# Querying Top-k Dominant Traffic Flows on Large Urban Road Networks

Stella Maropaki*
Norwegian University of Science and Technology
stella.maropaki@ntnu.no

Paolo Sottovia
Huawei Munich Research Center
paolo.sottovia@huawei.com

Stefano Bortoli
Huawei Munich Research Center
stefano.bortoli@huawei.com

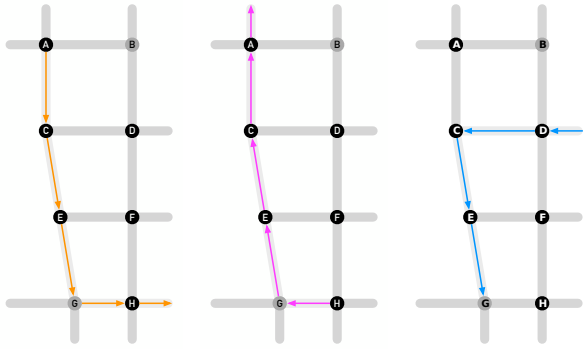
## ABSTRACT

With the ever-increasing urbanization, managing traffic and avoiding congestion becomes more and more challenging. Analysing the dynamics of vehicles is a crucial aspect of alleviating the problem of traffic congestion. One key aspect in assessing the traffic situation is to identify *dominant flows*, which are subject to high traffic volumes, and hence, are most prone to generate congestion. Real-world traffic data tracking vehicles in an urban network are proved to be extensive, subject to inconsistencies, and often detections are missing. These render many of the prior techniques inapplicable, highlighting the need for a novel robust and scalable technique. In this paper, we propose IST, an indexing technique for real-life traffic data, a scoring function for identifying the dominant flows and Flow-Scan, an algorithm for querying the dominant flows from the proposed index. Our experimental results demonstrate the efficiency and effectiveness of the presented method. Robustness and effectiveness were tested querying top-$k$ dominant flows on a real-life dataset. In addition, with synthetic data, we demonstrate that our method is scalable while comparing it to related existing methods.

## 1 INTRODUCTION

Predicting and preventing traffic congestion is important for multiple reasons, such as reducing the driving time from place to place, reducing pollution, reducing waste of fuel, and in general improving the efficiency of urban mobility. In order to prevent traffic congestion in urban roads, it is necessary to analyse the traffic dynamics and identify the areas where traffic jams are more likely to happen. Identifying these potentially problematic areas is the first step to pursue solutions alleviating the issue using traffic optimization techniques.

Dominant flows refer to the longest possible sequences of road segments where a significant number of vehicles travel in a given time window. They are crucial information to enable regional traffic optimization techniques and to coordinate traffic light plans in order to calibrate the capacity of roads. Detecting dominant flows enables smart traffic light plans optimization, allowing on the one hand to increase the throughput of vehicles crossing intersections in the directions affected by the dominant flows (traffic throttling), and on the other hand slow down traffic along with the dominant flows when road capacity cannot be further extended and backpressure must be applied to avoid saturation of road edges, i.e., congestions. Congestions can be interpreted as multiple overlapping flows that constrain each other and so creating traffic jams. In addition, the longer the dominant flows are the more areas they affect. Thus, detecting these flows allows adjusting the traffic light plans to avoid these traffic jams. Once the dominant flows are detected, dedicated analytics can be applied to them for further traffic and movement pattern analysis and for understanding macroscopic traffic trends. Additionally, knowing the dominant flows can be a powerful analytical tool for defining strategies for traffic management and control of traffic authorities.




(a) 03:00-04:00 (b) 08:00-09:00 (c) 13:00-14:00

**Figure 1: Example of dominant flows in an area during three different times of a day.**

As vehicles are used by people, it is not always straightforward to model their behaviour and moving patterns. For example, Figure 1 presents the dominant flows in an area at three different times of a day. As we can see, the flows are conflicting, especially between the one from 03:00-04:00 that goes from intersection $A$ to intersection $H$ and the one from 08:00-09:00 where it goes from intersection $H$ to intersection $A$. The most common traffic light systems allow traffic engineers to adjust traffic light plans during the day to accommodate different flows. Thus, existing adaptive traffic light systems seek dominant flow information during the day to avoid traffic jams and shorten commute time.

Our case study concerns an area in one of the largest Chinese metropolis where installed cameras allow to monitor traffic at intersections. Automated video processing allows detecting trajectories of vehicles across intersections, while detecting and tracking their plate number. Summaries of the information extracted from videos are made available for traffic optimization processes at 1 second frequency. The inherent complexity of real-life environment, video processing and software pipelines make the process prone to errors and failures that must be accounted for. This work is part of a solution combining cloud technologies and AI which strives for analysing and coordinating the traffic in the city to alleviate the traffic congestion.

Working with real-life data is a challenging task. Real-life data can be highly incomplete, with respect to missing values, and high noise. Our case study uses data from camera sensors that

*Work done during an internship at Huawei Munich Research Center.

cover only certain parts of the road network, leading to missing information. For example, in Figure 1, there are no camera sensors located in intersection $B$, so if multiple vehicles traveling from intersection $D$ to intersection $B$ create a dominant flow, this flow will be probably missed. Furthermore, camera sensors might suffer from various problems during their lifetime. Malfunction, bad visibility due to the weather or dirt may prevent precise detection of plate numbers. The aforementioned issues generate various types of inconsistencies in the detection data, and thus making the dominant flow identification more challenging.

The top-$k$ dominant flow extraction is not a new problem. Yet, the previous work [1–3, 6, 9] is considering only the frequency of the flows and not their length. However, if the length of the flows are ignored, single very frequent road segment flows would dominate the top-$k$ results and not allow the longer and more important flows to be identified and analysed. The PSSS method [3] suggests the use of a variant of a fundamental sequential pattern mining method [5, 8]. However, this technique is not applicable in our context due to our challenging real-life data where missing values in the detections are not rare. The method proposed in [3] is using the successor list of each sensor in order to find "hot routes", but in our setting where missing values are common and not all road edges are monitored, the successor lists will not achieve the desired outcome and end up giving wrong results.

This work is focused on identifying dominant flows in given time windows during the day, to support an adaptive traffic light system aimed at optimizing traffic by controlling road capacity and prevent congestion of vehicles on urban roads. In this paper, we aim to discover dominant flows occurring during ad-hoc time windows based on the frequency and length of the flow. The approach we are presenting in this work enables to overcome the issues that the previous methods have with the imperfect real-life data. Furthermore, we consider not only the frequency of a flow, but also its length. The main idea of this work is to use simple data structures to store all the vehicle detections from the camera sensors based both on their timestamps and their location. We adopt a combination of an inverted index and suffix trees to index only the necessary vehicle trip detections. From the index, we are then able to identify the most dominant flows very efficiently in multiple ad-hoc time windows.

**Contributions and Outline.** In summary, in this work we make the following contributions:

- We introduce a novel scoring function to determine the flows based on a weighted factor between their frequency and length.
- We propose a simple indexing technique for vehicle detections on road networks.
- We propose an efficient algorithm to identify dominant flows on multiple ad-hoc time windows from the proposed index.
- With thorough experiments, we demonstrate that our indexing and querying methods are efficient and viable solutions to our case study.

The rest of the paper is organised as follows. Section 2 reviews the related work, while in Section 3 the problem of finding the top-$k$ dominant flows is introduced. Then, Section 4 describes the indexing function for detecting the dominant flows. We evaluate our approach with the state-of-the-art approaches in Section 5. Finally, Section 6 concludes the paper with the final remarks and provides directions for future work.

## 2 RELATED WORK

A vehicle trip can be seen as an ordered sequence of detections where each detection represents a single itemset of the sequence. Thus, the identification of dominant flows can be abstracted, and therefore, can be related to the frequent sequential pattern mining topic. One of the early works on this topic is [10] in which the authors made a generalization of a sequential pattern and proposed the GSP algorithm to discover the frequent patterns. Later, two different methods, SPADE [12] and PrefixSpan [5, 8], were proposed that outperformed the GSP method. SPADE [12] was introduced as a method of frequent sequence mining using a lattice structure. PrefixSpan [5, 8] algorithm was later used in other works [3] for various applications like ours. We chose to use this method as our baseline, and not SPADE since this method was used in other works as well, even though none of them perfectly apply in our application, as described in the following Section 5.

An adaptation of PrefixSpan algorithm, the PSSS [3] method was proposed in order to mine hot routes on a road network using private vehicle Electronic Registration Identification data. The problem addressed in [3] can be considered similar to our work and we use the PSSS method as a baseline in our experiments. However, there are two main differences which make PSSS not applicable for our case. First, only the frequency of a route is considered; in our method we use a combination of the frequency and the length of a flow to define it as "hot" or dominant. Second, in PSSS method successor lists for each road node are introduced that determine the neighbouring road nodes where a following detection could happen. These successor lists are used to restrict the search of hot flows only on the neighbouring road edges. Although this makes PSSS method more efficient compared to the PrefixSpan, it cannot be applied in our case. In a real case scenario, we have missing data and gaps in the detections that prevent the successor list of each road node to benefit the identification of hot flows. For example, in Figure 1, the successor list of intersection $A$ includes intersections $B$ and $C$. If a dominant flow is from intersection $A$ to intersection $E$ through intersection $C$, and the camera on intersection $C$ is temporarily not giving any detections, this flow would be missed since the search from intersection $A$ will be stopped once there won't be any detections in intersection $C$. Furthermore, in our case study scenario, there are intersections that do not have detectors and thus by default there are detections missing. The adjacency property of the road network detections is also utilized in the GBM method [6], where it is assumed that the objects are moving in a grid space and each next move is happening to adjacent grid cells. Again, this property is not true for the real case scenarios with incomplete vehicle detection, that, as mentioned before, are due to factor like missing detectors, faulty hardware, poor weather conditions or network failure and the inherent complexity of real-time image processing.

Other works related to the frequent pattern on trajectories include [1, 2, 9] focus on detecting frequent patterns on trajectories that are based on GPS coordinates. These problems are incomparable to ours, with their biggest difference being their use of coordinates floating point data instead of fixed road network point data. Furthermore, only the frequency of the patterns is considered, in contrast to the combination of the length and the frequency that our work adopts.

## 3 PRELIMINARIES

Let $G = (U, E)$ be a directed unweighted graph that represents a road network. A node $u_i \in U$ represents a road intersection and a directed edge $e_i = (u_x, u_y) \in E$ represents a road segment that starts from intersection $u_x$ and ends in intersection $u_y$. In each edge of an intersection a detector is located, which gives information as a tuple in the form of $(c, t, e)$, where $c$ is the vehicle id or vehicle plate number, $t$ is the timestamp of the detection and $e$ is the directed road edge where the detection happened. In a real setting where each detector sends data every $1sec$, we can create sequences of detections for each vehicle, defined as vehicle trips.

*Definition 3.1 (Trip).* A vehicle trip $T$ is an ordered sequence of $o_i = (t_i, e_i)$, where $t_i$ is the timestamp and $e_i$ is the road edge of a detection that the vehicle moved on. To be considered part of the same trip, the time difference between two sequential entries must be less than a manually-defined time threshold $t_{min}$. Hence, we define a trip as $T = \{o_1, o_2, \ldots, o_{|T|}\}$, where $t_{i+1} - t_i \leq t_{min} \forall i, 1 \leq i \leq |T|$.

A flow $s = \{e_1, e_2, \ldots, e_{|s|}\}$ is an ordered sequence of road edges and its length is defined as the number of road edges that it contains, i.e. $len(s) = |e \in s|$. A trip $T$ contains a flow $s$ at time window $W = [t_a, t_b]$, denoted as $(s, W) \sqsubseteq T$ if and only if the trip $T$ has at least one index $i$ where the sequence of road edges starting at $i$ is the same sequence of ordered road edges during that time window $W$. More formally $\exists i, 1 \leq i \leq |T| : e_j = o_{i+j-1}.e \; \forall 1 \leq j \leq |s|$ and $[o_i.t, o_{i+|s|}.t] \subseteq W$ and $o_{i+j-1} \in T$ and $e_j \in s$. Given a time window $W$ and a set of trips $D_T$, we can define the frequency of a flow as the number of trips in which it is contained, i.e. $freq(s, W) = |T_i \sqsubseteq s|$.

In order to avoid considering not useful flows, we use minimum length, $l_{min}$, and minimum frequency, $f_{min}$, limits. If a flow has less frequency or less length than the minimum, we ignore it. In addition, we consider only maximal flows, where a flow is considered maximal if there are no sub-flows with the same frequency. A score can get assigned to a flow, based on its frequency and length. In this work the score is the following:

*Definition 3.2 (Flow Score).* Given a flow $s$, a time window $W$, a set of trips $D_T$, and a parameter $0 \leq \alpha \leq 1$, the score of the flow is the weighted combination of its frequency and length, i.e. $score(s, W, \alpha) = \alpha * freq(s, W) + (1 - \alpha) * len(s)$

Given a number $k \geq 1$, a set of trips $D_T$, a time window $W$ and a parameter $\alpha$, the $k-$Flows problem is to identify all the $k$ flows with the highest score.

## 4 METHOD FLOW-SCAN

The main idea of the proposed method is based on two observations. First, we need to identify the flows, or sub-trips, from the vehicle trips based on their frequency and length. For this purpose, we adopt a structure that utilizes a variant of the suffix tree [11]. Second, we don't need to maintain all the trips from the vehicles, but only the ones that are candidates for the results. In other words, we can discard the trips that have less than the required $l_{min}$. The trips with less than the required $f_{min}$ need to get retained, since their sub-trips could potentially have more than the required $f_{min}$.

**IST Index.** The main data structure of the method is an inverted index. The keys of the inverted index store all the sub-flows with $l_{min}$ that appear in the data. The values of the inverted index are suffix trees with all the sub-flows longer than $l_{min}$ having the key as prefix. The adopted suffix tree, is a compact version of the known suffix tree. Essentially, the nodes that have only one sub-tree are merged with the root of their sub-tree. In this way, the suffix tree has less nodes without loss of the necessary information. In addition, each node of the suffix tree has a pointer to a list of all the trip ids that contain the sub-flow of the node. Using this list, the frequency of the sub-flow can be easily determined.

| ID | Path | Vehicle | Timestamps |
|---|---|---|---|
| t1 | abcdef | A1 | 2020/03/20 07:12 – 2020/03/20 07:28 |
| t2 | abcg | A2 | 2020/03/20 07:25 – 2020/03/20 07:41 |
| t3 | hdebc | A3 | 2020/03/20 08:03 – 2020/03/20 08:23 |

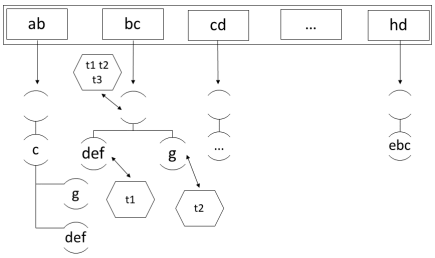



**Figure 2: IST index example using $l_{min} = 2$.**

An auxiliary data structure is used to store the information of all the trips; *trip id, vehicle id* that made the trip, and the starting and ending timestamps. This data structure is a simple list that is used as supplementary information of the main index. By scanning this list, the trips inside the time window can be identified.

*Example 4.1.* An example is shown in Figure 2 where the IST index is visualized for three vehicle trips. For this example, we use $l_{min} = 2$. All the sub-flows with length $= 2$ are indexed as keys in the IST, depicted as rectangles. Each key of the IST index is pointing to a suffix tree, depicted as circles, where all the sub-flows that have the certain keys as prefix are indexed. Furthermore, in the nodes of the suffix trees there are the pointers to the trip ids lists, depicted as hexagons. The IST index is not fully illustrated for presentation reasons. As seen in the figure the sub-flows *bcdef*, *bcg* and *bc* are indexed in the second key of the IST index *bc*. The suffix tree from that key includes the suffixes of the mentioned sub-flows *def* and *g*. The root of the tree is pointing to the list with the trip ids $\{t1, t2, t3\}$, since the sub-flow *bc* is contained in all the trips of our example. Accordingly, the other nodes of the suffix tree contain the respected trip ids in their id lists.

**Flow-Scan Query Method.** By indexing the trips using the IST data structure, we make sure that no unnecessary trips will become candidates for the dominant flows result and that no information of the trips is lost. In order to query the desired dominant flows, the Flow-Scan process of reading the data from the IST data structure is divided into four steps, shown in Algorithm 1. The first step is the identification of the trips that happen inside the time window $W$ (line 3). This step uses the auxiliary data structure where all the trips are stored is used. Once the trips inside the time window $W$ are retrieved, the keys from the IST inverted index are selected (line 4). This step is important in order to avoid traversing all the data in the IST data structure, but only the elements that could give candidate flows for the result. The third step is to retrieve all the maximal sub-flows that are longer than the $l_{min}$ and more frequent than the $f_{min}$ from

the suffix trees of the keys (lines 5-7). Lastly, the fourth step is to score the retrieved sub-flows using the scoring function from the Definition 3.2 (lines 8-9) and return the $k$ sub-flows with the highest scores.

**Algorithm 1** Flow-Scan Algorithm

```
1: function QUERY(IST index, query parameters α, k, W)
2:     Flows ← ∅
3:     D_{T,W} ← FindTripsInWindow(IST, W);
4:     KEYS ← FindIndexKeys(IST, D_{T,W});
5:     for each key ∈ KEYS do
6:         F ← from key.SuffixTree find all maximal flows with
       more than f_min
7:         add all F in Flows
8:     for each s ∈ Flows do
9:         score(s)
       return max_k[Flows]
```

## 5 EXPERIMENTS

In this section, we report the experimental evaluation of the proposed method, especially for what concerns efficiency and scalability. We first describe the experimental setup, the baseline methods and the datasets used for the evaluation. Then, we describe each experiment, report and discuss the results.

### 5.1 Environment and Setup

**Hardware and Implementation.** Our experiments were evaluated on a machine with 24 cores 3.40GHz Intel(R) Core(TM) i7-6700 CPUs, with 16GB memory, using Ubuntu 18.04.4 LTS, and all algorithms are implemented in Java 8.

**Compared Algorithms.** In our experiments, we use a Naive baseline algorithm to evaluate the efficiency of the Flow-Scan algorithm proposed in this work. The baseline method, Naive, utilizes the structure of the road network to store a list of detections in each road edge $e$ of the network. The list consists of tuples $(t, c, p)$ representing a detection of a vehicle $c$ at time $t$ at the given road edge $e$. The last, $p$, is a pointer to the next data point (detection) of the same vehicle $c$ within the same trip $T$, i.e. the two consequent detections occurred within $t_{min}$. In this way, the vehicle's trip can be recreated by traversing the vehicle's detections following the pointers $p$. During the query process, the algorithm scans all the road edges $e$ for detections that occur inside the time window $W$ and it constructs flows by traversing the detections using the pointers $p$. After identifying all flows, it calculates a score for each flow and returns the $k$ flows with the highest score. The Naive method can handle the missing data from the detections since it uses the road network only to store the detections and not to extract the flows from it, as opposed to the PSSS method [3]. In the Naive method, the search of the flows is happening by following the pointers of the tuples and not following the road network structure.

Additionally with the baseline Naive, we compare the proposed method Flow-Scan with the PrefixSpan method described in [5, 8] and with the PSSS method described in [3]. Since the PrefixSpan and the PSSS methods identify the dominant flows based only on their frequency, we use $\alpha = 1$ for the Naive and Flow-Scan methods in the experiments that all four algorithms are present. Furthermore, since the PrefixSpan and the PSSS methods were proposed for applications that didn't use incomplete data, we use these algorithms only in the experiments with the synthetic datasets as described in the following sections.

**Datasets.** For the evaluation of the aforementioned methods, we use four different datasets [4], two including real-life data and two synthetic data. The first dataset, RM, is a real-life dataset collected in a period of a month from traffic detectors in an area of a Chinese city. The detectors were located in 6 intersections monitoring the traffic on 44 road edges. Each detector was collecting data every 1 second for all the road edges in its radius. In total, 2894174 detections were collected, from 337089 different vehicles. The second dataset, RD, includes the data from RM for one of the days in that month. The data from that day include 116268 detections from 44643 vehicles.

As already mentioned, these real-life data are incomplete and the PSSS method [3] is not suitable for identifying the dominant flows over them. In order to be able to apply the PSSS and the PrefixSpan [5, 8] algorithms and to experiment with the scalability of the proposed algorithm Flow-Scan we use two datasets, with synthetic data. The synthetic data COM were generated in the same road network as the real-life data RM and RD, with the difference that we included detectors in the 2 intersections where the real scenario was missing. Then we generated equally random trips over the road network. In total we generated 18910 detections from 7500 vehicles. In order to evaluate the scalability of the methods in bigger road networks, we generated the synthetic data GRID using the SUMO Simulation of Urban Mobility [7]. We randomly generated trips on a $10x10$ intersections grid road network (having 100 intersections and 360 road edges) using a utility from SUMO that equally generates trips over the road network. Then, running the simulation and using TraCI Traffic Control Interface library[1] we read the simulation data and collect the detections. The simulation collected data include 1806141 detections from 135618 different vehicles.

### 5.2 Results and Discussion

Following, the experimental analysis is reported. We show the performance of the methods described based on their building and querying times, and discuss the effect that the various parameters have on the running time of the algorithms. The building time for each algorithm depends on the nature of the algorithm. For all of them the building time includes the reading of the data into memory. For Naive and Flow-Scan it also includes the time for building their respective index. The query time is the time used for each algorithm in order to retrieve the dominant flows from the data or the index. For all algorithms this time includes also the scoring of each flow. For PSSS algorithm the time for creating the successor list *only* for the road edges used in the dominant flows detection is measured in the query time. Note that the experimental results reported are the ones we believe are representative and not the exhaustive set of experiments performed.

**Dataset Size Scalability.** For testing the scalability, we run the four algorithms, PrefixSpan, PSSS, Naive, Flow-Scan, using the two synthetic datasets COM and GRID for different number of detections and vehicles. All algorithms were run with $t_{min} = 200$ sec, $\alpha = 1, k = 1$, $l_{min} = 2$ and $f_{min} = 2$.

Figure 3 shows the scalability of the building and query times for the four algorithms by changing the number of vehicles in the COM dataset. As expected, the more vehicles there are in

[1] https://sumo.dlr.de/docs/TraCI.html

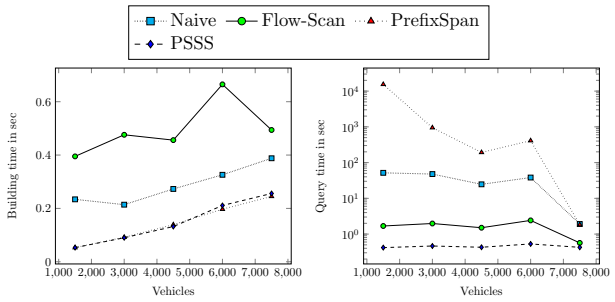



**Figure 3: Running times for the COM dataset changing the number of vehicles.**

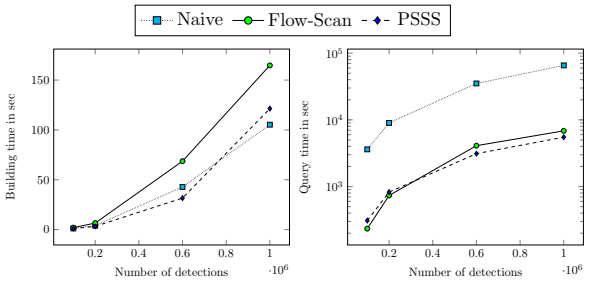
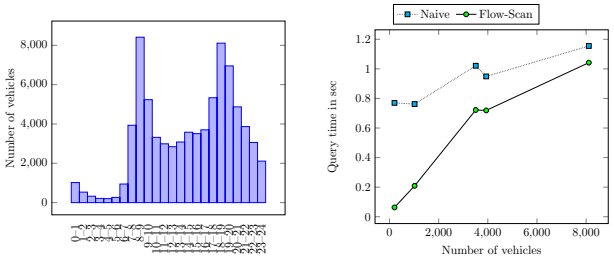



**Figure 4: Running times for the GRID dataset changing the number of detections.**




**(a) Histogram of number of vehicles per hour.**

**(b) Query time for changing time window $W$.**

**Figure 5: Time window analysis for the RD dataset.**

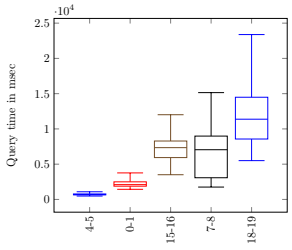



**Figure 6: Query time for Flow-Scan method using the RM dataset.**

the dataset, the more time the algorithms need for their building step. Method PrefixSpan and PSSS have the same performance on their building phase since they only read the data into memory in this step. On the other hand, the query time does not get affected by the number of the vehicles, excepting the PrefixSpan method that has to iterate multiple times through the vehicle trips to find the candidates for the result.

In Figure 4, the building and query times for the Naive, Flow-Scan and PSSS algorithms are shown by changing the number of detections in the GRID dataset. This dataset was too large for the PrefixSpan method and so we do not report running times for it. Similarly as the COM dataset, with the increase of the number of detections in the data, the building time of the methods is increased. For bigger scale of data, such as the GRID dataset, the building of the IST index is as efficient as the reading of the data into memory that the PSSS method is using, while in the query time the performance is also similar.

**Value of Time Window W.** The time window $W$ is the query parameter that affects the query time. It is used to restricts the dominant flows in the result, so that they are happening during the time window period. Intuitively, when the time window is in a rush hour period of the day, a significant number of vehicles will travel in the road edges, more flows will become candidates for the result and so, more time will be necessary for the query process. Figure 5a shows a histogram with the number of vehicles per hour for the RD dataset. The peak rush hours are 08:00-09:00 and 18:00-19:00 and the hours with the least traffic are 01:00-06:00.

We run the Naive and the Flow-Scan algorithms for five different time windows that had different number of vehicles. In this experiment we don't use the PrefixSpan and PSSS algorithms since we use the real-life RD dataset that is incomplete. The rest of the parameters were $t_{min} = 200$ sec, $\alpha = 0.5, k = 10, l_{min} = 1$ and $f_{min} = 1$. Figure 5b shows the query times in respect to the number of vehicles that were in each time window. We can see that with more vehicles in the time window, the query time increases. Furthermore, Naive algorithm is affected more by the number of vehicles in the time window, while Flow-Scan algorithm can handle better the increase.

In addition, for validating our method Flow-Scan over one month of data, we run the algorithm for the same time windows for all the days of a month using the RM dataset. The number of vehicles for each hour for all the different days of the month follow the same trend as in Figure 5a. Figure 6 shows the query times for each time window for 30 days. As seen in the figure, the different traffic conditions in the different days do not affect the time windows with small number of vehicles, like the time windows of 04:00-05:00 and 00:00-01:00. As the number of vehicles increases in the time window, the query time differences also increase.

**Value of Time Threshold $t_{min}$.** The time threshold affects the indexing time because the higher it is, the more detections will be connected into a single trip. Figure 7 shows the running times for the Naive and Flow-Scan algorithms, run for the RD dataset. Again, the PrefixSpan and PSSS algorithms are not present because of the incomplete real-life dataset. We chose as time window $W$ the period between 07:00-08:00 since this is the period with average number of vehicles. The rest of the parameters were $\alpha = 0.5, k = 10, l_{min} = 1$ and $f_{min} = 1$.

When the time threshold $t_{min}$ is increased, the building time for the Naive index is not affected compared to the building time for the IST index. When the $t_{min}$ is increased, as mentioned, more detections get connected into single trips and the trips grow in length. However, since the vehicle detections stay the same, we get less trips in total, and so the building of the IST index is decreased.

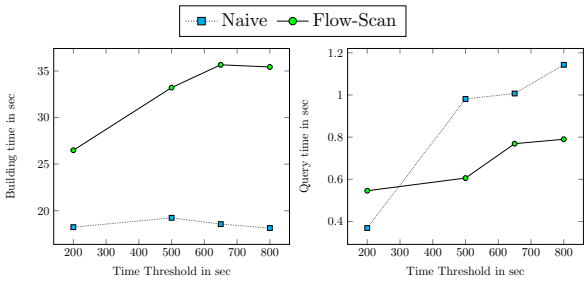



**Figure 7: Running times for the RD dataset changing the $t_{min}$ parameter.**

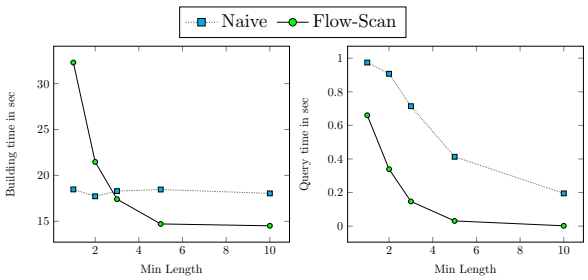



**Figure 8: Running times for the RD dataset changing the $l_{min}$ of the trips.**

On the contrary, the query time the Naive algorithm gets more affected by the increase of the $t_{min}$ since the trips become longer. The Naive query phase needs to iterate more times in the length of the trips than the Flow-Scan algorithm, and so the increase of the $t_{min}$ increases the query time of Naive.

**Value of Minimum Length $l_{min}$.** The $l_{min}$ of the trips controls the length of the flows that can be candidates as dominant flows and thus, indirectly can control how many trips can be considered from the dataset. If a flow is shorter than the $l_{min}$ then it is ignored in the result. In addition, the $l_{min}$ is the value that controls the length of the sub-flows stored as inverted index keys in the IST index. For evaluating how the value of the $l_{min}$ is affecting the Naive and the Flow-Scan methods, we run the algorithms for the RD dataset, with $t_{min} = 500$ sec, $\alpha = 0.5, k = 10$, $f_{min} = 1$ and time window $W$ = 07:00-08:00. As mentioned before, the PrefixSpan and PSSS algorithms are not included in this experiment because of the use of the real-life dataset.

Figure 8 shows the running times for the Naive and the Flow-Scan algorithms. When $l_{min} = 1$, all the flows and sub-flows are considered as candidates to be dominant, but when the $l_{min}$ increases, the number of flows longer than the threshold decrease, thus reducing the candidate flows. This shows in the query time for both Naive and Flow-Scan algorithms. For both algorithms as the $l_{min}$ is increasing, the query time is decreasing since less flows needs to be analysed. The IST index stores only the trips that have more than the $l_{min}$, and when the $l_{min}$ is increasing, the index building time is decreasing. The Naive algorithm has more stable building time with the change of the $l_{min}$ and that is because it does not check for $l_{min}$ of the trips before indexing them. However, Naive needs more time for querying than the Flow-Scan, approximately 30% more time, since it keeps all trips and not only the ones longer than $l_{min}$.

**Values of k, $\alpha$ and $f_{min}$.** The parameters $k$, $\alpha$ and $f_{min}$ affect which of the flows will be detected as the result dominant flows, but do not affect the running time of the algorithms. The parameter $k$ does not affect the algorithms' running times, since the algorithms do not use any pruning or early stopping condition. Similarly, changing $\alpha$ and $f_{min}$ affect only the scoring function of the flows and not the query process of the algorithms.

## 6 CONCLUSIONS

In this paper, we focused on the real-life scenario of identifying dominant traffic flows, which is crucial for traffic optimization techniques, such as avoiding traffic congestion on urban road networks. We introduced a scoring function that ranks the flows, proposed a simple data structure called IST, and proposed the Flow-Scan algorithm to identify the highest-ranking flows by utilizing the proposed IST index. Our experimental evaluation shows that the Flow-Scan method is efficient and scalable. It is equally efficient on the synthetic datasets with the state-of-the-art PSSS method while at the same time our method enables to overcome the limitations of the PSSS method that does not consider the length of a flow as an important factor and is incapable to handle missing values of the real-life data. Furthermore, although the Flow-Scan method uses more time to build the IST index compared to the Naive method, it outperforms the baseline in the query time and thus, it is a viable solutions for the large scale real-life urban road network scenario studied in this paper.

## ACKNOWLEDGMENTS

The authors would like to thank the Lamport team of the Huawei Munich Research Center members for their insightful comments. In addition, they would like to thank Daniele Foroni for his valuable help regarding the datasets and his constructive feedback.

## REFERENCES

[1] Huiping Cao, Nikos Mamoulis, and David W Cheung. 2005. Mining frequent spatio-temporal sequential patterns. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 8–pp.

[2] Lu Chen, Yunjun Gao, Ziquan Fang, Xiaoye Miao, Christian S Jensen, and Chenjuan Guo. 2019. Real-time distributed co-movement pattern detection on streaming trajectories. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1208–1220.

[3] Chen Cui, Linjiang Zheng, and Dihua Sun. 2019. Mining Private Vehicle Hot Routes Using Electronic Registration Identification Data. In *Proceedings of the 2019 International Conference on Big Data Engineering*. ACM, 51–56.

[4] Daniele Foroni, Paolo Sottovia, Stella Maropaki, and Stefano Bortoli. 2021. *Traffic Detection Datasets.* https://doi.org/10.5281/zenodo.4471357

[5] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*. Citeseer, 215–224.

[6] Anthony J. T. Lee, Yi-An Chen, and Weng-Chong Ip. 2009. Mining Frequent Trajectory Patterns in Spatial-Temporal Databases. *Information Sciences* 179, 13 (2009), 2218–2231.

[7] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie WieBner. 2018. Microscopic traffic simulation using sumo. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2575–2582.

[8] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. 2004. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on knowledge and data engineering* 16, 11 (2004), 1424–1440.

[9] Dimitris Sacharidis, Kostas Patroumpas, Manolis Terrovitis, Verena Kantere, Michalis Potamias, Kyriakos Mouratidis, and Timos Sellis. 2008. On-line discovery of hot motion paths. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 392–403.

[10] Ramakrishnan Srikant and Rakesh Agrawal. 1996. Mining sequential patterns: Generalizations and performance improvements. In *International Conference on Extending Database Technology*. Springer, 1–17.

[11] Peter Weiner. 1973. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE, 1–11.

[12] Mohammed J Zaki. 2001. SPADE: An efficient algorithm for mining frequent sequences. *Machine learning* 42, 1-2 (2001), 31–60.