Towards a Query-by-Example System for Knowledge Graphs

Nandish Jayaram[†] Arijit Khan[#] Chengkai Li[†] Xifeng Yan[§] Ramez Elmasri[†]

[Y]*University of Texas at Arlington,* [#]*Systems Group, ETH Zurich,* [X]*University of California, Santa Barbara*

ABSTRACT

We witness an unprecedented proliferation of knowledge graphs that record millions of heterogeneous entities and their diverse relationships. While knowledge graphs are structure-flexible and content-rich, it is difficult to query them. The challenge lies in the gap between their overwhelming complexity and the limited database knowledge of non-professional users. If writing structured queries over "simple" tables is difficult, it gets even harder to query complex knowledge graphs. As an initial step toward improving the usability of knowledge graphs, we propose to query such data by example entity tuples, without requiring users to write complex graph queries. Our system, GQBE (Graph Query By Example), is a proof of concept to show the possibility of this querying paradigm working in practice. The proposed framework automatically derives a hidden query graph based on input query tuples and finds approximate matching answer graphs to obtain a ranked list of top-k answer tuples. It also makes provisions for users to give feedback on the presented top-k answer tuples. The feedback is used to refine the query graph to better capture the user intent. We conducted initial experiments on the real-world *Freebase* dataset, and observed appealing accuracy and efficiency. Our proposal of querying by example tuples provides a complementary approach to the existing keyword-based and query-graph-based methods, facilitating user-friendly graph querying. To the best of our knowledge, GQBE is among the first few emerging systems to query knowledge graphs by example entity tuples.

1. INTRODUCTION

Consider the scenario where a Silicon Valley business analyst is interested in writing an article on entrepreneurs who have founded technology companies head-quartered in California. She could be aware of only a few such founder-company pairs; nevertheless, to write the article, she must have a more comprehensive list of such tuples. In general, query users might be interested in finding heterogeneous entities (e.g., persons, products, organizations) that are related in certain ways.

GRADES 2014, June 22, 2014, Snowbird, Utah, USA.

DOI: 10.1145/2621934.2621937.

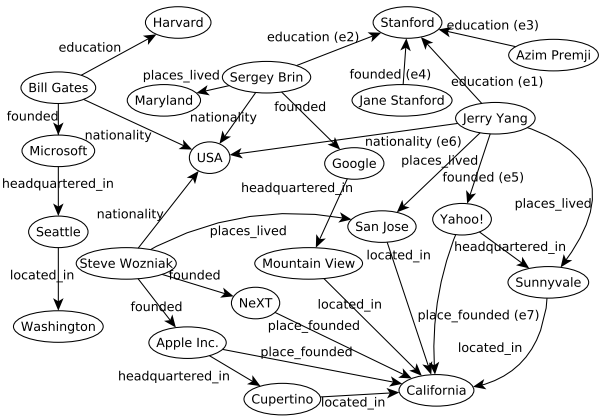


Figure 1: An Excerpt of a Knowledge Graph

Tasks like the above one are becoming increasingly common in several applications, including search, recommendation systems, and business intelligence. The analyst may have several ways to retrieve the founder-company list, such as querying a *knowledge graph* with some graph-querying procedure, e.g., using SPARQL. Particularly, the analyst may tap into knowledge graphs, which capture entities and their relationships. For example, Figure 1 is an excerpt of a knowledge graph, in which the edge labeled *founded* from node Jerry Yang to node Yahoo! captures the fact that the person founded the company. Instances of real-world knowledge graphs include *DBpedia* [3], *YAGO* [20], *Probase* [24], *Satori*[1], and *Freebase* [4] which powers Google's knowledge graph [2].

Although knowledge graphs are a great source of information, both users and application developers are often overwhelmed by the daunting task of understanding and using them, due to the sheer size and complexity of such data. Given its schema-less and semi-structured nature, a knowledge graph is typically represented as a set of ⟨subject, property, object⟩ triples. The Linking Open Data community has interlinked billions of RDF triples spanning over several hundred datasets, and these are stored in relational databases, graph databases, and triple-stores. In retrieving data from these databases, the norm is often to use structured query languages such as SQL, SPARQL, and those alike. However, writing structured queries requires extensive experiences in query language and data model, as well as a good understanding of the particular dataset [7].

Motivated by the aforementioned usability challenge, we present GQBE, a framework that can be used to query knowledge graphs by example tuples of entities without having to write complex struc-

[1] http://accounts.satori-design.com/index.php

[2] http://www.google.com/insidesearch/features/search/knowledge.html

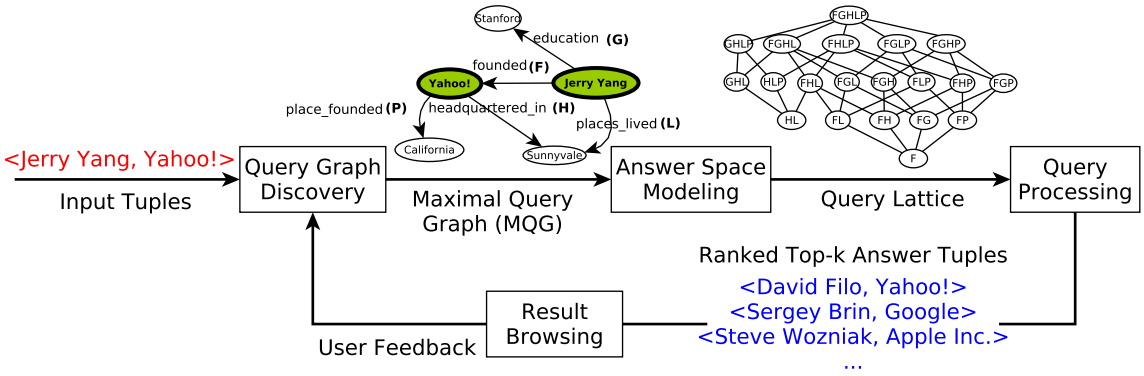


Figure 2: The Architecture and Components of GQBE

tured queries. [9] presents a proof of concept of this system which shows that such a querying paradigm is indeed a viable option.

Example 1 (Querying by Example Tuple): ⟨Jerry Yang, Yahoo!⟩ is an example input tuple that satisfies the analyst's query intent. This is shown in red as the input to GQBE in Fig. 2. Given the knowledge graph in Figure 1, GQBE finds answers such as ⟨David Filo, Yahoo!⟩ and ⟨Sergey Brin, Google⟩ as shown in blue in Fig. 2. It is to be noted here that the entities like Jerry Yang and Google are specific vertices in the knowledge graph, and not keywords.

Substantial progress has been made on query mechanisms that help users construct query graphs or even do not require explicit query graphs. Such mechanisms include keyword search (e.g., [11]), keyword-based query formulation [18, 25], and interactive and form-based query formulation [5, 8]. Note that query graphs or patterns are often used in the literature to graphically present queries over graphs. Underlyingly query graphs are formed by using structured query languages or some of the aforementioned query mechanisms. Therefore query graph is not what we refer to as "example". In fact, one of the components in GQBE discovers a hidden query graph from input entity tuples. Query by example (QBE) has a long positive history in relational databases [26]. Particularly, QBE and keyword-based methods are adequate for different usage scenarios. Using keyword-based methods, a user has to articulate query keywords, e.g., "technology companies head-quartered in California and their founders." for the aforementioned analyst. Not only a user may find it challenging to clearly articulate a query, but also a query system might not return accurate answers, since it is non-trivial to precisely separate these keywords and correctly match them with entities, entity types, and relationships. This has been verified through our own experience on a keyword-based system adapted from SPARK [15]. In contrast, a GQBE user only needs to know the names of some entities in example tuples, without being required to specify how exactly the entities are related. On the other hand, keyword-based querying is more adequate when a user does not know a few sample answers with respect to her query.

More formally, GQBE addresses the following problem. A ***data graph*** is a directed multi-graph G with node set $V(G)$ and edge set $E(G)$. Each node $v \in V(G)$ represents an entity and has a unique identifier $id(v)$. Each edge $e=(v_i, v_j) \in E(G)$ denotes a directed relationship from entity v_i to entity v_j. It has a label, denoted as $label(e)$. Multiple edges can have the same label. The user input and output of GQBE are both entity tuples, called ***query tuples*** and ***answer tuples***, respectively. A tuple $t=\langle v_1, \ldots, v_n \rangle$ is an ordered list of entities (i.e., nodes) in G. The constituting entities of query (answer) tuples are called *query (answer) entities*. Given a data graph G and a query tuple t, our goal is to find the top-k most similar answer tuples to t.

Since the users of GQBE are not required to construct a query graph, the system automatically discovers a hidden weighted query graph to capture the user's query intent. The system then finds a ranked list of similar answer tuples that are presented to the user. Provisions for the user to provide feedback on the answer tuples are made using which the user can re-rank the answer tuples and also mark their relevance. This iterative feedback mechanism is used by the system to refine the query graph until the user is satisfied with the results.

2. SYSTEM OVERVIEW

Figure 2 shows the various components involved in GQBE's framework. Since the input to the system is an example tuple, *Query Graph Discovery* module automatically discovers a weighted query graph that captures important relationships and entities between and around the entities in the example tuple. This module must discover the hidden query graph behind the example tuple that captures the user intent. Such a graph is called the *maximal query graph* (MQG) and edge-isomorphic answer graphs are projected to form various answer tuples.

It is unlikely that we will find an edge-isomorphic match to the MQG, so it becomes imperative to find approximate matches to the MQG. The *Answer Space Modeling* component of GQBE models the space of all answer graphs as a *query lattice* formed by the subsumption relationship between all subgraphs of the MQG. An edge-isomorphic match to a subgraph of the MQG is an approximate match to the MQG. The query lattice can be large and obtaining the top-k answer tuples requires evaluating these query graphs. For a large lattice, the brute-force approach of evaluating all the lattice nodes can be prohibitively expensive. The *Query Processing* module efficiently finds the top-k answer tuples, which would stop after partial lattice evaluation if the top-k answer tuples are found.

The top-k answer tuples obtained are presented to the user by the *Result Browsing* module. The user can browse through the presented top-k answer tuples and provide feedback to the system. The user can mark the relevant and irrelevant answers among the presented top-k answer tuples and also re-rank the answer tuples. This kind of feedback helps the system better understand the user intent since it gets both positive and negative example tuples from the feedback. This is an iterative process that will help the user in refining the query until her exact query intent is captured by the system and is satisfied with the answer tuples.

3. CURRENT SOLUTION

3.1 Query Graph Discovery

The goal of the *Query Graph Discovery* module is to discover a hidden maximal query graph that captures relationships that a user might be interested in for the given example tuple. Based on the data graph statistics, we assign weights to the edges which are used to retrieve the maximal query graph.

To illustrate the idea of assigning edge weights, let us consider the knowledge graph in Figure 1 and let ⟨Jerry Yang, Yahoo!⟩ be the input query tuple. First, we capture neighbors of query entities up to some pre-determined length d to construct a *neighborhood graph* (H), and then remove the *unimportant* edges from this neighborhood graph. For example, in Figure 1, edge e_1=(Jerry Yang, Stanford) labeled *education*, represents an important relationship between Stanford and a query entity Jerry Yang. But other edges labeled *education* and incident on Stanford (edges e_2 and e_3 in Figure 1) are deemed unimportant (and thus removed) with regard to query entity Jerry Yang, since there can be many graduates of Stanford in the knowledge graph. Despite pruning such clearly unimportant edges, the neighborhood graph H can still be large and its edges must thus be further pruned. We hence rank the edges of H by weighting them using several distance-based and frequency-based heuristics:

$$\mathsf{w}(e) = \mathsf{ief}(e) \;/\; (\mathsf{p}(e) \times \mathsf{d}^2(e)) \tag{1}$$

The weight $\mathsf{w}(e)$ of an edge e=(u, v) is: (1) directly proportional to its inverse edge frequency, $\mathsf{ief}(e)$, that captures how rare a relationship is globally in the data graph; (2) inversely proportional to its participation, $\mathsf{p}(e)$, that determines the number of edges in the data graph that share the same label and one of e's end nodes (u or v); and (3) inversely proportional to the distance, $\mathsf{d}(e)$, that captures the distance of edge e from the query entities. A greedy heuristic is used to choose the MQG (e.g., MQG in Figure 2), that is an m-edged *weakly connected* subgraph of H containing all query entities, while maximizing the total edge weight.

3.2 Query Processing

In order to find approximate matches to the maximal query graph, GQBE models the space of all answer graphs as a *query lattice* formed by the subsumption relationship between all subgraphs of the MQG. Fig. 3(a) is an example query lattice corresponding to the MQG in Fig. 2. An approximate answer graph is defined as an edge-isomorphic match to some query graph, which is a subgraph of the MQG, and is present in the query lattice as a lattice node. The query lattice can be large, and obtaining the top-k answer tuples [3] requires evaluating all these query graphs. Given a query lattice, a brute-force approach is to evaluate all lattice nodes (query graphs) to find all answer tuples. Its exhaustive nature leads to clear inefficiency, since we only seek the top-*k* answers. Moreover, the brute-force method evaluates all the queries separately, without sharing any computation.

We propose an efficient algorithm which allows sharing of computation. We employ an upper-bound based *bottom-up*, *best-first* strategy to explore the lattice. To process a query Q, at least one of its children Q'=$Q-e$ must have been processed. The results of Q' is used to efficiently process Q. At any given moment during query lattice evaluation, the lattice nodes belong to three mutually-exclusive sets—the evaluated, the unevaluated and the pruned. A subset of the unevaluated nodes, denoted the *lower-frontier* ($\mathcal{LF}$), are candidates for the node to be evaluated next. At the beginning, $\mathcal{LF}$ contains only the minimal query trees (nodes *F* and *HL* in Fig. 3(a)). After a node is evaluated, all its parents are added to $\mathcal{LF}$. Therefore, the nodes in $\mathcal{LF}$ either are minimal query trees or have at least one evaluated child. For each unevaluated candidate node Q in $\mathcal{LF}$, we define an *upper-bound score*, which is the best score Q's answer tuples can possibly attain. In the beginning all nodes in $\mathcal{LF}$ have the score of the MQG (node *FGHLP*) as its upper-bound score. The chosen node, Q_{best}, must have the highest upper-bound score among all the nodes in $\mathcal{LF}$. If evaluating Q returns an answer graph A, A has the potential to grow into an answer graph A' to an ancestor node Q'. In such a case, A and A' are projected to the same answer tuple t_A=$t_{A'}$. The answer tuple always gets the better score from A', under the simplified scoring function that sums up weights of all edges in query graph Q' and assigns it as A''s score.

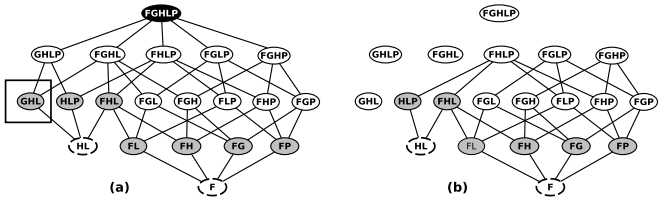


Figure 3: Query Lattice at Different Stages

Hence, Q's upper-bound score depends on its *upper boundary*—Q's unpruned ancestors that have no unpruned parents.The ***upper boundary*** of a node Q in $\mathcal{LF}$, denoted $\mathcal{UB}(Q)$, consists of nodes Q' in the ***upper-frontier*** ($\mathcal{UF}$), which is the set of unpruned nodes without unpruned parents, that subsume or is equal to Q. The ***upper-bound score*** of a node Q is the maximum score of any query graph in its upper boundary. Fig. 3(a) shows a lattice where nodes *F* and *HL* are the evaluated nodes and all the nodes shaded grey belong to $\mathcal{LF}$. Node *FGHLP* is the only node in the upper-frontier $\mathcal{UF}$. Node *GHL* is the Q_{best} next chosen for evaluation. A lattice node that does not have any answer graph is referred to as a *null node*. If Q_{best} (node *GHL*) turns out to be a null node after evaluation, all its ancestors are also null nodes as shown in Fig. 3(b). Such null nodes are pruned and thus the lattice changes dynamically. For nodes in $\mathcal{LF}$ that have at least one upper boundary node among the pruned ones, the change of $\mathcal{UF}$ leads to changes in their upper boundaries and, sometimes, their upper-bound scores too. The $\mathcal{UF}$ changes to nodes *FHLP*, *FGLP* and *FGHP* in Fig. 3(b). The new upper boundaries and upper-bound scores can efficiently recomputed, the details of which can be found in [10].

The algorithm terminates when the current score of the k^{th} best answer tuple so far is greater than the upper-bound score of the next Q_{best} chosen by the algorithm. This termination condition guarantees that we cannot get any answer tuple better than the current top-k by executing any other unevaluated node in the lattice.

Finally, given a query graph from the query lattice, we discuss the method to find all edge-isomorphic embeddings of that query graph. A query graph can be represented as a set of RDF triples (source, property, object). We adopt the vertical partitioning method [1] and maintain a table for each property with two columns ($subj$, obj), for the edges' source and destination nodes, respectively. For efficient query processing, two in-memory hash tables are created on each table, using *subj* and *obj* as the hash keys, respectively. Multi-way join query is used to evaluate a minimal query tree. For instance, lattice node *HL* in the lattice in Fig. 3(a) corresponds to the following query:

```
SELECT F.subj as n1, F.obj as n2, L.obj as n3
FROM F, L
WHERE F.subj=L.subj
```

[3] The details of the ranking criteria of the answer tuples are discussed in the extended version of the paper [10].

After a query graph is processed, its answers are materialized into a file (say file `HL` in this case) and used to process any of its parent nodes. For instance, lattice node *GHL* is evaluated using the following join query:

SELECT `HL.*, G.obj as` n_4
FROM `G,HL`
WHERE `G.subj=HL.`n_1

4. WORK IN PROGRESS

There are multiple optimizations and improvements that can be made to the system. For instance, as we can notice, the edge weighting function described in Eq. 2 is a best-effort based method of guessing what might be the important relationships to include. The MQG thus obtained tries to include edges that are important based on the data graph statistics and not on user query logs. The query processing method described in Section 3.2 performs multiple subgraph matching on a single machine. This method does not try to utilize various distributed computing platforms that can potentially be used to process various parts of the lattice independently. Lastly, obtaining the user feedback is of great importance for the system to identify the exact user intent. The user feedback can be used to learn and refine the original query graph.

4.1 Interactive and Iterative Query Suggestion

This approach treats user's example tuple as a partial query graph (or as anchor nodes A) and produces a complete query graph through interactions with user. It iteratively makes suggestions for new edges to be added to the partial query graph, which are either accepted or rejected by the user. X is the set of suggested edges and user responses to them such that $\forall x_i \in X, x_i = (e_i, r_i)$, where e_i is the suggested edge and $r_i \in \{yes, no\}$ is the user response. A suggested edge has to be connected to the partial query graph.

Example 2 (Query Graph Completion): Consider Fig. 4(a) as the user provided set of anchor nodes A and Fig. 4(b) as the target graph Q_t. Given the anchor nodes, if the system decided at iteration $t = 1$ the best edge to recommend is e_5, this edge is presented to the user. The user accepts this edge and $x_1 = (e_5, yes)$. If the next edge suggested at $t = 2$, given $X = ((e_5, yes))$ is e_6, the user response to this suggestion would be a no, and $x_2 = (e_6, no)$. If the system decides that the next best edge suggestion at $t = 3$, given $X = ((e_5, yes), (e_6, no))$ is e_1, the user response to this edge suggestion would be an yes, and $x_3 = (e_1, yes)$. Given $X = ((e_5, yes), (e_6, no), (e_1, yes))$, if the next edge suggested at $t = 4$ is e_7, the edge is accepted since it is in the target graph. The final edge suggestion e_7 leads to $X = ((e_5, yes), (e_6, no), (e_1, yes), (e_7, yes))$, which is the desired target graph.

The key to this approach is to recommend the next edge based on suggestions and user responses from previous m iterations, and a user log (the first edge suggestion is based only on the latter). Given data graph G and a set of possible user responses $R = \{yes, no\}$, a sequence of pairs of (suggested edge, user response) is $X = x_1, x_2, \ldots x_m$ where $x_i = (e_i, r_i) \in E(G) \times R$ for $i \in [1, m]$. The next pair of edge suggestion and user response, i.e., $x_{m+1} = (e_{m+1}, r_{m+1})$, is conditioned on X, and we would like to find the x_{m+1} that maximizes $P(x|x_1, \ldots x_m)$, i.e.,

$$x_{m+1} = \arg\max_{x \in E \times R} P(x|x_1, \ldots x_m) \quad (2)$$

Various models like naive Bayesian classifiers and Conditional Random Fields (CRF) [14] can be used to solve Eq. 2. A user query log that captures the edge suggestions and their corresponding user responses will be helpful to estimate the probability value.

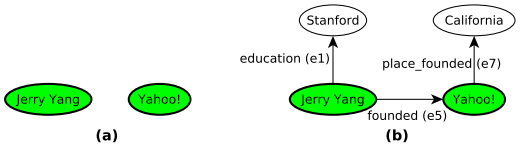


Figure 4: Query Graph Completion

Such a query log might not be easily available, since they are often proprietary to a company or unavailable. We thus plan to simulate such a query log, with the help of Wikipedia. Wikipedia sentences are parsed and entities in them are identified and mapped to nodes in the data graph. Edge labels between the mapped entities can be approximated as occurring in the sentences. Edges co-occurring in the neighboring sentences of a textual context can be viewed as co-occurring edges in a query. Some properties that appear in one set of sentences and not in another can be injected to simulate edge suggestions that were rejected by users.

4.2 Distributed Query Processing

As mentioned earlier, several graphs in the lattice are evaluated to find the top-k answer tuples. Popular distributed graph matching systems like [21, 19] find all exact matches of a single query graph efficiently. Since we must evaluate a lattice with several query graphs to find the top-k answer tuples, we need a distributed lattice traversal mechanism too.

One way to do this is to partition the lattice and process each lattice partition in parallel. If one of the lattice nodes in a partition is a null node, then this affects the number of lattice nodes to be evaluated in other partitions. The main challenge to address here is how to stop the lattice exploration once the top-k answer tuples are obtained, while ensuring a lattice node which is a super-graph of one of the null nodes of a different partition is not processed.

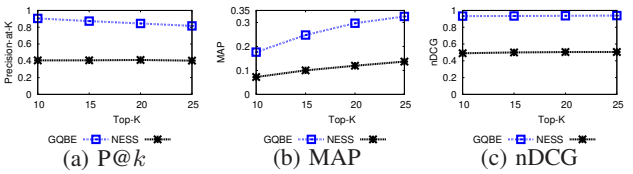
The other way of using a distributed framework for query processing is to partition the data graph across the various machines in the framework. Each lattice node may find matching answers in different partitions of the data graph. We have to design clever scheduling mechanisms to identify the various machines a single lattice node must be executed in, and also efficient join strategies if an answer graph is found across multiple partitions. Here we can leverage on the various existing distributed processing systems that partition the data graph, to evaluate a single lattice node. But, this has the additional challenge of scheduling the *best-first* lattice traversal method. Scheduling and processing various lattice nodes in parallel is a problem to be addressed.

4.3 Refining MQG with User Feedback

The feedback about the quality of the answer tuples can be used to indicate three different kinds of information. First, the user can mark the relevant answer tuples, which would provide positive example tuples to the system. Second, the user can mark irrelevant answer tuples, which would provide negative example tuples to the system. Third, the user can also re-rank the top-k answer tuples presented which can be used to indicate the degree of the impact each positive example tuple has.

Given a set of query tuples T (both positive and negative examples along with the ranking information) obtained through the feedback mechanism, our approach is to search for the optimal query graph under a clear optimization goal–the positive query tuples in T should be ranked as high as possible, adhering to the user given ranking, while the negative query tuples must be omitted.

Suppose M_T is the set of all possible MQGs (under a size constraint) given T. Thus we look for $\arg\max_{M \in M_T} f(M)$, where



(a) P@k (b) MAP (c) nDCG

Figure 5: Accuracy of GQBE and NESS on Freebase Queries

f is the optimization function. Given the optimization goal, f can naturally be a commonly used function for evaluating ranked retrieval results, e.g., Precision-at-k (P@k), Mean Average Precision (MAP) and Normalized Discounted Cumulative Gain (nDCG). A fundamental challenge to this approach of course is the potentially huge space of M_T. We thus shall look for randomized or approximation algorithms. Another challenge lies in the cost of calculating f for various queries in M_T , which in the most straightforward sense entails evaluating the queries themselves. To deal with this, we need to design efficient index structures and multi-query optimization methods for sharing the costs in evaluating queries.

A simple heuristics-based approach to accommodate multiple positive example tuples is to discover individual MQG M_{t_i}, for each example tuple t_i, and produce a merged and re-weighted MQG that captures the importance of edges with respect to their presence across multiple MQGs. The intuition here is that users will most likely mark answer tuples that have some relationships in common as relevant answers. So more weight is given to edges that are common across multiple MQGs. The edge weights of the final combined MQG is re-computed based on the number of MQGs a particular edge appears in. Corresponding query entities can be merged to virtual nodes and edges occurring in multiple MQGs can be merged only if they share both the end vertices too, and not just the label. The increase in weight of such merged edges is proportional to the number of MQGs the edge appears in. The weights of all edges that are not common this way are kept as is.

5. EXPERIMENTS

This section presents our initial experiment results on the accuracy and efficiency of GQBE. The experiments were conducted on a double quad-core 24 GB Memory 2.0 GHz Xeon server.

Dataset The experiments were conducted over a large real-world knowledge graph, the Freebase [4] dataset. We preprocessed the graph so that the kept nodes are all named entities (e.g., Yahoo!) and abstract concepts (e.g., Jewish people). The resulting Freebase graph contains 28M nodes, 47M edges, and 5,428 distinct edge labels.

Methods Compared GQBE was compared with a Baseline and NESS [13]. NESS is a graph querying framework that finds approximate matches of query graphs with unlabeled nodes which correspond to query entity nodes in MQG. Note that, like other systems, NESS must take a query graph (instead of a query tuple) as input. Hence, we feed the MQG discovered by GQBE as the query graph to NESS. Since, NESS does not consider edge-labeled graphs, we adapted it by requiring each candidate node v' of v to have at least one incident edge in the data graph bearing the same label of an edge incident on v in the query graph. Baseline explores a query lattice in a bottom-up manner and prunes ancestors of null nodes. However, differently, it evaluates the lattice by breadth-first traversal instead of in the order of upper-bound scores. Baseline terminates when every node is either evaluated or pruned.

Queries and Ground Truth Queries covering diverse domains, including people, companies, movies, sports, awards, religions, universities and automobiles were chosen. Twenty Freebase queries F_1–F_{20} are based on Freebase tables such as http:// www.freebase.com/view/computer/programming_language_designer?instances, except F_1 and F_6 which are from Wikipedia tables such as http://en.wikipedia.org/wiki/List_of_English_football_club_owners. Each such table is a collection of tuples, in which each tuple consists of one, two, or three entities. For each table, we used one or more tuples as query tuples and the remaining tuples as the ground truth for query answers.

Accuracy Based on Ground Truth

We measured the accuracy of GQBE and NESS by comparing their query results with the ground truth. The accuracy on a set of queries is the average of accuracy on individual queries. The accuracy on a single query is captured by three widely-used measures [16], P@k, MAP and nDCG.

Fig.5 shows these measures for different values of k on the Freebase queries. GQBE has high accuracy. For instance, its P@25 is over 0.8 and nDCG at top-25 is over 0.9. The absolute value of MAP is not high, merely because Fig.5(b) only shows the MAP for at most top-25 results, while the ground truth size (i.e., the denominator in calculating MAP) for many queries is much larger. Moreover, GQBE outperforms NESS substantially, as its accuracy in all three measures is almost always twice as better. This is because GQBE gives priority to query entities and important edges in MQG, while NESS gives equal importance to all nodes and edges except the pivot. Furthermore, the way NESS handles edge labels does not explicitly require answer entities to be connected by the same paths between query entities.

Efficiency Results

We compared the efficiency of GQBE, NESS and Baseline on Freebase queries. Fig.6 compares the three methods' query processing time, in logarithmic scale. The figure shows the query processing time for each of the 20 Freebase queries, and the edge cardinality of the MQG for each of those is shown below the corresponding query id. The query cost does not appear to increase by edge cardinality, regardless of the query method. For GQBE and Baseline, this is because query graphs are evaluated by joins and join selectivity plays a more significant role in evaluation cost than number of edges. NESS finds answers by intersecting postings lists on feature vectors. Hence, in evaluation cost, intersection size matters more than edge cardinality. GQBE outperformed NESS on 17 of the 20 queries and was more than 3 times faster in 10 of them. Baseline clearly suffered, due to its inferior pruning power compared to the best-first exploration employed by GQBE.

6. RELATED WORK

[17] also proposes the new querying paradigm of querying knowledge graphs by example tuples. But there are several features present in GQBE and not in [17]. (1) they do not specify a concrete way to discover the query graph, (2) they have no way to accept user feedback to better understand the query intent and (3) their query processing component mandates all answer graphs to be isomorphic to the initial query graph, thus precluding any approximate matching answer graphs from being found.

Several works [12, 6] identify the best subgraphs/paths in a data graph to describe how several input nodes are related. But the paths discovered by their techniques only connect the input nodes and have the further limitation of allowing only two input entities. Finding matching answer graphs to the discovered query graph is also not within the focus of the aforementioned works.

There are many studies on approximate/inexact subgraph matching in large graphs, such as G-Ray [23] and TALE [22] GQBE's query processing component is different from them on several aspects. First, GQBE only requires to match edge labels and match-

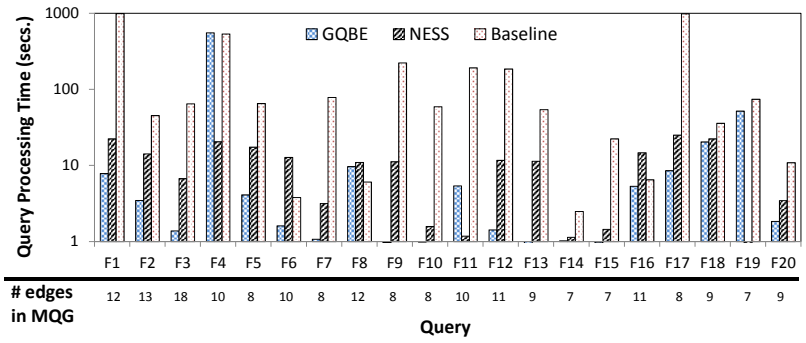


Figure 6: Query Processing Time

ing node identifiers is not mandatory. Second, in GQBE, the top-k query algorithm centers around query entities. More specifically, the weighting function gives more importance to query entities and the minimal query trees mandate the presence of entities corresponding to query entities. On the contrary, previous methods give equal importance to all nodes in a query graph, since the notion of query entity does not exist there. Finally, although the query relaxation DAG proposed in [2] is similar to GQBE's query lattice, the scoring mechanism of their relaxed queries is different and depends on XML-based relaxations.

7. CONCLUSION

We introduced a framework to query knowledge graphs by example entity tuples. As an initial step toward better usability of graph query systems, GQBE saves users the burden of forming explicit query graphs. If an example of what the user is looking for is fed to the system, a hidden query graph corresponding to the query tuple is automatically discovered. Top-k answer tuples are found by the query processing module and feedback on the quality of the answer tuples is solicited from the user. This feedback is used to better understand the user intent and refine the query graph. Initial experiments on Freebase dataset show that our proof of concept system GQBE outperforms the state-of-the-art system NESS on both accuracy and efficiency. Our future plan aims at finding better ways to discover and process the query graph. Concrete plans to improve the feedback mechanism to better capture the query intent are also laid out.

8. ACKNOWLEDGEMENTS

This work of Li is partially supported by NSF IIS-1018865, CCF-1117369, 2011 and 2012 HP Labs Innovation Research Awards, and the National Natural Science Foundation of China Grant 61370-019. The work of Yan was partially supported by the Army Research Laboratory under cooperative agreement W911NF-09-2-0053 (NSCTA). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB'07*.

[2] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and content scoring for xml. In *VLDB*, 2005.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a Web of open data. In *ISWC*, 2007.

[4] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250, 2008.

[5] E. Demidova, X. Zhou, and W. Nejdl. FreeQ: an interactive query interface for Freebase. In *WWW*, demo paper, 2012.

[6] L. Fang, A. D. Sarma, C. Yu, and P. Bohannon. REX: explaining relationships between entity pairs. In *PVLDB*, pages 241–252, 2011.

[7] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD'07*.

[8] M. Jarrar and M. D. Dikaiakos. A query formulation language for the data web. *TKDE*, 24:783–798, 2012.

[9] N. Jayaram, M. Gupta, A. Khan, C. Li, X. Yan, and R. Elmasri. GQBE: Querying knowledge graphs by example entity tuples. In *ICDE (demo description)*, 2014.

[10] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. *CoRR*, abs/1311.2100, 2013.

[11] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, pages 681–692, 2011.

[12] G. Kasneci, S. Elbassuoni, and G. Weikum. MING: mining informative entity relationship subgraphs. In *CIKM*, 2009.

[13] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD'11*.

[14] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*.

[15] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.

[16] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, NY, USA, 2008.

[17] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. In *VLDB*, 2014 (to appear).

[18] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, pages 423–434, 2010.

[19] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. SIGMOD '13, pages 505–516, 2013.

[20] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge unifying WordNet and Wikipedia. In *WWW'07*.

[21] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, pages 788–799, 2012.

[22] Y. Tian and J. M. Patel. TALE: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[23] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. *KDD*, 2007.

[24] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD*, pages 481–492, 2012.

[25] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword query reformulation on structured data. *ICDE*, pages 953–964, 2012.

[26] M. M. Zloof. Query by example. In *AFIPS*, 1975.