Tracking the Evolution of Congestion in Dynamic Urban Road Networks

Tarique Anwar*,†, Chengfei Liu‡, Hai L. Vu§ and Md. Saiful Islam#4

*‡Swinburne University of Technology, Melbourne, Australia
‡‡Data61, CSIRO, Melbourne, Australia
§§La Trobe University, Melbourne, Australia
*†tanwar, ‡ctt, ‡§hu@swin.edu.au, ‡¶m.islam5@latrobe.edu.au

ABSTRACT

The congestion scenario on a road network is often represented by a set of differently congested partitions having homogeneous level of congestion inside. Due to the changing traffic, these partitions evolve with time. In this paper, we propose a two-layer method to incrementally update the differently congested partitions from those at the previous time point in an efficient manner, and thus track their evolution. The physical layer performs low-level computations to incrementally update a set of small-sized road network building blocks, and the logical layer provides an interface to query the physical layer about the congested partitions. At each time point, the unstable road segments are identified and moved to their most suitable building blocks. Our experimental results on different datasets show that the proposed method is much efficient than the existing re-partitioning methods without significant sacrifice in accuracy.

Keywords

Road network motif, Incremental partitioning, Tracking congestion evolution, Urban road networks

1. INTRODUCTION

The roads of different localities usually 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. It leads to the fact that the different small sub-networks (of small areas like suburbs) of a large urban road network experience distinctive traffic flow within them. Previous works have applied congestion based partitioning of road networks to identify such sub-networks called spatial partitions or simply partitions [2]. 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 on urban road networks. For example, during the morning office-opening hours, the congestion generally starts developing in the outer suburbs and the roads connecting them to the city center, mostly occur inside the city during the day, and starts moving outwards again during the office-closing hours. The congestion can be simply understood as a congested partition that keeps on changing its structure, location, and level of congestion.

The maintenance and tracking of the differently congested partitions can potentially aid traffic management systems [4] and traffic visualization platforms like Google Traffic [3]. A naive way to track the change in partitions is to perform spatial partitioning of the road network at each time point, and analyze the change. But a complete re-partitioning is a computationally expensive task. Generally in successive time points, the traffic does not change abruptly, rather it is a gradual process. A logically better way is to incrementally update the previously obtained set of partitions by processing only the sections of probable change. It heavily reduces the computations, while may sacrifice the quality of partitions marginally. There exist works on the complete partitioning of road networks [4, 2], and some other related problems, including spatiotemporal propagation of congestion [5], identification of important road segments having high influence in propagating congestion [1], and incremental clustering of spatial data streams collected from sensors [7]. [3] identifies the congested partitions, and performs an experimental analysis by updating the partitions simply based on similarity in the traffic.

In this paper, we propose a two-layer method to incrementally compute the differently-congested partitions of a road network and use them to track the evolution of congestion. The physical layer performs low-level computations for the incremental update, whereas the logical layer presents the required congestion related information to the user after a light makeover. Our method in the physical layer starts with a set of building blocks (defined in Section 3.2) of the road network at the beginning time point. At each new time point, the unstable road segments are identified using a stability measure, indexed as a heap tree, and moved to their most suitable building blocks using the concepts of road network motifs.

2. PROBLEM DEFINITION

Definition 1: (Road Network) An urban road network is defined as $\mathcal{N} = (\mathcal{I}, \mathcal{R})$ comprising a set of intersection points $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ as nodes, which are connected among themselves by a set of directed road segments.
\( \mathcal{R} = \{r_1, r_2, \ldots, r_n\} \) as links, where each road segment \( r_i \) associates a measure of traffic density \( r_i.d \) with itself. ■

**Definition 2:** (Road Graph) Given a 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 \( u_i \) 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 \( v_i (\text{node}(r_i)) \in \mathcal{V} \) associates with it a feature value \( v_i.f \), which is the road traffic density \( r_i.d \).

**Definition 3:** (Partition) A given road network \( \mathcal{N} \) can be partitioned into multiple segments, each of whom is 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 \( \mathcal{P}_i \cap \mathcal{P}_j = \emptyset \) 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 a road network can be transformed to that of a road graph by following Definition 2, and vice versa.

**Definition 4:** (Intra-Partition Associativity) For a given set of partitions \( \mathcal{P} \), the intra-partition associativity is defined as the aggregation of the traffic similarity between all possible pairs \((r_i, r_j)\) for which \( r_i \) and \( r_j \) lie in different partitions.

**Definition 5:** (Intra-Partition Associativity) For a given set of partitions \( \mathcal{P} \), intra-partition associativity is defined as the aggregation of the traffic similarity between all possible linked pairs \((r_i, r_j)\) for which \( r_i \) and \( r_j \) lie in the same partition.

**Problem:** Let us suppose, we are given a set of road network partitions \( \mathcal{P}^{i-1} = \{\mathcal{P}^{i-1}_1, \mathcal{P}^{i-1}_2, \ldots, \mathcal{P}^{i-1}_k\} \) based on the traffic at time \( t_{i-1} \), such that \( \mathcal{P}^{i-1} \) has a minimum possible inter-partition associativity and a maximum possible intra-partition associativity. Our objective is to incrementally update \( \mathcal{P}^{i-1} \) to \( \mathcal{P}^i = \{\mathcal{P}^i_1, \mathcal{P}^i_2, \ldots, \mathcal{P}^i_k\} \) at each new time point \( t_i \), based on the respective traffic data, without re-partitioning the whole network, in such a way that the properties of inter-partition and intra-partition associativities are maintained.

3. PROPOSED METHOD

The proposed method tracks the evolution of traffic congestion over a period of 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 point 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** incrementally updates a large number of small-sized building blocks at each new time point.

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 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 \( \mathcal{B} = \{B_1, B_2, \ldots, B_n\} \). The logical layer constructs a building block graph \( \mathcal{G}^b = (\mathcal{V}^b, \mathcal{E}^b) \), where each building block forms a node \( \mathcal{v}_i \in \mathcal{V}^b \) and all pairs of neighboring (defined later) building blocks or nodes \( \langle \mathcal{v}_i, \mathcal{v}_j \rangle \in \mathcal{E}^b \) are connected by links \( e_i \in \mathcal{E}^b \). The number of nodes in this graph is much smaller than that in the road graph \( (|\mathcal{V}| = n_b < n_v) \). The nodes are first assigned their feature values \( \mathcal{v}_i.f \) as the mean of the corresponding building block. The links \( e_i \) between nodes \( \mathcal{v}_i \) and \( \mathcal{v}_j \) are weighted by the similarity between \( \mathcal{v}_i.f \) and \( \mathcal{v}_j.f \) as in [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, \( \alpha \)-Cut [2] is one such algorithm to do this task. Those with high means are considered as congested, and those with low means are considered as non-congested. 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, as the most fundamental constituents maintained in the physical layer.

**Definition 6:** (Building Block) Given a road graph \( \mathcal{G} \), a building block \( B_i = (\mathcal{V}_i, \mathcal{E}_i) \) is defined as a subgraph that forms one of the \( n_b \) fundamental constituents \( \mathcal{B} = \{B_1, B_2, \ldots, B_{n_b}\} \) in the physical layer at a time point, such that \( \bigcup_{i=0}^{n_b} \mathcal{V}_i = \mathcal{V} \) and \( \mathcal{V}_i \cap \mathcal{V}_j = \emptyset \) for all \( i \neq j \).

The building blocks are small constituents that represent the differently congested blocks of the spatial road network at a point of time in a fine granularity. Two blocks \( B_i \) and \( B_j \) are called neighbors, if there exist nodes \( v_i \) and \( v_j \) such that \((v_i, v_j) \in \mathcal{E}_i \) and \( v_i \in B_i \land v_j \in B_j \). A node \( v_i \in B_i \) is called a boundary node, if there exists another node \( v_j \) such that \((v_i, v_j) \in \mathcal{E}_i \) and \( v_j \notin B_i \). Each block has a set of spatially linked blocks as neighbors and a set of boundary nodes. The continuously changing traffic conditions affect their structure 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 blocks in fine granularity to effectively represent the traffic condition. At each subsequent time point their structure keeps changing, as much as the variation in network traffic, and the congestion hotspots keep evolving. A quality building block \( B_i \) is characterized by two properties: high homogeneity (in terms of their feature values) amongst the nodes inside, and a stable structure that tends to change the least with time. We capture the evolution of network congestion by tracking the evolution of blocks in a low level. The blocks for the beginning time point are mined by partitioning the network based

1. Building block and block are used synonymously hereafter.
on historical traffic data, and the proposed method in this paper starts tracking from this initial set.

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 [2], and consider the obtained partitions as the building blocks for the starting point. At each new time point, the most recent traffic data is fetched, based on which these blocks are incrementally updated by identifying and processing the unstable road segments. For any query being passed from the logical layer, it returns back the results instantly.

4. BUILDING BLOCK EVOLUTION

We start this section with some fundamental concepts, and then explain the algorithm working in the physical layer to incrementally update the building blocks.

4.1 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 7: (Node Stability) If a node \( v_i \) belongs to block \( B_j \) at time \( t_{i-1} \), then the stability \( \text{stab}'(v_i) \) is defined as the likelihood of \( v_i \) to remain in the same block \( B_j \) at time \( t_i \).

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 \( B_j \) at the current time \( t_i \); and ii) temporal stability, which looks into how much stable was \( v_i \) in the previous time points. Equation 1 shows the formulation to compute its measure, where \( \mu_j \) denotes the mean feature value inside \( B_j \) at time \( t_i \). The formula is an average of two quantities – the first one \( \text{stab}^{(r-1)}(v_i) \), which is its stability measure from the previous time point \( t_{i-1} \), stands for the temporal stability, and the second quantity, which is from the current time point \( t_i \), stands for the spatial stability. The second quantity firstly 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.

\[
\text{stab}^{(r)}(v_i) = \frac{\text{stab}^{(r-1)}(v_i) + (1 - \text{abs}(v_i - \mu_j)/\text{factor})}{2}
\]

\[
\text{factor} = \text{max}\{p_j^m, v_j^\text{min}\}, (v_j^\text{max} - p_j^m)
\]

Stability Tree: The blocks have one set of nodes lying on the boundaries (boundary nodes) and another set lying.
Algorithm 1: IncrementalUpdate(Block Set $B$, Short cycle index $SCI$)

1. Compute stability of boundary nodes at $t_i$;
2. Stability tree $ST ←$ Create a max-heap tree of boundary nodes;
3. Node $v_{root} ←$ Delete root of $ST$;
4. while $\text{stablity}(v_{root}) ≤ \epsilon_{stab}$ do
5.    Block $b_{current} ←$ Current block of $v_{root}$ in $B$;
6.    $setree ← SCI[v_{root}]$;
7.    Block $b_{next} ←$ IdentifyMSB($B$, $setree$);
8.    if $b_{next} ≠ b_{current}$ then
9.       $b_{current} ← b_{current}\setminus \{v_{root}\}$;
10.      $b_{next} ← b_{current} \cup \{v_{root}\}$;
11.      Update $ST ←$
12.         AddRemoveSTNodes($B$, $ST$, $v_{root}$, $b_{current}$, $b_{next}$);
13. return $B$;

bottom is shared between the two blocks. This observation leads to the fact that the number of occurrences of bounded road cycles inside each block is significantly higher than those across different blocks. These bounded cycles thus give information about the block structure and can be used to locate their suitable boundaries. 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 block structures.

Short Cycle Index: We pre-compute all the possible road cycles of path length smaller than or equal to $\epsilon_{path}$ in the road graph as an offline task and index them as follows. Firstly a sorted set of all nodes in $V$ is created based on their id. For each node $v_i \in V$, all the cycles that involve $v_i$ in the path are computed. Let us suppose $\{(v_i, v_a, v_b, v_e), (v_i, v_a, v_c, v_e), (v_i, v_d, v_e, v_i), (v_i, v_d, v_f, v_i)\}$ is the set of road 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 $v_a$, 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 $v_a$, $v_c$, $v_e$, and $v_f$. In this way the trie trees corresponding to all $v_i \in V$ are created and attached to the sorted set. This structure is called short cycle index (denoted by $SCI$).

4.3 Incremental Update

The incremental update algorithm looks into all the unstable nodes and moves them to the most suitable blocks at each time point. Let us suppose we have a given set of blocks $B$ at time $t_{i-1}$ and the short cycle index $SCI$. The complete algorithm to incrementally compute the blocks at time $t_i$ is shown in Algorithm 1. It starts with computing the stability measure from the traffic data at $t_i$ for all the boundary nodes and creating the stability tree $ST$ (lines 1-2). The iterative steps of computing the most suitable block for the most unstable boundary node and updating $B$ are carried out until all the unstable boundary nodes having their stability measure less than the threshold $\epsilon_{stab}$ have been processed (lines 3-12). After getting the most unstable node $v_{root}$ by deleting the root of $ST$ (line 3), its current block $b_{current}(\in B)$ and the short cycle tree $setree$ (from $SCI$) are accessed (lines 5-6). The most suitable block $b_{next}(\in B)$ is computed using the function IdentifyMSB(.) described later in Algorithm 2 (line 7). This computation is based on the new traffic data at $t_i$ in contrast to the current block $b_{current}$ based on the data at $t_{i-1}$. If the new block is different than the existing one (line 8), then $v$ is deleted from $b_{current}$ (line 9) and inserted into $b_{next}$ (line 10). It completes the update of $B$ for $v_{root}$. 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 using AddRemoveSTNodes(.) (line 11). This function first gets all the nodes $u$ linked to $v_{root}$ in the road graph. If the block of $u$ is same as the block of $v_{root}$ before the update ($b_{current}$), it means that $u$ is a boundary node. If this $u$ does not exist in $ST$, being a new boundary node it is added to the tree. If the block of $u$ is same as the block of $v_{root}$ after the update ($b_{next}$), it means that $u$ may not be a boundary node anymore. If this $u$ exists in $ST$ and does not lie on the boundary, being a new internal node it is removed from the tree. After completing the processing of $v_{root}$, the next most unstable node is extracted from $ST$ (line 12) and the same process (lines 4-12) is carried out repeatedly until all the unstable nodes have been processed.

4.4 Computing the most suitable block

As mentioned earlier in Section 4.2, the short length bounded road cycles are likely to be found in significantly large numbers within the 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 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 $\epsilon_{path}$, 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.

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

$$W(v, B_j) = \sum_{\forall C \in RCycles(v)} \frac{1}{\text{pathlength}(C) - 1} \quad (u \in C) \land (u \in B_j)$$

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 $B_j$, the weight for $B_j$ is incremented by $\frac{1}{\text{pathlength}(C) - 1}$. For example, if there is a cycle $\{v_1, v_2, v_3, v_4\}$, where $v_1(\in B_1)$, $v_2(\in B_2)$, $v_3(\in B_3)$, and $v_4(\in B_4)$, 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 $W(v_1, B_1)$, $v_2$ and $v_4$ both belonging to $B_1$ adds up to $\frac{1}{3}$, and the weight $\frac{1}{3}$ of $v_3 \in B_3$ is added to $W(v_1, B_j)$. In this manner, the weights for all the blocks are computed.
Algorithm 2: IdentifyMSB(Block Set B, Short cycle tree sctree)

1. s1, s2, s3 ← initialize new stack;
2. Node nparent ← sctree(root);
3. forall the Node nchild ∈ sctree(nparent).child do
4. if nchild ≠ φ AND dist(npARENT, nCHILD) ≤ γ then
5. Push nCHILD into s1;
6. repeat
7. Node nparent ← pop out from s1;
8. Push nparent into s2;
9. count = 0;
10. forall the Node nchild ∈ sctree(nparent).child do
11. if nchild ≠ φ AND dist(npARENT, nCHILD) ≤ γ then
12. Push nCHILD into s1;
13. count ← count + 1;
14. Push count into s3;
15. until s1 ≠ empty;
16. s1 ← re-initialize stack;
17. wblock[ ] ← initialize an array of values of size n;
18. repeat
19. Node node ← pop out from s2;
20. countchild ← pop out from s3;
21. if countchild = 0 then
22. Value weight = \( \frac{1}{\text{depth}(\text{node})} \);
23. else
24. Value weight ← initialize with 0;
25. for i ← 1 to countchild do
26. Value childweight ← pop out from s1;
27. weight ← weight + childweight;
28. Push weight into s1;
29. wblock[Block of node ] ← Increment by weight;
30. until s2 ≠ empty;
31. wmax ← maximum value in wblock[ ];
32. msh ← block corresponding to wmax;
33. return msh;

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 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 an 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 \( \{v, v_1, v_2\} \). 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 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 set of blocks \( B \) and the short cycle tree sctree from SCI, and computes the most suitable block (∈ \( B \)) for the 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 starts with pushing all the children of root node of sctree to the stack \( s_1 \), if the Euclidean distance between the parent and the child dist(npARENT, nCHILD) is less than or equal to \( \gamma \) (lines 1-5). The dist(.) function ensures that only the \( \gamma \)-bounded road cycles are explored. It is followed by iterative steps of popping out a node from \( s_1 \) (line 7), pushing it into the stack \( s_2 \) (line 8), pushing all its children back into \( s_1 \) if the parent-child satisfies the \( \gamma \) distance condition (lines 10-13), and pushing the count of these children into the stack \( s_3 \) (line 14). These steps are repeated until \( s_1 \) becomes empty (line 15). They compute the number of children of each node and keep them saved in the stacks for computing the block 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 17). Each value in the array correspond to a block in \( B \) at time \( t_i \). Then the nodes are popped out from \( s_2 \) one after another (line 19), the count of their children are popped out from \( s_3 \) (line 20), 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 in the branch and its depth gives the path length of the cycle. Hence the weight of \( \text{pathlength}(C) - \text{dist} \) that is carried by each node in the cycle \( C \) is computed as \( \frac{\text{pathlength}(C)}{\text{depth}(\text{node})} \) (line 22) and pushed into \( s_1 \) (line 28) to be used later to compute weights for its parents. The depth of the node is found from its leaf node child in sctree 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 \( s_1 \) as many times as the count (lines 23-27), which is again pushed back into \( s_1 \) (line 28). In addition to pushing the weight of nodes into \( s_1 \), the weight for the blocks in the array wblock[ ] is updated by adding weight to the array element corresponding to the block of node in \( B \) (line 29). These steps of popping out from \( s_2 \) and \( s_3 \) and computing the values of wblock[ ] using \( s_1 \) (lines 19-29) are repeated until \( s_2 \) becomes empty (lines 18,30), which marks the completion of processing of all the bounded short road cycles. At last we select the most suitable block msb for the current time \( t_i \) by getting the block corresponding to the maximum weight in wblock[ ] (lines 31-33).

5. EXPERIMENTS

Our datasets, shown in Table 1, include one real (M1) and two semi-synthetic (M1 and M2) datasets. M1 is recorded by the Sydney Coordinated Adaptive Traffic System (SCATS) from the Melbourne road network provided to us by VicRoads\(^2\). M1, and M2 are synthetically generated on the real road network of Melbourne using a web-based\(^3\) random road traffic generator MNTG [6].

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Place</th>
<th>Area(sq ml)</th>
<th>Road seg</th>
<th>Inter pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1 (real)</td>
<td>Melbourne</td>
<td>627.5</td>
<td>7245</td>
<td>2928</td>
</tr>
<tr>
<td>M1</td>
<td>CBD Melbourne</td>
<td>6.6</td>
<td>17,206</td>
<td>10,096</td>
</tr>
<tr>
<td>M2</td>
<td>CBD(+), Melbourne</td>
<td>31.5</td>
<td>53,494</td>
<td>28,460</td>
</tr>
</tbody>
</table>

The incremental partitioning results are evaluated by using the metric ANS (derived from Silhouette) that looks into the inter-partition heterogeneity and intra-partition homogeneity to quantify the overall quality of the obtained partitions [4, 2]. A value less than 1 for this measure indicates a good partitioning, and lower values indicate better partitioning.

Quality of incremental results: We start the experiments by firstly mining a set of blocks that are to be main-

\(^3\)It can be accessed through http://mntg.cs.umn.edu/tg/
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 blocks for an application environment is important for the method to have stable blocks and partitions. The longest running time for the real dataset $M_1$ is just a few seconds (for 1000 blocks), 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. We observe in the table that even our longest running time for any dataset is significantly lower than that of the existing re-partitioning method [2]. It suggests that our method can be effectively used with any real traffic management systems by properly setting its parameters.

### 6. CONCLUSION

In this paper, we proposed a two-layer method to incrementally update the differently congested partitions of a road network in an efficient manner, in order to track the evolution of traffic congestion. The physical layer maintains and incrementally updates a set of small-sized building blocks of the network, and the logical layer provides an interface to query the physical layer about the congested partitions. We conducted extensive experiments on real and synthetic datasets to demonstrate the efficacy of our method. Our method can effectively serve real traffic management systems for a continuous tracking of the evolution of congestion and aid in smart transportation services.

### Acknowledgment

This research was supported by the Data61 (formerly NICTA), ARC Discovery Projects DP140103499 and DP160102412, and Future Fellowship grant FT120100723.

### 7. REFERENCES


