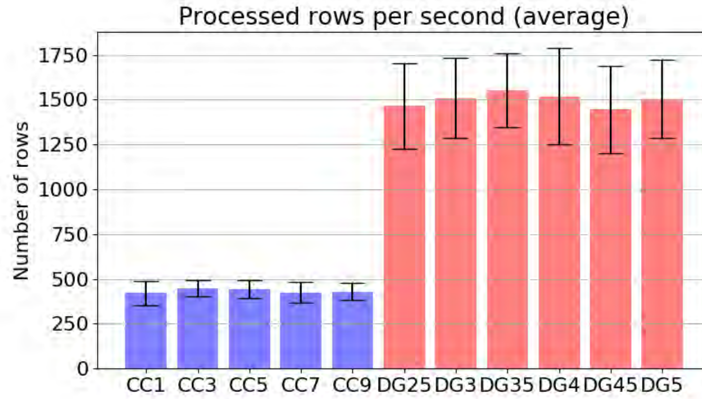


Figure 5.20: Number of processed rows per second. Average values over a run of the data from 2016-11-01 15:50-17:10, averaged over the values from each minute of the time span (80 minutes). Congested components in blue, Dengraph in red. CC stands for congested components, followed by a number representing the set congestion class threshold. DG stands for Dengraph, followed by a number representing the set neighborhood radius ϵ. All runs of Dengraph have a fixed $\eta = 2$ (min number of nodes in neighborhood).

6. And finally, since the connected components algorithm operates on numeric vertex identifiers, joining the streaming connected components algorithm result rows with a static DataFrame to get the road-kmref-lane vertex identifier for each result row.

In the case of Dengraph, the steps are as follows:

1. The sensor measurement stream is also joined with the static edge list DataFrame to generate a weighted edge stream.
2. The distance between the vertices of each edge is calculated.
3. Records are run through the Dengraph algorithm.
4. The result from the algorithm is exploded so that the result for each vertex is in a separate row.

The measured memory usage however only refers to the state that the algorithms themselves use (i.e. the actual connected components and Dengraph algorithms, not the streaming program set up around them).

5.8.3 Effect of parameter selection on performance

The effect of parameter selection on the performance of the congested components and dengraph algorithms was analyzed.

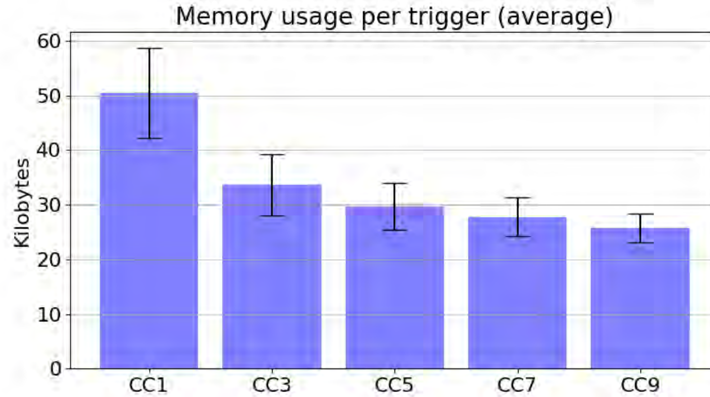


Figure 5.21: Congested components' memory usage per trigger. Average values over a run of the data from 2016-11-01 15:50-17:10, averaged over the values from each minute of the time span (80 minutes). CC stands for congested components, followed by a number representing the set congestion class threshold.

Congested components. In the case of congested components, a cursory look over the results in table 5.3 show that the trigger execution time for all values of the congestion class threshold is between 4 and 4.5 seconds on average, processing 420-450 rows per second. The average memory used per minute is however more varied. With a low congestion class threshold, the memory used is the highest, at 50.41 KB on average. With an increasing congestion class threshold, the memory used drops, down to 25.75 KB on average for congestion class threshold 9. This was to be expected, since a higher congestion class threshold means that more edges from the edge stream will be filtered out, resulting in less data being processed and stored in state by the connected components algorithm.

A one-way ANOVA test was performed to assess if the parameter selection has a significant effect on the performance of the algorithm. The null-hypothesis is that there is no statistical difference between the means of each group (congestion class thresholds 1-9, 80 measurements per group, one for each minute). A significance level of $\alpha = 0,05$ was used. The ANOVA tests show that there is a statistical difference between all measured variables with respect to the congestion class threshold; trigger execution time (p = 0.0251418), processed rows per second (p = 0.00451137) and memory (p = 4.9766e-116). The memory usage shows a clear trend, dropping with increasing congestion class threshold. There is however no clear trend when it comes to trigger execution time and processed rows per second.

Dengraph. A cursory look at the results presented in table 5.3 show that the Dengraph algorithm processes each trigger in just under 1,5 seconds on average, processing around 1500 rows per second on average, for all values of ϵ. The memory used is however constant, at an average of 965.71 KB, for all values of ϵ. This is to be expected since the value of ϵ has no effect on the size of the state used by

the algorithm. Each edge received by the algorithm is kept in state, and state is maintained for each vertex seen by the algorithm, irrespective of the value of ϵ. The variation in the size of the state is due to the fact that the number of input rows per trigger (minute) varies.

As with congested components, a one-way ANOVA test was performed to assess if the value of ϵ has an effect on the performance of the algorithm. The null-hypothesis is the same, that there is no statistical difference between the means of each group (the 6 different ϵ values, 80 measurements per group, one for each minute).

Again, a significance level of $\alpha = 0,05$ was used. The ANOVA tests show no significant difference between all measured variables with respect to the chosen ϵ value; trigger execution time (p = 0.751124), processed rows per second (p = 0.0532203) and memory (p = 1).

5.8.4 Comparison of congestion detection algorithms

To perform a more thorough performance comparison of the two congestion detection algorithms, each algorithm was run on the data from 2016-11-01 15:50-17:10 5 times. The congested components algorithm was run with a congestion class threshold of 5, while the Dengraph algorithm was run with $\epsilon = 0.035$, and $\eta = 2$ as usual. These parameters were chosen as they give the best congestion detection results (see section 5.5) and would therefore likely be used in production. The performance measurements for the two algorithms can be seen in table 5.4.

	Trigger execution time (ms) (average)	**Processed rows per second (average)**	**Memory per trigger (KB) (average)**
CC (c. class 5)	4335 ($\sigma = 762$)	432.70 ($\sigma = 52.79$)	29.73 ($\sigma = 4.30$)
DG ($\epsilon = 0,035$)	1264 ($\sigma = 436$)	1521.28 ($\sigma = 226.11$)	965.71 ($\sigma = 52.97$)

Table 5.4: Performance measurements. Average values over 5 runs of the data from 2016-11-01 15:50-17:10, averaged over the values from each minute of the time span (80 minutes per run, 5 runs). Standard deviation σ shown in parenthesis. CC stands for congested components, with c. class representing the set congestion class threshold. DG stands for dengraph, with ϵ representing the set neighborhood radius. Dengraph was run with $\eta = 2$ (min number of nodes in neighborhood).

There is a statistical difference between the two algorithms for all measured variables. Although quite clear from inspecting table 5.4, this was confirmed with a two-sample t-test with a significance value $\alpha = 0.01$.

The congestion class algorithm is significantly slower than Dengraph with regards to trigger execution time and the number of processed rows per second. How-

ever, it requires significantly less space for its state.

As previously discussed, the performance measurements taken include the entire streaming program set up around each algorithm. A possible performance bottleneck for the congested components streaming program is the fact that the connected components algorithm operates on numeric vertex IDs, while the sensor measurements stream contains vertex IDs of the form `RoadID-KmRef-LaneId`. The numeric IDs are generated within the streaming program as part of the static edge list DataFrame. A stream-static join is performed between the connected components result stream (which only contains numeric vertex IDs) and a static DataFrame to get the road-kmref-lane vertex IDs that the sensor measurements stream uses before sending the results downstream. It would be possible to piggy-back the `RoadID-KmRef-LaneId` vertex ID through the connected components algorithm, along with the numeric vertex ID. This would perhaps lead to better execution time performance, at the expense of a larger state.

5.8.5 Queue tracking algorithm

The queue tracking algorithm behaves differently than the congestion detection algorithms. It only calculates the parent clusters and emits results when every *final* row from a given pair of minutes is assumed to have arrived. Until that happens, it discards non-final rows and accumulates final rows in its state. This leads to uneven work being performed each trigger. Some triggers only accumulate rows in state, while others perform the actual computation of the algorithm. Table 5.5 shows the average trigger execution time, number of processed rows per second, and memory taken up by the algorithm during a run over the data from 2016-11-01 15:50-17:10.

As can be seen in table 5.5, the standard deviation of the measurement values is quite high, indicating a high dispersion of the measured data. This is in keeping with the uneven work being performed in each trigger. Figures 5.22 and 5.23 show kernel density estimation plots for the probability density functions of the trigger execution time and memory usage of each trigger.

A couple of outliers can be seen in the trigger execution times in figure 5.22, the max being 5325 ms in the run for congestion class threshold 9 (shown in purple). Recall that the amount of work performed in each trigger varies. Some triggers do not perform any computation and only accumulate incoming rows in the state, while other triggers may perform the parent cluster computation for a number of minute pairs, resulting in higher than average trigger execution times.

The relationship between the congestion class threshold and the memory used in each trigger is clearer. With increasing congestion threshold, the memory used decreases. This is to be expected since a higher congestion threshold means that more edges will be determined to be un-congested and filtered out. Fewer vertices will be assigned to a congested component, thus decreasing the size of the state that the queue tracker must maintain. Due to the uneven work performed in each trigger there are a number of outliers also visible in the measured memory usage, at points where data from a number of minutes has been accumulated in the state.

	Trigger execution time (ms) (average)	**Processed rows per second (average)**	**Memory per trigger (KB) (average)**
QT (CC c. class 1)	1405 ($\sigma = 402$)	662.73 ($\sigma = 903.36$)	535.07 ($\sigma = 205.53$)
QT (CC c. class 3)	1235 ($\sigma = 343$)	331.80 ($\sigma = 723.76$)	257.80 ($\sigma = 125.83$)
QT (CC c. class 5)	1411 ($\sigma = 485$)	214.03 ($\sigma = 409.57$)	187.74 ($\sigma = 96.10$)
QT (CC c. class 7)	1223 ($\sigma = 347$)	174.09 ($\sigma = 245.49$)	153.12 ($\sigma = 69.57$)
QT (CC c. class 9)	1423 ($\sigma = 560$)	92.57 ($\sigma = 148.83$)	105.91 ($\sigma = 55.95$)

Table 5.5: Performance measurements of the queue tracking algorithm, run on the results output stream of congested components with the given congestion class threshold. Average values over a run of the data from 2016-11-01 15:50-17:10, averaged over the minutes of the time span (80 minutes). Standard deviation σ shown in parenthesis.

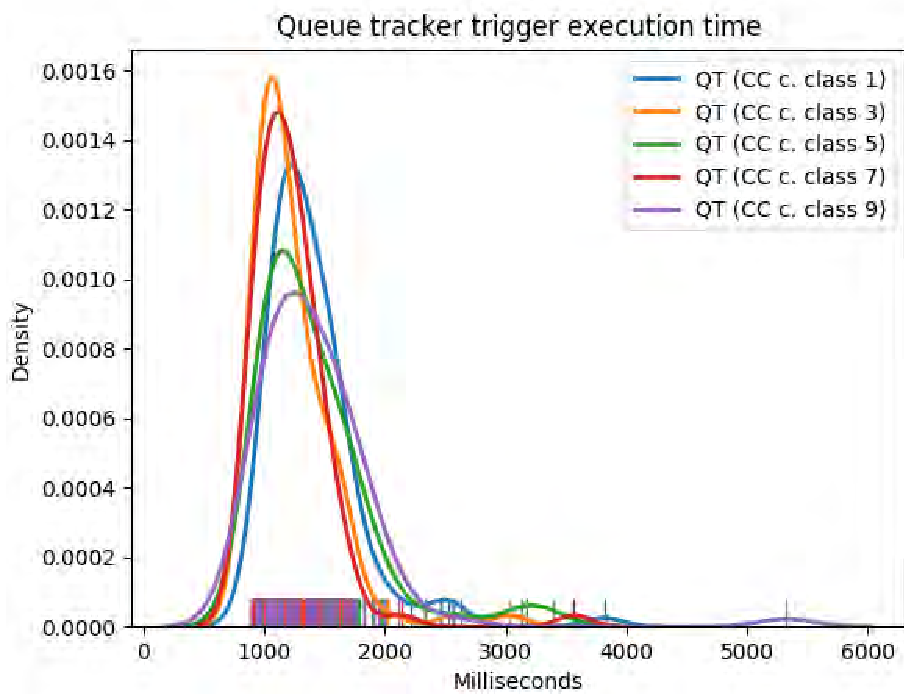


Figure 5.22: Kernel density plot for the queue tracker trigger execution times, when run of the results output stream of congested components with the given congestion class threshold. A rug plot for the individual data points can be seen along the horizontal axis. Data taken from a run over the sensor measurement data from 2016-11-01 15:50-17:10.

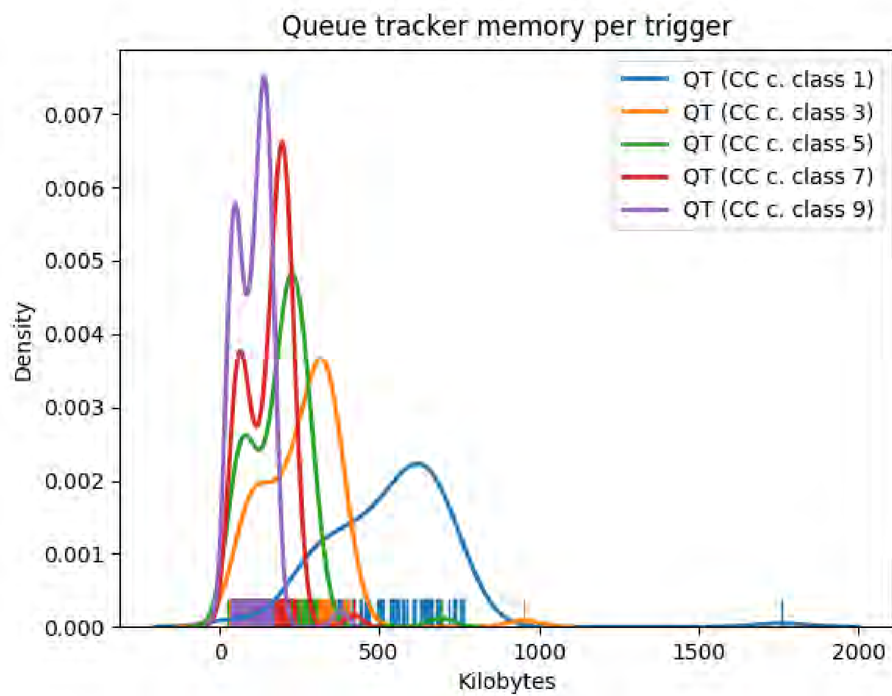


Figure 5.23: Kernel density plot for the queue tracker memory usage per trigger, when run of the results output stream of congested components with the given congestion class threshold. A rug plot for the individual data points can be seen along the horizontal axis. Data taken from a run over the sensor measurement data from 2016-11-01 15:50-17:10.

Chapter 6

Conclusions and future work

The questions set forward in the beginning of the thesis were successfully answered. The traffic sensors were represented as two types of graphs. The base graph, representing the flow of traffic through the road network while disallowing lane changes, and the reachability graph representing the flow of traffic with lane changes allowed. Both graphs represent the traffic sensors in the road system as vertices, and the possible routes for the flow of traffic as directed edges between the vertices. In the case of the base graph, the sensor measurements of the destination vertex of an edge are used as the weight for the edge, representing the traffic conditions on the road segment between two vertices. Although the reachability graph was not used in the methods implemented in the thesis, it could be used in a send warnings system as discussed in section 6.1.3.

Four different methods were implemented to detect congestion in the sensor graph on batch data. The problem was examined as a graph processing problem, adapting existing graph processing algorithms for the task of congestion detection in road network graphs.

The weighted Louvain modularity community detection algorithm was tested, as a representative of the popular modularity optimization methods for community detection. The results of the algorithm were poor. The modularity community division quality measure is not suitable for the simple road graph, where traffic sensors are represented mostly as a chain. The modularity community division quality measure relies on the comparison of the detected community division with what would be expected in a graph with the same community division and degree distribution, but randomized edges between the vertices of the graph. The comparison to a random graph is not suitable for the simple chain-like road graph.

As the traffic sensor base graph is simple (mainly a chain of vertices) an attempt was made to perform data clustering (as opposed to graph clustering) on the sensor data of each minute, using the graph after the fact to split the resulting clusters of vertices so that all vertices within a "split cluster" form a connected component in the road system graph. The goal being to group sensors into congested sensors and free flow sensors, and using the connected components algorithm to find connected

components of congested sensors which are taken to represent queues. Both average speed measurements and density measurements were used as features in the clustering. The method is able to detect congested regions while missing details such as shockwave patterns. Furthermore the method requires two steps, a clustering step followed by the computation of connected components.

The third congestion detection method implemented, named congested components, involved defining a congestion severity class for each sensor measurement, based on historical data from the the respective sensor, to detect congestion at each individual sensor. Connected components of congested sensors in the road graph were then found, allowing to track congestion in the spatial dimension, over a number of neighboring sensors. This method proved to yield the best results, detecting queues with an accuracy at best up to $\approx 94\%$ and shockwave patterns with an accuracy at best up to $\approx 93\%$.

Finally, the density based graph clustering algorithm Dengraph was implemented, with good results. Originally used for community detection in social network graphs, the algorithm was adapted for the task of congestion detection in traffic sensor networks. The algorithm gave good results, detecting queues with an accuracy at best up to $\approx 87.5\%$ and shockwaves with an accuracy at best $\approx 73.3\%$.

Overall comparison of the queue and shockwave detection capabilities of the congested components and Dengraph algorithms reveals that the congested components algorithm outperforms the Dengraph algorithm in terms of accuracy.

The congestion detection algorithms allow for the detection of each individual queue in the road network, where a chain of connected congested sensors are taken to represent a queue. A queue tracking algorithm based on the Jaccard distance between clusters was implemented to allow for the tracking the evolution of each individual queue through time.

The two best performing congestion detection algorithms in terms of accuracy, congested components and Dengraph, along with the queue tracking algorithm, were implemented as streaming systems using Spark Structured Streaming. The streaming implementations allow for real time congestion detection and tracking through the road network.

The performance of the streaming implementations was evaluated with respect to the time taken to process each minute's worth of sensor measurement data from the entire road network, their throughput, and memory footprint. Congested components processes each minute's worth of data in on average between 4 and 4.5 seconds, processing about 420-450 rows per second, depending on the set congestion threshold value. At worst, it uses just over 50 KB of memory per trigger for congestion class threshold 1, with the memory decreasing with increasing congestion threshold. Dengraph processes each minute's worth of data more than twice as fast, taking about 1.5 seconds and processing about 1500 rows per second. However, its memory footprint is larger, taking a constant 965.71 KB on average, as the entire road network graph is built up in memory.

6.1 Future work

The following sections discuss possible directions for future work.

6.1.1 Scalability considerations

The road sensor network under study in this thesis project is quite small, with only 2037 vertices and 2077 edges. It would be interesting to evaluate the scalability of the approaches with respect to a growing traffic sensor graph.

Scalability might also be achieved through graph partitioning. The road traffic graph might be partitioned into overlapping partitions, with separate instances of the stream processing algorithms handling each partition. If needed, the overlap of the partitions would allow for the congestion detection and tracking results to be reconciled into a single whole after the fact. With regards to the streaming systems implemented in this project, the graph partitioning approach could be implemented with minimal effort. The sensor measurements from each graph partition could be published in separate Kafka topics with a dedicated instance of the congestion detection and tracking algorithms processing the data from each topic.

Work is being done on the partitioning of the road network graph in another thesis project under the same research project at RISE SICS, in parallel to the work done in this thesis project. Combining the results from the two thesis projects could be of interest.

6.1.2 Ground truth and accuracy evaluation strategy

It would be beneficial to develop a better accuracy evaluation strategy. The evaluation done in this thesis project involves the visual comparison of labeled and detected congestion patterns on heat maps, taking the human expert approach. This introduces some subjectivity.

A human expert could label the data set to identify congested sensor readings, instead of congestion patterns. Each cell in the test congestion pattern heatmaps, belonging to a queue or shockwave of interest, could be labeled manually. The accuracy of the congestion detection algorithms might then be evaluated on a "per-cell" basis, identifying the number of true positives, false positives and false negatives, allowing for computation of per-sensor precision and recall metrics.

Furthermore, the implemented congestion detection and tracking methods could be evaluated on synthetic data generated by traffic simulation software. Traffic simulation software allows for the complete observation of the state of the (simulated) traffic flow, giving e.g. individual vehicle trajectories and precise queue lengths in terms of numbers of vehicles, essentially providing ground truth for the congestion state in the road network.

6.1.3 End-of-queue warning system

The work performed in the thesis involves creating methods for the detection and tracking of congestion through the road system. Connected sequences of congested sensors in the graph are taken to represent queues. For the detected queues, a warning system might be created to send warnings to the sensors upstream of the queue sensors. The reachability graph would be used for this. It gives information on all sensors upstream of the queue sensors while taking into account possible lane changes, making it possible to send warnings to all sensors upstream of the queue through which traffic can reach the end-of-queue.

6.1.4 End-of-queue detection system

The implemented methods do not put an emphasis on finding the end-of-queue sensors but rather the congestion patterns. A separate end-of-queue detection streaming system could be implemented, looking at sudden drops in average speed, or sudden increase in density, with regards to both space and time.

A version of the method proposed by Chou and Nichols [44] given in eq. 3.4 could be used. It could be modified to use the computed minimum free flow speed per sensor instead of relying on the average speed of the road network to figure out the required speed reduction percentage. This would remove the need to perform aggregations over a number of sensors to generate a result, the end-of-queue could be detected on a per sensor basis, as soon as a sensor reading from a given traffic sensor arrives in the streaming system.

Alternatively, it would be possible to run a separate streaming program maintaining average speed and average density values for each connected component in the base graph. These average values could then be used to identify congested areas (queues) within the base graph components.

The method could also be extended to take a look at the neighboring sensors up and downstream from a given sensor, taking the speed- and/or density differential between neighboring sensors into account when detecting the location of the end-of-queue.

This system could be used in conjunction with the congestion detection and tracking systems implemented in this thesis. The end-of-queue detection system identifying high-risk end-of-queue sensors, and the implemented congestion detection and tracking systems identifying the spatial extent and evolution of their respective queues.

6.1.5 Integrate with streaming predictions

It would be interesting to use machine learning methods to generate short term predictions for the output of the traffic sensors, and run the congestion detection and tracking methods implemented in this thesis on the prediction results in real time. A machine learning model would be integrated in the streaming system, giving predictions X minutes into the future continuously. It would be worth investigating

if this approach could essentially predict congestion and the location of the end-of-queue.

Work is being done on improving the accuracy of short term traffic prediction models in another thesis project under the same research project at RISE SICS, in parallel to the work done in this thesis project. Combining the results from the two thesis projects would be interesting.

6.1.6 Evolutionary clustering

Evolutionary clustering methods are clustering methods that work on time stamped data. As the name suggests, they are concerned with the evolution of clusters through time, creating a sequence of clustering results for each time step. Two conflicting criteria are optimized simultaneously, with a trade-off from one to the other. First, the clustering at a given time step should accurately represent the data from that time step (as if it was regular clustering, only focusing on the data from the one time stamp). This is called snapshot quality. The other is history cost, which measures how great the deviation of a clustering in time step t is from the clustering in the previous time step $t-1$. Combining the two criteria a trade-off is realized, staying faithful to the data from the current time step, and taking the clustering results from the previous time steps into account and not deviate drastically from them, providing a smooth evolution of the clusters [48].

The main benefits of employing evolutionary clustering in our case would be the following:

- *Noise removal.* Since care is taken to not deviate too much from the clustering results from the previous time steps, noise in the current time step will be smoothed out. This is essentially like smoothing through moving averages.
- *Smoothing.* The clusters should transition smoothly with time. This might give a nice view of the traffic queue as it moves through the road system.
- *Cluster correspondence.* Allows for tracking of clusters through time, making it is possible to tell that cluster B in time step t is the same cluster as cluster A in time step $t-1$. This is the goal of the queue tracking algorithm presented and implemented in this thesis.

It would be interesting to look at for instance the AFFECT evolutionary clustering framework [49]. It allows any static clustering algorithm which uses pairwise similarity or dissimilarities to be extended into an evolutionary clustering algorithm.

6.1.7 Continuously update minimum free flow speed per sensor

The congested components method relies on the minimum free flow speed of each sensor. The minimum free flow speed was calculated once, from the three day data set. However, in actual production, this minimum free flow speed might change

with time. Therefore it might be interesting to set up a second streaming system to determine the minimum free flow speed for each sensor periodically, based on new historical data.

Spark Structured Streaming allows for such periodical jobs to be run by using a one-time micro-batch trigger, in which case the streaming program is fired up periodically to process all the new data available since the last run, and then turns off.

This could be done for instance once a day. To handle any outliers and extreme values for the minimum free flow speed, a moving average may be used to smooth out the calculated daily values.

Appendix A

Test congestion pattern heat maps

This section shows the 8 spatio-temporal congestion patterns selected to evaluate the congestion detection methods implemented in the thesis project. There are two heat maps for each of the 8 patterns, one showing average speed and the other density. In the case of average speed, lighter colors represent lower speed. In the case of density, darker colors represent higher density. Blank cells represent missing sensor measurements.

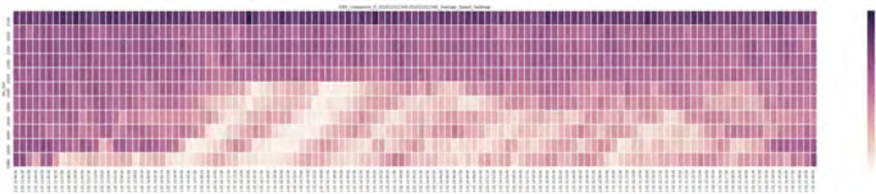


Figure A.1: E6N component 0, 2016-11-01 13:40 to 15:40, average speed heat map.

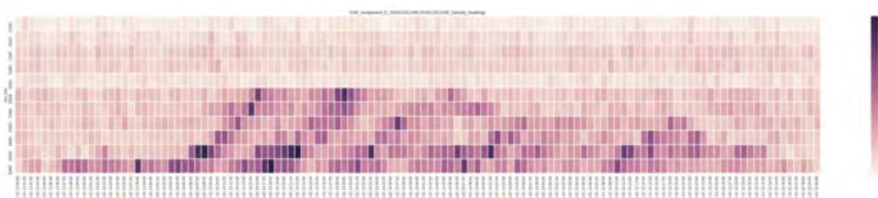


Figure A.2: E6N component 0, 2016-11-01 13:40 to 15:40, density heat map.

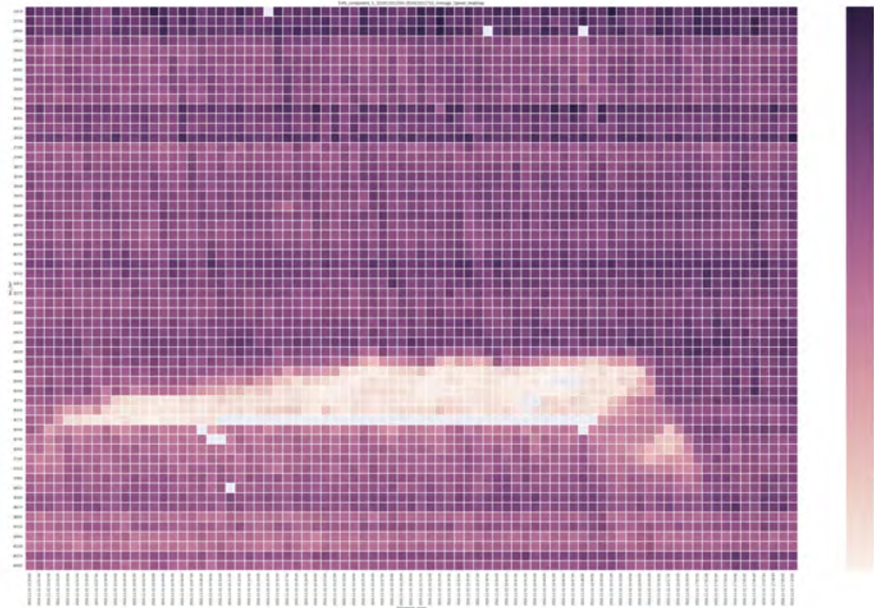


Figure A.3: E4N component 0, 2016-11-01 15:50 to 17:10, average speed heat map.

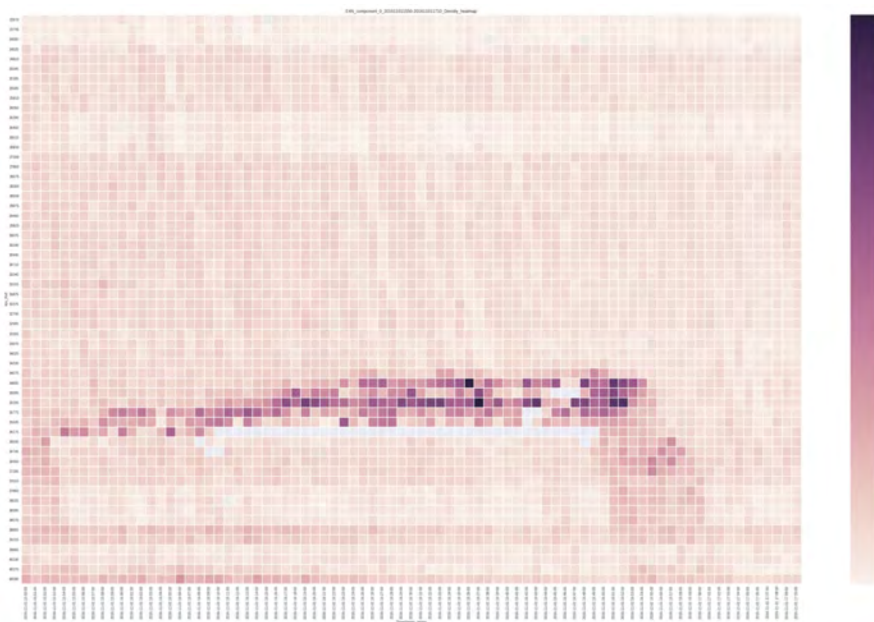


Figure A.4: E4N component 0, 2016-11-01 15:50 to 17:10, density heat map.

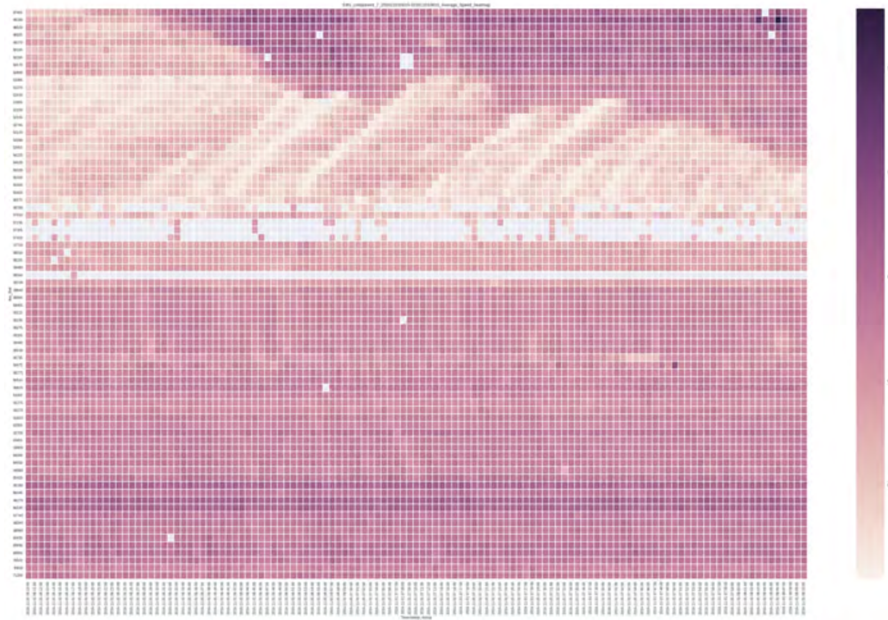


Figure A.5: E4N component 7, 2016-11-01 06:10 to 08:10, average speed heat map.

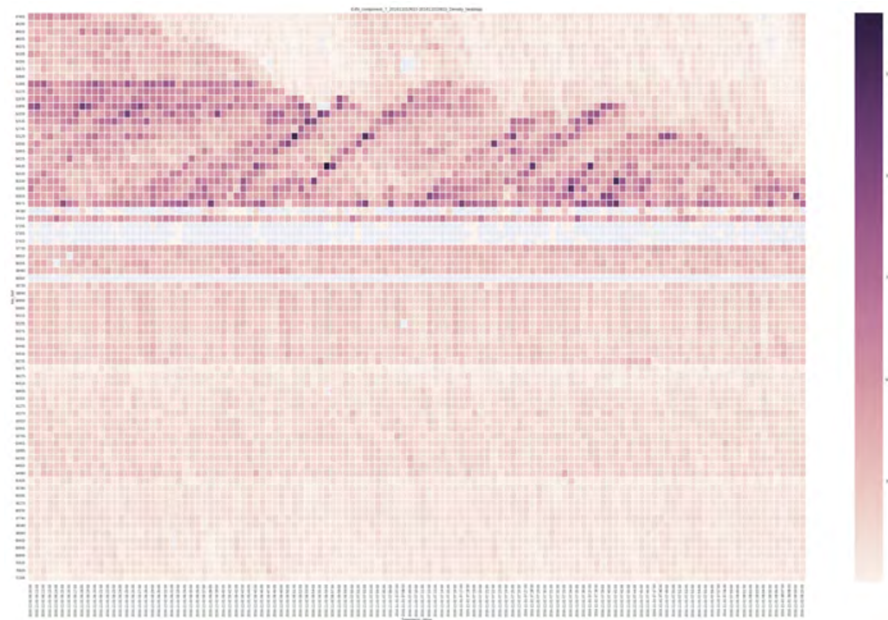


Figure A.6: E4N component 7, 2016-11-01 06:10 to 08:10, density heat map.

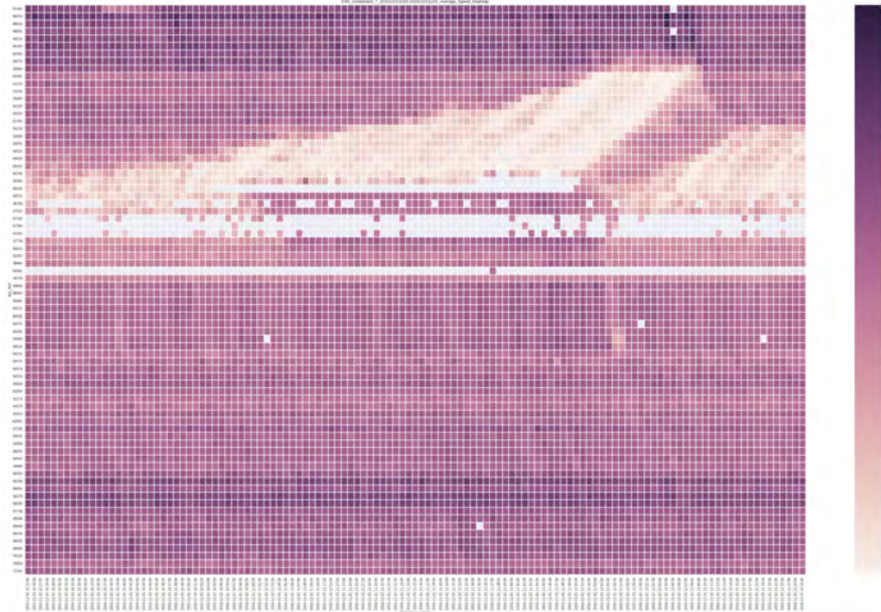


Figure A.7: E4N component 7, 2016-11-01 10:25 to 12:25, average speed heat map.

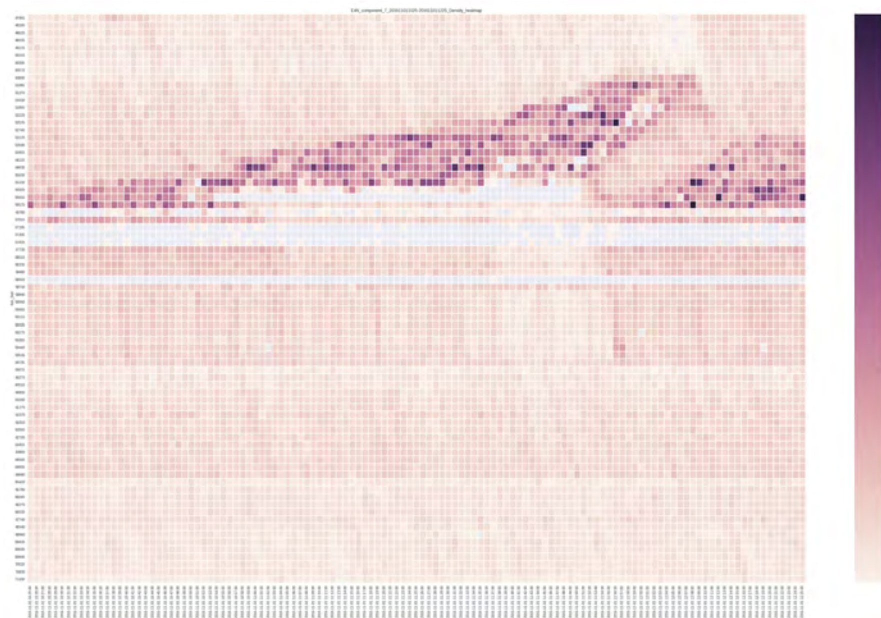


Figure A.8: E4N component 7, 2016-11-01 10:25 to 12:25, density heat map.

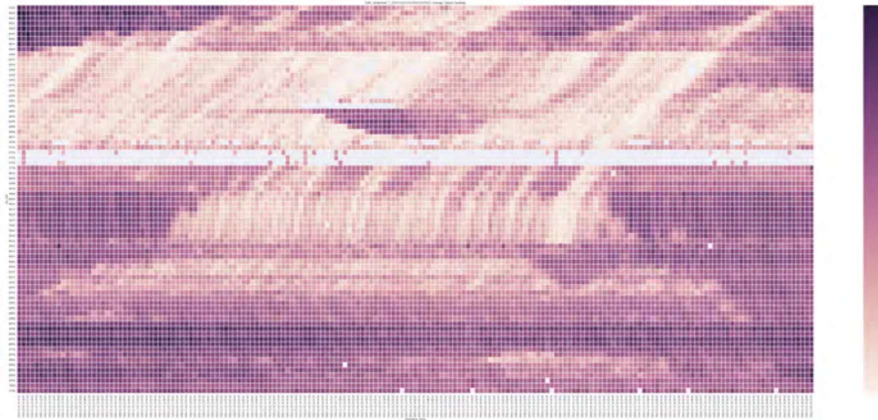


Figure A.9: E4N component 7, 2016-11-01 13:15 to 16:15, average speed heat map.

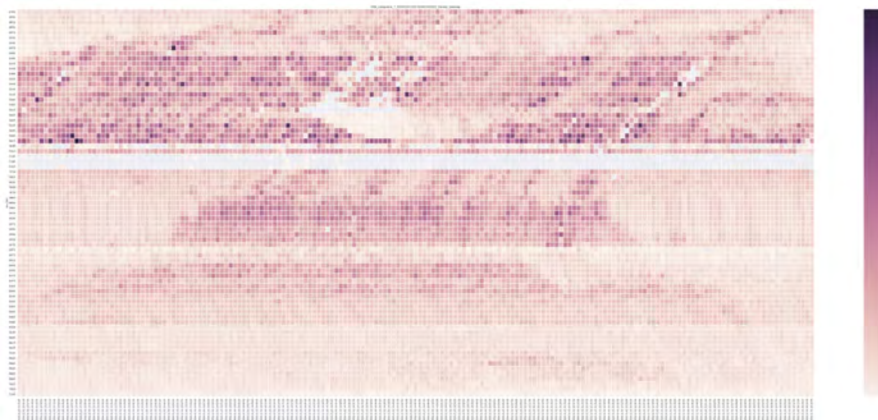


Figure A.10: E4N component 7, 2016-11-01 13:15 to 16:15, density heat map.

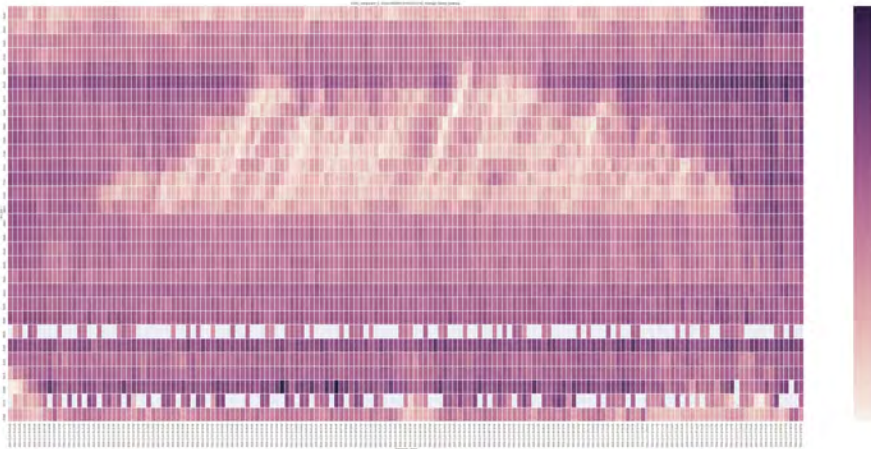


Figure A.11: E18O component 0, 2016-11-01 05:00 to 07:40, average speed heat map.

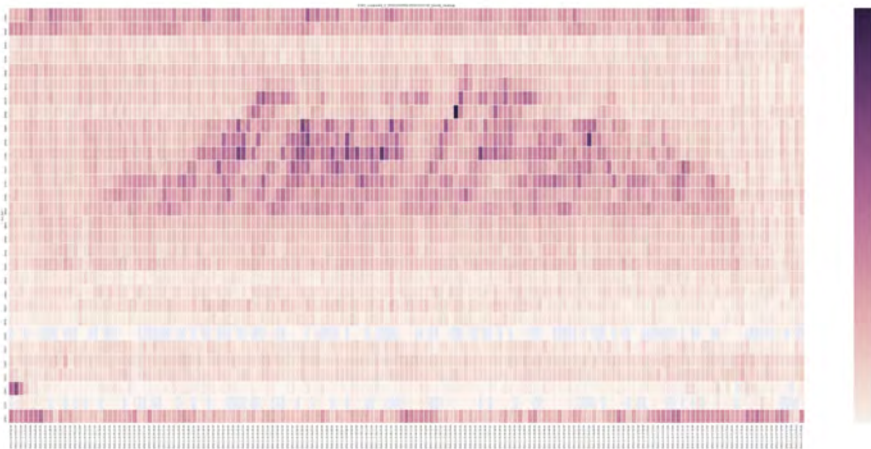


Figure A.12: E18O component 0, 2016-11-01 05:00 to 07:40, density heat map.

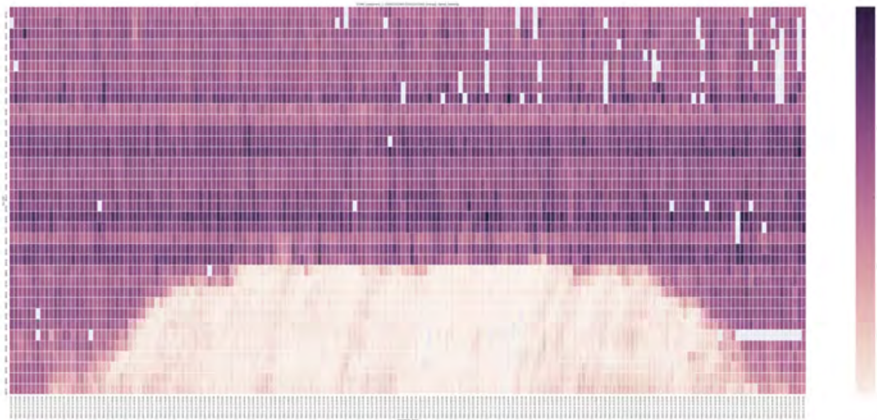


Figure A.13: E20W component 1, 2016-11-01 13:05 to 16:05, average speed heat map.

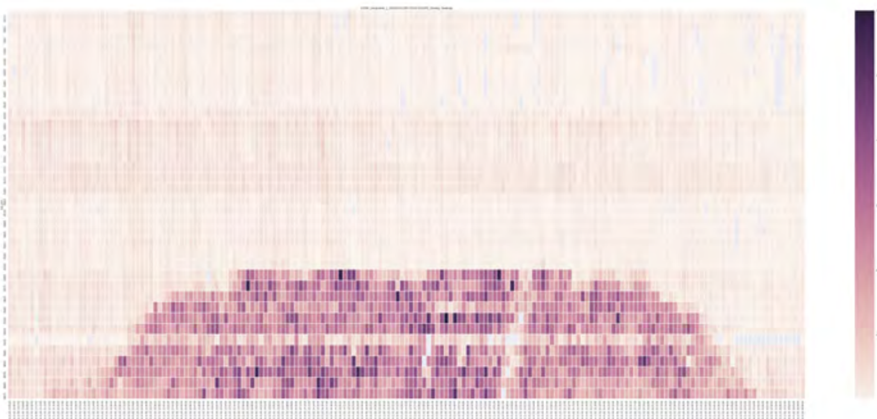


Figure A.14: E20W component 1, 2016-11-01 13:05 to 16:05, density heat map.

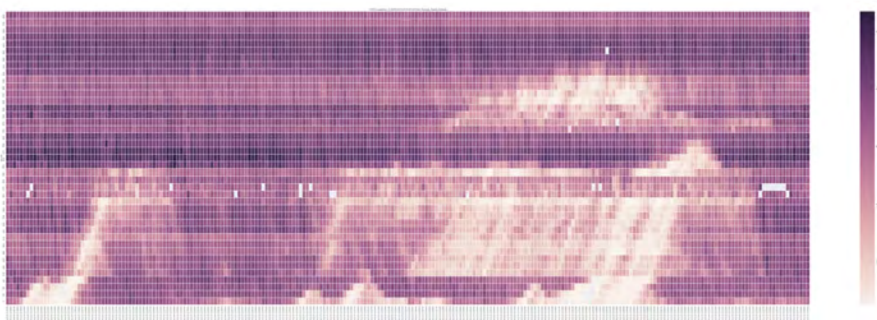


Figure A.15: E75W component 0, 2016-11-01 11:45 to 15:40, average speed heat map.

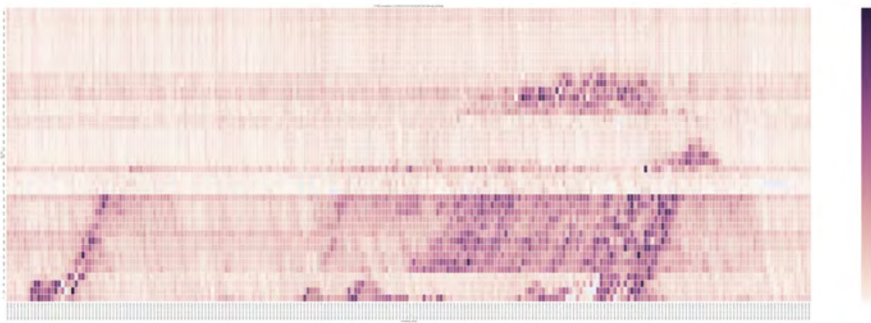
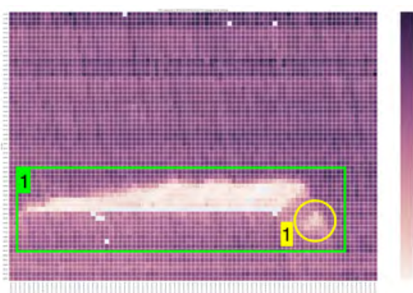


Figure A.16: E75W component 0, 2016-11-01 11:45 to 15:40, density heat map.

Appendix B

Congested components results

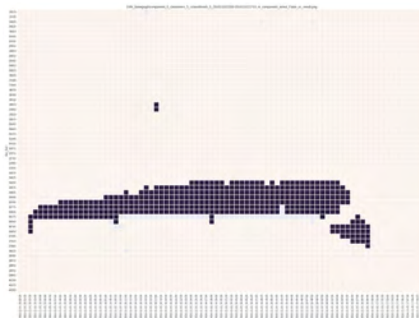
Following are the results from the congested components algorithm. The identified queues in the congestion patterns are delimited by green borders, and the identified shockwaves by yellow borders.



(a) Measured average speed values

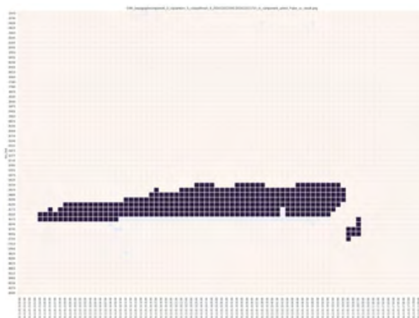
(b) Congestion class threshold 1

(c) Congestion class threshold 3



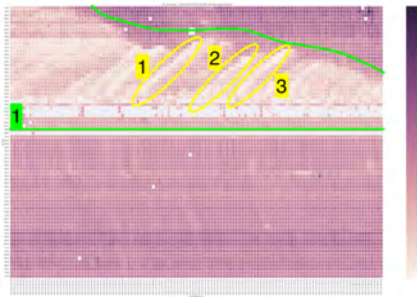
(d) Congestion class threshold 5

(e) Congestion class threshold 7

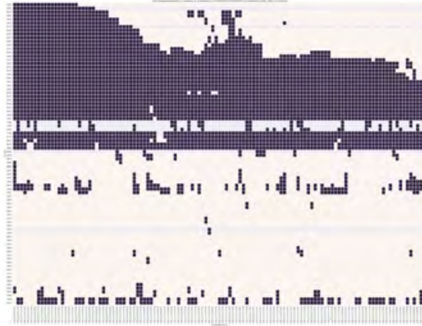


(f) Congestion class threshold 9

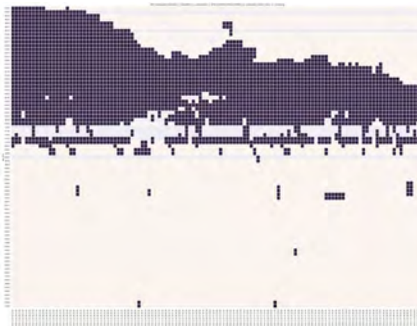
Figure B.1: Congested components results for test congestion pattern 1, E4N component 0, 2016-11-01 15:50-17:10.



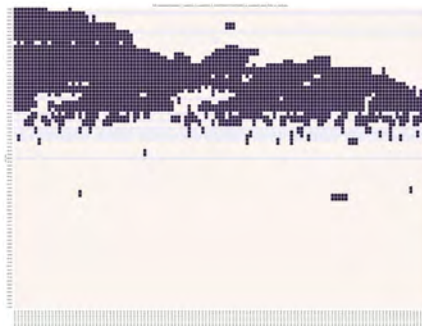
(a) Measured average speed values



(b) Congestion class threshold 1



(c) Congestion class threshold 3

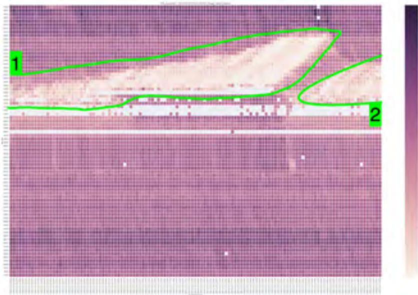


(d) Congestion class threshold 5

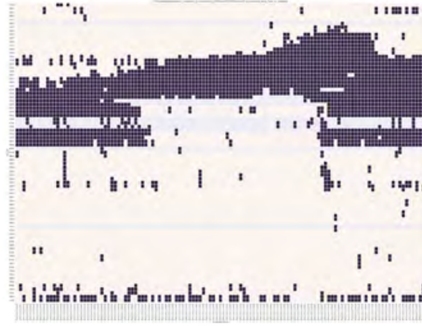
(e) Congestion class threshold 7

(f) Congestion class threshold 9

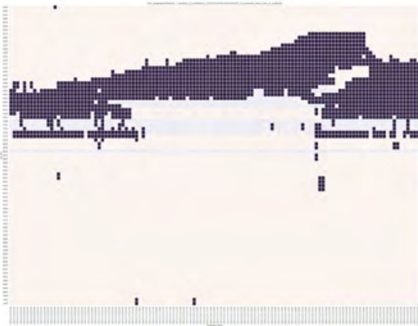
Figure B.2: Congested components results for test congestion pattern 2, E4N component 7, 2016-11-01 06:10-08:10.



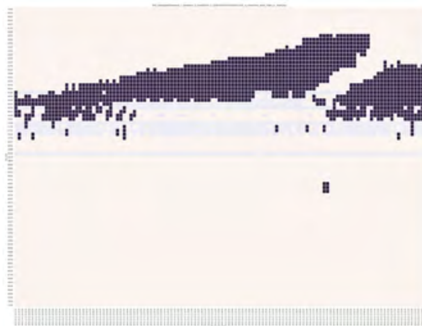
(a) Measured average speed values



(b) Congestion class threshold 1

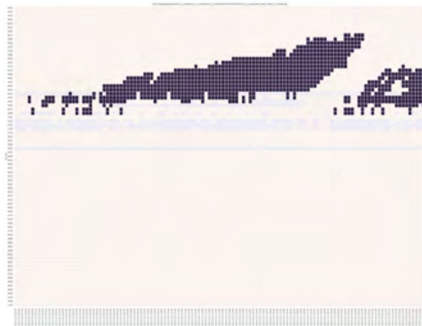


(c) Congestion class threshold 3



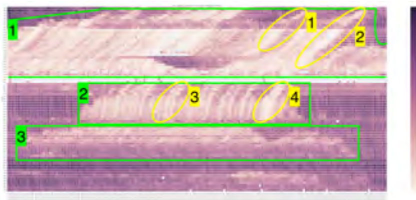
(d) Congestion class threshold 5

(e) Congestion class threshold 7

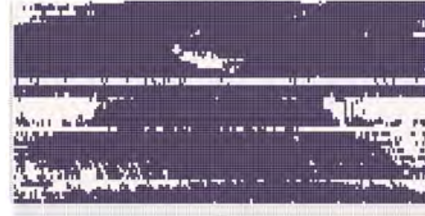


(f) Congestion class threshold 9

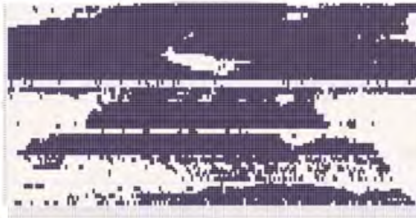
Figure B.3: Congested components results for test congestion pattern 3, E4N component 7, 2016-11-01 10:25-12:25.



(a) Measured average speed values



(b) Congestion class threshold 1



(c) Congestion class threshold 3

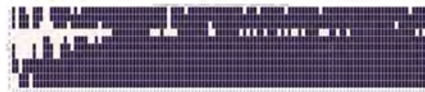
(d) Congestion class threshold 5

(e) Congestion class threshold 7

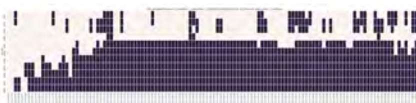
(f) Congestion class threshold 9

Figure B.4: Congested components results for test congestion pattern 4, E4N component 7, 2016-11-01 13:15-16:15.

(a) Measured average speed values

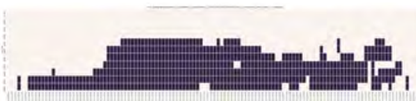


(b) Congestion class threshold 1



(c) Congestion class threshold 3

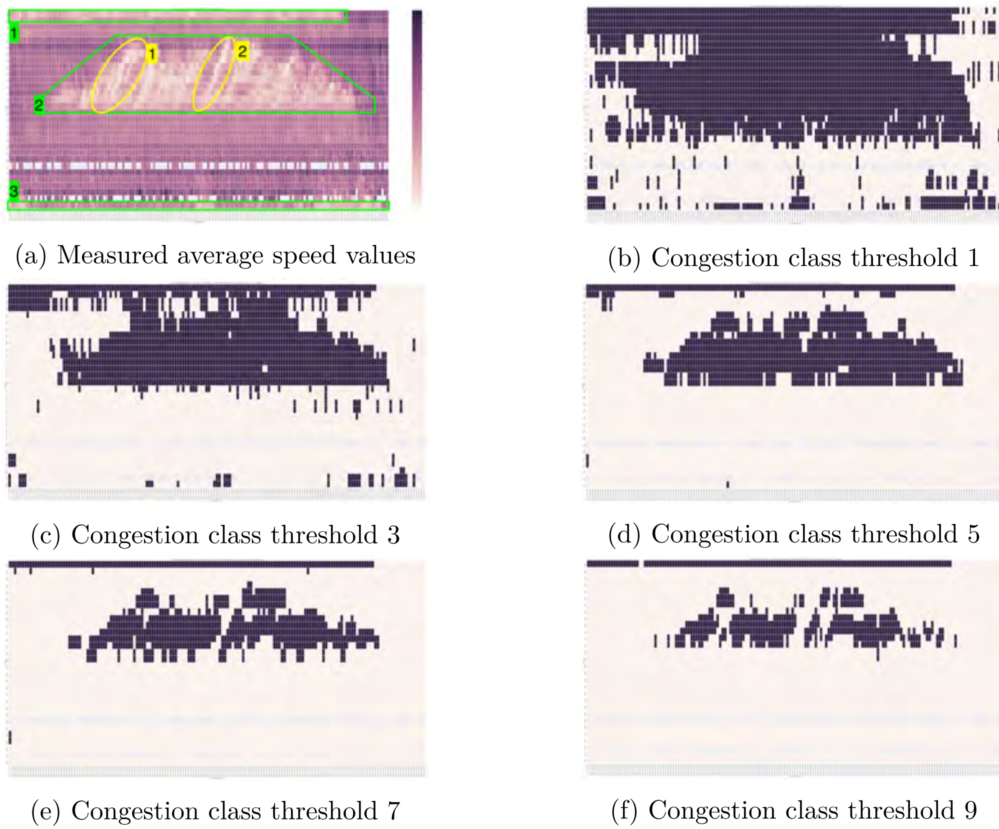
(d) Congestion class threshold 5



(e) Congestion class threshold 7

(f) Congestion class threshold 9

Figure B.5: Congested components results for test congestion pattern 5, E6N component 0, 2016-11-01 13:40-15:40.



(a) Measured average speed values

(b) Congestion class threshold 1

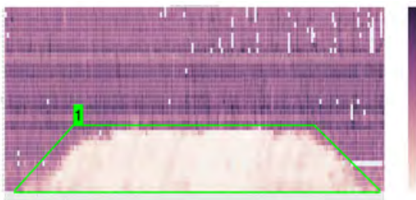
(c) Congestion class threshold 3

(d) Congestion class threshold 5

(e) Congestion class threshold 7

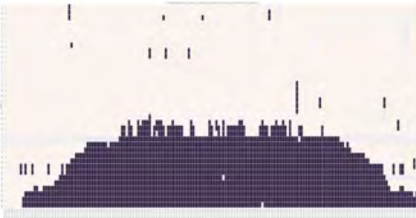
(f) Congestion class threshold 9

Figure B.6: Congested components results for test congestion pattern 6, E18O component 0, 2016-11-01 05:00-07:40.



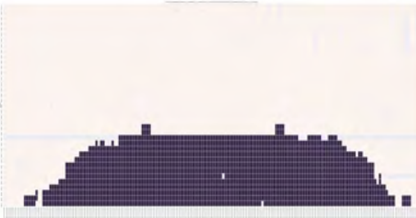
(a) Measured average speed values

(b) Congestion class threshold 1

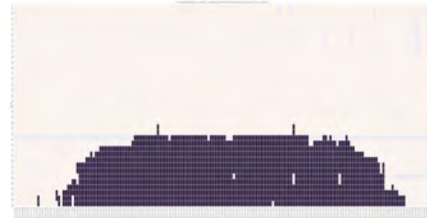


(c) Congestion class threshold 3

(d) Congestion class threshold 5

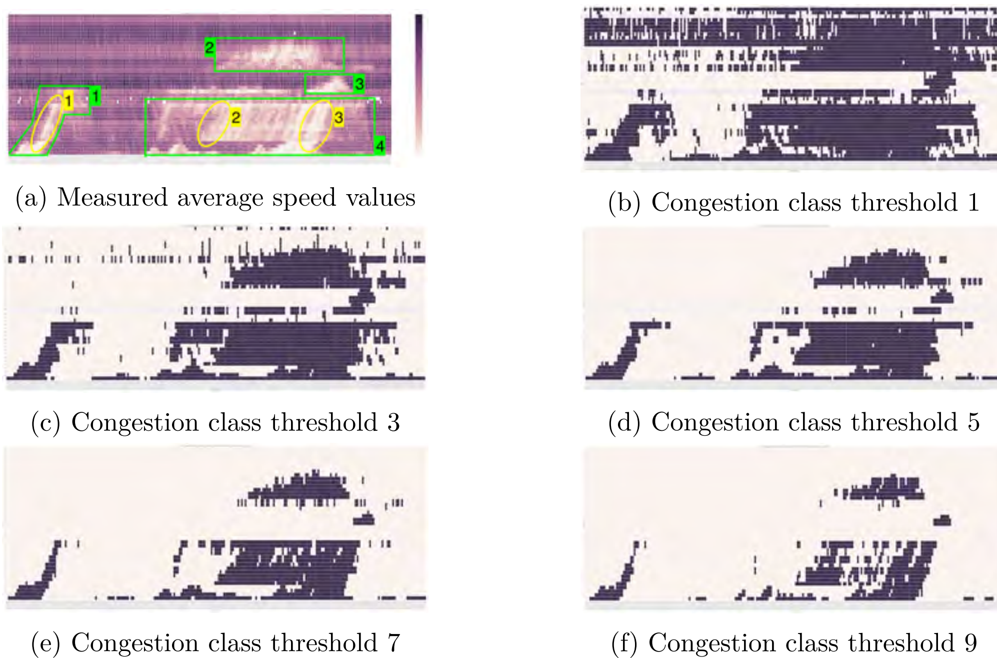


(e) Congestion class threshold 7



(f) Congestion class threshold 9

Figure B.7: Congested components results for test congestion pattern 7, E20W component 1, 2016-11-01 13:05-16:05.



(a) Measured average speed values

(b) Congestion class threshold 1

(c) Congestion class threshold 3

(d) Congestion class threshold 5

(e) Congestion class threshold 7

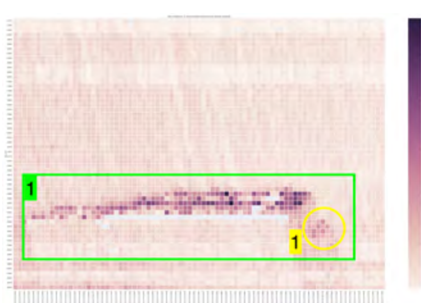
(f) Congestion class threshold 9

Figure B.8: Congested components results for test congestion pattern 8, E75W component 0, 2016-11-01 11:45-15:40.

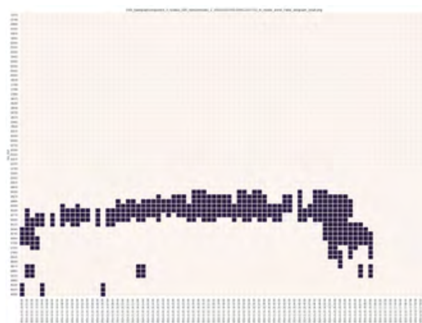
Appendix C

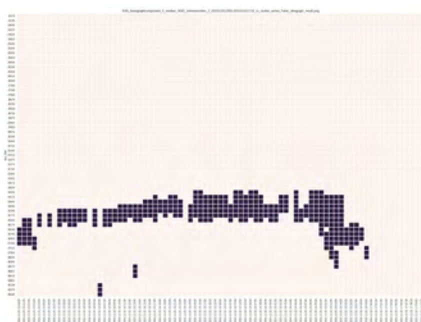
Dengraph results

Following are the results from the Dengraph algorithm. The identified queues in the congestion patterns are delimited by green borders, and the identified shockwaves by yellow borders.



(a) Measured density values


(b) $\epsilon = 0,05$


(c) $\epsilon = 0,045$

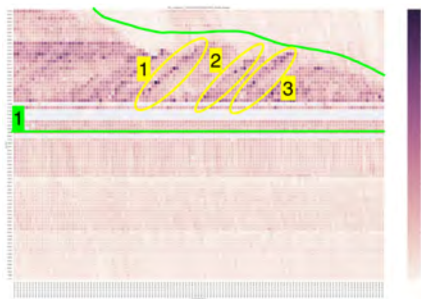
(d) $\epsilon = 0,04$

(e) $\epsilon = 0,035$

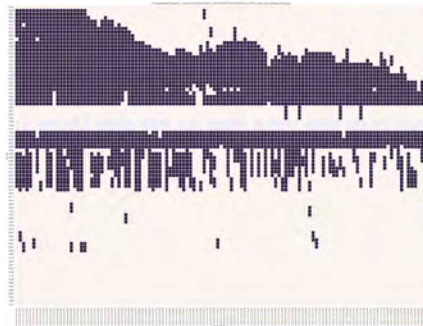
(f) $\epsilon = 0,03$

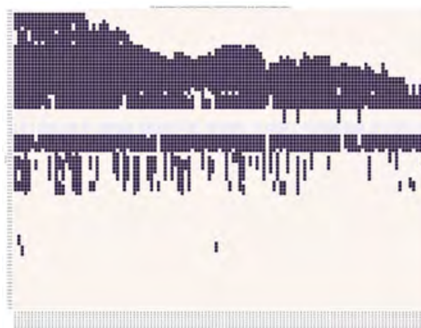
(g) $\epsilon = 0,025$

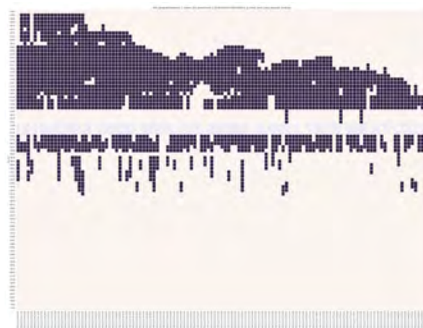
Figure C.1: Dengraph results for test congestion pattern 1, E4N component 0, 2016-11-01 15:50-17:10.



(a) Measured density values


(b) $\epsilon = 0,05$


(c) $\epsilon = 0,045$


(d) $\epsilon = 0,04$

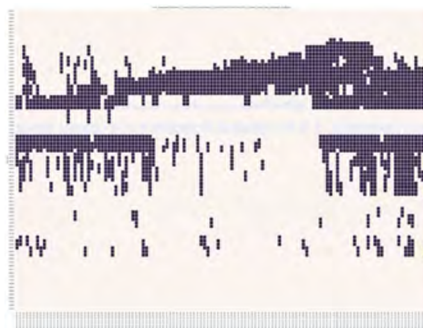
(e) $\epsilon = 0,035$

(f) $\epsilon = 0,03$

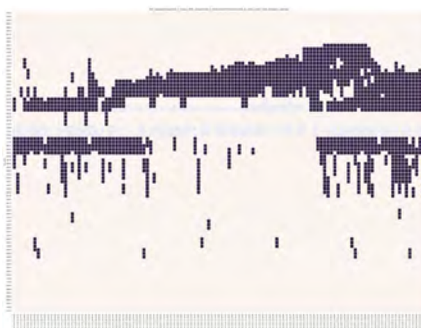
(g) $\epsilon = 0,025$

Figure C.2: Dengraph results for test congestion pattern 2, E4N component 7, 2016-11-01 06:10-08:10.

(a) Measured density values



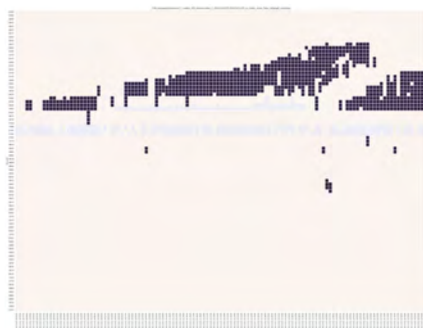
(b) $\epsilon = 0,05$



(c) $\epsilon = 0,045$

(d) $\epsilon = 0,04$

(e) $\epsilon = 0,035$

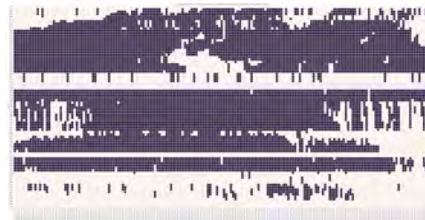


(f) $\epsilon = 0,03$

(g) $\epsilon = 0,025$

Figure C.3: Dengraph results for test congestion pattern 3, E4N component 7, 2016-11-01 10:25-12:25.

(a) Measured density values



(b) $\epsilon = 0,05$

(c) $\epsilon = 0,045$

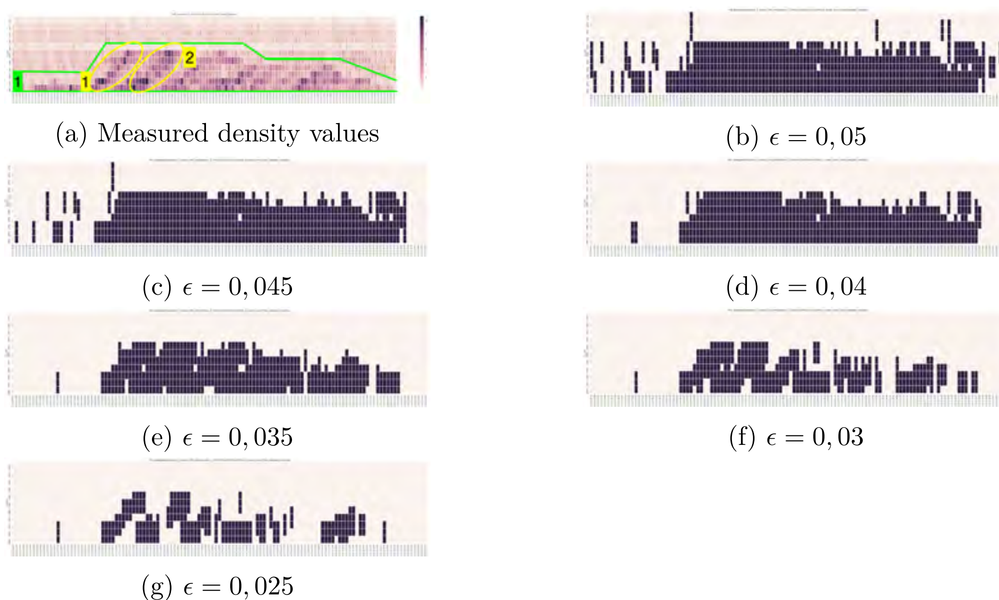
(d) $\epsilon = 0,04$

(e) $\epsilon = 0,035$

(f) $\epsilon = 0,03$

(g) $\epsilon = 0,025$

Figure C.4: Dengraph results for test congestion pattern 4, E4N component 7, 2016-11-01 13:15-16:15.



(a) Measured density values

(b) $\epsilon = 0,05$

(c) $\epsilon = 0,045$

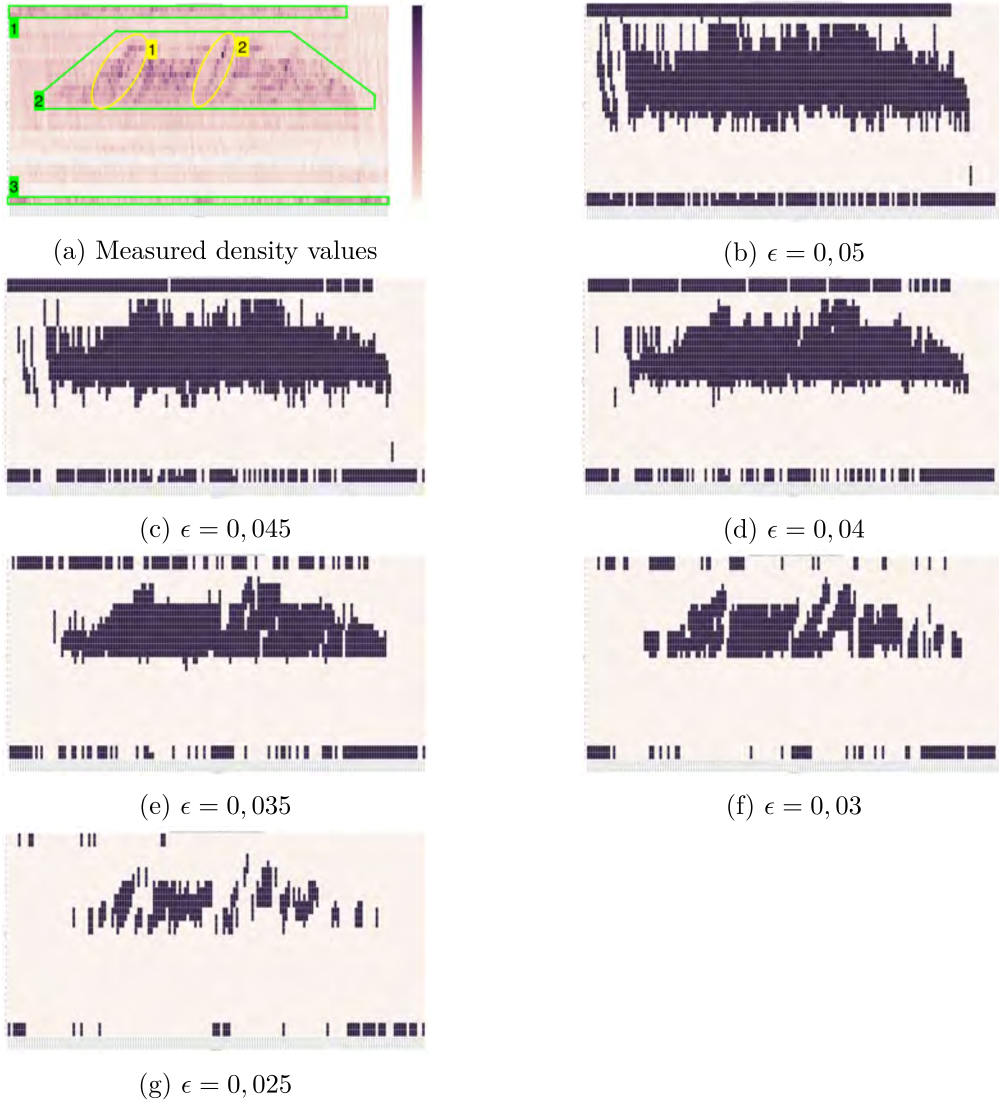
(d) $\epsilon = 0,04$

(e) $\epsilon = 0,035$

(f) $\epsilon = 0,03$

(g) $\epsilon = 0,025$

Figure C.5: Dengraph results for test congestion pattern 5, E6N component 0, 2016-11-01 13:40-15:40.



(a) Measured density values

(b) $\epsilon = 0,05$

(c) $\epsilon = 0,045$

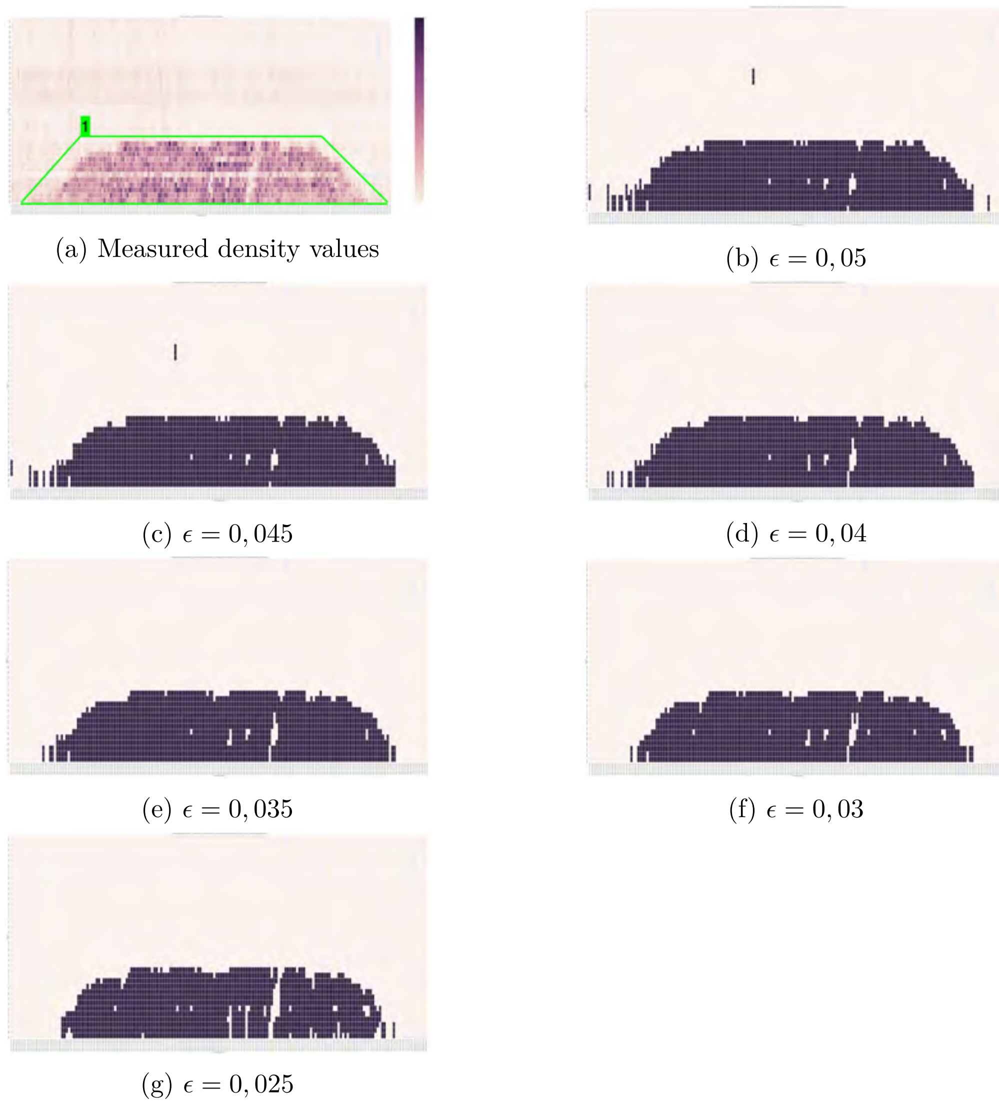
(d) $\epsilon = 0,04$

(e) $\epsilon = 0,035$

(f) $\epsilon = 0,03$

(g) $\epsilon = 0,025$

Figure C.6: Dengraph results for test congestion pattern 6, E18O component 0, 2016-11-01 05:00-07:40.



(a) Measured density values

(b) $\epsilon = 0,05$

(c) $\epsilon = 0,045$

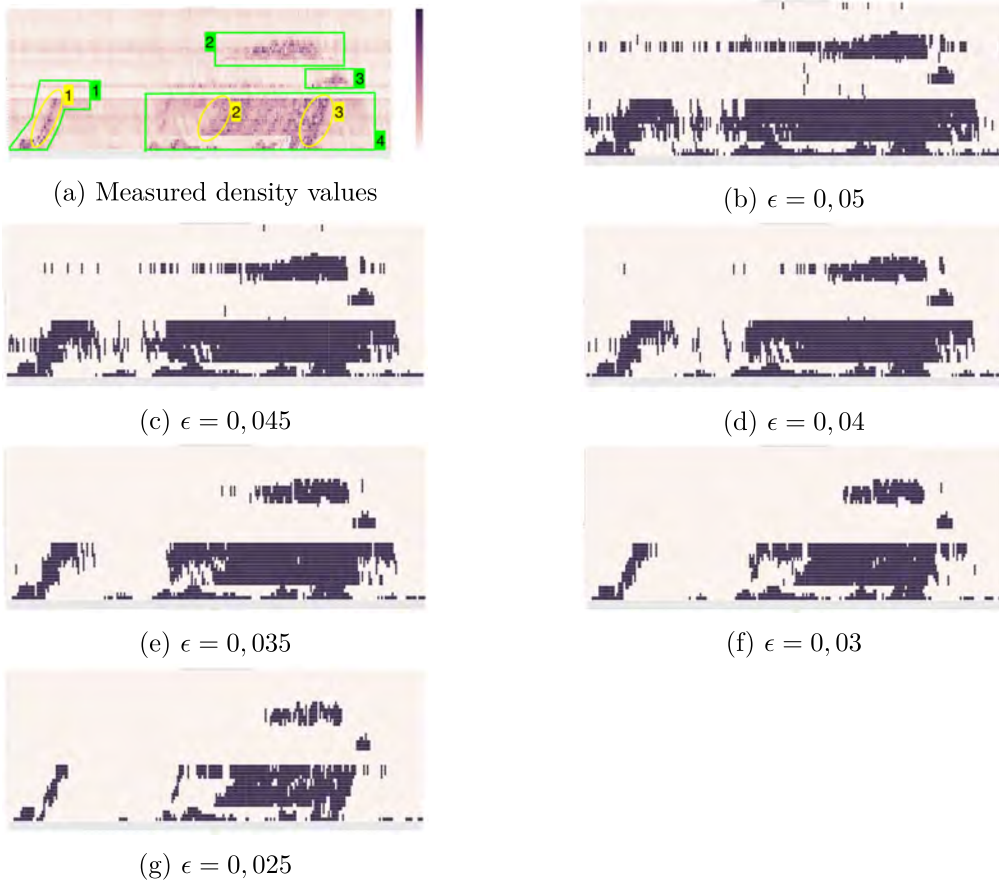
(d) $\epsilon = 0,04$

(e) $\epsilon = 0,035$

(f) $\epsilon = 0,03$

(g) $\epsilon = 0,025$

Figure C.7: Dengraph results for test congestion pattern 7, E20W component 1, 2016-11-01 13:05-16:05.



(a) Measured density values

(b) $\epsilon = 0,05$

(c) $\epsilon = 0,045$

(d) $\epsilon = 0,04$

(e) $\epsilon = 0,035$

(f) $\epsilon = 0,03$

(g) $\epsilon = 0,025$

Figure C.8: Dengraph results for test congestion pattern 8, E75W component 0, 2016-11-01 11:45-15:40.

Appendix D

Accuracy evaluation result tables

The following section shows the accuracy evaluation results for each selected test congestion pattern. There are a different number of labeled queues and shockwaves in each congestion pattern. The total number in each pattern can be seen in the table captions.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	0		0%	0		0%
CC (c. class 3)	0		0%	0		0%
CC (c. class 5)	1	1	100%	1	1	100%
CC (c. class 7)	1	1	100%	1	1	100%
CC (c. class 9)	1	1	100%	1	1	100%
DG ($\epsilon = 0,025$)	0		0%	0		0%
DG ($\epsilon = 0,03$)	0		0%	0		0%
DG ($\epsilon = 0,035$)	0		0%	1	1	100%
DG ($\epsilon = 0,04$)	1	1	100%	1	1	100%
DG ($\epsilon = 0,045$)	1	1	100%	1	1	100%
DG ($\epsilon = 0,05$)	0		0%	0		0%

Table D.1: Accuracy evaluation. Congestion pattern 1. 1 labeled queue and 1 labeled shockwave.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	0		0%	0		0%
CC (c. class 3)	1	1	100%	0		0%
CC (c. class 5)	1	1	100%	1	1	33.3%
CC (c. class 7)	0		0%	2	1,3	66.7%
CC (c. class 9)	0		0%	3	1,2,3	100%
DG ($\epsilon = 0,025$)	0		0%	2	1,3	66.7%
DG ($\epsilon = 0,03$)	0		0%	3	1,2,3	100%
DG ($\epsilon = 0,035$)	1	1	100%	3	1,2,3	100%
DG ($\epsilon = 0,04$)	1	1	100%	2	1,3	66.7%
DG ($\epsilon = 0,045$)	1	1	100%	0		0%
DG ($\epsilon = 0,05$)	0		0%	0		0%

Table D.2: Accuracy evaluation. Congestion pattern 2. 1 labeled queue and 3 labeled shockwaves.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	0		0%			
CC (c. class 3)	1	1	50%			
CC (c. class 5)	2	1,2	100%			
CC (c. class 7)	2	1,2	100%			
CC (c. class 9)	2	1,2	100%			
DG ($\epsilon = 0,025$)	0		0%			
DG ($\epsilon = 0,03$)	0		0%			
DG ($\epsilon = 0,035$)	2	1,2	100%			
DG ($\epsilon = 0,04$)	1	1	50%			
DG ($\epsilon = 0,045$)	1	1	50%			
DG ($\epsilon = 0,05$)	0		0%			

Table D.3: Accuracy evaluation. Congestion pattern 3. 2 labeled queues and 0 labeled shockwaves.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	0		0%	0		0%
CC (c. class 3)	0		0%	0		0%
CC (c. class 5)	3	1,2,3	100%	3	2,3,4	75%
CC (c. class 7)	2	1,2	66.7%	4	1,2,3,4	100%
CC (c. class 9)	1	1	33.3%	3	1,2,4	75%
DG ($\epsilon = 0,025$)	0		0%	2	3,4	50%
DG ($\epsilon = 0,03$)	0		0%	2	3,4	50%
DG ($\epsilon = 0,035$)	3	1,2,3	100%	3	2,3,4	75%
DG ($\epsilon = 0,04$)	2	1,3	66.7%	0		0%
DG ($\epsilon = 0,045$)	2	1,3	66.7%	0		0%
DG ($\epsilon = 0,05$)	1	1	33.3%	0		0%

Table D.4: Accuracy evaluation. Congestion pattern 4. 3 labeled queues and 4 labeled shockwaves.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	0		0%	0		0%
CC (c. class 3)	0		0%	0		0%
CC (c. class 5)	1	1	100%	1	1	50%
CC (c. class 7)	1	1	100%	1	1	50%
CC (c. class 9)	0		0%	2	1,2	100%
DG ($\epsilon = 0,025$)	0		0%	1	1	50%
DG ($\epsilon = 0,03$)	0		0%	2	1,2	100%
DG ($\epsilon = 0,035$)	1	1	100%	1	1	50%
DG ($\epsilon = 0,04$)	1	1	100%	0		0%
DG ($\epsilon = 0,045$)	1	1	100%	0		0%
DG ($\epsilon = 0,05$)	0		0%	0		0%

Table D.5: Accuracy evaluation. Congestion pattern 5. 1 labeled queue and 2 labeled shockwaves.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	0		0%	0		0%
CC (c. class 3)	0		0%	0		0%
CC (c. class 5)	2	1,2	66.7%	2	1,2	100%
CC (c. class 7)	2	1,2	66.7%	2	1,2	100%
CC (c. class 9)	1	1	33.3%	2	1,2	100%
DG ($\epsilon = 0,025$)	0		0%	2	1,2	100%
DG ($\epsilon = 0,03$)	1	2	33.3%	2	1,2	100%
DG ($\epsilon = 0,035$)	2	1,2	66.7%	2	1,2	100%
DG ($\epsilon = 0,04$)	3	1,2,3	100%	2	1,2	100%
DG ($\epsilon = 0,045$)	2	1,3	66.7%	0		0%
DG ($\epsilon = 0,05$)	2	1,3	66.7%	0		0%

Table D.6: Accuracy evaluation. Congestion pattern 6. 3 labeled queues and 2 labeled shockwaves.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	0		0%			
CC (c. class 3)	1	1	100%			
CC (c. class 5)	1	1	100%			
CC (c. class 7)	1	1	100%			
CC (c. class 9)	1	1	100%			
DG ($\epsilon = 0,025$)	1	1	100%			
DG ($\epsilon = 0,03$)	1	1	100%			
DG ($\epsilon = 0,035$)	1	1	100%			
DG ($\epsilon = 0,04$)	1	1	100%			
DG ($\epsilon = 0,045$)	1	1	100%			
DG ($\epsilon = 0,05$)	1	1	100%			

Table D.7: Accuracy evaluation. Congestion pattern 7. 1 labeled queue and 0 labeled shockwaves.

	Found queues	**Queue IDs**	**Queue recall**	**Found s.waves**	**S.wave IDs**	**S.wave recall**
CC (c. class 1)	1	1	25%	0		0%
CC (c. class 3)	2	1,4	50%	1	1	33.3%
CC (c. class 5)	4	1,2,3,4	100%	2	1,2	66.7%
CC (c. class 7)	4	1,2,3,4	100%	2	1,2	66.7%
CC (c. class 9)	3	1,2,3	75%	3	1,2,3	100%
DG ($\epsilon = 0,025$)	2	1,4	50%	3	1,2,3	100%
DG ($\epsilon = 0,03$)	2	1,4	50%	2	1,2	66.7%
DG ($\epsilon = 0,035$)	3	1,2,4	75%	1	1	33.3%
DG ($\epsilon = 0,04$)	4	1,2,3,4	100%	1	1	33.3%
DG ($\epsilon = 0,045$)	3	1,3,4	75%	1	1	33.3%
DG ($\epsilon = 0,05$)	1	3	25%	0		0%

Table D.8: Accuracy evaluation. Congestion pattern 8. 4 labeled queues and 3 labeled shockwaves.

Bibliography

[1] M. Barth and K. Boriboonsomsin, “Real-World CO2 Impacts of Traffic Congestion,” *Transportation Research Record: Journal of the Transportation Research Board*, p. 24, 2008.

[2] D. Stokols, R. W. Novaco, J. Stokols, and J. Campbell, “Traffic congestion, Type A behavior, and stress.” *Journal of Applied Psychology*, vol. 63, no. 4, pp. 467–480, 1978. doi: 10.1037/0021-9010.63.4.467. [Online]. Available: http://doi.apa.org/getdoi.cfm?doi=10.1037/0021-9010.63.4.467 [Accessed: 2018-08-23]

[3] J. Currie and R. Walker, “Traffic Congestion and Infant Health: Evidence from E-ZPass,” *American Economic Journal: Applied Economics*, vol. 3, no. 1, pp. 65–90, Jan. 2011. doi: 10.1257/app.3.1.65. [Online]. Available: http://pubs.aeaweb.org/doi/10.1257/app.3.1.65 [Accessed: 2018-08-23]

[4] P. Goodwin, “The Economic Costs of Road Traffic Congestion,” 2004.

[5] “Vehicle- and Infrastructure-Based Technology for the Prevention of Rear-End Collisions,” p. 59. [Online]. Available: https://www.ntsb.gov/safety/safety-studies/Pages/SIR0101.aspx [Accessed: 2018-09-17]

[6] “The bright side of sitting in traffic: Crowdsourcing road congestion data.” [Online]. Available: https://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html [Accessed: 2018-08-04]

[7] E. D’Andrea and F. Marcelloni, “Detection of traffic congestion and incidents from GPS trace analysis,” *Expert Systems with Applications*, vol. 73, pp. 43–56, May 2017. doi: 10.1016/j.eswa.2016.12.018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0957417416306935 [Accessed: 2018-09-13]

[8] B. Anbaroglu, B. Heydecker, and T. Cheng, “Spatio-temporal clustering for non-recurrent traffic congestion detection on urban road networks,” *Transportation Research Part C: Emerging Technologies*, vol. 48, pp. 47–65, Nov. 2014. doi: 10.1016/j.trc.2014.08.002. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0968090X14002186 [Accessed: 2018-08-05]

[9] B. Coifman, “Identifying the onset of congestion rapidly with existing traffic detectors,” *Transportation Research Part A: Policy and Practice*, vol. 37, no. 3, pp. 277–291, Mar. 2003. doi: 10.1016/S0965-8564(02)00016-2. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0965856402000162 [Accessed: 2018-08-04]

[10] Y. Liu, W.-B. Zhang, Z.-L. Wang, and C.-Y. Chan, “DSRC-based end of queue warning system.” IEEE, Jun. 2017. doi: 10.1109/IVS.2017.7995844. ISBN 978-1-5090-4804-5 pp. 993–998. [Online]. Available: http://ieeexplore.ieee.org/document/7995844/ [Accessed: 2018-07-11]

[11] N. Chintalacheruvu and V. Muthukumar, “Video Based Vehicle Detection and its Application in Intelligent Transportation Systems,” *Journal of Transportation Technologies*, vol. 02, no. 04, pp. 305–314, 2012. doi: 10.4236/jtts.2012.24033. [Online]. Available: http://www.scirp.org/journal/doi.aspx?DOI=10.4236/jtts.2012.24033 [Accessed: 2018-09-13]

[12] J. Leitloff, S. Hinz, and U. Stilla, “Vehicle queue detection in satellite images of urban areas,” in *3rd International Symposium Remote Sensing and Data Fusion Over Urban Areas*, vol. 36, no. Part 8, 2005, p. W27.

[13] B. Barbagli, I. Magrini, G. Manes, A. Manes, G. Langer, and M. Bacchi, “A Distributed Sensor Network for Real-Time Acoustic Traffic Monitoring and Early Queue Detection,” in *2010 Fourth International Conference on Sensor Technologies and Applications*. Venice, Italy: IEEE, Jul. 2010. doi: 10.1109/SENSORCOMM.2010.102. ISBN 978-1-4244-7538-4 pp. 173–178. [Online]. Available: http://ieeexplore.ieee.org/document/5558049/ [Accessed: 2018-09-13]

[14] R. Smith, “Directive 2010/41/EU of the European Parliament and of the Council of 7 July 2010,” in *Core EU Legislation*. London: Macmillan Education UK, 2015, pp. 352–355. ISBN 978-1-137-54501-5 978-1-137-54482-7. [Online]. Available: http://link.springer.com/10.1007/978-1-137-54482-7_33 [Accessed: 2018-08-24]

[15] “Apache Spark - Unified Analytics Engine for Big Data.” [Online]. Available: https://spark.apache.org/ [Accessed: 2018-09-15]

[16] “Hops Hadoop - Data on Tap.” [Online]. Available: https://www.hops.io/ [Accessed: 2018-09-15]

[17] A. Håkansson, “Portal of Research Methods and Methodologies for Research Projects and Degree Projects,” in *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July*. CSREA Press USA, 2013, pp. 67–73.

[18] L. Elefteriadou, *An introduction to traffic flow theory*, ser. Springer optimization and its applications. New York: Springer, 2014, no. volume 84. ISBN 978-1-4614-8434-9 OCLC: ocn857109800.

[19] B. S. Kerner, *The physics of traffic: empirical freeway pattern features, engineering applications, and theory.* Berlin: Springer, 2010. ISBN 978-3-642-05850-9 OCLC: 795910643.

[20] Y. Sugiyama, M. Fukui, M. Kikuchi, K. Hasebe, A. Nakayama, K. Nishinari, S.-i. Tadaki, and S. Yukawa, "Traffic jams without bottlenecks - experimental evidence for the physical mechanism of the formation of a jam," *New Journal of Physics*, vol. 10, no. 3, p. 033001, Mar. 2008. doi: 10.1088/1367-2630/10/3/033001. [Online]. Available: http://stacks.iop.org/1367-2630/10/i=3/a=033001?key=crossref.d9c6328c540467e5d2beaf6961d03278 [Accessed: 2018-07-17]

[21] H. Rehborn and J. Palmer, "ASDA/FOTO based on Kerner's three-phase traffic theory in North Rhine-Westphalia and its integration into vehicles." IEEE, Jun. 2008. doi: 10.1109/IVS.2008.4621192. ISBN 978-1-4244-2568-6 pp. 186–191. [Online]. Available: http://ieeexplore.ieee.org/document/4621192/ [Accessed: 2018-07-17]

[22] *Highway Capacity Manual: Volume 1: Concepts.* Washington: Transportation research board, 2010. ISBN 978-0-309-16077-3 978-0-309-16078-0 OCLC: 780843453.

[23] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002. doi: 10.1073/pnas.122653799. [Online]. Available: http://www.pnas.org/cgi/doi/10.1073/pnas.122653799 [Accessed: 2018-07-22]

[24] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, Feb. 2010. doi: 10.1016/j.physrep.2009.11.002 ArXiv: 0906.0612. [Online]. Available: http://arxiv.org/abs/0906.0612 [Accessed: 2018-07-22]

[25] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, Feb. 2004. doi: 10.1103/PhysRevE.69.026113. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.69.026113 [Accessed: 2018-07-26]

[26] M. E. J. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, Jun. 2006. doi: 10.1073/pnas.0601602103 ArXiv: physics/0602124. [Online]. Available: http://arxiv.org/abs/physics/0602124 [Accessed: 2018-07-28]

[27] W. Li and D. Schuurmans, "Modular Community Detection in Networks," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 1, 2011, p. 1366.

[28] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, Oct. 2008. doi: 10.1088/1742-5468/2008/10/P10008 ArXiv: 0803.0476. [Online]. Available: http://arxiv.org/abs/0803.0476 [Accessed: 2018-07-28]

[29] H.-P. Kriegel, P. Kröger, J. Sander, and A. Zimek, "Density-based clustering," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 231–240, May 2011. doi: 10.1002/widm.30. [Online]. Available: http://doi.wiley.com/10.1002/widm.30 [Accessed: 2018-07-30]

[30] T. Falkowski, A. Barth, and M. Spiliopoulou, "DENGRAPH: A Density-based Community Detection Algorithm." IEEE, Nov. 2007. doi: 10.1109/WI.2007.74. ISBN 978-0-7695-3026-0 pp. 112–115. [Online]. Available: http://ieeexplore.ieee.org/document/4427076/ [Accessed: 2018-07-28]

[31] ——, "Studying community dynamics with an incremental graph mining algorithm," *AMCIS 2008 Proceedings*, p. 29, 2008.

[32] T. Falkowski, "Community analysis in dynamic social networks," Ph.D. dissertation, Fakultät für Informatik der Otto-von-Guericke-Universität Magdeburg, 2009.

[33] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[34] "RDD Programming Guide - Spark 2.3.1 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/rdd-programming-guide.html [Accessed: 2018-08-01]

[35] "Spark SQL and DataFrames - Spark 2.3.1 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/sql-programming-guide.html [Accessed: 2018-08-01]

[36] "Cluster Mode Overview - Spark 2.3.1 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html [Accessed: 2018-08-01]

[37] "Spark Streaming - Spark 2.3.1 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html [Accessed: 2018-08-02]

[38] "Structured Streaming Programming Guide - Spark 2.3.1 Documentation." [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html [Accessed: 2018-08-02]

[39] "Apache Kafka." [Online]. Available: https://kafka.apache.org/ [Accessed: 2018-09-17]

[40] "Apache Kafka." [Online]. Available: https://kafka.apache.org/intro [Accessed: 2018-07-31]

[41] "Apache Kafka." [Online]. Available: https://kafka.apache.org/documentation [Accessed: 2018-07-31]

[42] "View places, traffic, terrain, biking, and transit - Computer - Google Maps Help." [Online]. Available: https://support.google.com/maps/answer/3092439 [Accessed: 2018-08-04]

[43] X. Li, J. Han, J.-G. Lee, and H. Gonzalez, "Traffic Density-Based Discovery of Hot Routes in Road Networks," in *Advances in Spatial and Temporal Databases*, D. Papadias, D. Zhang, and G. Kollios, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4605, pp. 441–459. ISBN 978-3-540-73539-7 978-3-540-73540-3. [Online]. Available: http://link.springer.com/10.1007/978-3-540-73540-3_25 [Accessed: 2018-08-05]

[44] C.-S. Chou and A. P. Nichols, "Deriving a surrogate safety measure for freeway incidents based on predicted end-of-queue properties," *IET Intelligent Transport Systems*, vol. 9, no. 1, pp. 22–29, Feb. 2015. doi: 10.1049/iet-its.2013.0199. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/iet-its.2013.0199 [Accessed: 2018-08-06]

[45] A. Khan, "Intelligent infrastructure-based queue-end warning system for avoiding rear impacts," *IET Intelligent Transport Systems*, vol. 1, no. 2, p. 138, 2007. doi: 10.1049/iet-its:20060086. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/iet-its_20060086 [Accessed: 2018-08-06]

[46] "Spark / graphX implementation of the distributed louvain modularity algorithm." [Online]. Available: https://github.com/Sotera/spark-distributed-louvain-modularity [Accessed: 2018-09-17]

[47] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "Graph Distances in the Data-Stream Model," *SIAM Journal on Computing*, vol. 38, no. 5, pp. 1709–1727, Jan. 2009. doi: 10.1137/070683155. [Online]. Available: http://epubs.siam.org/doi/10.1137/070683155 [Accessed: 2018-09-17]

[48] D. Chakrabarti, R. Kumar, and A. Tomkins, "Evolutionary clustering," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge*

Discovery and Data Mining, ser. KDD '06. New York, NY, USA: ACM, 2006. doi: 10.1145/1150402.1150467. ISBN 1-59593-339-5 pp. 554–560. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150467

[49] K. S. Xu, M. Kliger, and A. O. Hero III, "Adaptive Evolutionary Clustering," *Data Mining and Knowledge Discovery*, vol. 28, no. 2, pp. 304–336, Mar. 2014. doi: 10.1007/s10618-012-0302-x ArXiv: 1104.1990. [Online]. Available: http://arxiv.org/abs/1104.1990 [Accessed: 2018-08-24]

TRITA TRITA-EECS-EX-2018:652

www.kth.se