MIT-LL Graph QuBE: Probabilistic Query-by-Example of Large Graphs

Charlie Dagli dagli@ll.mit.edu
Michael Yee myee@ll.mit.edu
Gordon Vidaver gordon.vidaver@ll.mit.edu
Wade Shen myee@ll.mit.edu
Joseph Campbell jpc@ll.mit.edu

1. Introduction and Motivation

The ability to efficiently search for patterns-of-behavior among entities transacting over time becomes increasingly valuable as the amount of transactional data continues to increase. Despite this wealth of information, there is a relative poverty of automatic (or semi-automatic) tools to help answer the following basic question: "If we have an interesting pattern of behavior between some entities-of-interest, can we look through all our data to find other such examples of this type of behavior?"

MIT Lincoln Laboratory (in collaboration with Oculus and Giant Oak) has developed a query-by-example tool called Graph Query-by-Example (QuBE) to help address this capability gap.

This paper outlines the Graph QuBE system explaining the technical approach and system architecture, and implementation.

2. Graph QuBE System Architecture

Graph QuBE has been designed to facilitate query-by-example functionality over transactional data over time. Transactional data over time can be represented as a time-varying graph where nodes represent entities and edges the transactions between them. In this abstraction, a pattern query can be represented as the time-varying sub-graph induced between interactions among a collection of entities over time (a repeated fan-out fan-in pattern among a subset of entities, for example).

Traditional query-by-example technologies fail to adequately address the inherent challenges posed by this type of data. Sub-graph search algorithms find structurally similar graphs, but cannot match temporal characteristics. Conversely, powerful temporal matching algorithms are rendered impractical by the exponential number of sub-graphs against which to match. In this work, we marry the strengths of these traditional approaches to develop an efficient two-stage graph query-by-example (Graph QuBE) system for time-series graphs. This system architecture is shown in the figure below.

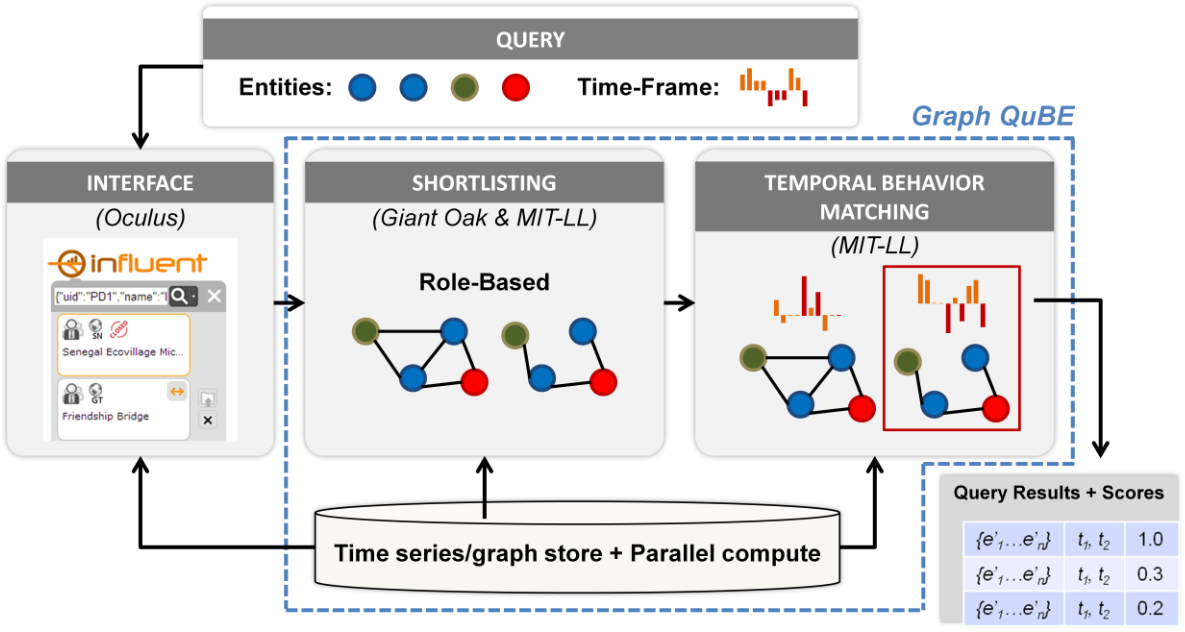

Figure 1: MIT-LL Graph QuBE System Architecture

Given a query specified by the user as a collection of entities and a time window, the first stage of our system shortlists a small number of sub-graphs structurally similar to the query graph. These results are passed to the second stage where a transaction-based Hidden Markov Model (HMM) re-orders the shortlisted results with respect to temporal similarity. This two-stage approach allows us to leverage the relative strengths of each of these traditional approaches while balancing speed and performance. The following sections outline the technical details of each of these stages.

2.1 Short-listing

Graph QuBE relies on an efficient short-listing phase in advance of the more computationally intensive temporal behavior matching stage outlined in the Section 2.2. In particular, this short-listing must be fast, high-precision and *prioritized*. In this context, we define prioritized to mean that we want to return sub-graphs that are both connected similarly *and* contain entities similar to those appearing in the query.

Accordingly, Graph QuBE relies on what we will refer to as *Role-Based Short-listing*. Role-based short-listing proceeds as follows. First, for each entity e_i in the query graph, we find other entities similar to e_i, with respect to some entity feature-space. Next, given the set of similar entities for each entity in the query, we find and filter group of entities, $\{e'_1 \dots e'_n\}$, with coincident activity to find sub-graphs. Finally, we score these short-listed sub-graphs using the aggregation of individual entity-search scores and return the results. The following two sections summarize the two main techniques we use to perform the second step, coincident activity filtering: *Greedy Cartesian Product Search* and *Greedy Depth-First Search*.

2.1.1 *Greedy Cartesian Product Short-listing*

Greedy Cartesian product Short-listing is a technique to form and filter sub-graphs greedily derived from the Cartesian Product of entities from the similar entity lists from each entity in the query graph. This approach is outlined in the Figure below.

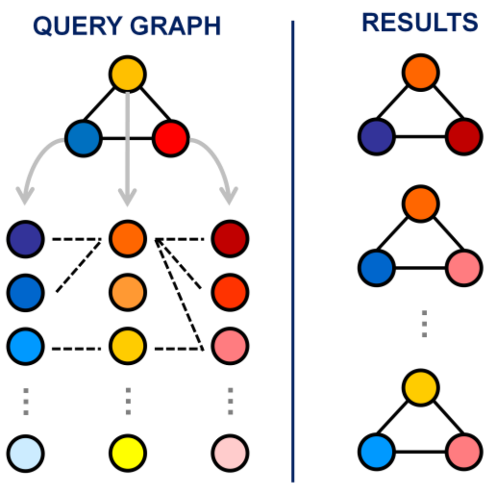

Figure 2: Greedy Cartesian product Short-listing: Given similar entity lists for each entity in the query graph, candidate sub-graphs from the Cartesian product between are derived greedily from the similar entity lists.

As can be seen in Figure 2, we begin by arraying the most-similar entity lists for each of the entities in the query graph (blue, yellow and red similar entity lists). From these aligned lists, we scan across horizontally, greedily forming members of sub-graphs from the Cartesian product of the similar entity lists. For each potential sub-graph found in this manner, we check for coincident activity, keeping only those hypothetical sub-graphs which are connected. Final scores for these valid sub-graphs are found by aggregating the similarity scores for each of the constituent entities.

2.1.2 Greedy Depth-First Search Short-listing

Greedy Depth-First Search is a technique to form and filter sub-graphs derived from a depth-first search of the connectivity structure of the full graph. This approach is outlined in the Figure below.

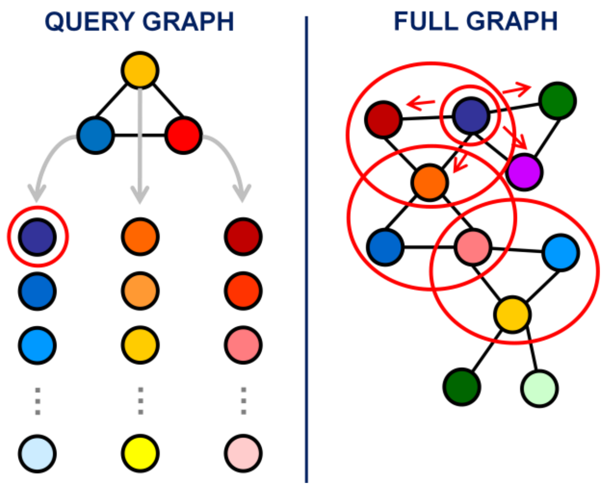

Figure 3: Greedy Depth-First Search Short-listing: Given the most similar entity among all similar entities, greedy depth-first search along the global graph is used to derive candidate sub-graphs.

As can be seen in Figure 3, starting with the most similar entity among all similar entities, we perform depth-first search along the entity connectivity structure of the full graph. The depth-first search recursively finds matching sub-graphs, maintaining a priority queue of aggregate scores for each found sub-graph. The top-K scoring sub-graphs from this queue are returned to the user as the final sub-graph search result.

2.2 Temporal Behavior Matching

Graph QuBE relies on Hidden Markov Models (HMM) for temporal behavior matching among sub-graphs. HMMs are doubly-stochastic probability models used to model temporal data (e.g. speech signals) and encode complex temporal behavior in a small number of parameters, θ. The Figure below illustrates how temporal behavior matching on graphs can be abstracted to an HMM model.

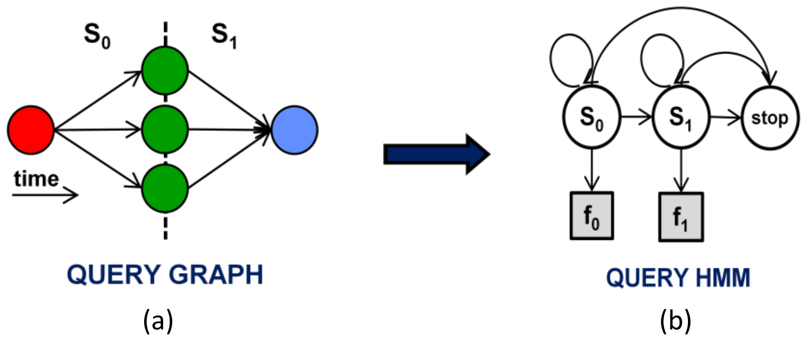

Figure 4: Abstracting a temporal query graph illustrating money laundering, (a), into a query HMM, (b).

Consider the Query Graph in Figure 4a. This graph exhibits a fan-out fan-in structure where each node represents an entity (e.g. account) and each directed edge represents an interaction (e.g. financial transaction). This structure evolves temporally, as illustrated by the time arrow.

We can abstract the temporal query graph show on the left to a query HMM shown on the right. This is done by first observing the temporal structure of this graph has two distinct phases, disbursement and collection, each of which has its own characteristics in term of observed behavior. Each stage, disbursement and collection, can be mapped to an unobserved HMM state, s_0 and s_1, and the observed behaviors in each stage can be mapped to feature vectors, f_0 and f_1. In this way, a complex series of temporal transactions between entities can be represented by a small set of parameters, θ, of a probability model.

Using this abstraction, temporal behavior matching via HMMs can be performed by the following steps. Model parameters, θ, for the query HMM model, Q_θ, are learned from the query graph. We use temporal featuring of transaction features to estimate state boundaries. To estimate state transition probabilities we use raw counts of state transitions found from the estimate state boundaries. Emission probabilities ($(f_0|s_0)$ and $p(f_1|s_1)$ in the example) are found using kernel density estimation using transaction and event-level features.

Following this, the likelihood score of each short-listed dynamic sub-graph is computed using Q_θ (e.g. score each sub-graph assuming Q_θ generated the data). This is done by computing the data likelihood given the model and the Maximum Likelihood (ML) estimate of the state-sequence that could have generated the data using an efficient version of the Viterbi algorithm. Finally, we re-rank the short-listed results based on these likelihoods and return the results to the user.

2.3 **Feature Extraction**

For practical application to transactional data, the Graph QuBE system utilizes *entity* features (short-listing) and *transactional* features (temporal behavior matching).

Entity-based features describe characteristics of each of the entities interacting with one another (e.g. average behavior, demographic information, etc.) which may be useful for matching entities against each other.

Transactional features are computed to represent the behaviors of entities interacting with one another. There are two types of transactional features: *transaction-level* features and *event-level* features.

Transaction-level features are features describing each independent transaction in the data set. These features basically describe how this transaction compares to other transactions this entity has had as well as to the broader population of all transactions.

Event-level features are features describing characteristics of transactions occurring within a specific time frame. These features are the *emission* features used in both HMM learning and inference. Since these features depend on both a specific set of interacting entities and a unique time frame, they are computed on-the-fly.

The exact form these features will take will vary on the application domain of interest. Below we highlight an example set of such features for transactional datasets released as part of the XDATA Summer Workshop Challenge Problems.

2.3.1 Example Features Extracted for XDATA Challenge Problem Datasets

As part of work in XDATA, MIT-LL along with Giant Oak, hand-constructed a set of features tailored to transactional data from the XDATA challenge problem datasets, specifically for the Kiva and Bitcoin financial data sets. A summary of entity and transaction-level features Graph QuBE used for these datasets are shown in Tables 1, 2 and 3 below, respectively.

Table 1: Entity Features

FEATURE	DESCRIPTION
credit_mean	Average of credits to entity
credit_std	Standard deviation of credits to entity
credit_interarr_mean	Average amount of time (milliseconds) between credits
credit_interarr_std	Standard deviation of time between credits
debit_mean	Average of debits to entity
debit_std	Standard deviation of debits to entity
debit_interarr_mean	Average amount of time (milliseconds) between debits
debit_interarr_std	Standard deviation of time between debits
perp_in	Perplexity[1] of the distribution of the number of incoming transactions from all other entities
perp_out	Perplexity of the distribution of the number of outgoing transactions to all other entities

[1] http://en.wikipedia.org/wiki/Perplexity

Table 2: Transaction-Level Features

FEATURE	DESCRIPTION
dev_pop	Deviation of the amount of a transaction from all other transactions
credit_dev	Deviation of a credit transaction from all other credits for the entity making the transaction
debit_dev	Deviation of a debit transaction from all other debits for the entity making the transaction

Table 3: Event-Level Features

FEATURE	DESCRIPTION
dev_pop_window	Deviation of the amount of a transaction from all other transactions in the window
dev_own_credits	Deviation of the amount of a transaction from entity's other credits in the window
dev_own_debits	Deviation of the amount of a transaction from entity's other debits in the window
t_since_previous	Time since previous transaction in window
rel_amt	Relative amount change between this and previous transaction

Before any of these features are used in our system, quantile-based univariate feature standardization is performed to normalize the data. Additionally, for the entity features, kd-tree indexing is performed for computational speed-up at run-time.

2.4 Technical Implementation and Technology Transfer

GraphQuBE is primarily written in Java with Apache Maven for build automation and dependency management. The GraphQuBE service runs as a REST-ful web service (via the Spark micro-framework) which exposes entity and pattern search functionality via Avro queries.

To improve modularity and forward-compatibility, for both entity and pattern search, the API for REST queries will continue to be the same even as underlying technologies improve. Additionally, our open source distribution contains example methods for data ingest and feature extraction from flat-files of an example dataset (small sample of Bitcoin transactions) to support developers adapting these tools to their own data.

As of the end of 2013, the initial version of Graph QuBE has been integrated into Influent, an Oculus XDATA Framework tool for flow analysis. Influent provides a framework for visualizing and interactively analyzing flow between entities, including the flow of entities. Graph QuBE serves as a back-end service for Influent's transactional pattern search interface which provides the ability for users to query for patterns of transactions involving multiple entities.