Capturing the Spatiotemporal Evolution in Road Traffic Networks

Tarique Anwar, Chengfei Liu, *Member, IEEE*, Hai L. Vu, *Senior Member, IEEE*, Md. Saiful Islam, *Member, IEEE*, and Timos Sellis, *Fellow, IEEE*

Abstract—The urban road networks undergo frequent traffic congestions during the peak hours and around the city center. Capturing the spatiotemporal evolution of the congestion scenario in real-time in an urban-scale can aid in developing smart traffic management systems, and guiding commuters in making informed decision about route choice. The congestion scenario is often represented by a set of distinguishable network partitions that have a homogeneous level of congestion inside them but are heterogeneous to others. Due to the dynamic nature of traffic, these partitions evolve with time in terms of their structure and location. In this paper, we propose a comprehensive framework to capture the evolution by incrementally updating the partitions in an efficient manner using a two-layer approach. The physical layer maintains a set of small-sized road network building blocks in a fine granularity, and performs low-level computations to incrementally update them, whereas the logical layer performs high-level computations in order to serve as an interface to query the physical layer about the congested partitions in a coarse granularity. We also propose an in-memory index called `Bin` that compactly stores the historical sets of building blocks in the main memory with no information loss, and facilitates their efficient retrieval. Our experimental results show that the proposed method is much efficient than the existing re-partitioning methods without significant sacrifice in accuracy. The proposed `Bin` consumes a minimum space with least redundancy at different time stamps.

Index Terms—Road network motifs, Incremental partitioning, evolution of road traffic, urban road traffic networks

1 Introduction

Generally the roads of different localities or suburbs in a city experience specific traffic flow patterns based on their *spatiotemporal* significance. In the spatial perspective, roads inside the city center or an area having popular venues like a stadium or hospital, usually remain more congested than others without such significance. The different small sub-networks (small areas like suburbs) of a large urban road network experience distinctive traffic flow depending on the significance of venues inside them. Previous works have applied the partitioning of urban road networks based on their traffic level to identify such sub-networks called *spatial partitions* or simply *partitions* [1], [2]. On the other hand, in the temporal perspective, the roads usually remain busier during the peak times than the off-peak times. It reflects the dynamic nature of congestion in the spatial partitions. These spatiotemporal behaviors altogether lead to the evolution of traffic congestion. For example, during the morning office-opening hours, the congestion generally starts developing in the outer suburbs and the roads connecting the city center, mostly occurs inside the city during the day, and starts moving outwards again during the office-closing hours. The congestion can be simply understood and represented as a congested partition that keeps on changing its structure, location, and level of congestion. The continuous maintenance, tracking, and capturing of the differently congested partitions in an incremental approach has two applications [3], [4]. First, it can potentially aid smart traffic management systems for a homogeneous distribution of traffic among different partitions, and second, it can be used in journey planning applications to provide information-rich visualization of the evolving traffic scenario and guide commuters in making informed decisions. The existing research on the evolution of road traffic networks deals with several problems, including prediction of future traffic condition [5], [6] and causal discovery of anomalies and structures [7], [8], but a partition-based evolution of road traffic networks has not been investigated. The works dealing with road network partitioning consider only static partitioning [1], [2], [4].

The `Google Traffic` feature of `Google Maps` visualizes real-time road traffic, based on anonymously collected data. It simply presents a heatmap of the actual traffic on the corresponding roads. Fig. 1 shows a snapshot of a typical Monday 11:00 AM. The commuters can plan their journey using this visual information. As the congestion forms and dissolves via the linked road segments, the level of congestion on a road segment is dependent on the preceding and following segments. Instead of naively considering the congestion level of individual road segments, they can be effectively grouped in the form of differently congested partitions (including both congested and non-congested ones), and visualized to show the real-time congestion scenario. It would further enrich the visual information about the congestion spread and connectivity of the

- *T. Anwar is with the Indian Institute of Technology Ropar, Rupnagar 140001, India. E-mail: tarique@iitrpr.ac.in.*
- *C. Liu and T. Sellis are with the Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: {cliu, tsellis}@swin.edu.au.*
- *H.L. Vu is with the Institute of Transport Studies, Monash University, Clayton, VIC 3800, Australia. E-mail: hai.vu@monash.edu.*
- *M.S. Islam is with Griffith University, Nathan, QLD 4111, Australia. E-mail: saiful.islam@griffith.edu.au.*

Manuscript received 22 Jan. 2017; revised 27 Dec. 2017; accepted 2 Jan. 2018. Date of publication 18 Jan. 2018; date of current version 5 July 2018.
(Corresponding author: Tarique Anwar.)
Recommended for acceptance by G. Yu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2018.2795001

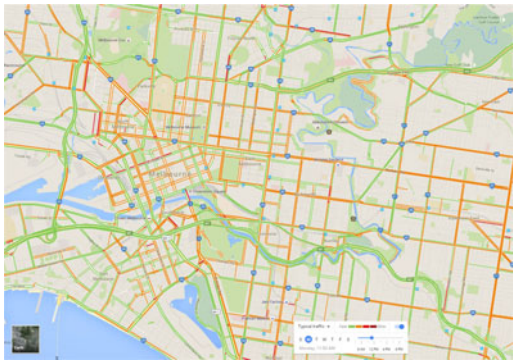


Fig. 1. Google Traffic visualization at 11:00 AM (typical, Monday).

different individual congestions [9]. Some other applications of the maintenance of urban road network partitions are partition-based route-guidance, trip planning, recovery of missing traffic data, and other complex graph processing methods for traffic-aware smart travel services [1], [2], [10].

The naive way to track the change in partitions is to perform spatial partitioning of the urban road network at each time stamp based on the traffic measures, and analyze the change. But a complete re-partitioning is a computationally expensive task and may even require more time than the time-interval of data collection. On the other hand, grid-based indexing of road networks fail to distinguish the two oppositely directed traffic flows of the same road, which may experience different traffic load at the same time. Generally in successive time stamps, the traffic does not change abruptly. A logically better and efficient way is to incrementally update the previously obtained set of partitions by processing only the sections of probable change. It significantly reduces the computations, while may sacrifice the quality of partitions marginally. There exist works on the complete partitioning of urban road networks [1], [2], [4], but the problem of incremental maintenance of their partitions is still unexplored. The main challenges in this problem are two folds. First, the computations need to be efficient enough to complete the incremental update before the arrival of data from the next time stamp. Second, there needs to be a mechanism to economically store the historical information in primary or secondary memory, which provides its efficient retrieval.

In our recent preliminary work [11], we developed a two-layer method to track the evolution of congestion by incrementally updating a set of urban road network partitions. The physical layer performs low-level computations for the incremental update, whereas the logical layer presents those obtained results to the user after a light makeover. Our method in the physical layer starts with a set of *building blocks* (defined in Section 3.3) of the urban road network at the beginning time stamp. During the period of evolution, the unstable road segments are identified at each time stamp, indexed as a heap tree, and moved to their most suitable building blocks.We extend this work in the current paper, and develop a comprehensive framework to capture the spatiotemporal evolution of the traffic scenario. We model the building blocks in the form of a congestion evolution graph and propose an in-memory index called Bin to compactly store the historical sets of building blocks. Bin is referenced and updated by the physical layer during the incremental updates. The logical layer accesses Bin to efficiently retrieve and present the congestion evolution graph, and support queries related to congested partitions.

In summary, we make the following main contributions in this paper. First, we model the urban road network congestion in the form of a *congestion evolution graph* to effectively capture its spatiotemporal evolution (Section 3.3). Second, we develop an in-memory index for the building blocks (referred as Bin) to compactly store the historical information in the main memory. It facilitates efficient retrieval, visualization, and understanding of the evolution using the *congestion evolution graph* (Section 4.1.1). Third, we adapt the two-layer method for dynamic incremental maintenance of the differently congested partitions (proposed in our previous preliminary work [11]) with respect to Bin, and develop a comprehensive framework. The method incrementally updates the *building blocks* at each new time stamp in Bin. The *stability measure*, used to identify the unstable road segments, and the concepts of *road network motifs* used to understand the grouping patterns, are the two main highlights, which set the foundation of our method (Sections 4.2, 4.3, and 4.4). Lastly, we perform extensive experiments on both real and synthetic data to demonstrate the efficacy of our framework (Section 5).

The rest of the paper is organized as follows. We start with some preliminaries and problem definition in Section 2. Section 3 presents a high level description of the our two-layer approach, followed by the proposed method in Section 4. The experimental results are presented in Section 5, followed by some related works in Section 6. We conclude the paper in Section 7.

2 Preliminaries

2.1 Urban Road Networks

Urban roads exist in the form of a directed physical network, defined in Definition 1, spatially spread over a large urban area. To simplify this network for computational purpose, we give it a representation of an undirected graph in Definition 2 (please refer [2] for complete details).

Definition 1 (Road Network). *An urban road network is defined as $\mathcal{N} = (\mathcal{I}, \mathcal{R})$ comprising a set of intersection points $\mathcal{I} = \{\iota_1, \iota_2, \ldots, \iota_{n_\iota}\}$ as nodes that are connected among themselves by directed road segments $\mathcal{R} = \{r_1, r_2, \ldots, r_{n_r}\}$ as links, where each road segment r_i associates a measure of traffic density $r_i.d$ with itself.*

Definition 2 (Road Graph). *Given an urban road network $\mathcal{N}$, the corresponding road graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is constructed by representing each road segment $r_i \in \mathcal{N}$ as a node v_i, and establishing an undirected link e_i between each possible node pair (v_j, v_k) if there exists at least one intersection point ι_l which is a common intersection for the roads r_j and r_k, and the traffic can flow either from r_j to r_k or vice versa. Each node $v_i(node(r_i)) \in \mathcal{V}$ associates with it a feature value $v_i.f$, which is the road traffic density $r_i.d$.*

Definition 3 (Partition). *An urban road network $\mathcal{N}$ can be partitioned into multiple segments, each called a partition $\mathcal{P}_i$ of $\mathcal{N}$. All the different segments form a set of partitions $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k\}$, such that i) $\bigcup_{i=1}^{k} \mathcal{P}_i = \mathcal{R}$ and $\mathcal{P}_i \bigcap \mathcal{P}_j = \emptyset$ for all $i \neq j$, and ii) each $\mathcal{P}_i$ is connected inside and all adjacency relations, except the cross-partition relations (inter-partition links), are maintained as in $\mathcal{N}$.*

A partition of an urban road network can be transformed to that of a road graph by following Definition 2, and vice versa. Most of the urban roads exist as two-way roads,

which are divided into two parts from the middle for traffic of the two opposite directions. These two parts undergo different kinds of traffic flow patterns. For example, on a road that connects outskirts with the city center, the morning office hours would find more traffic heading towards the city center, whereas the opposite would be seen in the evening hours. This is accommodated in the road graph by considering the two directions as separate adjacent road segments that share common intersection points.

Traffic congestion broadly refers to the traffic condition on a road segment, which is identified by different traffic flow measures. It does not have a uniform or standard definition [12]. We consider the traffic density (Definition 2) as a measure to characterize the *level of congestion*. It is a spatial measure of the traffic on a particular road segment, calculated as number of vehicles per meter. Thus, a high traffic density value indicates a high level of congestion.

Definition 4 (Inter-Partition Associativity). *For a given partition $\mathcal{P}_i$, the inter-partition associativity is defined as the normalized summation of the level of congestion similarity between all possible linked pairs (r_a, r_b) such that $r_a \in \mathcal{P}_i$ and $r_b \in \overline{\mathcal{P}_i}$. $\frac{W(\mathcal{P}_i^{j-1}, \overline{\mathcal{P}_i^{j-1}})}{|\mathcal{P}_i^{j-1}|}$ denotes the inter-partition associativity of $\mathcal{P}_i$ at time stamp t_{j-1}.*

Definition 5 (Intra-Partition Associativity). *For a given partition $\mathcal{P}_i$, intra-partition associativity is defined as the normalized summation of the level of congestion similarity between all possible linked pairs (r_a, r_b) such that $\{r_a, r_b\} \in \mathcal{P}_i$. $\frac{W(\mathcal{P}_i^{j-1}, \mathcal{P}_i^{j-1})}{|\mathcal{P}_i^{j-1}|}$ denotes the intra-partition associativity of $\mathcal{P}_i$ at time stamp t_{j-1}.*

2.2 Problem Definition

Given an urban road network $\mathcal{N} = (\mathcal{I}, \mathcal{R})$ and its road graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the problem addressed in this paper is to incrementally update the road network partitions with the aim to capture the evolution of traffic congestion in a dynamic environment. Let us suppose, we are given a set of road network partitions $\mathcal{P}^{j-1} = \{\mathcal{P}_1^{j-1}, \mathcal{P}_2^{j-1}, \ldots, \mathcal{P}_k^{j-1}\}$ based on the traffic at time stamp t_{j-1}, such that $\mathcal{P}^{j-1}$ is $\min_{\mathcal{P}^{j-1}} \sum_{i=1}^{k} (\alpha \times \frac{W(\mathcal{P}_i^{j-1}, \overline{\mathcal{P}_i^{j-1}})}{|\mathcal{P}_i^{j-1}|} - (1-\alpha) \times \frac{W(\mathcal{P}_i^{j-1}, \mathcal{P}_i^{j-1})}{|\mathcal{P}_i^{j-1}|})$ for a suitable α, as in [1]. In this manner, $\mathcal{P}^{j-1}$ has a minimum inter-partition associativity and a maximum intra-partition associativity.

i) The first objective is to dynamically update $\mathcal{P}^{i-1}$ to $\mathcal{P}^i = \{\mathcal{P}_1^i, \mathcal{P}_2^i, \ldots, \mathcal{P}_{k'}^i\}$ at each new time stamp t_i based on the respective traffic data in an incremental manner without re-partitioning the whole network, in such a way that the properties of inter-partition and intra-partition associativities are maintained. The number of updated partitions k' may not be equal to k.

ii) The second objective is to develop an in-memory indexing scheme for a compact storage of the incrementally obtained historical sets of partitions $\mathcal{P}^0, \mathcal{P}^1, \ldots, \mathcal{P}^j$ and facilitate their efficient retrieval.

3 Proposed Method

The proposed method dynamically tracks the evolution of traffic congestion in real-time. Instead of incrementally maintaining the partitions directly, we embed the functionalities in two different layers. The *logical layer* gets the query to identify the congested partitions at a time stamp from the user, passes it to the physical layer, lightly processes the returned data, and returns the results to the user, whereas the *physical layer* efficiently maintains a large number of evolving building blocks using an index structure.

3.1 Logical Layer

From the user end, the logical layer provides the service to get the congested/non-congested partitions at any point of time. It is based on a set of so-called building blocks that are maintained up-to-date by the physical layer. After getting a query from the user, this layer transforms the granularity of the query from partitions to the building blocks, and passes to to the physical layer. For example, a query to fetch k *differently congested partitions* of the network at the current time is transformed to fetch the building blocks of the current time. The physical layer returns the result as $\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{n_b}\}$. The logical layer constructs a building block graph $\mathcal{G}^b = (\mathcal{V}^b, \mathcal{E}^b)$, where each building block forms a node $\varsigma_i \in \mathcal{V}^b$ and all pairs of *neighboring* (defined later) building blocks or nodes $\{\varsigma_i, \varsigma_j\} \in \mathcal{V}^b$ are connected by links $\varepsilon_l \in \mathcal{E}^b$. The number of nodes in this graph is much smaller than that in the road graph ($|\mathcal{V}^b| = n_b << n_r$). The nodes are first assigned their feature values $\varsigma_i.f$ as the mean of the corresponding building block. The links ε_l between nodes ς_i and ς_j are weighted by the similarity between $\varsigma_i.f$ and $\varsigma_j.f$ as in [1], [2]. Then a partitioning is performed on $\mathcal{G}^b$ to obtain a set of k differently congested partitions, utilizing any existing method. For example, α-Cut [1], [2] is one such algorithm to do this task. Higher mean values indicate higher levels of congestion. Using this information the user query is responded accordingly.

The physical layer always keeps itself up-to-date with the building blocks that are to be served to the logical layer, and the logical layer partitions a small graph of building blocks, which takes fractions of a second. Thus the method is able to produce the results immediately for any query. Below we will focus on the development of the physical layer and the algorithms to maintain the building blocks.

3.2 Physical Layer

The traffic congestion has the property to form and gradually grow from small regions to spread into other parts via the linked road segments. It is also very natural to have multiple blocks of independent congestions at the same time, which sometimes even merge with others. Considering both the congested and non-congested blocks of road segments, we propose the concept of *building blocks* in road networks.

Definition 6 (Building Block). *Given a road graph $\mathcal{G}$, a building block $\mathcal{B}_i = (\mathcal{V}_i^b, \mathcal{E}_i^b)$ is defined as a subgraph that forms one of the n_b fundamental constituents $\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{n_b}\}$ in the physical layer at a time stamp, such that $\bigcup_{i=0}^{n_b} \mathcal{V}_i^b = \mathcal{V}$ and $\mathcal{V}_i^b \bigcap \mathcal{V}_j^b = \emptyset$ for all $i \neq j$. The building blocks at time t are denoted by $\mathcal{B}^t = \{\mathcal{B}_1^t, \mathcal{B}_2^t, \ldots, \mathcal{B}_{n_{bt}}^t\}$*

The physical layer is the backbone of the proposed tracking method. It continuously maintains the evolving building blocks by incrementally updating them based on the most recent traffic data. To efficiently perform the incremental update, we start with an off-line preprocessing step to mine the road network building blocks. In the illustration examples of this paper, we will partition the road network based on the historical traffic data using α-Cut because of its suitability [1], [2], and consider the obtained partitions as the building blocks for the starting point. Nevertheless,

other alternatives can also be used in place of α-Cut. At each new time stamp, the most recent traffic data is fetched, based on which these blocks are incrementally updated by identifying and processing the *unstable* road segments. The building blocks for all the time stamps are stored and maintained in an index structure that facilitates their compact storage for later reference and efficient retrieval. For any query being passed from the logical layer, it retrieves the result from the index and returns back instantly.

3.3 Congestion Evolution

Definition 7 (Boundary Node). *Given a set of building blocks $\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{n_b}\}$ of a road graph $\mathcal{G}$, a node (of $\mathcal{G}$) $v_i \in \mathcal{B}_p$ is called a* boundary node *if there exists another node v_j such that $\langle v_i, v_j \rangle \in \mathcal{E}$ and $v_j \notin \mathcal{B}_p$.*

Definition 8 (Neighbor). *Given a set of building blocks $\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{n_b}\}$ of a road graph $\mathcal{G}$, $\mathcal{B}_p$ is called a* neighbor *of $\mathcal{B}_q$ if there exist nodes v_i and v_j such that $\langle v_i, v_j \rangle \in \mathcal{E}$ and $v_i \in \mathcal{B}_p \wedge v_j \in \mathcal{B}_q$.*

The continuously changing traffic conditions affect the structure of building blocks in terms of size, shape and location. For example, in the day time the traffic generally remains varied in the different regions, and thus require building blocks in fine granularity to effectively represent the traffic condition. At each subsequent time stamp their structure keeps changing, as much as the variation in network traffic, and the congestion hotspots keep evolving. The possible operations leading to the change are shifting the boundary nodes of $\mathcal{G}$ from one block to another, splitting one into multiple blocks, and merging of multiple blocks into one. We conceptually model this temporal evolution of spatially connected building blocks into a structure called *congestion evolution graph*, defined below, and illustrated later in Fig. 5(b).

Definition 9 (Congestion Evolution Graph). *For given sets of building blocks $\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^t$ at t consecutive time stamps and the neighbors in each $\mathcal{B}^i$, the* congestion evolution graph *is represented as a window of t time stamps. Each time stamp consists of spatially linked blocks, which have temporal links only to the blocks of preceding and following time stamps. The spatial links exist between two building blocks $\mathcal{B}^i_p$ and $\mathcal{B}^i_q$ only if they are* neighbor, *and the temporal links exist between two building blocks $\mathcal{B}^i_p$ and $\mathcal{B}^{i+1}_q$ only if $\exists v_j | v_j \in \mathcal{B}^i_p \wedge v_j \in \mathcal{B}^{i+1}_q$.*

3.4 Stability

During the 24 hours of a day, the traffic load on an urban road network varies from time to time. For example, in early morning the roads are mostly free, and as peak hour draws near, they become busy quite rapidly. The period of time during which the traffic changes from free to congested (or vice versa) is very short for some roads, depending on their spatial importance, which makes the vicinity unstable. After sometime, the traffic gradually approaches towards being stable. Thus *stability* is an important feature of road networks that leads to a better understanding of the spatiotemporal aspects of traffic congestion.

Definition 10 (Node Stability). *If a node v_i of road graph belongs to building block $\mathcal{B}_j$ at time t_{r-1}, then the stability $stab^r(v_i)$ is defined as the likelihood of v_i to remain in the same block $\mathcal{B}_j$ at time t_r.*

We consider two different kinds of *node stability*, i) *spatial* stability, which looks into how well the feature values of v_i match with those of the rest in $\mathcal{B}_j$ at the current time t_r; and ii) *temporal* stability, which looks into how much stable was v_i in the previous time stamps. Equation 1 shows the formulation to compute its measure, where μ^r_j denotes the mean feature value inside $\mathcal{B}_j$ at time t_r. The formula is an average of two quantities—the first one $stab^{(r-1)}(v_i)$, which is its stability measure from the previous time stamp t_{r-1}, stands for the temporal stability, and the second quantity, which is from the current time stamp t_r, stands for the spatial stability. The second quantity first gets the normalized distance of the node from the centroid of its block, and then subtracts it from 1 to get the closeness, which determines the spatial stability. Its value ($\in [0, 1]$) becomes 1 when the node feature value is exactly the same as the block mean value. A low value of this measure indicates that the node is less suitable for being part of the corresponding block.

$$stab^{(r)}(v_i) = \frac{\left(stab^{(r-1)}(v_i) + \left(1 - \text{abs}\left(\frac{v_i.f - \mu^r_j}{factor}\right)\right)\right)}{2} \quad (1)$$

$$factor = \max\{(\mu^r_j - v^{\min}_j), (v^{\max}_j - \mu^r_j)\} \quad (2)$$

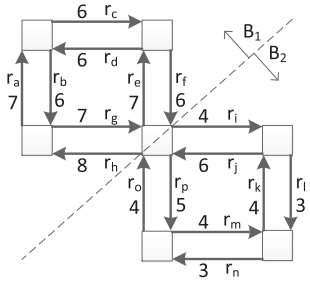
3.5 Road Network Motifs

Graph (or network) motifs are small connected components that exist in significantly large numbers in a graph globally [13], and form the elementary structures or patterns to make up the whole graph. Motifs have been found to be very useful in understanding the local structural principles of real world graphs. Road networks too have small connected structures that are commonly found all over the network. Generally the intersection points connect four different roads, each of which have their own traffic in the two opposite directions. This kind of intersections lead to the formation of directed cycles of length 4 (4-cycle or rectangle), 6 (6-cycles), and other cycles of higher *even* length in the road network. Triangles, 5-cycles, and other cycles of higher odd length are rare. We define the following motif concepts in road networks.

Definition 11 (Road Cycle). *Given a road network $\mathcal{N} = (\mathcal{I}, \mathcal{R})$, a path $p = (r_1, r_2, \ldots, r_l)$ comprising l different road segments is said to be an l-*roadcycle *if there exists a path $p' = (r_1, r_2, \ldots, r_l, r_{l+1})$ such that $r_1 = r_{l+1}$ and the traffic flow is directed as $r_1 \rightarrow r_2 \rightarrow\rightarrow r_l \rightarrow r_{l+1}$. An* l-roadcycle *$(r_1, r_2, \ldots, r_l)$ is said to be a γ-*bounded-l-roadcycle *if the euclidean distance between each r_i and r_{i+1} is less than or equal to γ.*

A *bounded road cycle* ensures that it is a tightly bound cycle where all the participating road segments have similar traffic measures up to a certain extent. Due to the static structure of the road networks, the *road cycles* remain the same throughout, but the dynamic traffic keeps the set of *bounded road cycles* open to change at any point of time.

The building blocks maintained by our method have the properties of high homogeneity and high connectivity inside them, whereas these properties are marked low (i.e., high heterogeneity and low connectivity) between different blocks in the road graph. Fig. 2 shows an example of a small road sub-network (each road segment has its own name and indicative traffic measure) and the list of all γ-bounded-l-roadcycles where γ and l are set to 2 and 4 respectively. The network in Fig. 2(a) is divided by a dashed line into blocks B_1 and B_2, in which both the blocks have homogeneous traffic measures inside them, but



(a) A road sub-network

$\mathbf{B}_1$	$\mathbf{B}_2$
r_a-r_c-r_f-r_h	r_i-r_l-r_n-r_o
r_a-r_c-r_d-r_b	r_i-r_l-r_k-r_j
r_a-r_b-r_g-r_h	r_i-r_j-r_p-r_o
r_b-r_g-r_e-r_d	r_j-r_p-r_m-r_k
r_c-r_f-r_e-r_d	r_k-r_l-r_n-r_m
r_e-r_f-r_h-r_g	r_m-r_n-r_o-r_p
r_e-r_f-r_i-r_j	

(b) 2-bounded-4-roadcycles

Fig. 2. Example of γ-*bounded-*l-*roadcycles*.

heterogeneous to each other, with B_1 having higher measures than B_2. Fig. 2(b) shows all the 2-bounded-4-roadcycles. It is found that out of the total of 13 cycles, 6 lie completely inside B_1, the same amount inside B_2, and only one cycle shown at the bottom is shared between the two blocks. This observation leads to the fact that the number of occurrences of bounded cycles inside each building block is significantly higher than those across different blocks. These bounded cycles give information about the block structure and can be used to locate their suitable boundaries, in order to maintain blocks of *low inter-block and high intra-block associativities*. The lengthy cycles are mostly composed of short cycles (e.g., rectangles). Therefore, instead of all the possible cycles, we consider only the short-length bounded cycles to give an indication about the building block structures.

4 Capturing the Evolution

This section presents our method to continuously track the change by incrementally updating the building blocks in an efficient manner, and thus capture the spatiotemporal evolution in a fine granularity in the form of historical building blocks.

4.1 Index Structure

For efficient computation and economic memory consumption, we index the historical sets of building blocks, the current boundary nodes, and the static short cycles using suitable structures.

4.1.1 Building Block Index (Bin)

The tracking of the evolution of building blocks needs to maintain the historical information (of selected time stamps) in the main memory. It demands an index structure that provides a compact in-memory storage and efficient retrieval. We propose an in-memory block index called Bin (illustrated in Fig. 3) to address these requirements. It consists of three different components: a tree structure Tr, a list of linked lists TL, and an inverted list NI. The design of Tr enables a compact storage, and TL and NI facilitate efficient retrieval. Each block at time t_i is a set of road segments having a unique identifier. These blocks are indexed in the form of a tree Tr having a root node R, a set of internal nodes that correspond to the road segments, and a set of leaf nodes that stand for the different time stamps. All nodes except the root have one parent, and all nodes except the leaves have one or more children. Fig. 3 shows the index for sets of building blocks at three time stamps – $i)$ $\{\{12,6,1\},\{4,8,7,3,2\},\{11,10,5,9\}\}$ at t_0, $ii)$ $\{\{11,12,6,1\},\{8,7,3,2\},\{4,10,5,9\}\}$ at t_1; and $iii)$ $\{\{10,11,12,6,1\},\{5,8,7,3,2\},\{9\},\{4\}\}$ at t_2. A block can be retrieved by traversing up from the leaf node until the root node is reached, and the time stamp of that leaf node indicate the time stamp of the retrieved block. TL maintains a list of all the time stamps where each one of them is linked to all the leaf nodes referring to that time. As shown in the figure, t_0 from the list is linked to all the leaf nodes referring to t_0 in different branches of the tree. To access blocks of a particular time stamp, say t_0, the linked list for t_0 can be accessed from TL, using which all the leaf nodes corresponding to t_0 can be accessed, which further lead to retrieval of all the building blocks by upward tree-traversal. For a fast lookup of the latest time stamp (supports faster incremental update computations), NI maintains an inverted list of the leaf nodes, which lead to access the corresponding node at the latest time stamp. The figure shows that at time t_2, the node v_1 can be accessed by the leaf node in the left most branch (child of 10), and v_2 can be accessed by the leaf child of 5. NI is simply used to refer to the building block to which a particular node belongs.

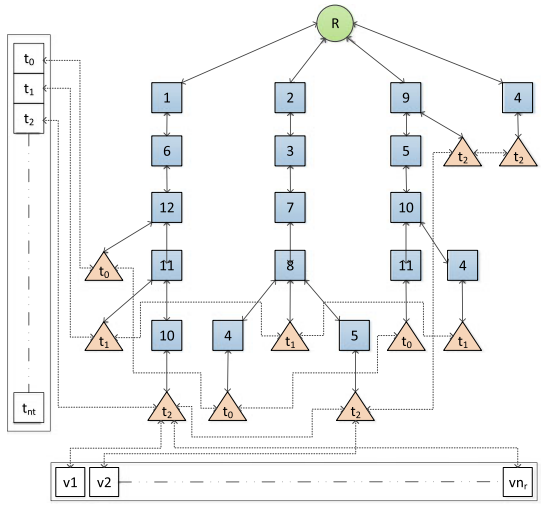
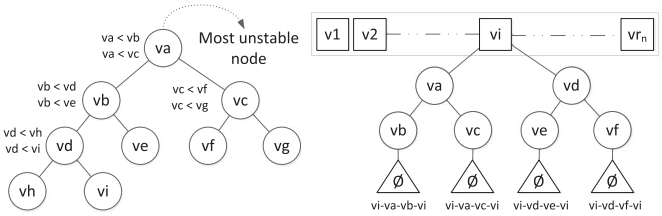


Fig. 3. Bin.

4.1.2 Stability Tree

The building blocks are small subgraphs where road segments are represented as nodes of $\mathcal{G}$ and linked by the neighborhood relationship. The building blocks have one set of nodes lying on the boundaries (boundary nodes) and another set lying in the interior (internal nodes). As the boundary nodes are directly linked to nodes of a neighboring building block, they are more likely to undergo a change (change in the block to which a node belongs) than the internal nodes in the next time stamp. Moreover, through boundary adjustment an internal node can undergo a change only after at least one of its linked nodes has undergone a change that makes this internal node a boundary node. Even among the boundary nodes, they all differ in their likelihood of change, which is captured by our stability measure. We use this measure to prioritize the boundary adjustment of nodes with high unstablity. At the beginning time stamp, we compute all the boundary nodes and index them as follows. A min-heap tree is created based on the stability measure, in which the least stable or the most unstable node of $\mathcal{G}$ becomes the root. This tree is called *stability tree* (denoted by ST) and



(a) Stability tree (b) Short cycle index

Fig. 4. Index for boundary nodes and their short cycles ('$<$' denotes *less stable than*).

shown in Fig. 4(a). During the process of boundary adjustment the node with highest priority is processed first, and this min-heap tree is maintained all throughout the incremental update by adding all the newly becoming boundary nodes and removing all the newly becoming internal nodes.

4.1.3 Short Cycle Index

We pre-compute all the possible $cycles$ of path length smaller than or equal to ϵ_{path} in the road graph as an offline task and index them as follows. First a sorted set of all nodes in $\mathcal{V}$ is created based on their id. For each node $v_i \in \mathcal{V}$, all the cycles that involve this node in the path are computed. Let us suppose $\{(v_i, v_a, v_b, v_i), (v_i, v_a, v_c, v_i), (v_i, v_d, v_e, v_i), (v_i, v_d, v_f, v_i)\}$ is the set of cycles involving v_i. A trie tree is created having v_i as the root node, v_a and v_d as the children of v_i, v_b and v_c as the children of v_b, and v_e and v_f as the children of v_d. Then the end-marker leaf nodes having the information of their depth in the tree are added as children of v_b, v_c, v_e, and v_f. In this way the trie trees corresponding to all $v_i \in \mathcal{V}$ are created and attached to the sorted set as shown in Fig. 4 (b). This structure is called *short cycle index* (SCI). We achieve two things with the help of SCI. First, having all the pre-computed short cycles available in the memory, they are not required to be computed repeatedly for each node during the runtime of the incremental update algorithm in Section 4.2. Second, indexing them in a trie structure consumes a minimum memory space, and enables to re-use the partly computed informations in the algorithm.

4.2 Incremental Update

The incremental update algorithm looks into all the unstable nodes of $\mathcal{G}$ and moves them to the most suitable building blocks at each time stamp. Broadly it consists of two main tasks—the incremental computation of the set of building blocks for the current time stamp followed by updating the index `Bin`. Let us suppose we have a given `Bin` and short cycle index SCI at time t_{i-1} for sets of building blocks from time t_0 to t_{i-1}. The complete algorithm to incrementally compute and update `Bin` at time t_i is shown in Algorithm 1. It starts with computing the stability measures from the traffic data at t_i for all the boundary nodes and creating the stability tree ST (lines 1-2). Then the Tr, TL, and NI components of `Bin` are updated to add the leaf nodes for t_i and re-direct the links in TL and NI (lines 3-5). The iterative steps of computing the most suitable building block for the most unstable boundary node and updating `Bin` is carried out until all the boundary nodes having their stability measure less than the threshold ϵ_{stab} have been processed (lines 6-15). After getting the most unstable node v by deleting the root of ST (line 6), the short cycle tree $sctree$ is accessed from SCI (line 9), and the most suitable leaf node $lnext$ ($\in$ `Bin`.Tr) is computed using the function `IdentifyMSL(.)` described later in Algorithm 2 (line 10). This computation is based on the new traffic data at t_i in contrast to the current leaf node $lcurrent$ ($\in$ `Bin`.Tr) based on the data at t_{i-1}. If the new leaf node is different than the existing one (line 11), then v is deleted from the branch of $lcurrent$ using function `DeleteFromBranch(.)` described later in Algorithm 3 (line 12). It is followed by inserting v into the branch of $lnext$ using function `InsertIntoBranch(.)` described later in Algorithm 4 (line 13). This step completes the update of `Bin` for v. The stability tree ST is then updated by inserting all the newly created boundary nodes and deleting those nodes which no more lie on the boundary because of the change using function `AddRemoveSTNodes(.)` described later in Algorithm 5 (line 14). After completing the processing of v, the next most unstable node is extracted from ST (line 15) and the same process (lines 7-15) is carried out repeatedly until all the unstable nodes have been processed.

4.3 Computing the Most Suitable Block

As mentioned earlier in Section 3.5, on the basis of the properties of network motifs and building blocks, the short length bounded road cycles are likely to be found in significantly large numbers within the building blocks rather than crossing multiple of them. A naive way to find the most suitable block for a node v at time t_i is to select the one having the highest number of bounded road cycles with all the nodes of $\mathcal{G}$ lying in the same block. But often there are cycles passing through multiple blocks, where most part of the cycle lie within the most suitable block leaving some fractions in neighboring blocks. The naive method ignores these fractions. Our main idea here is to identify the block that is involved in most part of the bounded road cycles of node v. For this, we consider all the road cycles of path length shorter than or equal to ϵ_{path}, making the range as $[3, \epsilon_{path}]$. Each block is quantified by a weight function $W(.)$ that considers the total of fractions from all the cycles lying in the respective block. For this quantification, all cycles account for a weight of 1, which is equally divided among all the cycle nodes other than v. For longer cycles the value being divided among more nodes gives lesser power to each. Therefore, the shorter the cycle, the bigger the impact of its nodes.

Algorithm 1. IncrementalUpdate(Block Index `Bin`, Stability tree ST, Short Cycle Index SCI)

1 Compute stability of boundary nodes at t_i;
2 Stability tree $ST \leftarrow$ Create a max-heap tree;
3 Update `Bin`.$Tr \leftarrow$ Add a leaf node T_i as sibling of each T_{i-1};
4 Update `Bin`.$TL \leftarrow$ Link all T_i and add the list to TL;
5 Update `Bin`.$NI \leftarrow$ Set $(node.leaf \leftarrow T_i)$ instead of T_{i-1}, $\forall node \in$ `Bin`.NI;
6 Node $v \leftarrow$ Delete root of ST;
7 **while** $stablity(v) \leq \epsilon_{stab}$ **do**
8 Leaf $lcurrent \leftarrow$ `Bin`.$NI.v.leaf$;
9 $sctree \leftarrow SCI[v]$;
10 Leaf $lnext \leftarrow$ IdentifyMSL(`Bin`, $sctree$);
11 **if** $lnext \neq lcurrent$ **then**
12 Update `Bin` $\leftarrow$ DeleteFromBranch(`Bin` , $lcurrent, v$);
13 Update `Bin` $\leftarrow$ InsertIntoBranch(`Bin`, $lnext, v$);
14 Update $ST \leftarrow$ AddRemoveSTNodes(`Bin`, $ST, v, lcurrent$, $lnext$);
15 Node $v \leftarrow$ Delete root of ST;
16 **return** `Bin`;

Formulated in Equation 3, $W(v, \mathcal{B}_j)$ computes the weight assigned to block $\mathcal{B}_j$ to identify the most suitable block for v, where $RCycles(v)$ gives all the γ-bounded short road cycles involving v, and u is another node of $\mathcal{G}$ that is involved in the same cycle and belongs to $\mathcal{B}_j$ at time t_i.

$$W(v, \mathcal{B}_j) = \sum_{\substack{\forall C \in RCycles(v) \\ (u \in C) \wedge (u \in \mathcal{B}_j)}} \frac{1}{pathlength(C) - 1} \tag{3}$$

In other words, each road cycle C that involves v is traversed, and for each node u in that cycle, if it belongs to block $\mathcal{B}_j$, the weight for $\mathcal{B}_j$ is incremented by $\frac{1}{pathlength(C)-1}$. For example, if there is a cycle $\{v_1(\in \mathcal{B}_1), v_2(\in \mathcal{B}_2), v_3(\in \mathcal{B}_1), v_4(\in \mathcal{B}_1)\}$, each node (except the one for which the weight is being computed) will have the power to make an affect by $\frac{1}{3}$, which in total equals to 1. To compute $W(v_1, \mathcal{B}_1)$, v_3 and v_4 both belonging to $\mathcal{B}_1$ adds up to $\frac{2}{3}$, and the weight $\frac{1}{3}$ of $v_2 \in \mathcal{B}_2$ is added to $W(v_1, \mathcal{B}_2)$. In this manner, the weights for all the blocks are computed by adding the values from all the different cycles, and the one with the highest weight is selected as the most suitable block. However, as the number of road cycles is usually large, traversing all of them individually to compute the weight for the building blocks in this way adds a lot of computations, and affects the running time.

In most of the road cycles in which a node v is involved, there exists overlapping of some parts of the complete path of multiple cycles. For example, the cycles $C_1 = \{v, v_1, v_2, v_3\}$ and $C_2 = \{v, v_1, v_2, v_4\}$ of v have three overlapping nodes (v, v_1 and v_2). Computing the weights by traversing through C_1 and C_2 independently, repeats the computations done for v, v_1 and v_2. We make use of these overlappings in computing the building block weights, thereby avoid redundant computations. This is done by our *multi-stack based algorithm* (shown in Algorithm 2) with the help of our short cycle index SCI that keeps the cycles indexed as a tree, having no repeating nodes even for the overlapping cycles. It accesses the block index `Bin` and the short cycle tree $sctree$ from SCI, and computes the most suitable leaf ($\in$ `Bin`.Tr) for node v (root node of $sctree$) at time t_i. The stacks used in the algorithm explore the cycles in $sctree$ and keep part of the computed information saved for its reuse later for the overlapping cycles.

The algorithm uses three stacks $s1$, $s2$, and $s3$ to store and process the computed values throughout (line 1). It starts with pushing all the children of root node of $sctree$ to the stack $s1$, if the euclidean distance between the parent and the child $dist(nparent, nchild)$ is less than or equal to γ (lines 2-5). The $dist(.)$ function ensures that only the γ-bounded road cycles are explored. It is followed by iterative steps of popping out a node from $s1$ (line 7), pushing it into the stack $s2$ (line 8), pushing all its children back into $s1$ if the parent-child satisfies the γ distance condition (lines 10-12), and pushing the count of these children into the stack $s3$ (line 13). These steps are repeated until $s1$ becomes empty (line 14). They compute the number of children of each node in the short cycle tree, and keep them saved in the stacks for computing the block weights (or leaf weights) later. Thereafter an array of values is initialized to store the weights $W(.)$ computed for each block in order to select the most suitable one (line 16). As the blocks are retrieved by the leaf nodes in `Bin`, the weights correspond to the leaf nodes of `Bin`.Tr at time t_i. Then the nodes are popped out from $s2$ one after another (line 18), the count of their children are popped out from $s3$ (line 19), followed by a set of steps to compute the weights. A value of 0 as the count of children indicates that it is the last node of a branch in the short cycle tree, and its *depth* gives the path length of the cycle. Hence the weight of $\frac{1}{pathlength(C)-1}$ that is carried by each node in the cycle C is computed as $\frac{1}{depth(node)}$ (line 21) and pushed into $s1$ (line 27) to be used later to compute weights for its parents. The depth of the node is found from its leaf in the short cycle tree that contains the depth information. If the count of children is non-zero, the weight is computed by adding the weights of all its children obtained by popping up $s1$ as many times as the count (lines 22-26), which is again pushed back into $s1$ (line 27). In addition to pushing the weight of nodes into $s1$, the weight for the leaves in the array $wleaf[\,]$ is updated by adding $weight$ to the array element corresponding to the leaf of $node$, i.e., `Bin`.$Tr.node.leaf$ (line 28). These steps of popping out from $s2$ and $s3$ and computing the values of $wleaf[\,]$ using $s1$ (lines 18-28) are repeated until $s2$ becomes empty (lines 17, 29), which marks the completion of processing of all the bounded short road cycles. At last we select the most suitable leaf for the current time t_i by getting the leaf corresponding to the maximum weight in $wleaf[\,]$ (lines 30-35).

Algorithm 2. IdentifyMSL(Block Index `Bin`, Short Cycle Tree $sctree$)

1 $s1, s2, s3 \leftarrow$ initialize new stack;
2 Node $nparent \leftarrow sctree(root)$;
3 **for all** *Node* $nchild \in sctree(root).child$ **do**
4 **if** $nchild \neq \phi$ AND $dist(nparent, nchild) \leq \gamma$ **then**
5 Push $nchild$ into $s1$;
6 **repeat**
7 Node $nparent \leftarrow$ pop out from $s1$;
8 Push $nparent$ into $s2$;
9 $count = 0$;
10 **for all** *Node* $nchild \in sctree(nparent).child$ **do**
11 **if** $nchild \neq \phi$ AND $dist(nparent, nchild) \leq \gamma$ **then**
12 Push $nchild$ into $s1$, $count \leftarrow count + 1$;
13 Push $count$ into $s3$;
14 **until** $s1 \neq empty$
15 $s1 \leftarrow$ re-initialize stack;
16 $wleaf[\,] \leftarrow$ initialize an array of values for each leaf of current time stamp in `Bin`.Tr;
17 **repeat**
18 Node $node \leftarrow$ pop out from $s2$;
19 $countchild \leftarrow$ pop out from $s3$;
20 **if** $countchild = 0$ **then**
21 Value $weight = \frac{1}{depth(node)}$;
22 **else**
23 Value $weight \leftarrow$ initialize with 0;
24 **for** $i \leftarrow 1$ **to** $countchild$ **do**
25 Value $childweight \leftarrow$ pop out from $s1$;
26 $weight \leftarrow weight + childweight$;
27 Push $weight$ into $s1$;
28 $wleaf$[`Bin`.$NI.node.leaf$] $\leftarrow$ Increment by $weight$;
29 **until** $s2 \neq empty$
30 Leaf $lcurrent \leftarrow$ `Bin`.$NI.sctree(root).leaf$;
31 $msl \leftarrow lcurrent, wmax \leftarrow wleaf[lcurrent]$;
32 **for all** $wl \in wleaf[\,]$ **do**
33 **if** $wmax < wl$ **then**
34 $msl \leftarrow$ leaf corresponding to wl, $wmax \leftarrow wl$;
35 **return** msl;

4.4 Updating Index

After identifying the most suitable leaf in the index Bin at time t_i for a node v of $\mathcal{G}$, the remaining task is to update Bin and ST with this change. This is done by the functions DeleteFromBranch(.), InsertIntoBranch(.), and AddRemoveSTNodes(.). The operations of inserting/deleting a node into/from a branch defined on Bin are performed in such a manner that the data redundancy at different time stamps is minimum and there is no information loss in the historical data. This section explains each of the three mentioned functions individually.

The function DeleteFromBranch(.) (Algorithm 3) is called to delete a node v from its current branch in Bin Its main objective is to delete v in such a way that the block retrieved for time t_{i-1} includes v in the set of nodes but excludes it for time t_i. For this we start from the leaf of its branch, and traverse upwards towards the root using two pointers $nodepre$ and $node$ until v is found (lines 1-11). During the traversal, we check if there are other nodes that do not lie in the traversal path (branch out somewhere). It is done by looking into the set of children of $node$. If there are other children in addition to $nodepre$ (line 5), then a new node $vclone$ is created as a clone of v (line 6), and inserted as parent of all the siblings of $nodepre$ (line 7) and as child of $node$ (line 8). This process is continued until $nodepre$ reaches v (line 11), after which v is simply removed from there by making all its children (or that of $nodepre$) as the children of $node$ (line 12). This node is added as child of the parent of leaf $lcurrent.parent$ (line 13) and as parent of all the children of $lcurrent.parent$ (line 14). At the end, the leaf $lcurrent$ (referring to T_i) is set as the child of $nodepre.parent$ or sibling of $nodepre$ (line 15). After doing this change, the block retrieved by traversing from T_i to root includes all the previous nodes except v, whereas the set of nodes remain the same for all other time stamps or leaves. By inserting additional clones of v, we add some redundancy, but it is kept the minimum that is required to keep all the information for previous time stamps without any loss. A *delete* operation takes the node closer to the leaf. The unstable nodes, which frequently undergo *deletion* from a branch, are quickly found in the upward traversal from the leaf, thereby improves its efficiency for such nodes.

Algorithm 3. DeleteFromBranch(Block Index Bin, Leaf $lcurrent$, Node v)

1 Node $nodepre \leftarrow lcurrent.parent$;
2 Node $node \leftarrow nodepre.parent$;
3 **if** $nodepre \neq v$ **then**
4 **repeat**
5 **if** $\{node.child \backslash nodepre\} \neq \phi$ **then**
6 Node $vclone \leftarrow$ new clone of v;
7 $vclone \overset{\text{parent-childlink}}{\longleftrightarrow} \{node.child \backslash nodepre\}$;
8 $node \overset{\text{parent-childlink}}{\longleftrightarrow} vclone$;
9 $nodepre \leftarrow node$;
10 $node \leftarrow node.parent$;
11 **until** $nodepre = v$
12 $node \overset{\text{parent-childlink}}{\longleftrightarrow} \{nodepre.child\}$;
13 $lcurrent.parent \overset{\text{parent-childlink}}{\longleftrightarrow} nodepre$;
14 $nodepre \overset{\text{parent-childlink}}{\longleftrightarrow} \{lcurrent.parent.child\}$;
15 Set T_i as child of $nodepre.parent$;
16 **return** Bin;

After the deletion of v from its previous branch it is inserted into the most suitable branch $lnext$ in Bin using the function InsertIntoBranch(.) (Algorithm 4). The main idea here is that if v already exists there as a sibling of $lnext$ (referring to T_i), then without adding any new node, $lnext$ is simply set as the child of v (line 5). Otherwise (line 2), a new node is created for v (line 3) and added as a child of the parent of $lnext$ (line 4), after which $lnext$ is set as the child of v (line 5). After this update, the complete block including v can be retrieved directly by traversing upwards from $lnext$ to the root.

Algorithm 4. InsertIntoBranch(Block Index Bin, Leaf $lnext$, Node v)

1 Node $node \leftarrow lnext.parent$;
2 **if** $v \notin \{node.child\}$ **then**
3 Create a node for v;
4 $node \overset{\text{parent-childlink}}{\longleftrightarrow} v$;
5 Set T_i as child of v;
6 **return** Bin;

The stability tree ST contains all the boundary nodes in the road network at a point of time, heapified based on their stability measure. After deleting its root to get the unstable nodes and process them, if the node changes its branch in Bin.Tr (or changes it block), several nodes newly become boundary nodes. On the other hand, several nodes that were on the boundary earlier, no more lie on the boundary. The function AddRemoveSTNodes(.) (Algorithm 5) updates ST by adding the new boundary nodes into the tree and removing the internal nodes from the tree. It starts with getting all the nodes u (can be 7 at most) linked to v in the road graph (line 1). If the leaf of u referred in the node inverted list Bin.$NI.u.leaf$ is same as the leaf node of v before the update $lcurrent$ (line 2), it means that u is a boundary node. If it does not exist in ST, being a new boundary node it is added to the tree (lines 2-3). If the leaf of u referred in Bin.$NI.u.leaf$ is same as the leaf node of v after the update $lnext$ (line 4), it means that u may not be a boundary node anymore. If it does not exist in ST and does not lie on the boundary, it is removed from the tree (lines 4-5). Everytime a node v changes its branch in Bin.Tr, all its linked nodes are checked to keep an up-to-date collection of the boundary nodes.

Algorithm 5. AddRemoveSTNodes(Block Index Bin, Stability tree ST, Node v, Leaf $lcurrent$, Leaf $lnext$)

1 **for all** *Node* $u \in neighbour(v)$ **do**
2 **if** Bin.$NI.u.leaf = lcurrent$ AND $u \notin ST$ **then**
3 Insert u into ST;
4 **if** Bin.$NI.u.leaf = lnext$ AND $u \in ST$ AND u does not lie on boundary **then**
5 Delete u from ST;
6 **return** ST;

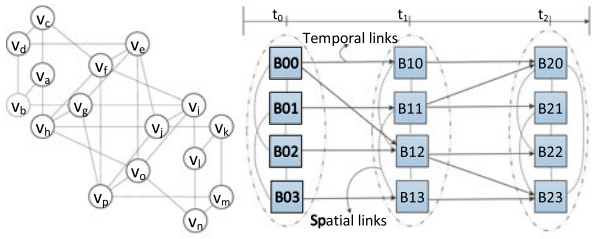
4.5 An Illustrative Example

Let us consider the road network shown in Fig 2(a), with new indicative traffic measures of the road segments at three time stamps, as shown in Table 1. The road graph of this network is shown in Fig. 5(a). While the road segments r_i are presented as part of the actual road network at the user end, they are considered as nodes v_i of the road graph for internal

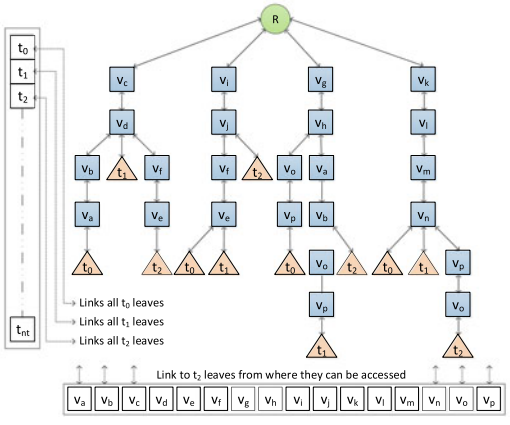
TABLE 1
Indicative Traffic Measures

	r_a	r_b	r_c	r_d	r_e	r_f	r_g	r_h	r_i	r_j	r_k	r_l	r_m	r_n	r_o	r_p
t_0	7	6	6	6	4	3	10	9	3	2	6	7	4	5	10	11
t_1	7	7	5	4	3	2	9	9	4	3	6	7	5	6	10	11
t_2	10	9	3	2	3	2	9	10	5	4	7	8	8	9	7	8

processing. To set the starting point at time t_0, we obtain 4 building blocks by applying α-Cut, as explained in Section 3.2. The different colors in Table 1 denote the different building blocks ($\mathcal{B}_0$ orange, $\mathcal{B}_1$ purple, $\mathcal{B}_2$ blue, and $\mathcal{B}_3$ green). The Bin is constructed for these four blocks as four linear branches from the root in Tr, ending with leaf nodes. We now incrementally update these blocks by considering only the *2-bounded-4-roadcycles* at t_1 and t_2, which are $\{(r_a\text{-}r_b\text{-}r_g\text{-}r_h), (r_k\text{-}r_l\text{-}r_n\text{-}r_m), (r_e\text{-}r_f\text{-}r_i\text{-}r_j), (r_g\text{-}r_p\text{-}r_o\text{-}r_h)\}$ and $\{(r_a\text{-}r_b\text{-}r_g\text{-}r_h), (r_c\text{-}r_f\text{-}r_e\text{-}r_d), (r_k\text{-}r_l\text{-}r_n\text{-}r_m), (r_m\text{-}r_n\text{-}r_o\text{-}r_p)\}$, respectively. In Algorithm 1, the stability tree processes the nodes (of road graph) in the following order at t_1: $v_p, v_f, v_o, v_e, v_j, v_g, v_h, v_d, v_i, v_c, v_m, v_a, v_b, v_l, v_k, v_n$. Based on Algorithm 2, nodes v_a and v_b are moved from block $\mathcal{B}_0$ to $\mathcal{B}_2$. Similarly at t_2, v_f and v_e are moved from $\mathcal{B}_1$ to $\mathcal{B}_0$, and v_p and v_o are moved from $\mathcal{B}_2$ to $\mathcal{B}_3$ in this order. The colors in the rows for t_1 and t_2 in Table 1 show the blocks thus obtained, Fig. 5(c) shows the resulting structure of Bin, and Fig. 5(b) shows the congestion evolution graph. Any block of any historical time stamp can be directly accessed from Bin.Tr with the help of Bin.TL. We also observe that the unstable nodes which frequently change blocks remain close to the leaves in Bin.Tr, and as mentioned earlier, this makes the *delete* operation efficient. All these computations are performed in the physical layer. On passing a query for 2 partitions at t_2, the logical layer produces the resulting partitions of road segments as $\{r_c, r_d, r_e, r_f, r_i, r_j\}$ and $\{r_g, r_h, r_a, r_b, r_k, r_l, r_m, r_n, r_o, r_p\}$.



(a) Road graph (b) Congestion evolution graph



(c) Structure of Bin at t_2

Fig. 5. Illustrative example.

TABLE 2
Dataset Statistics

Dataset	Place	Area(sq ml)	Road seg	Inter pts
$\mathtt{M}_s$(*real*)	Melbourne	627.5	7245	2928
$\mathtt{M}_1$	CBD Melbourne	6.6	17,206	10,096
$\mathtt{M}_2$	CBD(+) Melbourne	31.5	53,494	28,465
$\mathtt{M}_3$	Melbourne	42.03	79,487	42,321
$\mathtt{M}_4$	Melbourne(+)	57.5	102,292	55,518

4.6 Time Complexity

The worst case time complexity of the proposed method (Algorithm 1) is $\mathcal{O}(n^2)$, but most of the times it is much lower. In Algorithm 1, lines 1-2 are performed only for the boundary nodes, which would be n in worst case. Thus the worst case time complexity of line 2 is $\mathcal{O}(n \log n)$, which is the cost of creating a heap tree. Lines 3-6 are performed in constant time. Lines 7-15 execute only for unstable nodes. In worst case if all the nodes are unstable, which has almost no chance of happening, these lines would be executed n times. Lines 8, 9 and 15 are performed in constant time, and line 10 executes Algorithm 2. Algorithms 3, 4 and 5 are performed in lines 12-14 only if the unstable node is to be shifted to another block. The worst case time complexity of Algorithm 2 is $\mathcal{O}(n)$. Lines 2-14, 17-29, and 32-34 are performed at the cost of $\mathcal{O}(ncycle_i)$, $\mathcal{O}(ncycle_i)$, and $\mathcal{O}(n_b)$ respectively, where n_b is the number of building blocks and $ncycle_i$ is the number of different nodes involved in cycles of the unstable node v_i. All other lines are performed in constant time. Algorithm 3 traverses upwards from the leaf towards the root in Bin until the node to be deleted v is found. The unstable nodes generally exist closer to the leaf, and therefore v is found quickly. In worst case, the complete branch has to be traversed at the cost of $\mathcal{O}(7\times$ *number of nodes in the corresponding block*$)$. Algorithm 4 is performed in constant time as the insertion takes place at the leaf, and Algorithm 5 has a worst case time complexity of $\mathcal{O}(7\times$ *number of nodes in the corresponding block*$)$ as a node in the road graph can have a maximum of 7 links.

5 Experiments

5.1 Datasets

Our datasets, shown in Table 2, include one real ($\mathtt{M}_s$) and four semi-synthetic ($\mathtt{M}_1$, $\mathtt{M}_2$, $\mathtt{M}_3$, and $\mathtt{M}_4$) datasets. $\mathtt{M}_s$ is recorded by the Sydney Coordinated Adaptive Traffic System (SCATS)[1] from the Melbourne road network provided to us by VicRoads.[2] It is an accumulation of the traffic records of individual road segments for each signal cycle from 1st Jan 2011 to 1st Jan 2013. The considered Melbourne network consists of 7245 road segments and 2928 intersection points. The traffic measures include traffic volume (number of vehicles crossing a road segment during the green time) and degree of saturation (the ratio of the effectively used green time to the total available green time). We consider degree of saturation as the feature value of road segments, which approximates the traffic density. $\mathtt{M}_1$, $\mathtt{M}_2$,, $\mathtt{M}_3$, and $\mathtt{M}_4$ are synthetically

1. SCATS is a fully adaptive urban traffic control system developed in Australia in 1970. It manages the signal phases (cycle times, phase splits, and offsets) of the traffic signals dynamically in real-time, based on the traffic data collected by the vehicle sensors (inductive loops) installed within road pavements of each traffic signal.

2. https://www.vicroads.vic.gov.au/

generated on the real road network of Melbourne using a web-based[3] random road traffic generator MNTG [14]. We populated 25,246, 62,300, 84,999, and 127,285 vehicles respectively, and obtained their movement trajectories for 100 continuous time stamps. The trajectories are sequences of 100 or less ⟨latitude,longitude⟩ pairs corresponding to vehicle positions at each timestamp. The vehicle positions are mapped to corresponding road segments and the traffic density measures (in terms of vehicles/meter) are computed at each time stamp. We use traffic density as the feature value of road segments for these datasets. All our datasets consider urban-scale networks, because generally outside the city area there are not many branches of roads, and the connections are sparse. It leads to a natural separation of the different cities, and therefore larger networks, e.g., state road networks, do not require any technical treatment for partitioning.

5.2 Evaluation Metrics

The overall quality of the incremental partitioning results is evaluated in several ways. First, we assume the partitions obtained by the normal re-partitioning approach (α-Cut [1]) as ground truth $\mathcal{P}^g$, and relatively measure the accuracy of the dynamically obtained (proposed method) partitions $\mathcal{P}$ using *purity F-measure* (or F_P) $\in [0, 1]$ [15]. It is the harmonic mean of *purity* $(\sum_{\mathcal{P}_i \in \mathcal{P}} \frac{1}{n_r} \times \max_{\mathcal{P}^g_j \in \mathcal{P}^g} |\mathcal{P}_i \cap \mathcal{P}^g_j|)$ and *inverse purity* $(\sum_{\mathcal{P}^g_i \in \mathcal{P}^g} \frac{1}{n_r} \times \max_{\mathcal{P}_j \in \mathcal{P}} |\mathcal{P}^g_i \cap \mathcal{P}_j|)$. Thereafter, we compare different methods using *ANS* (developed in [4], explained below).

A value of less than 1 for this measure indicates a good partitioning, and lower values indicate better partitioning. We also use *intra* (average of the intra-partition distances, the lower the better) and *inter* (average of the spatially adjacent inter-partition distances, the higher the better) metrics, described in detail in [2], to evaluate the intra and inter properties separately.

The NCutSilhouette (NS) between two partitions $\mathcal{P}_i$ and $\mathcal{P}_j$, defined in Equation 4, computes the dissimilarity between $\mathcal{P}_i$ and $\mathcal{P}_j$. A small value of $NS(\mathcal{P}_i, \mathcal{P}_j)$ indicates that $\mathcal{P}_i$ and $\mathcal{P}_j$ have low dissimilarity (or high similarity). The quality of a partition $\mathcal{P}_i \in \mathcal{P}$ is measured by $NS(\mathcal{P}_i)$, defined in Equation 5, where the numerator is the intra-partition dissimilarity of $\mathcal{P}_i$ and the denominator $NSN(\mathcal{P}_i, \mathcal{P}_j) = \min\{NS(\mathcal{P}_i, \mathcal{P}_x) | \mathcal{P}_x \in neighbor(\mathcal{P}_j)\}$ is the inter-partition dissimilarity of $\mathcal{P}_i$ with the remaining partitions of $\mathcal{P}$. Finally, the ANS is computed as the average of $NS(\mathcal{P}_i)$ for all $\mathcal{P}_i \in \mathcal{P}$.

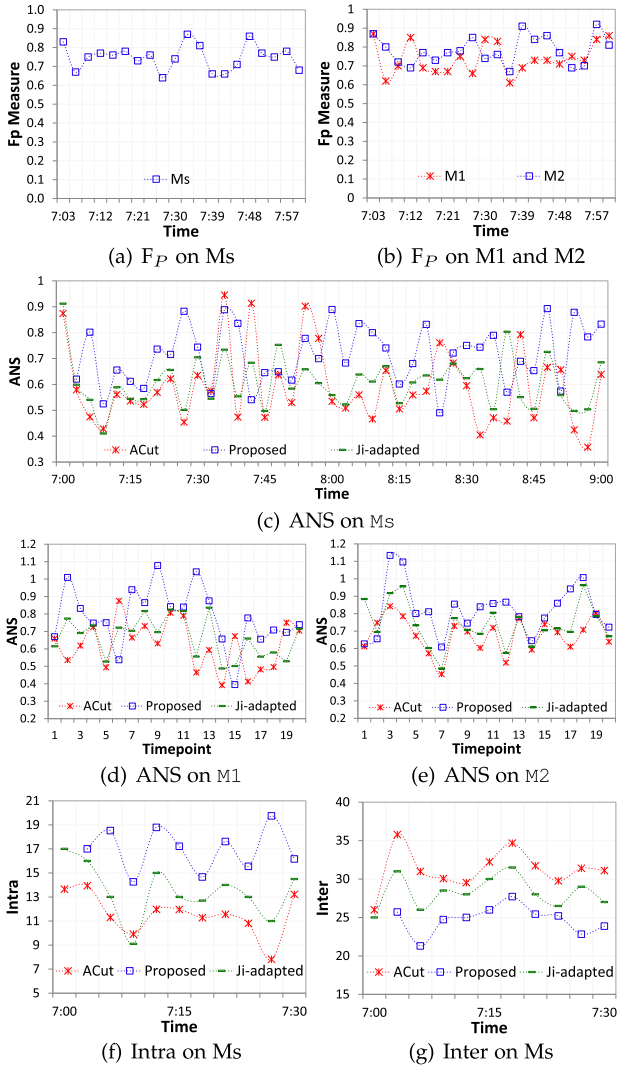
$$NS(\mathcal{P}_i, \mathcal{P}_j) = \frac{\sum_{v_p \in \mathcal{P}_i} \sum_{v_q \in \mathcal{P}_j} \left(v_p.f - v_q.f\right)^2}{|\mathcal{P}_i| \times |\mathcal{P}_j|} \quad (4)$$

$$NS(\mathcal{P}_i) = \frac{NS(\mathcal{P}_i, \mathcal{P}_i)}{NSN(\mathcal{P}_i, \mathcal{P}_j)} \quad (5)$$

5.3 Quality of Incremental Results

We partition the urban road network into a relatively large number of small sized partitions and consider them as the building blocks for the beginning time stamp.

The proposed dynamic maintenance approach processes only the recently changed portions for a real-time response. Alternatively, it can be achieved by re-partitioning the network at regular time intervals. As opposed to the dynamic approach, the static re-partitioning approach processes the whole network repeatedly, and its heavy computations make it unacceptable for real-time response. Here we evaluate the quality of our dynamically obtained results (dynamic approach) by comparing them with those obtained by direct application (re-partitioning approach) of two other methods on the considered dataset- i) the two-stage α-Cut [1], and ii) an adapted two-stage version of [4] (`Ji-adapted`). The original method of [4] is not scalable for large networks, and therefore we adapt this method by adding an extra stage, same as [1], before the application of [4]. Fig. 6 shows this comparison on `Ms`, `M1` and `M2` datasets in terms of F_P, *ANS*, *intra* and *inter*. The measure for the proposed method is computed by incrementally maintaining a set of 200 building blocks in the physical layer and obtaining a set of 10 partitions from the logical layer at each of the shown time stamps. The results shown on the `Ms` dataset refer to the real scats traffic data on 03-12-2012 (Monday). We observe that the F_P measure is mostly between 0.7 and 0.8, which indicates a high relative accuracy of our dynamic incremental partitioning method. A low measure of F_P means that the dynamically obtained results are dissimilar to that of the assumed ground truth. As this is not the actual ground truth, sometimes even after having a low F_P measure, the results



(a) F_P on Ms (b) F_P on M1 and M2

(c) ANS on `Ms`

(d) ANS on `M1` (e) ANS on `M2`

(f) Intra on Ms (g) Inter on Ms

Fig. 6. Partitioning quality.

3. It can be accessed through http://mntg.cs.umn.edu/tg/

TABLE 3
Running Time (in Seconds)

Number of maintained building blocks		Dataset				
		Ms	M1	M2	M3	M4
100	Logical layer	<1	<1	<1	<1	<1
	Physical layer	<1	1	72	129	263
200	Logical layer	<1	<1	<1	<1	<1
	Physical layer	<1	1	86	184	397
400	Logical layer	<1	<1	1	1	1
	Physical layer	<1	1	96	218	451
600	Logical layer	1	1	1	1	2
	Physical layer	<1	1	117	265	570
800	Logical layer	3	3	5	7	9
	Physical layer	1	1	145	307	702
1000	Logical layer	10	12	15	16	19
	Physical layer	1	6	173	354	885
α-Cut [1]		98	129	1905	5907	19736
`Ji-adapted` [4]		101	138	1964	5992	19995

TABLE 4
Running Time without Using Index Structures (in Seconds)

Number of maintained building blocks		Dataset				
		Ms	M1	M2	M3	M4
100	Physical layer	1	2	117	225	490
200	Physical layer	1	2	138	328	753
400	Physical layer	1	3	163	357	816
600	Physical layer	1	3	186	487	1084
800	Physical layer	2	3	231	599	1311
1000	Physical layer	2	10	297	673	1732

may actually be very good in quality, as is evident from the following ANS-based results. In Fig. 6(c), 6(d), and 6(e), we observe that the ANS curve of the proposed method is most of the time higher but close to that of the α-Cut and `Ji-adapted`, which means that the incrementally obtained partitions by the proposed method are not significantly inferior than those obtained by partitioning the urban road network directly using α-Cut and `Ji-adapted`. Sometimes the proposed method achieves even better quality than the existing methods. This is due to some randomness coming from the traffic data and the k-means used in spectral clustering after eigen-decomposition in ACut and Ji-adapted. Also, the ANS measure is always less than 1 for the real `Ms` dataset and most of the times less than 1 for the `M1` and `M2` datasets, which indicates that our results are still considerably good in quality. The main objective of the proposed dynamic method is to maintain the partitions efficiently, which lacks in the existing urban road network (or graph) partitioning algorithms.

5.4 Efficiency of Incremental Computations

Table 3 shows the average running time[4] of the logical layer for producing partitions from the building blocks and that of the physical layer for incrementally updating the building blocks at each time stamp, for all the considered datasets. Its shows the running time by varying the number of maintained building blocks, in comparison to α-Cut and `Ji-adapted`. We observe that the lower the number of maintained building blocks, the faster is the method. The logical layer performs faster because it has to partition a smaller graph where the number of nodes is the number of maintained building blocks, and the physical layer is faster because of lesser overhead. When there is a large number of building blocks, it creates a large number of unstable and noisy road segments lying on the boundaries, which too often shift themselves from one block to another, leading to an overhead. Therefore selecting the right number of building blocks for an application environment is important for the method to have stable building blocks and partitions.

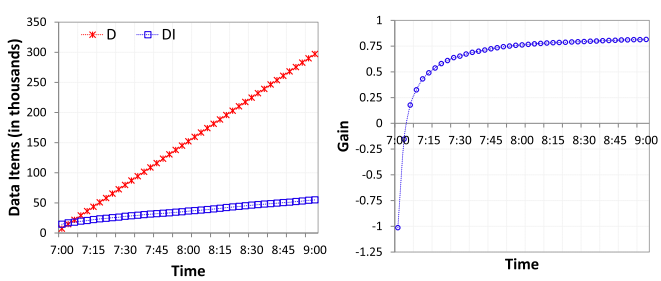
4. On a Core i5 computer with 8 GB RAM.

The appropriate number of blocks is to be chosen based on the computing environment and the dataset size, in such a way that the computations are homogeneously divided into the physical and logical layer. It is also based on requirements from the user end about the size of blocks, one would be interested to see inside the partitions. In our case, we found 200 blocks as a reasonable number. The longest running time for the real dataset `Ms` (Melbourne road network) is just a few seconds, which shows its applicability for the real urban road networks. It can also be performed in fractions of a second by maintaining less number of blocks. The longest running time shown in the table is around 6 minutes for the largest dataset (`M3`) on an ordinary PC, which may drastically improve on a high performance computer. Moreover it can also be performed in shorter time with less number of blocks. Comparing these times with the existing repartitioning method ([1], [4]), we observe that even our longest running time is significantly lower than the existing method. Thus it suggests that our method can be effectively used with real traffic management systems by correctly setting its parameters to deal with the real situation in urban-scale road networks.

Table 4 shows the running time of physical layer of the proposed method implemented without using the proposed index structures (Bin, stability tree, and short cycle index). We observe that it takes 1.5 to 2 times longer running time. The running time for the logical layer remains same in both the versions, as it does not use any index. It demonstrates the effectiveness of our index structures in improving the efficiency.

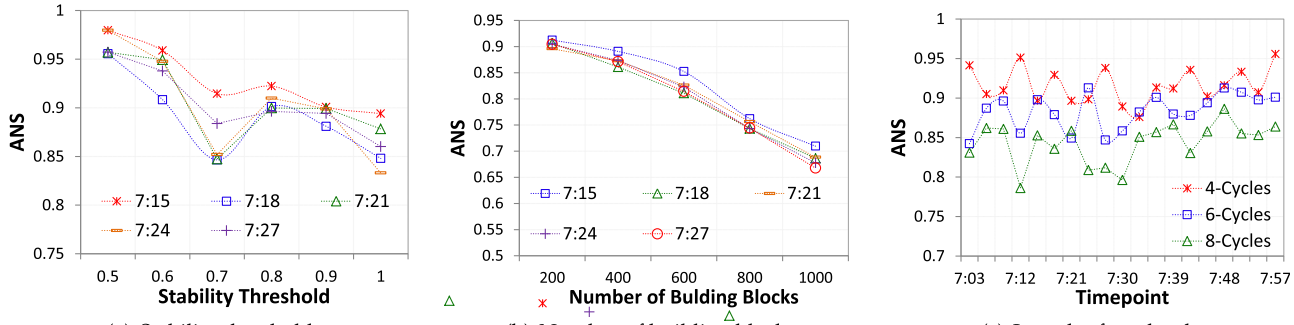
5.5 Memory Consumption in `Bin`

Without using any index, a system would simply dump the sets of building blocks at each new time stamp, which would proportionally keep on increasing the memory usage with lots of redundant information. In contrast, our `Bin` stores the complete information with minimum redundancy. The compactness of our index is quantified by the measure $gain = \frac{D-DI}{D}$, where D is the number of data items



(a) Stored data items (b) Gain

Fig. 7. Memory consumption in `Bin`.



(a) Stability threshold (b) Number of building blocks (c) Length of roadcycle

Fig. 8. Effects of external parameters.

stored in memory when no index is used, and DI is the number of data items stored in memory using Bin. Fig. 7(a) shows the number of data items stored for maintaining 100 building blocks of the Ms dataset, and Fig. 7(b) shows our gain in memory consumption. The gain starts with even less than -1, but keeps on increasing with time and crosses 0.8 within two hours. The reason for a negative gain in the beginning is that it needs to store the extra information in Bin.TL and Bin.NI in addition to the blocks in Bin.Tr. For the Ms dataset, the number data items stored in these components in the first time stamp are 7245 (for 7245 road segments), 100 (for maintaining 100 blocks), and 7246 (road segments + root node in the tree), respectively. Over the period of time, Bin.NI consumes the same space, and Bin.TL and Bin.Tr keep increasing, but in a slow rate.

5.6 Effects of External Parameters on Blocks

Fig. 8 shows the effects of stability threshold ϵ_{stab}, the number of incrementally maintained building blocks n_b, and the roadcycle length l. The results are shown from experiments on the real data Ms. Fig. 8(a) shows the ANS measures of the set of building blocks at varying ϵ_{stab} at five different time stamps. Observe that the quality of results are best at $\epsilon_{stab} = 0.7$ and 1. When it is 1, all the boundary nodes are considered unstable processed once, because of which it gives good results. Interestingly 0.7 also produces good results for all the time pints, which may be because nodes above this ϵ_{stab} are very stable, and processing them worsens the situation until all the nodes in the stability tree are processed. Similarly, Fig. 8(b) shows the ANS measure of the building blocks at five different time stamps when the number of such maintained blocks are from 200 to 1000.

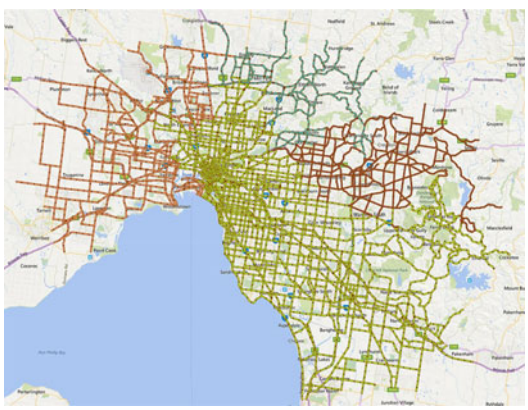


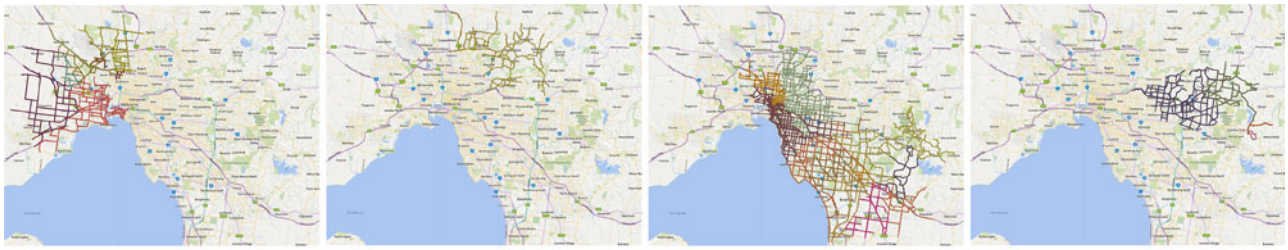
Fig. 9. Partitions in the Logical layer at 07:09 AM (Best viewed in color).

Observe that the more the number of partitions maintained, the better is the quality. Fig. 8(c) shows the ANS measure of the building blocks obtained by setting the roadcycle length l to 4, 6, and 8. There exists 26622, 165971, and 959528 roadcycles respectively for each value of l in the considered network. As the odd length roadcycles are very rare to find, we test with only even lengths. We observe that the quality of results generally gets better upto some extent with increasing cycle length. The reason for this behavior is that the cycles with high l include all the shorter cycles, and thus check the homogeneity in a larger region. With a small l, it checks only a very local region, and may lead to results that are not very good with respect to the global data. However, exceptional situations may occur in case of abrupt changes in the traffic. One such example is 07:24 AM, where 4-roadcycles give better result than 6-roadcycles.

5.7 Visualization Comparison with Google Traffic

This section demonstrates the usefulness of the proposed framework in traffic visualization in an informative way. Google Traffic (Fig. 1) simply visualizes a heatmap of the traffic data on individual road segments. It does not do any further mining or processing of the raw data. Its limitations are, i) there is no information about how the multiple independent or linked congestions are spread, and ii) very limited features to understand the temporal evolution. In contrast, our framework processes the raw traffic data with the proposed method to produce an information-rich visualization shown in Figs. 9 and 10. To keep the pictures simple and easy to understand, only one direction[5] of traffic flow is shown. Fig. 9 shows 4 color-coded partitions from the Ms at 07:06 AM computed in the logical layer. It gives a high level information of the traffic scenario at this time. Each single color has homogeneous traffic inside. If we are interested further, we can zoom-in into the partitions to see the actual building blocks being maintained in Bin in the physical layer. Fig. 10 shows the building blocks of the four partitions. A large number of maintained building blocks would produce the zoomed-in view in a fine granularity. Looking into the partitions and building blocks, it is easy to understand how the congestion is spread. Some partitions or building blocks may be congested, others may be non-congested (can be identified from the color codes). There may be multiple independent congestion blocks in different regions at the same time. By

5. The framework considers the two traffic directions as separate connected road segments.



(a) Building blocks in Partition 1 (b) Building blocks in Partition 2 (c) Building blocks in Partition 3 (d) Building blocks in Partition 4

Fig. 10. Building blocks in the Physical layer at 07:09 AM (Best viewed in color)

scrolling over the temporal domain, the evolution in those congestion blocks can be seen as snapsots in adjacent time-frames. A common phenomenon is that different independent congestions start growing from different regions in busy hours, merge with others to form a big congestion in a the peak time, and then gradually disperse back again. In our study on this dataset, we found that the biggest partiton in light green starts shrinking from the south as time passes by. At 08:06 AM the dark brown partition shifts to the south, the blue partition covers the north and north-west regions, and the light green partition covers the CBD and nearby regions.

6 Related Work

Evolution in Road Traffic Networks: The evolution of specific events or characteristics in complex dynamic networks is studied from different perspectives in different networks. In road networks, studying the evolution of traffic congestion is an important problem for a smartly managed transportation. While the spatial partitioning of an urban road network to identify the differently congested individual sub-networks is being studied [1], [4], tracking their evolution is yet to receive attention. In our previous work [9], we identify the urban road network partitions using [1] and track the congested partitions by considering only the similarity in feature values. We show that the incremental maintenance of partitions can be used for effective traffic visualization in platforms like `Google Traffic`. In our recent preliminary work [11], we develop a two-layer method to track the evolution of congestion using the concepts of road network motifs, which we in the current paper for a comprehensive study. [16] investigates the spatiotemporal relation of congested road segments, and performs an empirical observation on the propagation of congestion. [10] incrementally clusters the spatial data streams collected from sensors. They first predict the clusters roughly using the previous clustering results, and then refine them further in the next stage. In [17], we identify the road segments having high influence in propagating congestion in a road network. At each point of time we incrementally update the road segment influence scores and rank them.

The time series data of road networks are often modelled using Markov models for future traffic predictions [5], [6]. [5] models the travel-cost time series of road segments using a Spatiotemporal Hidden Markov Model and learns its parameters from historical travel data. Using the real-time travel data, the learned model infers near future travel costs, with an aim to be used for fastest route computation. [6] takes into account the historical traffic and weather conditions to mine a landmark graph in the cloud offline. Then in the online process, it models the traffic condition time series of landmarks as an mth-order Markov chain after discretizing the real-time traffic condition measures into different states, and infers future traffic conditions. Another line of research discovers the spatiotemporal causal properties in traffic data streams [7], [8]. [7] first builds a region graph, where regions are identified by an image segmentation method, and links are created based on the transitions in traffic data. Thereafter the outliers are detected from the links, and the outlier causal trees are constructed by analyzing the adjacent time frames. [8] discovers of causal structures by considering time-varying properties with dynamic Bayesian network, and define a causal boundary.

Spatial Network Partitioning: Spatial partitioning of urban road networks is the first step of our framework, which leads to identification of the distinguishable partitions. The well studied graph partitioning lays the foundation of this problem. In [4], the authors proposed a normalized cut based method for spatial partitioning of transportation networks. Their method suffers from high time and space complexity issues for large urban road networks [1], [2]. The two heuristic methods proposed in [18] works fine for small-sized networks, but the running time increases significantly with the increase in network size. We developed a scalable road network partitioning method to identify the differently congested partitions [1]. The method starts with constructing a road graph from the given road network, followed by mining a road supergraph, and then partitioning the supergraph. The actual road network partitions are then extracted from the supergraph partitions by mapping them to the road network. The partitioning of the supergraph is done by optimizing a measure called α-Cut, by following a spectral clustering based solution. We extended the α-Cut algorithm to develop a further efficient algorithm called `FaDSPa` in [2] using density based clustering concepts. There also exist other works that treat road network partitioning as a secondary problem to solve some other problems of primary concern [3], [19], [20].

Efficient spatial query processing is another related research area in road networks. In [21], an efficient and scalable index called `G-Tree` is proposed to support shortest path, kNN, and keyword-based kNN queries. Another work [22] proposed an efficient method to process collective spatial keyword queries (CSKQ) considering the actual road network distance.

7 Conclusion

In this paper, we proposed a two-layer based comprehensive framework to incrementally update the differently congested partitions of an urban road network in an efficient manner, and thus capture the spatiotemporal evolution of

traffic congestion. We also proposed Bin, an in-memory index for a compact storage and efficient retrieval of the historical building blocks. We conducted extensive experiments on real and synthetic datasets to demonstrate the efficacy of our method. The proposed method significantly outperforms the existing re-partitioning methods in terms of efficiency, with a small sacrifice on accuracy. The in-memory building block index saves a significant amount of memory space to store the history. Thus, the proposed framework can effectively serve real traffic management systems for continuously monitoring the evolution of congestion and aid in smart transportation services.

Acknowledgments

This research was supported by Data61, ARC DP140103499, DP160102412, and FT120100723.

References

[1] T. Anwar, C. Liu, H. L. Vu, and C. Leckie, "Spatial partitioning of large urban road networks," in *EDBT*, 2014, pp. 343–354.
[2] T. Anwar, C. Liu, H. L. Vu, and C. Leckie, "Partitioning road networks using density peak graphs: Efficiency vs. accuracy," *Inf. Syst.*, vol. 64, pp. 22–40, 2017.
[3] B. Zhang, K. Xing, X. Cheng, L. Huang, and R. Bie, "Traffic clustering and online traffic prediction in vehicle networks: A social influence perspective," in *Proc. INFOCOM*, 2012, pp. 495–503.
[4] Y. Ji and N. Geroliminis, "On the spatial partitioning of urban transportation networks," *Transp. Res. Part B: Methodological*, vol. 46, no. 10, pp. 1639–1656, 2012.
[5] B. Yang, C. Guo, and C. S. Jensen, "Travel cost inference from sparse, spatio temporally correlated time series using Markov models," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 769–780, Jul. 2013.
[6] J. Yuan, Y. Zheng, X. Xie, and G. Sun, "Driving with knowledge from the physical world," in *Proc. ACM SIGKDD*, 2011, pp. 316–324.
[7] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xie, "Discovering spatio-temporal causal interactions in traffic data streams," in *Proc. ACM SIGKDD*, 2011, pp. 1010–1018.
[8] V. W. Chu, R. K. Wong, W. Liu, and F. Chen, "Causal structure discovery for spatio-temporal data," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2014, pp. 236–250.
[9] T. Anwar, H. L. Vu, C. Liu, and S. P. Hoogendoorn, "Temporal tracking of congested partitions in dynamic urban road networks," *Trans. Res. Record: J. Trans. Res. Board*, vol. 2595, pp. 88–97, 2016.
[10] L.-Y. Wei and W.-C. Peng, "An incremental algorithm for clustering spatial data streams: exploring temporal locality," *Knowl. Inf. Syst.*, vol. 37, no. 2, pp. 453–483, 2013.
[11] T. Anwar, C. Liu, H. L. Vu, and M. S. Islam, "Tracking the evolution of congestion in dynamic road networks," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 2323–2328.
[12] M. Aftabuzzaman, "Measuring traffic congestion- a critical review," in *Proc. 30th Australasian Transp. Res. Forum*, 2007, https://trid.trb.org/view/855242
[13] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: Simple building blocks of complex networks," *Sci.*, vol. 298, no. 5594, pp. 824–827, 2002.
[14] M. F. Mokbel, et al., "MNTG: An extensible web-based traffic generator," in *Proc. 13th Int. Conf. Adv. Spatial Temporal Databases*, 2013, pp. 38–55.
[15] E. Amigó, J. Gonzalo, J. Artiles, and F. Verdejo, "Combining evaluation metrics via the unanimous improvement ratio and its application to clustering tasks," *J. Artif. Int. Res.*, vol. 42, no. 1, pp. 689–718, 2011.
[16] Y. Ji, J. Luo, and N. Geroliminis, "Empirical observations of congestion propagation and dynamic partitioning with probe data for large-scale systems," *Transp. Res. Record*, vol. 2422, pp. 1–11, 2014.
[17] T. Anwar, C. Liu, H. L. Vu, and M. S. Islam, "Roadrank: Traffic diffusion and influence estimation in dynamic urban road networks," in *Proc. CIKM*, 2015, pp. 1671–1674.
[18] H. Etemadniaa, K. Abdelghanya, and A. Hassan, "A network partitioning methodology for distributed traffic management applications," *Transportmetrica A: Transport Sci.*, vol. 10, no. 6, pp. 518–532, 2014.
[19] Z. Xu and H.-A. Jacobsen, "Processing proximity relations in road networks," in *Proc. ACM SIGMOD*, 2010, pp. 243–254.
[20] G. Kellaris and K. Mouratidis, "Shortest path computation on air indexes," *Proc. VLDB Endow.*, vol. 3, no. 1/2, pp. 747–757, Sep. 2010.
[21] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong, "G-tree: An efficient and scalable index for spatial search on road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 8, pp. 2175–2189, 2015.
[22] Y. Gao, J. Zhao, B. Zheng, and G. Chen, "Efficient collective spatial keyword query processing on road networks," *IEEE Trans. Intell. Transp. Syst.*, vol. 17, no. 2, pp. 469–480, Feb. 2016.

Tarique Anwar received the master's and PhD degrees in computer science from Jamia Millia Islamia, India, in 2010, and the Swinburne University of Technology, Australia, in 2017, respectively. Currently, he is an assistant professor with the Indian Institute of Technology Ropar, India. His research interests include data management, spatiotemporal data mining, road traffic networks, and social networks.

Chengfei Liu received the BS, MS, and PhD degrees in computer science from Nanjing University, China in 1983, 1985, and 1988, respectively. Currently, he is a professor with, Swinburne University of Technology, Australia. His research interests include keywords search on structured data, query processing and refinement for advanced database applications, query processing on uncertain data and big data, and data-centric workflows. He is a member of the IEEE and ACM.

Hai L. Vu received the BSc, MSc, and PhD degrees in electrical engineering from the Technical University of Budapest, Hungary, in 1994 and 1999, respectively. Currently, he is a professor and Australian Research Council (ARC) future fellow in the area of Intelligent Transport Systems (ITS) with Monash University, Australia. His research interests include performance analysis and design of wireless networks, and stochastic optimization with applications to intelligent transport systems. He is a senior member of the IEEE.

Md. Saiful Islam received the BSc (Hons) and MS degrees in computer science and engineering from the University of Dhaka, Bangladesh, in 2005 and 2007, respectively, and the PhD degree from the Swinburne University of Technology, Australia, in February 2014. He is currently a lecturer with Griffith University, Australia. His research interests include database usability, data analytics, exploration, and big and spatial data management. He is a member of the IEEE and ACM.

Timos Sellis received the PhD degree in computer science from the University of California, Berkeley, in 1986. He is a professor and the director of the Data Science Research Institute, Swinburne University of Technology, Australia. Between 2013 and 2015, he was a professor with RMIT University, Australia, and before 2013 the director of the Institute for the Management of Information Systems (IMIS) and a professor with the National Technical University of Athens, Greece. His research interests include big data, data streams, personalization, data integration, and spatio-temporal database systems. He is a fellow of the IEEE and ACM.