VTeller: Telling the Values Somewhere, Sometime in a Dynamic Network of Urban Systems

Yan Li

University of Massachusetts, Lowell
yan_li1@student.uml.edu

Tingjian Ge

University of Massachusetts, Lowell
ge@cs.uml.edu

Cindy Chen

University of Massachusetts, Lowell
cchen@cs.uml.edu

ABSTRACT

Dynamic networks are very common in urban systems today. As data are acquired, unfortunately, they are rarely complete observations of the whole system. It is important to reliably infer the unobserved attribute values anywhere in the graphs, at certain times—either in the past or in the future. Previous work does not sufficiently capture the correlations inherent with graph topology and with time. We propose a machine learning approach using a novel probabilistic graphical model. We devise a series of algorithms to efficiently group the vertices, to learn the model parameters, and to infer the unobserved values for query processing. Furthermore, we propose a method to incrementally and automatically update the model. Finally, we perform an extensive experimental study using two real-world dynamic graph datasets to evaluate our approach.

CCS CONCEPTS

• Information systems → Data mining;

KEYWORDS

dynamic network; graphical model; inference

ACM Reference Format:

Yan Li, Tingjian Ge, and Cindy Chen. 2018. VTeller: Telling the Values Somewhere, Sometime in a Dynamic Network of Urban Systems. In *The 27th ACM International Conference on Information and Knowledge Management (CIKM '18)*, October 22–26, 2018, Torino, Italy. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3269206.3271778>

1 INTRODUCTION

An important subset of big data has to do with the enormous surge in the growth and expansion of urban regions in the world. The data in urban systems such as transportation, roads and bridges, water and energy systems, and telecommunication networks can often be modeled as dynamic graphs, i.e., graphs/networks that change over time. In this paper, we focus on attribute value changes (of nodes/edges), but not the graph topology changes. Due to resource, time, or monetary constraints, the data and updates that people acquire or observe are typically only a small subset of all. Users often need to query missing data values somewhere at a certain time in the dynamic network. For example, we are working with

civil and mechanical engineering researchers on traffic/driving guidance and control with smart vehicles under severe weather, such as snow storms. Traffic, road, and weather conditions over a large road network form a rich dynamic graph. What can be sensed and detected from smart vehicles and roadway sensors is only a small portion of the overall dynamic network environment. In order to provide travel guidance and traffic control, we need to query the traffic condition at unobserved road segments, or the slippery condition of certain roads at an unobserved time instant, among many other queries.

There are many previous approaches on inferring incomplete data, for example [12, 13, 16, 23]. Most of them, however, do not capture the data correlations both inherent in the *graph topology* and over the *time horizon*. There are a few exceptions (e.g., [20, 21, 25]), with the best performing one recently in [11]. Yet its capture of the two types of correlations is still quite simple and limited. As shown in our experiments (Section 7), even though it is more efficient than our inference algorithm, its inference accuracy is far below our method. Moreover, our query processing algorithm after optimizations is still very fast. Our basic idea is to devise a novel probabilistic graphical model (PGM) that scales and that leverages the topology of the original data graph. Specifically, the PGM topology is derived from the topology of the original data graph. Our model consists of multiple levels of latent variables, each capturing the co-evolution of a small number of nodes over time. We summarize our contributions as follows:

- We propose a novel probabilistic graphical model with layers of latent variables to capture system states that evolve over time and that capture the correlations across both graph topology and time (§3).
- We propose vertex grouping and parameter learning algorithms with analysis of their results and performance (§4).
- We devise a query processing algorithm, along with optimizations, that unrolls a subtree from our model into a dynamic Bayesian network, and that performs inference based on belief propagation [17] (§5).
- We devise an adaptive, incremental, and lightweight model parameter partial update algorithm to solve the problem of dynamic model parameter changes in urban systems (§6).
- We perform a comprehensive experimental evaluation using two real world datasets (§7).

2 PROBLEM STATEMENT

A dynamic graph $\mathbb{G}$ consists of an infinite sequence of graph snapshots $\mathcal{G}_1, \dots, \mathcal{G}_t, \dots$, where $\mathcal{G}_t$ is the graph snapshot at time t . All the snapshots are based on the same underlying graph $G = (V, E)$. That is, all vertices and edges in $\mathcal{G}_t (t \geq 1)$ come from G . In the event that the graph topology also changes over time, we may

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIKM '18, October 22–26, 2018, Torino, Italy

© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6014-2/18/10...\$15.00
<https://doi.org/10.1145/3269206.3271778>

consider G as the *union* of the vertices (resp. edges) of all graph snapshots—thus, this formulation is general enough, and we focus on dynamic attribute value changes. We use the terms *dynamic graph* and *dynamic network* interchangeably.

We consider an attribute a of either each $v \in V$ or each $e \in E$ (i.e., either a vertex attribute or an edge attribute); we call it a *target attribute*. If the target attribute a is an edge-attribute, then logically, there is a corresponding value graph $G_v = (V_v, E_v)$ —each edge in G corresponds to a vertex in G_v , and two vertices in G_v are connected by an undirected edge if their corresponding edges in G intersect at a node. If the target attribute a is a vertex-attribute, then the value graph G_v is the same as G .

EXAMPLE 1. Figure 1(a) shows the original road network graph G , where our target attribute is an edge attribute, i.e., the traffic condition of a road segment edge, such as (1, 2) or (2, 3). Figure 1(b) illustrates a snippet of the corresponding value graph G_v , where we show that the upper 10 road-segment edges in G map to 10 vertices in G_v . An edge is created between two vertices in (b) if the corresponding road-segments in (a) intersect—e.g., the edge between (1, 2) and (1, 5) in (b). In the graph snapshots $\mathcal{G}_1, \dots, \mathcal{G}_t, \dots$, some observed values of the target attribute (traffic condition) arrive at each time step t , in snapshot $\mathcal{G}_t$.

Figure 1: Illustrating a value graph G_v . (a) is the original road network graph G , in which an edge attribute, traffic condition of a road, is our target attribute. (b) is part of the value graph G_v , in which a vertex maps to an edge in G .

In general, there can be correlation between the attribute a values at two adjacent vertices of G_v (topological correlation); there can also be correlation between the a values of a single vertex of G_v at two time instants (temporal correlation). We are given a set of m observed values of the target attribute a within a time interval $T = [\tau_b, \tau_e]$, i.e., the attribute a values at (v_i, t_i) for each $1 \leq i \leq m$, where v_i is a vertex in G_v , $t_i \in T$ is a time instant, $\tau_b = \min_{1 \leq i \leq m} t_i$, and $\tau_e = \max_{1 \leq i \leq m} t_i$. Our goal is to return the value (distribution) of the target attribute a at an unobserved vertex/time pair (v_q, t_q) , where v_q is a vertex in G_v and t_q may be a time instant within T (i.e., $\tau_b \leq t_q \leq \tau_e$), after T (i.e., $t_q > \tau_e$), or before T (i.e., $t_q < \tau_b$). Note that we focus on point queries in this work; they are a building block for other types of queries such as aggregate and join queries, which we leave as future work.

3 THE MODEL

In our solution, we devise a novel probabilistic graphical model. We intend to model the time-variant nature of the whole value graph. To do this in a scalable manner, we use one latent variable to characterize the correlation and evolution of a *small number* of

values in the graph. Thus, we have many such latent variables—these are the first-tier latent variables. For each first-tier latent variable, we use a Markov chain to model its state changes over time, where a *state* is treated as a value of the latent variable. Each first-tier latent variable “emits” a number of values in the graph (through conditional distributions). This is illustrated in Figure 2, where each first-tier latent variable emits five base-level graph values (in a red solid oval).

The intuition is that each first-tier latent variable is not overwhelmed with too much information from the base-level graph values, and that we can have a model of a reasonable size and complexity. Analogously, to model the correlation and co-evolution of the network values at a greater scope, we have the second-tier latent variables, each of which follows a Markov chain and emits a small number of correlated/co-evolving first-tier latent variables. Figure 2 only depicts one second-tier latent variable (the root node) that emits five first-tier latent variables in the dashed oval. For clarity, Figure 2 only shows a small snippet of the graph; we may have more tiers/levels.

Figure 2: Illustrating our probabilistic graphical model.

The intuitions of our whole model are as follows. There is a reason for a real network to have its graph structure—its edges typically contain all important relationships and correlations; otherwise missing edges should be added. There is a large body of work on missing link predictions (e.g., see [14]), which is beyond the scope of this paper. Based on graph topology, our model *directly* encodes stronger correlations of random variables that are closer in the graph, and *indirectly* (at a higher level of the model) encodes weaker correlations of random variables that are distant in the graph. The hierarchical structure of our model also helps inference performance and scalability—which can be run on a local subtree of the whole model.

Note that if we “unroll” each internal node across adjacent time steps (as each internal node follows a Markov chain), our model is essentially a dynamic Bayesian network. This is exactly our treatment in Section 5 for query processing, illustrated in Figure 3.

4 DATA-DRIVEN MODEL LEARNING

4.1 Vertex Grouping

As we model each group of value vertices with a latent variable, it is desirable to have groups with close sizes, but they do not have to be of the exact same size. Intuitively, other conditions being equal, the more variables/vertices in a group, the more complex it is for the latent variable to characterize their state co-evolution over time through a hidden Markov model. This motivates us to have approximately balanced group sizes. Thus, for our specific purpose,

we devise a very efficient algorithm (with a linear time amortized cost), as shown in ONELEVELVERTEXGROUPING. The key idea of the algorithm is to pick the next heaviest weight edges (i.e., most correlated vertices) to assign into a group, and to simultaneously balance the group sizes and ensure that each group size be in $[\alpha, 2\alpha)$, where α is a parameter of a small value (e.g., $\alpha = 5$).

Algorithm 1: ONELEVELVERTEXGROUPING(G)

Input: G : graph, where we add a weight to each edge (u, v) as the Pearson correlation coefficient [23] of the values at u and v

Output: groups of the vertices in G

```

1 initially let each vertex be its own group
2 create a priority queue  $Q$  for all edges  $E$  (weight as priority)
3 while pop  $e$  from  $Q$  with highest weight do
4   if  $e$  connects two groups  $c_1$  and  $c_2$  where  $|c_1| \leq |c_2|$  then
5     if  $|c_1| < \alpha$  then
6       merge  $c_1$  and  $c_2$  into one group  $c_{1,2}$ 
7       if  $|c_{1,2}| \geq 2\alpha$  then
8         let  $e = (v_1, v_2)$  where  $v_1 \in c_1$  and  $v_2 \in c_2$ 
9          $n \leftarrow \lfloor \frac{|c_{1,2}|}{2} \rfloor - |c_1|$  //number of vertices to move
10        BFS from  $v_2$  in  $c_2$  to get  $n$  vertices  $V_{2 \rightarrow 1}$ 
11        move  $V_{2 \rightarrow 1}$  from  $c_2$  to  $c_1$ , resulting in  $c'_1$  and  $c'_2$ 
12        replace  $c_{1,2}$  by  $c'_1$  and  $c'_2$  (split)
```

Line 2 of the algorithm creates a priority queue of edges based on their weights. We use the Pearson correlation coefficient [23] as the weight. Thus, edges connecting most correlated value vertices will be popped out first in line 3. As long as one of the groups connected by the current edge e is smaller than α , we merge the two groups in line 6, making e an intra-group edge. If it turns out that the other group is too large and the total size exceeds the “invariant” range $[\alpha, 2\alpha)$, lines 8-12 split the merged group into two, effectively balancing the sizes of the original two groups. Line 9 computes the number of vertices to be moved from the bigger group to the smaller one, while line 10 does a breadth-first search (BFS) in the bigger group to get the set of vertices $V_{2 \rightarrow 1}$ to move.

Note that although this algorithm is initially similar to hierarchical clustering [23], the bulk of the algorithm (lines 4-12) is completely different, for a different goal—hierarchical clustering is only concerned with homogeneous clusters regardless of their sizes, while our algorithm aims at two simultaneous targets: (1) high correlations within each group (which could be negative correlation, not necessarily homogeneous), and (2) approximate balance of group sizes. In addition, the related problem of graph partitioning is an NP-hard problem; see [6] for a survey of approximate solutions. Most of them are more expensive than linear cost. In particular, a state-of-the-art algorithm [4] achieves a (k, v) balanced partition, i.e., it partitions G into k components of at most size $v \cdot (n/k)$, where $v > 1$ is a constant. It has an expensive computation cost of $O(n^2)$. More importantly, it would not meet our need since it does not bound from below the number of nodes in a partition (it could be 1). We have two different aims, as discussed above, in addition to being efficient. We prove the following properties.

THEOREM 1. *After running ONELEVELVERTEXGROUPING, every group has at least α but less than 2α vertices, unless there is an*

isolated small group with less than α vertices and no edges coming out of it.

PROOF. We say that an edge is an *inter-group* edge if its two endpoints are in two groups, and an *intra-group* one if its two endpoints are in one group. Initially, all edges are inter-group ones as each vertex is a group. Then in descending weight order, all these edges are traversed exactly once. First, no group will have size 2α or more in the end. We prove this by induction over the while loop in line 3. Initially it must be true as each group is of size 1. Now suppose it is true after the previous while loop (line 3); we show that it is true after the current loop. As each group is less than 2α , the merged group size $|c_{1,2}|$ in line 7 is no more than $4\alpha - 2$. Since the two groups from the split in lines 11-12 have sizes $\lfloor \frac{|c_{1,2}|}{2} \rfloor$ and $\lceil \frac{|c_{1,2}|}{2} \rceil$, respectively, they are both less than 2α again.

We next show that at the end of the algorithm, no inter-group edge is connected to a group that is smaller than size α . Initially all inter-group edges are put in the priority queue Q (line 2), and each of them is visited in descending weight order (line 3). When a merge happens in line 6, the inter-group edge e in the current iteration (line 3) will become an intra-group one. On the other hand, when a merge does not happen, it could be one of the two reasons: (1) edge e has already become intra-group when a previous iteration merges the two groups it connects; (2) the two groups it connects are already of size at least α . In either case, it still holds that no inter-group edge is connected to a group that is smaller than size α .

A noteworthy situation arises when a split is done after a merge (lines 11-12). This is the only situation where some intra-group edges may revert back to be inter-group. However, such an inter-group edge must be connecting c'_1 and c'_2 , both of which are of size at least α , because their sizes are $\lfloor \frac{|c_{1,2}|}{2} \rfloor$ and $\lceil \frac{|c_{1,2}|}{2} \rceil$ and $|c_{1,2}| \geq 2\alpha$ (line 7). To summarize the proof above, by the time we iterate through all the edges in Q , there is no inter-group edge connected to a group smaller than size α . Thus, Theorem 1 is true. $\square$

Most real world large graphs are connected (such as telecommunication networks or road networks). Hence it is not a problem to have groups all of size $[\alpha, 2\alpha)$. Even if the graph were not connected, the group size invariant $[\alpha, 2\alpha)$ in Theorem 1 would only require each *component* have size at least α . If indeed there are smaller than α components/groups, that is the nature of the graph and we can just leave them as they are. We next prove the complexity of the algorithm.

THEOREM 2. *The amortized cost of ONELEVELVERTEXGROUPING is $O(|E| + |V|)$. In particular, it is not a function of α .*

PROOF. A naïve analysis will give the cost $O(\alpha \cdot |E| + |V|)$, as we may have to traverse and move $O(\alpha)$ vertices (lines 10-11) in an iteration of an edge. We use an amortized analysis to give a tighter bound. First, observe that each of the new groups c'_1 and c'_2 from the split (lines 11-12) always has a size in $[\alpha, \frac{3}{2}\alpha)$. We already show the lower bound of α as part of the proof of Theorem 1. The upper bound $\frac{3}{2}\alpha$ is because c'_1 and c'_2 have sizes $\lfloor \frac{|c_{1,2}|}{2} \rfloor$ and $\lceil \frac{|c_{1,2}|}{2} \rceil$, and because $|c_{1,2}| < 3\alpha$ as $|c_1| < \alpha$ (line 5) and $|c_2| < 2\alpha$ (proof of Theorem 1). We say that a vertex is *on the small side* if it is in a group of size less than α , and *on the big side* if it is in a group of size

at least α . Initially all $|V|$ vertices are on the small side. Observe that once a vertex goes from the small side to the big side, it never goes back, because even a split (line 12) will have groups of size at least α . Thus, this is a one-way move. Initially, at least 2α small-side points are “consumed” (i.e., moved to the big side) before triggering a split, and a split costs $O(\alpha)$ (for BFS and moving vertices). After a split, since each of the two groups has size less than $\frac{3}{2}\alpha$ (proven above), it will consume at least $\frac{\alpha}{2}$ small-side points before triggering another split. Overall, $O(\alpha)$ small-side points are consumed to trigger a split that also costs $O(\alpha)$. The initial number of small-side points is $|V|$; thus the total split cost in the whole algorithm is $O(|V|)$. The amortized cost of the algorithm is $O(|E| + |V|)$. $\square$

We repeat the same algorithm for upper levels of the model, where each node is the parent of a group of nodes in the lower level, and each edge (v_1, v_2) in the lower level has a corresponding edge between v_1 and v_2 's parents in the upper level (if they are different).

4.2 Learning Model Parameters

Given training data for a period of time, we now discuss how to determine our model's parameters. The parameters at each internal node of the model consist of three parts: (1) the number of states, (2) the transition probabilities between states, and (3) the emission probabilities to lower level child nodes. Recall that the bottom level of our model is the original value graph nodes, which we call the leaf nodes. The parameter learning proceeds in a bottom-up order, starting from each node N right above the bottom level. To learn the parameters of N , our basic idea is to extend the Baum-Welch algorithm [18] to multiple emission variables, and hence multiple emission probability distributions, one for each child node of N . In order to do this, we propagate up a distribution function $f_{t,N}(j)$, i.e., the probability of state/value j at node N at time step t . In a way, this f function bottom-up propagation can be regarded as a bottom-up order *variable elimination* [15] for the Bayesian network at time step t . The algorithm is shown in ALLPARAMETERLEARNING.

The loop in line 2 is to iterate over all possible numbers of states of node N from 2 to the maximum number of states σ allowed for a variable (e.g., $\sigma = 32$). Later on, in lines 21-26, we will use the minimum description length (MDL) principle [19] to determine the optimal number of states. Lines 3-4 initialize the state transition probabilities at node N , $A_N[i][j]$ (from state i to state j), and the emission probabilities to each child C_i , $p_{C_i}[j][k]$ (from state j of N to state/value k of C_i) to uniform distributions.

Then the “repeat” loop in lines 6-20 performs the E and M steps (as in the *expectation maximization* method of the extended Baum-Welch algorithm described earlier), until the distributions A_N and p_{C_i} 's converge. Lines 8-14 are the E-step. In line 9, the f function value, as discussed above, is propagated from all child nodes to N , taking into account the f values at the children and the emission probabilities. The α , β , and γ functions in lines 10-14 are similar to Baum-Welch, except that we use f function values as the emission probabilities. Likewise, lines 16-19 are the M-step where we re-estimate A_N and p_{C_i} 's. Finally, in lines 21-26, we calculate and compare the description length dl , and pick the number of states that results in the minimum dl , following the minimum description

Algorithm 2: ALLPARAMETERLEARNING(G)

Input: G : graph with observed values in times 1 to T
Output: all parameters of the model

```

1 for each internal node  $N$  in a bottom-up order do
2   for  $|S| \leftarrow 2, \dots, \sigma$  do
3     initialize  $A_N[i][j]$  to  $\frac{1}{|S|}$  ( $1 \leq i, j \leq |S|$ )
4     for each child node  $C_i$  do
5       initialize  $p_{C_i}[j][k]$  to  $\frac{1}{|S|}$  ( $1 \leq j \leq |S|, 1 \leq k \leq |S_i|$ )
        //prob at state  $j$  of  $N$  emitting state  $k$  of
         $C_i$ 
6     repeat
7       //E-step
8       for  $t \leftarrow 1 \dots T$  and  $j \leftarrow 1 \dots |S|$  do
9          $f_{t,N}[j] \leftarrow \prod_{C_i} \{ \sum_{k=1}^{|S_i|} f_{t,C_i}[k] \cdot p_{C_i}[j][k] \}$ 
10         $\alpha[t][j] \leftarrow \sum_{i=1 \dots |S|} \alpha[t-1][i] \cdot A_N[i][j] \cdot f_{t,N}[j]$ 
11       for  $t \leftarrow T \dots 1$  and  $i \leftarrow 1 \dots |S|$  do
12         $\beta[t][i] \leftarrow \sum_{j=1 \dots |S|} \beta[t+1][j] \cdot A_N[i][j] \cdot f_{t,N}[j]$ 
13       for  $t \leftarrow 1 \dots T, i \leftarrow 1 \dots |S|, j \leftarrow 1 \dots |S|$  do
14         $\gamma_t[i][j] \leftarrow \alpha[t-1][i] \cdot A_N[i][j] \cdot f_{t,N}[j] \cdot \beta[t][j]$ 
15       //M-step
16       for  $i \leftarrow 1 \dots |S|, j \leftarrow 1 \dots |S|$  do
17         $A_N[i][j] \leftarrow \frac{\sum_{t=1}^T \gamma_t[i][j]}{\sum_{k=1}^{|S|} \sum_{t=1}^T \gamma_t[i][k]}$ 
18       for each child node  $C_i, j \leftarrow 1 \dots |S|, k \leftarrow 1 \dots |S_i|$  do
19         $p_{C_i}[j][k] \leftarrow \frac{\sum_{t=1}^T \sum_{i=1}^{|S|} \gamma_t[i][j] \cdot f_{t,C_i}[k]}{\sum_{i=1}^{|S|} \sum_{t=1}^T \gamma_t[i][j]}$ 
20     until converged
21      $dl \leftarrow 0; z \leftarrow \sum_{i,j} A_N[i][j] + \sum_{C_i,j,k} p_{C_i}[j][k]$ 
22     for  $i \leftarrow 1 \dots |S|, j \leftarrow 1 \dots |S|$  do
23        $p \leftarrow \frac{A_N[i][j]}{z}; dl \leftarrow dl + p \log \frac{1}{p}$ 
24     for each child node  $C_i, j \leftarrow 1 \dots |S|, k \leftarrow 1 \dots |S_i|$  do
25        $p \leftarrow \frac{p_{C_i}[j][k]}{z}; dl \leftarrow dl + p \log \frac{1}{p}$ 
26     choose the min  $dl$  among different  $|S|$  for the  $S$  of node  $N$ 
```

length (MDL) principle [19] (which basically says that the best model should be the most concise one).

To summarize, the ALLPARAMETERLEARNING algorithm is correct since (1) it runs a variant/generalization of the Baum-Welch algorithm at each internal node, and (2) instead of using emission probabilities as in Baum-Welch, we use f functions as in bottom up variable elimination (VE) to express the likelihood of producing all the leaf nodes of the subtree rooted at N . The f functions take the observed values at the leaf nodes. We analyze its complexity as follows.

THEOREM 3. *Suppose that it takes θ iterations for the loop in lines 6-20 to converge, and that we try a constant number of states at each node. Then the complexity of ALLPARAMETERLEARNING is $O(\theta T|V|)$.*

PROOF. This complexity result is based on the following two insights: (1) The algorithm goes through every internal node once to learn its parameters. (2) Even though there are loops in line 4 and line 18 to iterate through all child nodes of N , each node has only one parent. Thus, counting through a node's parent, the code

in a loop is only done once for every node during the *entire* run of the algorithm, which implies that the amortized (aggregate) cost of this part is linear to $|V|$. $\square$

5 INFERENCE TO ANSWER QUERIES

We now study the query answering problem. We are given a number of observed values of variables within the time interval $[\tau_b, \tau_e]$, and we have a number of queries $Q_1, \dots, Q_c$. Each query Q_i ($1 \leq i \leq c$) inquires about the (unobserved) value of a variable at time t (where t is either inside or outside $[\tau_b, \tau_e]$). For each query Q_i , the user may specify the minimum set of observations $\mathbb{E}_i$ that must be used to answer the query.

The basic idea of the query processing algorithm is to first “unroll” the model across multiple time steps in the observed evidence interval T and the queried time steps. This is illustrated in Figure 3, where the model is unrolled across 4 time steps. The roots of the four trees are just one node (the *lowest common ancestor* of all evidence nodes and query nodes) unrolled at four adjacent time steps. The green oval leaf nodes are the observed variables, while the 4 leaf nodes marked with “?” (one at each time step) are the query nodes. In this example, all observations are between time step 1 and 3. Query nodes could be before observation start time too (e.g., time step 0).

The unrolled model is essentially a dynamic Bayesian network, and fortunately it does not have any cycles, which means we can perform *exact inference* by belief propagation [17]. To do this, we first add auxiliary “factor” nodes, one in the middle of each edge, as shown in Figure 3. Informally, a factor node for a Bayesian network just contains the conditional probability function between the nodes it connects. The product of all factor node functions is the joint distribution of the whole model.

We present the basic query processing algorithm in BELIEFPROPAGATIONOVERBRIDGES. The multiple versions of the lowest common ancestor (LCA) of the evidence and query nodes are connected by “bridges”, which are from the Markov model of the LCA, shown as the root nodes in Figure 3. We will later present some optimizations on top of this basic algorithm.

Algorithm 3: BELIEFPROPAGATIONOVERBRIDGES

Input: M : model, Q : a query leaf node, $\mathbb{E}$: observed evidence nodes for Q
Output: estimated probability distribution of Q

- 1 identify the minimum subtree T_{eq} that contains $\mathbb{E}$ and Q
- 2 build a dynamic graph by connecting the roots of T_{eq} at different times
- 3 **for each bridge edge do**
- 4 add a factor node that has the transition function of the chain
- 5 **for each edge in T_{eq} in all time steps do**
- 6 add a factor node that has the emission function
- 7 $R \leftarrow$ root of T_{eq} at time $\lceil \frac{\tau_b + \tau_e}{2} \rceil$
- 8 do belief propagation from each leaf of the trees to R , taking $\mathbb{E}$
- 9 do belief propagation from R to all leaves
- 10 return the product of the messages to Q

Line 1 of the algorithm finds the minimum subtree, rooted at the LCA of the relevant nodes. Line 2 is the “unrolling” of the model

across the relevant time steps. Lines 3-6 add the factor nodes that are either the transition probability function of the Markov chain at the LCA, or the emission functions from a parent to a child. Line 7 picks a single “root” R at the “middle” of the times of the observed nodes, as the root of the whole unrolled dynamic Bayesian network. This root R can in fact be any node and belief propagation will give the same result for the query node [17]. We pick this particular root R for the ease of presentation of the algorithm in lines 8-9. Basically, belief propagation comprises two rounds: from all leaves to root (line 8), and then from root to all leaves (line 9). We illustrate how this works via an example below.

EXAMPLE 2. We illustrate the algorithm through a simple example in Figure 4, where we show three time steps and the chosen root R of this unrolled model is in the middle. For the ease of illustration, each tree only has 2 levels, and we focus on the observed evidence node B (marked with a green oval) and the query node Q (marked with a “?”). Suppose there are only two states (values) of each node (0 and 1) and the transition matrix of the Markov chain at R is $A = \begin{bmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{bmatrix}$, and that the emission distribution from C to B (or R to Q) is $P = \begin{bmatrix} 0.2 & 0.8 \\ 0.8 & 0.2 \end{bmatrix}$. That is, factor node f_2 contains A , while f_1 and f_3 contain P . Suppose the observed value at B is 0. Our goal is to answer Q .

First consider round 1 of belief propagation from leaves to R (line 8). For leaf B , we have $\sum_B \mu_{B \rightarrow f_1}(B) \Pr[B|C] = \mu_{f_1 \rightarrow C}(C)$, where $\mu_{B \rightarrow f_1}(B)$ stands for the message from B to f_1 , and it is a function of B , and $\Pr[B|C]$ is just the function f_1 has (i.e., P). Thus $\mu_{f_1 \rightarrow C}(0) = P(0|0) = 0.2$ and $\mu_{f_1 \rightarrow C}(1) = P(0|1) = 0.8$. This is because we are given the observed B value 0; hence $\mu_{B \rightarrow f_1}(0) = 1$ and $\mu_{B \rightarrow f_1}(1) = 0$. We get $P(0|0)$ and $P(0|1)$ from looking up the P matrix above. Once we get $\mu_{f_1 \rightarrow C}(C)$, round 1 message passing to root continues. Next, C will collect the messages it receives from all its neighbors except f_2 (which is on the path to R), multiply them together, and pass it to f_2 . Since the subtree under C 's right branch does not contain observed nodes, the message to C from its right child is simply 1 (which we will discuss in the optimizations shortly). Thus, $\mu_{C \rightarrow f_2}(C) = \mu_{f_1 \rightarrow C}(C)$, which is obtained earlier.

Then similar to the message from f_1 to C , we can compute the message from f_2 to R , $\mu_{f_2 \rightarrow R}(R)$, as follows: $\mu_{f_2 \rightarrow R}(0) = 0.2 \cdot A(0|0) + 0.8 \cdot A(0|1) = 0.32$ and $\mu_{f_2 \rightarrow R}(1) = 0.2 \cdot A(1|0) + 0.8 \cdot A(1|1) = 0.68$. Here, the conditional probabilities such as $A(0|0)$ are looked up from the transition matrix A above. Now we proceed to round 2 of belief propagation (line 9). R will send a message to f_3 , i.e., $\mu_{R \rightarrow f_3}(R)$, as a product of all messages R receives from all other neighbors. This is just the $\mu_{f_2 \rightarrow R}(R)$ we have above (since, again, other neighbors give a message 1). Then finally, we compute our goal $\mu_{f_3 \rightarrow Q}(Q)$ as: $\mu_{f_3 \rightarrow Q}(0) = 0.32 \cdot P(0|0) + 0.68 \cdot P(0|1) = 0.608$, and $\mu_{f_3 \rightarrow Q}(1) = 0.32 \cdot P(1|0) + 0.68 \cdot P(1|1) = 0.392$, which is the estimated distribution of Q .

The correctness of BELIEFPROPAGATIONOVERBRIDGES is due to the fact that the unrolled graph is still a tree without loops, and hence we can perform exact inference. It is easy to see that the complexity of the algorithm is linear to the number of vertices in the value graph and to the length of the time interval T , as message passing is linear to the tree size.

Further Optimizations. We propose a few refinements and optimizations to the query processing algorithm. First, we observe that,

Figure 3: Part of our graphical model (a subtree) is unrolled across 4 time steps. The root of each tree is the unrolled version of the lowest common ancestor of all observed evidence nodes (marked as green ovals) and query nodes (marked by “?”). Factor nodes are added in the middle of each edge (as small squares), containing the conditional probability functions.

Figure 4: Illustrating the BELIEFPROPAGATIONOVERBRIDGES algorithm.

rather than spending $O(n \cdot |T|)$ time to process one query at a time, we batch a set $\mathcal{Q}$ of many queries that can be answered together using the same set of observed evidence $\mathcal{E}$. The rationale of doing this is analogous to the multi-query optimization, i.e., to share the computation efforts among concurrent queries.

Our second refinement is for the first round of belief propagation in line 8 of BELIEFPROPAGATIONOVERBRIDGES. We only initiate the messages from the observed evidence nodes $\mathcal{E}$, but ignore all other leaves. We will show that the end result is the same.

The third refinement is for the second round of belief propagation in line 9 of BELIEFPROPAGATIONOVERBRIDGES. We only propagate messages along the paths to the set of query nodes $\mathcal{Q}$. To figure out the paths first, we can start from the query nodes and trace back towards R . The message passing paths will be the reverse of these paths. Again, we show that the final query result is equivalent with this optimization.

THEOREM 4. *Assuming that the Markov chain at root R in BELIEFPROPAGATIONOVERBRIDGES has a stationary distribution, our query processing algorithm will return exactly the same results with or without the three refinements.*

PROOF. It is clear that the first refinement does not change query results. Let us consider the second refinement. In the first round of message passing, a leaf without an observed value passes a message of constant value “1” to the factor node above by the definition of belief propagation. Recall that, as illustrated in Example 2, $\sum_B \mu_{B \rightarrow f_1}(B) \Pr[B|C] = \mu_{f_1 \rightarrow C}(C)$. When $\mu_{B \rightarrow f_1}(B) = 1$, we have $\mu_{f_1 \rightarrow C}(C) = \sum_B \Pr[B|C] = 1$. Thus, this constant message 1 keeps getting propagated throughout the whole subtree that does not have any observed evidence. If the message 1 gets to the root of the tree at a particular time step, e.g., node C in Figure 4, the message will need to be passed towards the designated root R . This message

passing could be in two directions: in increasing time order (the same as the Markov chain) or in decreasing time order (reverse of the Markov chain) if it is from a node to the right of R in Figure 4. It is well known that, if a Markov chain has a stationary distribution, the reverse chain is also a Markov chain. Hence, in either direction, the summation equation as shown above still holds, and message “1” will be propagated all the way.

Next we consider the third refinement (for round 2). We can prove this inductively. Consider a query node (leaf) Q . There is only one path from the root to Q . To get the message to Q , we only need to consider its parent node P (as there is only one edge to Q). For P to pass a message to Q in round 2, P needs to multiply messages from all its other children (from round 1) and from its parent. This argument goes on inductively, until we reach the root node of Q ’s subtree, e.g., node C in Figure 4. C ’s messages from its neighbors are all either from round 1 or from the path from R . This holds all the way to the designated root R . $\square$

Note that the refinements proposed here make it convenient to handle other types of queries such as aggregates and joins, which we plan to study in detail in future work.

6 ADAPTIVE AND INCREMENTAL MODEL

In massive dynamic networks, the model parameters may change after some time. We propose an approach that addresses this issue. The basic idea is that during query processing, we do leave-one-out cross-validation [23] for a small constant number of observed leaves. If the accuracy is below a threshold, we decide to partially rebuild the model around the observed evidence $\mathcal{E}$. To partially (incrementally) rebuild the model, we invoke the ALLPARAMETER-LEARNING method in Section 4.2, but only learn the nodes that are ancestors of the observed evidence nodes $\mathcal{E}$. Therefore, this is much more efficient than a complete rebuild/re-learn of the model. Effectively, the part of the model that is accessed more frequently will be maintained more up-to-date. The algorithm is presented in CROSSVALIDATIONPARTIALUPDATE, where BPOB is short for BELIEFPROPAGATIONOVERBRIDGES, the algorithm that answers queries.

The loop in lines 1-8 does leave-one-out cross validation test on k (a small constant number) values randomly chosen from the observed evidence values. If all k tests have errors greater than some threshold, we partially re-build the model over the observed values $\mathcal{E}$.

Algorithm 4: CROSSVALIDATIONPARTIALUPDATE

```

1 repeat while running BPOB
2   choose one evidence leaf  $E_q$  uniformly at random
3   let the path from  $R$  to  $E_q$  be  $P_{eq}$ 
4   round 1 of BPOB (line 8) is exactly the same as before
5   in round 2 of BPOB (line 9), create a second version of messages
     for nodes in  $P_{eq}$  by treating  $E_q$  as a query
6   if  $E_q$  result differs from the observed within a bound then
7     return
8 until  $k$  times
9  $\mathbb{A} \leftarrow$  set of ancestors of the leaves in  $\mathbb{E}$ 
10 run ALLPARAMETERLEARNING but only for nodes in  $\mathbb{A}$ 

```

One may wonder about the correctness of cross-validating the query result on E_q , as line 4 of the algorithm does not exclude E_q from passing its value/evidence messages to R in round 1. For clarity, we formally summarize the correctness of the algorithm and its complexity in the following theorem.

THEOREM 5. *The cross-validation in lines 4-5 is sound: specifically, the E_q result we get is based on all observed evidence except E_q itself. Furthermore, the cross-validation does not increase the asymptotic complexity of BPOB. Let the probability that a query result has error exceeding a bound be ϵ (line 6). Then the probability that a partial update is run is ϵ^k , and the expected cost is $\epsilon^k \theta e |T| \log |V|$, where θ is the number of iterations for the loop in BPOB to converge, e is the number of evidence nodes, and $|T|$ is the time interval length of evidence.*

PROOF. We first show the soundness of cross-validation. The key point is that, in line 4, even though we do not keep a second version of messages without E_q participating in round 1 of message passing, any messages that the observed E_q value contributes to will not affect the message eventually passed to E_q in round 2. To see why this is true, consider the messages that goes through path P_{eq} (from R to E_q) in round 2. Regardless of whether such a message is “horizontal” (e.g., from R to f_2 to C in Figure 4) or “vertical” (from a root down to E_q), the messages from the neighbors that form the product do not contain any factors from the message of E_q in round 1.

Since the loop in lines 1-8 is repeated at most a constant number of times, and the extra message passing in line 5 is subsumed by the asymptotic cost of BPOB query processing itself, the complexity is the same. Lines 6-7 and the loop in lines 1-8 indicate that only if all k tests fail, does it do the partial model update; hence the probability is ϵ^k . It is also clear that the expected cost is $\epsilon^k \theta e |T| \log |V|$, similar to the analysis in Theorem 3. $\square$

7 EXPERIMENTS

Datasets and Setup. We use two real-world dynamic graph datasets: **(1) Hong Kong traffic data.** We get this dataset from [7], which provides a link to download very large historical traffic data of Hong Kong. For example, using an interval from 8/1/2009 to 9/22/2017 gives a dataset of 42.59 GB. The graph update frequency is on average about one report of a road every 0.38 second. A road report consists of a *timestamp*, *two ends* of the road, *region*, *type*, *saturation*

level, and observed *travel speed*. The attribute we are most interested in, other than time, is the travel speed, which indicates the traffic condition. **(2) New York taxi data.** This 11 GB New York city’s 2013 taxi trip data contains information such as medallion number, vendor ID, pick-up and drop-off date, time and location, passenger count, trip time, and trip distance [8]. A new data update—a report about a pick-up and drop-off trip information comes in on average every 0.186 second.

Dataset	#vertices	#edges	Average inter-up-date time	Data Size
HK traffic	605	365,420	0.48 sec	42.59GB
NY taxi	5,654	31,962,062	0.186 sec	11GB

Table 1: Statistics of the datasets.

Some statistics of the datasets are summarized in Table 1. We implement all the algorithms presented in the paper, as well as two baseline methods for comparison, which we will describe below. The experiments are performed on a MacBook Pro machine with OS X version 10.11.4, a 2.5 GHz Intel Core i7 processor, a 16 GB 1600 MHz DDR3 memory, and a Macintosh hard disk.

Experimental Results. In the first set of experiments, we examine our graph and model building, including the vertex grouping algorithm. For the HK traffic data, the *travel speed* on each road segment is the attribute value of our interest, which is dynamic over time and indicates the traffic condition of the roads. We discretize the speed value into 0 (below 10km/hr), 1 (between 10km/hr and 20km/hr), ..., 6 (above 60km/hr). Thus, each road is a *vertex*, and there is an *edge* between two vertices if the roads they correspond to intersect. For the NY taxi data, we partition the whole area into 0.0075 latitude by 0.01 longitude (i.e., about 0.5 mile by 0.5 mile) grids, and each grid area is a vertex. The attribute value in question is the number of pick-ups at the grid area during the time step. We discretize this count to 0-4 by partitioning it into five intervals. An edge is added between two grid areas (vertices) that are neighbors.

In Figure 5, we show the graph and model building time for various training data sizes from 500MB to 2.5 GB, using the NY taxi data (HK traffic data gives us a similar trend in result). As discussed in Section 4.1, general partition algorithms such as [4] do not meet our needs. While the group size balance has been proven (Theorem 1), to evaluate the effectiveness of our algorithm in preserving value correlations within a group, we design a baseline that, like in previous work [6], does a breadth-first-search and ensures a balanced group size α without explicitly optimizing intra-group value correlation. We can see that even for large training data sizes, the graph and model building of our algorithm takes only 10’s or 20’s seconds, and grows linearly with the training data size. It is only slightly slower than the baseline algorithm. This result is consistent with our analysis in Theorem 2, which shows that our vertex grouping algorithm has an $O(|E| + |V|)$ amortized cost. In Figure 6, using 30 hours of the HK traffic data, we examine the performance of the vertex grouping algorithm alone under different group-size parameter α values ranging from 3 to 15, again compared with baseline algorithm. Our vertex grouping algorithm is slightly slower than the baseline, and the speed is independent of α . Again, this is consistent with Theorem 2 which indicates that the amortized cost is not a function of α .

Fig 5 Graph/model building time

Fig 6 Vertex grouping time vs. α Fig 7 Correlation coefficients vs. α

Fig 8 Parameter learning time

Fig 9 Average # of states vs. α

Fig 10 Brier scores/errors (HK traffic)

Fig 11 Brier scores/errors (NY taxi)

Fig 12 Execution time (HK traffic)

Furthermore, in Figure 7, we test the vertex grouping result under different group-size parameter α values using the NY taxi dataset, by calculating the average (absolute value of) Pearson correlation coefficients [23] of pairs of vertices connected by edges either inside each group partition (marked as “intra” in Figure 7), or across two group partitions (marked as “inter”). We also compare the result with the baseline method as described above. It is clear that the correlations of vertices connected by intra-group edges are much higher than those of inter-group edges, under our grouping algorithm. This is because our algorithm systematically ensures both high correlations within each group and the balance of group sizes in the range of $[\alpha, 2\alpha - 1]$. This is not the case with the baseline algorithm, which results in more or less equal division of edges with respect to vertex correlation. In addition, the difference between intra- and inter- group edges is most significant when α is a certain value (8 in our setting). When α is either too small or too large, more fraction of low-correlation edges must be put inside a group, as all vertices must be grouped with their neighbors by the algorithm. As shown later, α also has an effect on model parameters such as the number of states of a node.

In the next set of experiments, we study the model parameter learning algorithm (ALLPARAMETERLEARNING). Figure 8 shows the running times for both datasets, varying the training data duration from 15 hours to 90 hours. We can see that they are both linear functions of the size of training data, with slightly different rates of change. Moreover, the parameter learning times are acceptable, ranging from 2 seconds to 17 seconds. In Figure 9, we further look at the result of this learning, in the average number of states of

each node of the model, while varying the group size parameter α . For both datasets, the average number of states per node increases as group size increases. The reason is that, for a larger group size, the parent node in our model, as a latent variable, must be able to characterize more complex co-evolution of the child nodes as a group—therefore more “states” are needed. As it turns out, on average more states are needed for the travel speed attribute in the Hong Kong traffic dataset than the number of taxi pick-ups in a grid area of the New York taxi dataset. In subsequent experiments, we use a default $\alpha = 5$ unless otherwise specified.

In the next set of experiments, we examine the query result accuracy, as well as the performance of answering queries. We design six query templates considering variations such as the density of observed data, whether the queried vertex is among the one observed (but at a different time) or an unobserved one, and whether the queried time is within the interval $T = [\tau_b, \tau_e]$ of the observed vertices, or precede or succeed T . The query generation templates are as follows:

(Q1) Perform a random walk for 10 time steps from an arbitrary vertex; observe one vertex at each time step; query an observed vertex at an unobserved time within T .

(Q2) Same as Q1, except that we observe *three* vertices at each time step (i.e., walk through more vertices/edges).

(Q3) Same as Q1, except that we query an unobserved vertex to be seen at time $\tau_e + 1$ (for $T = [\tau_b, \tau_e]$, i.e., the next vertex to be traversed if the random walk were to continue).

(Q4) Same as Q3, except that we query the vertex at time $\tau_e + 3$, i.e., two more time steps later.

Fig 13 Execution time (NY taxi) Fig 14 Query batch optimization Fig 15 Effect of incremental update Fig 16 Overhead of incremental update

(Q5) Same as Q2, except that we query an unobserved vertex at time $\tau_b - 1$ (for $T = [\tau_b, \tau_e]$, i.e., the preceding vertex if the random walk started a time step earlier).

(Q6) Same as Q5, except that we query an unobserved vertex at time $\tau_b - 3$.

Recall that our query processing algorithms return a probability distribution as a query answer. Fortunately, both of our two real-world datasets have observed values as the ground truth for us to evaluate the accuracy of query answers. Simply using the difference between the expected value of the distribution and the ground truth value is undesirable, as that does not take the uncertainty (i.e., the “spread”) of the query answer into consideration. Inspired by the Brier score [5] which works only for *nominal* attributes, we define a variant of it that is more suitable for *ordinal* values, as in our case. Our Brier score variant is defined as $\frac{\sum_i p_i(v_i - o)^2}{d_m^2(o)}$, for a query result distribution $\{(v_i, p_i)\}$ and the ground truth value o , where $d_m(o)$ is the maximum possible distance to o . For example, if the whole value range is $[0, 4]$, the ground truth is $o = 2$, and the query result is $\{(2, 0.5), (3, 0.5)\}$, then the Brier score variant is $\frac{0.5 \times 1^2}{2^2} = 0.125$. Thus, it is always a value in $[0, 1]$, with 0 being the most accurate, considering both the *location* and *spread* of a distribution.

In Figure 10, using the Hong Kong traffic dataset, we show the average Brier scores of the results of six groups of queries Q1 to Q6 generated from the six query templates as described above. Figure 11 is likewise for the New York taxi dataset. We compare against two baselines. Baseline 1 is as described earlier, doing BFS for vertex grouping, but otherwise is the same as our approach in learning parameters and answering queries. Baseline 2 is the previous state-of-the-art algorithm as described in [11] that does a combined Kernel Regression and k -NearestNeighbor with $k = 10$.

We can see that our method has small Brier scores (errors) for both datasets, significantly smaller than the two baseline approaches. Among the six types of queries, Q2, Q5, and Q6 have smaller errors due to the fact that they have three times of the observed vertices, i.e., more evidence that sheds light to the queried values. Moreover, in general, the farther away in time is the queried vertex from the observed vertex interval T , the more error is incurred (e.g., Q1 within T , Q3 at $\tau_e + 1$, and Q4 at $\tau_e + 3$). This is because when belief propagation to the query vertices has to go through more unobserved nodes, greater errors are accumulated.

Baseline 1 has larger errors because its vertex grouping is rather random, which makes its model inferior to ours, even though it uses the same parameter learning and query processing algorithms. Baseline 2 is less accurate than our method because it is not as sophisticated as our model in capturing the value correlations near and far in the graph topology and over time. Finally, we observe that the New York taxi dataset tends to give slightly greater errors in general than the Hong Kong traffic dataset.

Then we measure the query execution times of the three approaches, as shown in Figure 12 for the Hong Kong traffic dataset, and in Figure 13 for the New York taxi dataset. Query groups 1, 3, and 4 have the same number of observed (evidence) vertices, thus having almost identical execution times, and we show them together; similar for query groups 2, 5, and 6. The figures show that, for both datasets, our method is slightly faster than baseline 1 in answering queries, while baseline 2 is much faster than both of them—baseline 2 has a significantly smaller computation overhead.

Baseline 1 is slightly slower than our method because the vertices in each group are less correlated; hence a higher entropy is present in each group of child nodes, causing their parent node to have more states on average, slowing down the inference during query processing. Query groups 2, 5, and 6 are slower than 1, 3, and 4 since the former query groups involve more observed vertices, which require more time in the first pass belief propagation (from observed leaves to root). In addition, we observe from Figures 10 and 11 that our method takes longer time for the Hong Kong traffic dataset. This is because, as indicated in Figure 9, the average number of states per node is higher for the Hong Kong traffic dataset, which makes the BELIEFPROPAGATIONOVERBRIDGES algorithm slower.

In Figure 14, we further evaluate the benefit of the refinements and optimizations proposed in Section 5. We only test the query batch technique because the other two refinements involve belief propagation over fewer vertices, and are obviously faster. Figure 14 shows the execution times under various query batch sizes (i.e., the number of queries in a batch). We can see that the batch versions considerably improve the performance due to the sharing of the belief/message propagations among the queries in a batch.

Our last set of experiments examine the effectiveness and efficiency of the incremental model update algorithm described in Section 6. We first arbitrarily change the parameters of our model, causing the model to be unsuitable. Then we run six query batches QB1 to QB6 in a sequential order, each containing 12 queries, two

from each query templates described earlier. We compare three versions: (1) no model update, and always use the unsuitable model, (2) a full rebuild of model first, before answering the six query batches, and (3) run `CROSSVALIDATIONPARTIALUPDATE` while answering each query batch in order. In Figure 15, we measure the average Brier scores (i.e., result errors). The no model update version incurs the most errors, while a full rebuild gives small errors. Our incremental update algorithm achieves accuracy improvement as it answers each batch of queries in order. This is because it partially updates the model for the portion of the model parameters relevant to the current queries. The query result accuracy improves as this update progresses, eventually approaching the level achieved by a full model rebuild in the beginning.

Figure 16 shows the execution times of the relevant operations in the approaches above. Note that the y axis is in logarithm scale. We can see that a full model rebuild is two orders of magnitude slower than an incremental update, which only accesses a small part of the model. We also show this cost side-by-side with the query processing cost of a batch of queries described above, which is the smallest among the three operations. The combined results in Figures 13 and 14 show that the incremental update approach is practical and useful in adapting the model for answering queries.

Summary. The comprehensive experimental results indicate that our vertex grouping is fast and produces grouping that preserves high vertex correlations within groups. The model building and parameter learning algorithms are efficient and result in models that can answer queries accurately. Factors that affect query result accuracy include the density of observed evidence vertices in the graph and over time, as well as the distance of query vertex/time from the observed vertices. Finally, our proposal of incremental model update is effective and efficient in rebuilding models and improving result accuracy.

8 OTHER RELATED WORK

Most closely related work has been discussed inline in the previous sections; we survey other related work here. Dealing with missing data is an important problem in many fields. For example, Osborne [16] and Tan et al. [23] give good surveys on this topic. Predictive database and query processing are also studied, e.g., [3], [10]. In particular, Akdere et al. [3] argue that the next generation DBMS should incorporate a predictive model management component. All the previous work discussed above, however, cannot be applied to our problem since we specifically target dynamic networks, where both correlations implied by graph topology and time-wise state evolution patterns over the whole large graph must be incorporated. There are a few exceptions which we have discussed in §1 and compared against in §7. Kim and Leskovec [14] study the “network completion” problem, which bears some similarity with our work. However, their work is about inferring missing nodes and edges in large graphs, while we do not intend to change the graph structure, but only focus on the values.

Probabilistic graphical model (PGM) has seen its many uses in the literature. For example, Deshpande et al. [1] give a survey of the usage of PGM in managing and querying probabilistic databases, where PGM is to model the correlations among the uncertain data entities. One particular project is the BayesStore at UC Berkeley

[24], where Wang et al. use Bayesian networks extensively in their system. It has also been proposed by Deshpande et al. to use PGM to guide *data acquisition* [9] in systems such as sensor networks. Finally, dynamic graphs and graph streams are also studied in the literature, e.g., [2], [22]. However, in this paper, we study a completely different problem.

9 CONCLUSIONS AND ACKNOWLEDGMENTS

Dynamic graphs are increasingly common in urban systems today. It is important to be able to answer queries on unobserved variables in the graph, given the observed ones. In this paper, we propose a novel probabilistic graphical model which embodies the correlations of variables across graph topology and time. We devise a number of algorithms for model learning and query answering. Our experiments on two large real-world dynamic graph datasets demonstrate the effectiveness and efficiency of our approach.

This work is supported by NSF grants IIS-1149417 and IIS-1633271.

REFERENCES

- [1] C. Aggarwal. 2009. Graphical models for uncertain data. In *Managing and Mining Uncertain Data*. Springer.
- [2] C. Aggarwal, Y. Li, P. Yu, and R. Jin. 2010. On Dense Pattern Mining in Graph Streams. *VLDB* (2010).
- [3] M. Akdere, U. Cetintemel, M. Riondato, E. Upfal, and S. Zdonik. 2011. The Case for Predictive Database Systems: Opportunities and Challenges. In *CIDR*.
- [4] K. Andreev and H. Räcke. 2004. Balanced Graph Partitioning. In *SPAA*.
- [5] G. Brier. 1950. Verification of Forecasts Expressed in Terms of Probability. *Monthly Weather Review* 78 (1950), 1.
- [6] A. Buluc, H. Meyerhenke, I. Saffro, P. Sanders, and C. Schulz. 2013. *Recent Advances in Graph Partitioning*. arXiv:1311.3144.
- [7] Dataset 2017. Hong Kong traffic data. Available at <https://data.gov.hk/en-data/dataset/hk-td-tis-traffic-speed-map>.
- [8] Dataset 2017. New York taxi data. Available at http://chriswhong.com/open-data/foil_nyc_taxi/.
- [9] A. Deshpande, C. Guestrin, and S. Madden. 2005. *CIDR* (2005).
- [10] S. Duan and S. Babu. 2007. Processing Forecasting Queries. In *VLDB*.
- [11] J. Haworth and T. Cheng. 2012. Non-parametric Regression for Space-time Forecasting under Missing Data. *Computers, Environment and Urban Systems* 36 (2012).
- [12] J. Honaker and G. King. 2010. In *American Journal of Political Science*, Vol. 54.
- [13] N. Horton and K. Kleinman. 2007. Much Ado About Nothing: A Comparison of Missing Data Methods and Software to Fit Incomplete Data Regression Models. *The American Statistician* (2007).
- [14] M. Kim and J. Leskovec. 2011. The Network Completion Problem: Inferring Missing Nodes and Edges in Networks. In *SIAM*.
- [15] D. Koller and N. Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*.
- [16] J. Osborne. 2013. Dealing with Missing or Incomplete Data: Debunking the Myth of Emptiness. In *Best Practices in Data Cleaning: A Complete Guide to Everything You Need to Do Before and After Collecting Your Data*.
- [17] J. Pearl. 1982. Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. In *AAAI*.
- [18] L. Rabiner. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE* (1989).
- [19] J. Rissanen. 1978. Modeling by shortest data description. In *Automatica*.
- [20] T. Schneider. 2001. Analysis of incomplete climate data: Estimation of mean values and covariance matrices and imputation of missing values. *Journal of Climate* 14 (2001).
- [21] R. Smith, S. Kolenikov, and L. Cox. 2003. Spatio-temporal modeling of PM2.5 data with missing values. *J. of Geophysical Research - Atmospheres* 128 (2003).
- [22] C. Song, T. Ge, C. Chen, and J. Wang. 2015. Event Pattern Matching over Graph Streams. *VLDB* (2015).
- [23] P. Tan, M. Steinbach, and A. K. Kar. 2006. *Introduction to Data Mining*. Pearson Addison Wesley.
- [24] D. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein. 2008. BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models. *VLDB* (2008).
- [25] J. Wang, N. Zou, and G. Chang. 2008. Travel time prediction: Empirical analysis of missing data issues for advanced traveler information system applications. *J. of the Transportation Research Board* 2049 (2008).