Querying Knowledge Graphs by Example Entity Tuples*

Nandish Jayaram, Arijit Khan, Chengkai Li *Member, IEEE*, Xifeng Yan *Member, IEEE*, Ramez Elmasri

Abstract—We witness an unprecedented proliferation of knowledge graphs that record millions of entities and their relationships. While knowledge graphs are structure-flexible and content-rich, they are difficult to use. 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, complex graphs are only harder to query. 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 form complex graph queries. Our system, GQBE (Graph Query By Example), automatically discovers a weighted hidden maximum query graph based on input query tuples, to capture a user's query intent. It then efficiently finds and ranks the top approximate matching answer graphs and answer tuples. We conducted experiments and user studies on the large Freebase and DBpedia datasets and observed appealing accuracy and efficiency. Our system provides a complementary approach to the existing keyword-based methods, facilitating user-friendly graph querying. To the best of our knowledge, there was no such proposal in the past in the context of graphs.

Index Terms—Knowledge Graphs, Entity Graphs, Query by Example, Graph Query Processing

1 Introduction

1.1 Motivation

There is an unprecedented proliferation of *knowledge graphs* that record millions of entities (e.g., persons, products, organizations) and their relationships. Fig.1 is an excerpt of a knowledge graph, in which the edge labeled *founded* between nodes Jerry Yang and Yahoo! captures the fact that the person is a founder of the company. Examples of real-world knowledge graphs include DBpedia [3], YAGO [19], Freebase [4] and Probase [24]. Users and developers are tapping into knowledge graphs for numerous applications, including search, recommendation, and business intelligence.

Both users and application developers are often overwhelmed by the daunting task of understanding and using knowledge graphs. This largely has to do with the sheer size and complexity of such data. As of March 2012, the Linking Open Data community had interlinked over 52 billion RDF triples spanning over several hundred datasets. More specifically, the challenges lie in the gap between complex data and non-expert users. Knowledge graphs are often stored in relational databases, graph databases and triplestores. 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, data model, and a good understanding of particular datasets [10]. *If querying "simple" tables is difficult, aren't complex graphs harder to query?*

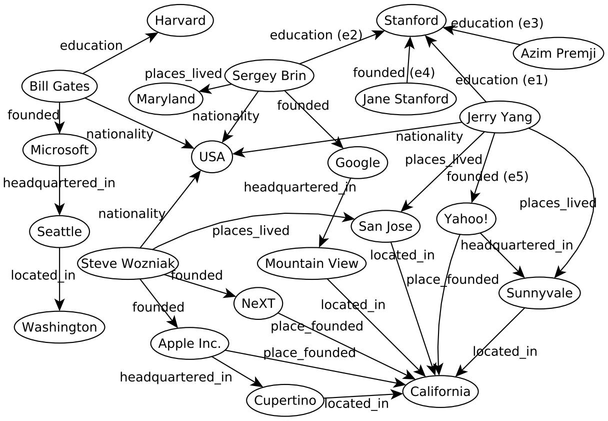


Fig. 1: An Excerpt of a Knowledge Graph

Motivated by the aforementioned usability challenge, we build GQBE[1] (Graph Query by Example), a system that queries knowledge graphs by example entity tuples instead of graph queries. Given a data graph and a query tuple consisting of entities, GQBE finds similar answer tuples. Consider the data graph in Fig.1 and an scenario where a Silicon Valley business analyst wants to find entrepreneurs who founded technology companies head-quartered in California. Suppose she knows an example query tuple such as ⟨Jerry Yang, Yahoo!⟩ that satisfies her query intent. Entering such an example tuple to GQBE is simple, especially assisted by user interface tools such as auto-completion in identifying the exact entities in the data graph. The answer tuples can be ⟨Steve Wozniak, Apple Inc.⟩ and ⟨Sergey Brin, Google⟩, which are founder-company pairs.

- *N. Jayaram, C. Li and R. Elmasri are with the Department of Computer Science and Engineering, The University of Texas at Arlington. E-mail: nandish.jayaram@mavs.uta.edu, cli@uta.edu, elmasri@uta.edu*
- *A. Khan is with the Systems Group, ETH Zurich. The work was done at UCSB. E-mail: arijit.khan@inf.ethz.ch*
- *X. Yan is with the Department of Computer Science, University of California, Santa Barbara. E-mail: xyan@cs.ucsb.edu*

. *This work is partially supported by NSF IIS-1018865, IIS-14-08928, CCF-1117369, the Army Research Laboratory under cooperative agreement W911NF-09-2-0053 (NSCTA), 2011 and 2012 HP Labs Innovation Research Awards, and the National Natural Science Foundation of China Grant 61370019. We also thank Ning Yan for his useful contributions during our discussions.

[1] A demonstration video of the system can be found at http://www.youtube.com/watch?v=4QfcV-OrGmQ.

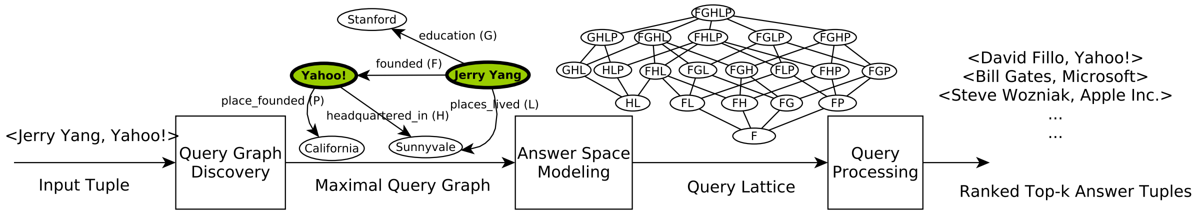


Fig. 2: The Architecture and Components of GQBE

If the query tuple consists of 3 or more entities (e.g., ⟨Jerry Yang, Yahoo!, Sunnyvale⟩), the answers will be similar tuples of the same cardinality (e.g., ⟨Steve Wozniak, Apple Inc., Cupertino⟩).

1.2 Overview and Contributions

GQBE is among the first to query knowledge graphs by example entity tuples. An brief overview of the entire system and a demonstration description can be found in [12] and [11], respectively. There are several challenges in building GQBE. Below we provide a brief overview of our approach in tackling these challenges. The ensuing discussion refers to the system architecture and components of GQBE, as shown in Fig. 2.

(1) With regard to *query semantics*, since the input to GQBE is a query tuple instead of an explicit query graph, it must derive a hidden query graph based on the query tuple, to capture the user's query intent. GQBE's *query graph discovery* component (Sec.3) fulfills this requirement and the derived graph is termed a *maximum query graph* (MQG). The edges in MQG, weighted by several frequency-based and distance-based heuristics, represent important "features" of the query tuple to be matched in answer tuples. More concretely, they capture how entities in the query tuple (i.e., nodes in a data graph) and their neighboring entities are related to each other. Answer graphs matching the MQG are projected to answer tuples, which consist of answer entities corresponding to the query tuple entities. GQBE further supports multiple query tuples as input which collectively better capture the user intent.

(2) With regard to *answer space modeling* (Sec.4), there can be a large space of approximate answer graphs (tuples), since it is unlikely to find answer graphs exactly matching the MQG. GQBE models the space of answer tuples by a *query lattice* formed by the subsumption relation between all possible query graphs. Each query graph is a subgraph of the MQG and contains all query entities. Its answer graphs are also subgraphs of the data graph and are edge-isomorphic to the query graph. Given an answer graph, its entities corresponding to the query tuple entities form an answer tuple. Thus the answer tuples are essentially approximate answers to the MQG. For ranking answer tuples, their scores are calculated based on the edge weights in their query graphs and the match between nodes in the query and answer graphs.

(3) The query lattice can be large. To obtain top-k ranked answer tuples, the brute-force approach of evaluating all query graphs in the lattice can be prohibitively expensive. For *efficient query processing* (Sec.5), GQBE employs a top-k lattice exploration algorithm that only partially evaluates the lattice nodes in the order of their corresponding query graphs' upper-bound scores.

We summarize the contributions of this paper as follows:

- For better usability of knowledge graph querying systems, we propose a novel approach of querying by example entity tuples, which saves users the burden of forming explicit query graphs.
- The query graph discovery component of GQBE derives a hidden maximum query graph (MQG) based on input query tuples, to capture users' query intent. GQBE models the space of query graphs (and thus answer tuples) by a query lattice based on the MQG.
- GQBE's efficient query processing algorithm only partially evaluates the query lattice to obtain the top-k answer tuples ranked by how well they approximately match the MQG.
- We conducted extensive experiments and user study on the large Freebase and DBpedia datasets to evaluate GQBE's accuracy and efficiency (Sec.7). The comparison with a state-of-the-art graph querying framework NESS [14] and an exemplar query system EQ [18] shows that GQBE is over twice as accurate as NESS and EQ. GQBE also outperforms NESS on efficiency in most of the queries.

2 Problem Formulation

GQBE runs queries on knowledge data graphs. 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 $id(v)$. [2] 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 answer tuples t' with the highest similarity scores $\mathsf{score}_t(t')$.

We define $\mathsf{score}_t(t')$ by matching the inter-entity relationships of t and that of t'. The best matches for individual entities in t may not form the best match for the query tuple t as a whole. It is thus imperative to form a query graph involving the entities of the query tuple and other neighboring relationships and entities. These neighboring relationships and entities are important "features" that might be of interest to users. Thus $\mathsf{score}_t(t')$ entails matching two graphs constructed from t and t', respectively.

To this end, we define the *neighborhood graph* for a tuple, which is based on the concept of undirected path. An *undirected path* is a path whose edges are not

[2] Without loss of generality, we use an entity's name as its identifier in presenting examples, assuming entity names are unique.

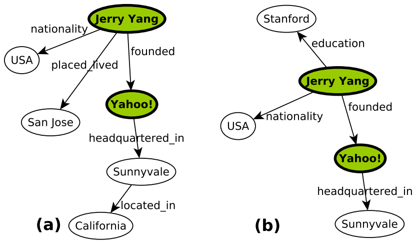


Fig. 4: Two Query Graphs in Fig.3

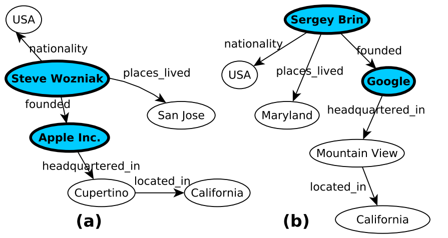


Fig. 5: Two Answer Graphs for Fig.4(a)

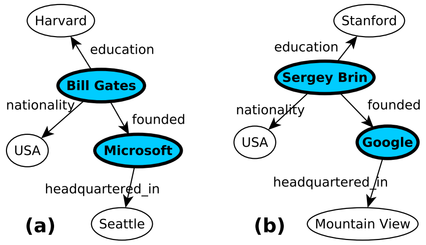


Fig. 6: Two Answer Graphs for Fig.4(b)

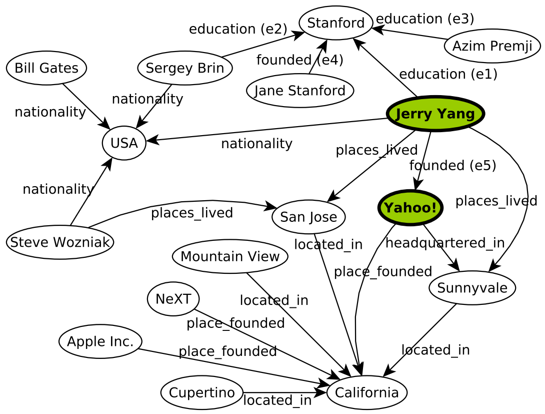


Fig. 3: Neighborhood Graph for ⟨Jerry Yang, Yahoo!⟩

necessarily oriented in the same direction. Unless otherwise stated, we refer to undirected path simply as "path". We consider undirected path because an edge incident on a node can represent an important relationship with another node, regardless of its direction. More formally, a path p is a sequence of edges $e_1, \ldots, e_n$ and we say each edge $e_i \in p$. The path connects two nodes v_0 and v_n through intermediate nodes $v_1, \ldots, v_{n-1}$, where either e_i=(v_{i-1}, v_i) or e_i=(v_i, v_{i-1}), for all $1 \leq i \leq n$. The path's length, $len(p)$, is n and its endpoints, $ends(p)$, are $\{v_0, v_n\}$. There is no undirected cycle in a path, i.e., $v_0, \ldots, v_n$ are all distinct.

Definition 1 The ***neighborhood graph*** of query tuple t, denoted H_t, is the *weakly connected subgraph*[3] of data graph G that consists of all nodes reachable from at least one query entity by an undirected path of d or less number of edges (including query entities themselves) and the edges on all such paths. The *path length threshold*, d, is an input parameter. More formally, the nodes and edges in H_t are defined as follows:

$V(H_t) = \{v|v \in V(G)$ and $\exists p$ s.t. $ends(p)$=$\{v_i, v\}$ where $v_i \in t$, $len(p) \leq d\}$;

$E(H_t) = \{e|e \in E(G)$ and $\exists p$ s.t. $ends(p)$=$\{v_i, v\}$ where $v_i \in t, len(p) \leq d$, and $e \in p\}$.

Example 1 (Neighborhood Graph) Given the data graph in Fig.1, Fig.3 shows the neighborhood graph for query tuple ⟨Jerry Yang, Yahoo!⟩ with path length threshold d=2. The nodes in dark color are the query entities.

Intuitively, the neighborhood graph, by capturing how query entities and other entities in their neighborhood are related to each other, represents features of the query tuple that are to be matched in query answers. It can thus be viewed as a hidden query graph derived for capturing user's query intent. We are unlikely to find query answers that exactly match the neighborhood graph. It is however possible to find exact matches to its subgraphs. Such subgraphs are all query graphs and their exact matches are approximate answers that match the neighborhood graph to different extents.

Definition 2 A ***query graph*** Q is a weakly connected subgraph of H_t that contains all the query entities. We use $\mathcal{Q}_t$ to denote the set of all query graphs for t, i.e., $\mathcal{Q}_t$=$\{Q|Q$ is a weakly connected subgraph of H_t s.t. $\forall v \in t$, $v \in V(Q)\}$.

Continuing the running example, Fig.4 shows two query graphs for the neighborhood graph in Fig.3.

Echoing the intuition behind neighborhood graph, the definitions of answer graph/tuple are based on the idea that an answer tuple is similar to the query tuple if their entities participate in similar relationships in their neighborhoods.

Definition 3 An ***answer graph*** A to a query graph Q is a weakly connected subgraph of G that is edge-isomorphic to Q. Formally, there exists a bijection f:$V(Q) \rightarrow V(A)$ such that:

- For every edge $e = (v_i, v_j) \in E(Q)$, there exists an edge $e' = (f(v_i), f(v_j)) \in E(A)$ such that $label(e) = label(e')$;
- For every edge $e' = (u_i, u_j) \in E(A)$, there exists $e = (f^{-1}(u_i), f^{-1}(u_j)) \in E(Q)$ such that $label(e) = label(e')$.

For a query tuple t=$\langle v_1, \ldots, v_n \rangle$, the ***answer tuple*** in A is t_A=$\langle f(v_1), \ldots, f(v_n) \rangle$. We also call t_A the *projection* of A.

We use $\mathcal{A}_Q$ to denote the set of all answer graphs of Q. We note that a query graph (tuple) trivially matches itself, therefore is not considered an answer graph (tuple).

Example 2 (Answer Graph and Answer Tuple) Fig.5 and Fig.6 each show two answer graphs for query graphs Fig.4(a) and Fig.4(b), respectively. The answer tuples in Fig.5 are ⟨Steve Wozniak, Apple Inc.⟩ and ⟨Sergey Brin, Google⟩. The answer tuples in Fig.6 are ⟨Bill Gates, Microsoft⟩ and ⟨Sergey Brin, Google⟩.

The set of answer tuples for query tuple t are $\{t_A|A \in \mathcal{A}_Q, Q \in \mathcal{Q}_t\}$. The ***score of an answer*** t' is given by:

$$\mathsf{score}_t(t') = \max_{A \in \mathcal{A}_Q, Q \in \mathcal{Q}_t} \{\mathsf{score}_Q(A)|t' = t_A\} \qquad (1)$$

The score of an answer graph A ($\mathsf{score}_Q(A)$) captures A's similarity to query graph Q. Its equation is given in Sec.4.2.

The same answer tuple t' may be projected from multiple answer graphs, which can match different query graphs. For instance, Figs. 5(b) and 6(b), which are answers to different query graphs, have the same projection—⟨Sergey Brin, Google⟩. By Eq. (1), the highest score attained by the answer graphs is assigned as the score of t', capturing how well t' matches t.

[3] A directed graph is *weakly connected* if there exists an undirected path between every pair of vertices.

3 Query Graph Discovery

3.1 Maximum Query Graph

The concept of neighborhood graph H_t (Def.1) was formed to capture the features of a query tuple t to be matched by answer tuples. Given a well-connected large data graph, H_t itself can be quite large, even under a small path length threshold d. For example, using Freebase as the data graph, the query tuple ⟨Jerry Yang, Yahoo!⟩ produces a neighborhood graph with 800K nodes and 900K edges, for d=2. Such a large H_t makes query semantics obscure, because there might be only few nodes and edges in it that capture important relationships in the neighborhood of t.

GQBE's query graph discovery component constructs a weighted *maximum query graph* (MQG) from H_t. The MQG is expected to be drastically smaller than H_t and capture only important features of the query tuple. It is worth noting that a small and plausible MQG can be a Steiner tree connecting all the query entities. But it will fail to capture features that are not on any simple path between a pair of query entities. We thus need a more comprehensive, yet small MQG. We now define MQG and discuss its discovery algorithm.

Definition 4 The ***maximum query graph*** MQG_t, given a parameter m, is a weakly connected subgraph of the neighborhood graph H_t that maximizes total edge weight $\sum_e \mathsf{w}(e)$ while satisfying (1) it contains all query entities in t and (2) it has m edges. The importance of an edge e in H_t, given by its weight $\mathsf{w}(e)$, is defined in Sec.6.

Two challenges exist in finding MQG_t by directly going after the above definition. First, a weakly connected subgraph of H_t with exactly m edges may not exist for an arbitrary m. A trivial value of m that guarantees the existence of the corresponding MQG_t is $|E(H_t)|$, because H_t is weakly connected. This value could be too large, which is exactly why we aim to make MQG_t substantially smaller than H_t. Second, even if MQG_t exists for an m, finding it requires maximizing the total edge weight, which is a hard problem as given in Theorem 1.

Theorem 1 The decision version of finding the maximum query graph MQG_t for an m is NP-hard.

Proof: We prove the NP-hardness by reduction from the NP-hard constrained Steiner network (CSN) problem [15]. Given an undirected connected graph $G_1 = (V, E)$ with non-negative weight $w(e)$ for every edge $e \in E$, a subset $V_n \subset V$, and a positive integer m, the CSN problem finds a connected subgraph $G' = (V', E')$ with the minimum total edge weight, where $V_n \subseteq V'$ and $|E'| = m$. The polynomial-time reduction from the CSN problem to MQG problem is by transforming G_1 to G_2, where each edge e in G_1 is given an arbitrary direction and a new weight $w'(e) = W - w(e)$, where $W = \sum_{e \in E} w(e)$. There are two important observations here: (1) the edge directions do not matter for the MQG problem as we only look for a *weakly* connected subgraph; and therefore, one can add arbitrary edge directions while constructing G_2 from G_1. (2) Given an instance of the CSN problem, $W = \sum_{e \in E} w(e)$ is constant, and also $W \geq w(e)$ for all $e \in E$. Therefore, the new edge weights $w'(e) = W - w(e)$ are non-negative numbers. Now, let V_n be the query tuple for the MQG problem. The maximum query graph MQG_{V_n} found from G_2 provides a CSN in G_1. This is because maximizing $\sum_{e \in MQG_{V_n}} w'(e)$ is equivalent to minimizing $\sum_{e \in MQG_{V_n}} w(e)$, which is the objective function for the CSN problem. This completes the proof. □

Algorithm 1: Discovering the Maximum Query Graph

```
   Input: neighborhood graph $H_t$, query tuple $t$, an integer $r$
   Output: maximum query graph $MQG_t$
1  $m \leftarrow \frac{r}{|t|+1}$; $V(MQG_t) \leftarrow \phi$; $E(MQG_t) \leftarrow \phi$; $\mathcal{G} \leftarrow \phi$;
2  foreach $v_i \in t$ do
3      $G_{v_i} \leftarrow$ use DFS to obtain the subgraph containing vertices (and
       their incident edges) that connect to other $v_j$ in $t$ only through $v_i$;
4      $\mathcal{G} \leftarrow \mathcal{G} \cup \{G_{v_i}\}$;
5  $G_{core} \leftarrow$ use DFS to obtain the subgraph containing vertices and
   edges on undirected paths between query entities;
6  $\mathcal{G} \leftarrow \mathcal{G} \cup \{G_{core}\}$;
7  foreach $G \in \mathcal{G}$ do
8      $step \leftarrow 1$; $s_1 \leftarrow 0$; $s \leftarrow m$;
9      while $s > 0$ do
10         $M_s \leftarrow$ the weakly connected component found from the
           top-$s$ edges of $G$ that contains all of $G$'s query entities;
11         if $M_s$ exists then
12             if $|E(M_s)| = m$ then break;
13             if $|E(M_s)| < m$ then
14                 $s_1 \leftarrow s$;
15                 if $step = -1$ then break;
16             if $|E(M_s)| > m$ then
17                 if $s_1 > 0$ then
18                     $s \leftarrow s_1$; break;
19                 $s_2 \leftarrow s$; $step \leftarrow -1$;
20         $s \leftarrow s + step$;
21     if $s = 0$ then $s \leftarrow s_2$;
22     $V(MQG_t) \leftarrow V(MQG_t) \cup V(M_s)$;
23     $E(MQG_t) \leftarrow E(MQG_t) \cup E(M_s)$;
```

Based on the theoretical analysis, we present a greedy method (Alg.1) to find a plausible sub-optimal graph of edge cardinality *close* to a given m. The value of m is empirically chosen to be much smaller than $|E(H_t)|$. Consider edges of H_t in descending order of weight $\mathsf{w}(e)$. We use G_s to denote the graph formed by the top s edges with the largest weights, which itself may not be weakly connected. We use M_s to denote the weakly connected component (a maximum subgraph where an undirected path exists for every pair of vertices) of G_s containing all query entities in t, if it exists. Our method finds the smallest s such that $|E(M_s)|$=m (Line 12). If such an M_s does not exist, the method chooses s_1, the largest s such that $|E(M_s)|$<m. If that still does not exist, it chooses s_2, the smallest s such that $|E(M_s)|$>m, whose existence is guaranteed because $|E(H_t)|$>m. For each s value, the method employs a depth-first search (DFS) starting from a query entity in G_s, if present, to check the existence of M_s (Line 10).

The M_s found by this method may be unbalanced. Query entities with more neighbors in H_t likely have more prominent representation in the resulting M_s. A balanced graph should instead have a fair number of edges associated with each query entity. Therefore, we further propose a divide-and-conquer mechanism to construct a balanced MQG_t. The idea is to break H_t into n+1 weakly connected subgraphs. One is the *core graph*, which includes all the n query entities in

t and all undirected paths between query entities. Other n subgraphs are for the n query entities individually, where the subgraph for entity v_i includes all entities (and their incident edges) that connect to other query entities only through v_i. The subgraphs are identified by a DFS starting from each query entity (Lines 4-6 of Alg.1). During the DFS from v_i, all edges on the undirected paths reaching any other query entity within distance d belong to the core graph, and other edges belong to v_i's individual subgraph. The method then applies the aforementioned greedy algorithm to find $n{+}1$ weakly connected components, one for each subgraph, that contain the query entities in corresponding subgraphs. Since the core graph connects all query entities, the $n{+}1$ components altogether form a weakly connected subgraph of H_t, which becomes the final MQG_t. For an empirically chosen small r as the target size of MQG_t, we set the target size for each individual component to be $\frac{r}{n+1}$, aiming at a balanced MQG_t.

The greedy approach described in Alg. 1 makes a best effort at pruning unimportant features and finding an MQG that captures the user intent, by ensuring that only highly weighted edges are present in the MQG. Ability to capture the user intent well depends on how good the edge weighting function $\mathsf{w}(e)$ is in assigning high weights to edges that are intended by users.

Complexity Analysis of Alg.1 The complexity analysis of this and other algorithms can be found in the Appendix that appears in the online supplemental material.

3.2 Multi-tuple Queries

The query graph discovery component derives a user's query intent from input query tuples. For that, a single query tuple might not be sufficient. While the experiment results in Sec.7 show that a single-tuple query obtains excellent accuracy in many cases, the results also exhibit that allowing multiple query tuples often help in improving query answer accuracy. It is because important relationships commonly associated with multiple tuples express the user intent more precisely. Suppose a user provides two query tuples—⟨Jerry Yang, Yahoo!⟩ and ⟨Steve Wozniak, Apple Inc.⟩. The entities in both tuples share common properties such as *places_lived* in San Jose and *headquartered_in* a city in California, as Fig.1 shows. This might indicate the user is interested in finding people from San Jose who founded technology companies in California.

Given a set of tuples T, GQBE finds top-k answer tuples similar to T collectively. To accomplish this, one approach is to discover and evaluate the maximum query graphs (MQGs) of individual query tuples. The scores of a common answer tuple for multiple query tuples can then be aggregated. This has two potential drawbacks: (1) Our concern of not being able to well capture user intent still remains. If k is not large enough, a good answer tuple may not appear in enough individual top-k answer lists, resulting in poor aggregated score. (2) It can become expensive to evaluate multiple MQGs.

We approach this problem by producing a merged and re-weighted MQG that captures the importance of edges with respect to their presence across multiple MQGs. The merged MQG is then processed by the same method for single-tuple queries. GQBE employs a simple strategy to merge multiple MQGs. The individual MQG for a query tuple $t_i{=}\langle v_1^i, v_2^i, \ldots, v_n^i\rangle{\in}T$ is denoted M_{t_i}. A virtual MQG M'_{t_i} is created for every M_{t_i} by replacing the query entities $v_1^i, v_2^i, \ldots, v_n^i$ in M_{t_i} with corresponding virtual entities $w_1, w_2, \ldots, w_n$ in M'_{t_i}. Formally, there exists a bijective function $g{:}V(M_{t_i}){\rightarrow}V(M'_{t_i})$ such that (1) $g(v_j^i){=}w_j$ and $g(v){=}v$ if $v{\notin}t_i$, and (2) $\forall e{=}(u,v){\in}E(M_{t_i})$, there exists an edge $e'{=}(g(u), g(v))\ {\in}E(M'_{t_i})$ such that $label(e){=}label(e')$; $\forall e'{=}(u',v'){\in}E(M'_{t_i})$, $\exists e\ {=}(g^{-1}(u'), g^{-1}(v')){\in}E(M_{t_i})$ such that $label(e){=}label(e')$.

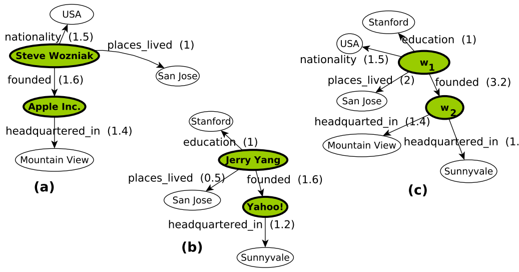


Fig. 7: Merging Maximum Query Graphs

The merged MQG, denoted MQG_T, is produced by including vertices and edges in all M'_{t_i}, merging identical virtual and regular vertices, and merging identical edges that bear the same label and the same vertices on both ends, i.e.,

$$V(MQG_T) = \bigcup_{t_i \in T} V(M'_{t_i}) \text{ and } E(MQG_T) = \bigcup_{t_i \in T} E(M'_{t_i}).$$

The edge cardinality of MQG_T might be larger than the target size r. Thus Alg.1 (Sec.3.1) is also used to trim MQG_T to a size close to r. In MQG_T, the weight of an edge e is given by $c * \mathsf{w}_{max}(e)$, where c is the number of M'_{t_i} containing e and $\mathsf{w}_{max}(e)$ is its maximum weight among all such M'_{t_i}. (For complexity analysis, refer to the Appendix that appears in the online supplemental material.)

Example 3 (Merging Maximum Query Graphs) Let Figs. 7 (a) and (b) be the M_{t_i} for query tuples ⟨Steve Wozniak, Apple Inc.⟩ and ⟨Jerry Yang, Yahoo!⟩, respectively. Fig.7(c) is the merged MQG_T. Note that entities Steve Wozniak and Jerry Yang are mapped to w_1 in their respective M'_{t_i} (not shown, for its mapping from M_{t_i} is simple) and are merged into w_1 in MQG_T. Similarly, entities Apple Inc. and Yahoo! are mapped and merged into w_2. The two *founded* edges, appearing in both individual M_{t_i} and sharing identical vertices on both ends (w_1 and w_2) in the corresponding M'_{t_i}, are merged in MQG_T. Similarly the two *places_lived* edges are merged. However, the two *headquartered_in* edges are not merged, since they share only one end (w_2) in M'_{t_i}. The edges *nationality* and *education*, which appear in only one M_{t_i}, are also present in MQG_T. The number next to each edge is its weight.

4 Answer Space Modeling

Since it is unlikely to find exactly matching answer graphs to the discovered MQG, approximate matches have to be found. Given the maximum query graph MQG_t for t, we thus model the space of possible query graphs by a lattice. We further discuss the scoring of answer graphs by how they match query graphs.

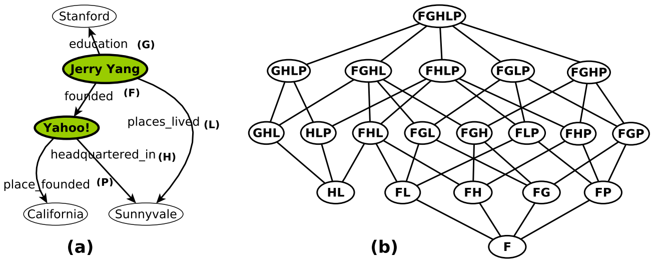


Fig. 8: Maximum Query Graph and Query Lattice

4.1 Query Lattice

The ***query lattice*** $\mathcal{L}$ is a partially ordered set (poset) ($\mathcal{QG}_t$, $\prec$), where $\prec$ represents the subgraph-supergraph subsumption relation and $\mathcal{QG}_t$ is the subset of query graphs (Def.2) that are subgraphs of MQG_t, i.e., $\mathcal{QG}_t$=$\{Q|Q \in \mathcal{Q}_t$ and $Q \preceq MQG_t\}$. The top element (root) of the poset is thus MQG_t. When represented by a Hasse diagram, the poset is a directed acyclic graph, in which each node corresponds to a distinct query graph in $\mathcal{QG}_t$. Thus we shall use the terms *lattice node* and *query graph* interchangeably. The *children* (*parents*) of a lattice node Q are its subgraphs (supergraphs) with one less (more) edge, as defined below.

$$\mathsf{Children}(Q) = \{Q'|Q' \in \mathcal{QG}_t, Q' \prec Q, |E(Q)| - |E(Q')| = 1\}$$
$$\mathsf{Parents}(Q) = \{Q'|Q' \in \mathcal{QG}_t, Q \prec Q', |E(Q')| - |E(Q)| = 1\}$$

The leaf nodes of $\mathcal{L}$ constitute of the *minimal query trees*, which are those query graphs that cannot be made any simpler and yet still keep all the query entities connected. A query graph Q is a minimal query tree if none of its subgraphs is also a query graph. In other words, removing any edge from Q will disqualify it from being a query graph—the resulting graph either is not weakly connected or does not contain all the query entities. Note that such a Q must be a tree.

Example 4 (Query Lattice and Minimal Query Tree) Fig.8(a) shows a maximum query graph MQG_t, which contains two query entities in shaded circles and five edges F, G, H, L, and P. Its corresponding query lattice $\mathcal{L}$ is in Fig.8(b). The root node of $\mathcal{L}$, denoted $FGHLP$, represents MQG_t itself. The bottom-most nodes, F and HL, are the two minimal query trees. Each lattice node is a subgraph of MQG_t. For example, the node FG represents a query graph with only edges F and G. Note that there is no lattice node for GLP since it is not a valid connected query graph.

The construction of the query lattice, i.e., the generation of query graphs corresponding to its nodes, is integrated with its exploration. In other words, the lattice is built in a "lazy" manner—a lattice node is not generated until the query algorithm (Sec.5) must evaluate it. The lattice nodes are generated in a bottom-up way. A node is generated by adding exactly one appropriate edge to the query graph for one of its children. The generation of bottom nodes, i.e., the minimal query trees, is described below.

By definition, a minimal query tree can only contain edges on undirected paths between query entities. Hence, it must be a subgraph of the weakly connected component M_s found from the core graph described in Sec.3.1. To generate all minimal query trees, our method enumerates all distinct spanning trees of M_s by the technique in [9] and then prune them. Specifically, given one such spanning tree, all non-query entities (nodes) of degree one along with their edges are deleted. The deletion is performed iteratively until there is no such node. The result is a minimal query tree. Only distinct minimal query trees are kept. Enumerating all spanning trees in a large graph is expensive. However, in our experiments on the Freebase dataset, the MQG_t discovered by the approach in Sec.3 mostly contains less than 15 edges. Hence, the M_s from the core graph is also empirically small, for which the cost of enumerating all spanning trees is negligible.

4.2 Answer Graph Scoring Function

The score of an answer graph A ($\mathsf{score}_Q(A)$) captures A's similarity to the query graph Q. It is defined below and is to be plugged into Eq. (1) for defining answer tuple score.

$$\begin{aligned} \mathsf{score}_Q(A) &= \mathsf{s\_score}(Q) + \mathsf{c\_score}_Q(A) \\ \mathsf{s\_score}(Q) &= \sum_{e \in E(Q)} \mathsf{w}(e) \\ \mathsf{c\_score}_Q(A) &= \sum_{\substack{e=(u,v) \in E(Q) \\ e'=(f(u),f(v)) \in E(A)}} \mathsf{match}(e, e') \end{aligned} \tag{2}$$

In Eq. (2), $\mathsf{score}_Q(A)$ sums up two components—the *structure score* of Q ($\mathsf{s\_score}(Q)$) and the *content score* for A matching Q ($\mathsf{c\_score}_Q(A)$). $\mathsf{s\_score}(Q)$ is the total edge weight of Q. It measures the important structure in MQG_t that is captured by Q and thus by A. $\mathsf{c\_score}_Q(A)$ is the total extra credit for identical nodes among the matching nodes in A and Q given by f—the bijection between $V(Q)$ and $V(A)$ as in Def.3. For instance, among the 6 pairs of matching nodes between Fig.4(a) and Fig.5(a), the identical matching nodes are USA, San Jose and California. The rationale for the extra credit is that although node matching is not mandatory, the more nodes are matched, the more similar A and Q are.

The extra credit is defined by the following function $\mathsf{match}(e, e')$. Note that it does not award an identical matching node excessively. Instead, only a fraction of $\mathsf{w}(e)$ is awarded, where the denominator is either $|E(u)|$ or $|E(v)|$. ($E(u)$ are the edges incident on u in MQG_t.) This heuristic is based on that, when u and $f(u)$ are identical, many of their neighbors can be also identical matching nodes.

$$\mathsf{match}(e, e') = \begin{cases} \frac{\mathsf{w}(e)}{|E(u)|} & \text{if } u = f(u) \\ \frac{\mathsf{w}(e)}{|E(v)|} & \text{if } v = f(v) \\ \frac{\mathsf{w}(e)}{min(|E(u)|, |E(v)|)} & \text{if } u = f(u), v = f(v) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

5 Query Processing

GQBE's query processing component takes MQG_t (Sec.3) and the query lattice $\mathcal{L}$ (Sec.4) and finds answer graphs matching the query graphs in $\mathcal{L}$. Before we discuss how $\mathcal{L}$ is evaluated (Sec.5.2), we introduce the storage model and query plan for processing one query graph (Sec.5.1).

Algorithm 2: Best-first Exploration of Query Lattice

```
Input: query lattice $\mathcal{L}$, query tuple $t$, and an integer $k$
Output: top-$k$ answer tuples
1  lower frontier $\mathcal{LF}$ ← leaf nodes of $\mathcal{L}$; Terminate ← false;
2  while not Terminate do
3      $Q_{best}$ ← node with the highest upper-bound score in $\mathcal{LF}$;
4      $\mathcal{A}_{Q_{best}}$ ← evaluate $Q_{best}$; (Sec.5.1)
5      if $\mathcal{A}_{Q_{best}}$=$\emptyset$ then
6          prune $Q_{best}$ and all its ancestors from $\mathcal{L}$;
7          recompute upper-bound scores of nodes in $\mathcal{LF}$; (Alg. 3)
8      else
9          insert Parents($Q_{best}$) into $\mathcal{LF}$;
10     if top-$k$ answer tuples found [Theorem 3] then Terminate←true;
```

5.1 Processing One Query Graph

The abstract data model of knowledge graph can be represented by the Resource Description Framework (RDF)—the standard Semantic Web data model. In RDF, a data graph is parsed into a set of triples, each representing an edge e=(u, v). A triple has the form (subject, property, object), corresponding to $(u, label(e), v)$. Among different schemes of RDF data management, one important approach is to use relational database techniques to store and query RDF graphs. To store a data graph, we adopt this approach and, particularly, the vertical partitioning method [1]. This method partitions a data graph into multiple two-column tables. Each table is for a distinct edge label and stores all edges bearing that label. The two columns are $(subj, obj)$, for the edges' source and destination nodes, respectively. For efficient query processing, two in-memory search structures (specifically, hash tables) are created on the table, using $subj$ and obj as the hash keys, respectively. The whole data graph is hashed in memory by this way, before any query comes in.

Given the above storage scheme, to evaluate a query graph is to process a multi-way join query. For instance, the query graph in Fig.8(a) corresponds to SELECT F.subj, F.obj FROM F,G,H,L,P WHERE F.subj=G.sbj AND F.obj=H.subj AND F.subj=L.subj AND F.obj=P.subj AND H.obj=L.obj. We use right-deep hash-joins to process such a query. Consider the topmost join operator in a join tree for query graph Q. Its left operand is the *build relation* which is one of the two in-memory hash tables for an edge e. Its right operand is the *probe relation* which is a hash table for another edge or a join subtree for Q'=$Q-e$ (i.e., the resulting graph of removing e from Q). For instance, one possible join tree for the aforementioned query is $G \bowtie (F \bowtie (P \bowtie (H \bowtie L)))$. With regard to its topmost join operator, the left operand is G's hash table that uses $G.sbj$ as the hash key, and the right operand is $(F \bowtie (P \bowtie (H \bowtie L)))$. The hash-join operator iterates through tuples from the probe relation, finds matching tuples from the build relation, and joins them to form answer tuples.

5.2 Best-first Exploration of Query Lattice

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 top-*k* answers. Moreover, the potentially many queries are evaluated separately, without sharing of computation. Suppose query graph Q is evaluated by the aforementioned hash-join between the build relation for e and the probe relation for Q'. By definition, Q' is also a query graph in the lattice, if Q' is weakly connected and contains all query entities. In other words, in processing Q, we would have processed one of its children query graph Q' in the lattice.

We propose Alg.2, which allows sharing of computation. It explores the query lattice in a *bottom-up* way, starting with the minimal query trees, i.e., the bottom nodes. After a query graph is processed, its answers are materialized in files. To process a query Q, at least one of its children Q'=$Q-e$ must have been processed. The materialized results for Q' form the probe relation and a hash table on e is the build relation.

While any topological order would work for the bottom-up exploration, Alg.2 employs a *best-first* strategy that always chooses to evaluate the most promising lattice node Q_{best} from a set of candidate nodes. The gist is to process the lattice nodes in the order of their upper-bound scores and Q_{best} is the candidate with the highest upper-bound score (Line 3). If processing Q_{best} does not yield any answer graph, Q_{best} and all its ancestors are pruned (Line 6) and the upper-bound scores of other candidate nodes are recalculated (Line 7). The algorithm terminates, without fully evaluating all lattice nodes, when it has obtained at least k answer tuples with scores higher than the highest possible upper-bound score among all unevaluated nodes (Line 10).

For an arbitrary query graph Q, its upper-bound score is given by the best possible score Q's answer graphs can attain. Deriving such upper-bound score based on $\mathsf{score}_Q(A)$ in Eq. (2) leads to loose upper-bound. $\mathsf{score}_Q(A)$ sums up the structure score of Q ($\mathsf{s\_score}(Q)$) and the content score for A matching Q ($\mathsf{c\_score}_Q(A)$). While $\mathsf{s\_score}(Q)$ only depends on Q itself, $\mathsf{c\_score}_Q(A)$ captures the matching nodes in A and Q. Without evaluating Q to get A, we can only assume perfect $\mathsf{match}(e, e')$ in Eq. (2), which is clearly an over-optimism. Under such a loose upper-bound, it can be difficult to achieve an early termination of lattice evaluation.

To alleviate this problem, GQBE takes a two-stage approach. Its query algorithm first finds the top-k' answers (k'>k) based on the structure score $\mathsf{s\_score}(Q)$ only, i.e., the algorithm uses a simplified answer graph scoring function $\mathsf{score}_Q(A) = \mathsf{s\_score}(Q)$. In the second stage, GQBE re-ranks the top-k' answers by the full scoring function Eq. (2) and returns the top-*k* answer tuples based on the new scores. Our experiments showed the best accuracy for k ranging from 10 to 25 when k' was set to around 100. Lesser values of k' lowered the accuracy and higher values increased the running time of the algorithm. In the ensuing discussion, we will not further distinct k' and k.

5.3 Details of the Best-first Exploration Algorithm

(1) Selecting $\mathbf{Q_{best}}$

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 (Line 1 of Alg.2). After a node is evaluated, all its parents are added to $\mathcal{LF}$ (Line 9). Therefore, the nodes in $\mathcal{LF}$ either are minimal query trees or have at least one evaluated child:

$$\mathcal{LF} = \{Q| \ Q \text{ is not pruned}, \mathsf{Children}(Q){=}\emptyset \text{ or } \\ (\exists Q' \in \mathsf{Children}(Q) \text{ s.t. } Q' \text{ is evaluated})\}.$$

To choose Q_{best} from $\mathcal{LF}$, the algorithm exploits two important properties, dictated by the query lattice's structure.

Property 1 If $Q_1 \prec Q_2$, then $\forall A_2 \in \mathcal{A}_{Q_2}$, $\exists A_1 \in \mathcal{A}_{Q_1}$ s.t. $A_1 \prec A_2$ and $t_{A_1}{=}t_{A_2}$.

Proof: If there exists an answer graph A_2 for a query graph Q_2, and there exists another query graph Q_1 that is a subgraph of Q_2, then there is a subgraph of A_2 that corresponds to Q_1. By Definition 3, that corresponding subgraph of A_2 is an answer graph to Q_1. Since the two answer graphs share a subsumption relationship, the projections of the two yield the same answer tuple. □

Property 1 says, if an answer tuple t_{A_2} is projected from answer graph A_2 to lattice node Q_2, then every descendent of Q_2 must have at least one answer graph subsumed by A_2 that projects to the same answer tuple. Putting it in an informal way, an answer tuple (graph) to a lattice node can always be "grown" from its descendant nodes and thus ultimately from the minimal query trees.

Property 2 If $Q_1{\prec}Q_2$, then $\mathsf{s\_score}(Q_1){<}\mathsf{s\_score}(Q_2)$.

Proof: If $Q_1 \prec Q_2$, then Q_2 contains all edges in Q_1 and at least one more. Thus the property holds by the definition of $\mathsf{s\_score}(Q)$ in Eq. (2). □

Property 2 says that, if a lattice node Q_2 is an ancestor of Q_1, Q_2 has a higher structure score. This can be directly proved by referring to the definition of $\mathsf{s\_score}(Q)$ in Eq. (2).

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. The chosen node, Q_{best}, must have the highest upper-bound score among all the nodes in $\mathcal{LF}$. By the two properties, if evaluating Q returns an answer graph A, A has the potential to grow into an answer graph A' to an ancestor node Q', i.e., $Q{\prec}Q'$ and $A{\prec}A'$. 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 answer scoring function $\mathsf{score}_Q(A) = \mathsf{s\_score}(Q)$, which Alg.2 adopts as mentioned in Sec. 5.2. 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*** ($\mathcal{UF}$) that subsume or equal to Q:

$$\mathcal{UB}(Q) = \{Q'| \ Q' \succeq Q, Q' \in \mathcal{UF}\},$$

where $\mathcal{UF}$ are the unpruned nodes without unpruned parents: $\mathcal{UF}{=}\{Q| \ Q \text{ is not pruned}, \nexists Q' \succ Q \text{ s.t. } Q' \text{ is not pruned}\}$.

The ***upper-bound score*** of a node Q is the maximum score of any query graph in its upper boundary:

$$U(Q) = \max_{Q' \in \mathcal{UB}(Q)} \mathsf{s\_score}(Q') \tag{4}$$

Example 5 (Lattice Evaluation) Consider the lattice in Fig.9(a) where the lightly shaded nodes belong to the $\mathcal{LF}$ and the darkly shaded node belongs to $\mathcal{UF}$. At the beginning, only the minimal query trees belong to the $\mathcal{LF}$ and the maximum query graph belongs to the $\mathcal{UF}$. If *HL* is chosen as Q_{best} and evaluating it results in matching answer graphs, all its parents (*GHL*, *HLP* and *FHL*) are added to $\mathcal{LF}$ as shown in Fig.9(b). The evaluated node *HL* is represented in bold dashed node.

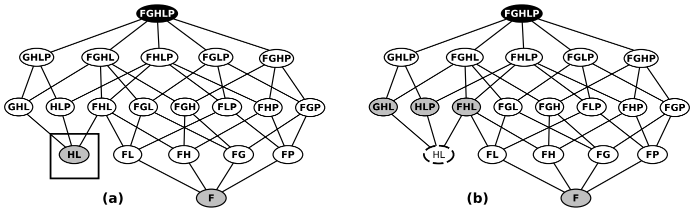


Fig. 9: Evaluating Lattice in Figure 8(b)

Algorithm 3: Recomputing Upper-bound Scores

```
Input: query lattice $\mathcal{L}$, null node $Q_{best}$, and lower-frontier $\mathcal{LF}$
Output: $U(Q)$ for all $Q$ in $\mathcal{LF}$
foreach $Q \in \mathcal{LF}$ do
    $\mathcal{NB} \leftarrow \phi$; // set of new upper boundary candidates of $Q$.
    foreach $Q' \in \mathcal{UB}(Q) \cap \mathcal{UB}(Q_{best})$ do
        $\mathcal{UB}(Q) \leftarrow \mathcal{UB}(Q) \setminus \{Q'\}$;
        $\mathcal{UF} \leftarrow \mathcal{UF} \setminus \{Q'\}$;
        $V(Q'') \leftarrow V(Q')$;
        foreach $e \in E(Q_{best}) \setminus E(Q)$ do
            $E(Q'') \leftarrow E(Q') \setminus \{e\}$;
            find $Q_{sub}$, the weakly-connected component of $Q''$,
            containing all query entities;
            $\mathcal{NB} \leftarrow \mathcal{NB} \cup \{Q_{sub}\}$;
    foreach $Q_{sub} \in \mathcal{NB}$ do
        if $Q_{sub} \nprec$ (any node in $\mathcal{UF}$ or $\mathcal{NB}$) then
            $\mathcal{UB}(Q) \leftarrow \mathcal{UB}(Q) \cup \{Q_{sub}\}$, $\mathcal{UF} \leftarrow \mathcal{UF} \cup \{Q_{sub}\}$;
    recompute $U(Q)$ using Eq. (4);
```

(2) Pruning and Lattice Recomputation

A lattice node that does not have any answer graph is referred to as a *null node*. If the most promising node Q_{best} turns out to be a null node after evaluation, all its ancestors are also null nodes based on Property 3 below which follows directly from Property 1.

Property 3 If $\mathcal{A}_{Q_1} = \emptyset$, then $\forall Q_2 \succ Q_1$, $\mathcal{A}_{Q_2} = \emptyset$.

Proof: Suppose there is a query node Q_2 such that $Q_1 \prec Q_2$ and $\mathcal{A}_{Q_1} = \emptyset$, while $\mathcal{A}_{Q_2} \neq \emptyset$. By Property 1, for every answer graph A in $\mathcal{A}_{Q_2}$, there must exist a subgraph of A that belongs to $\mathcal{A}_{Q_1}$. This contradiction completes the proof. □

Based on Property 3, when Q_{best} is evaluated to be a null node, Alg.2 prunes Q_{best} and its ancestors, which changes the upper-frontier $\mathcal{UF}$. It is worth noting that Q_{best} itself may be an upper-frontier node, in which case only Q_{best} is pruned. In general, due to the evaluation and pruning of nodes, $\mathcal{LF}$ and $\mathcal{UF}$ might overlap. 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. We refer to such nodes as *dirty nodes*. The rest of this section presents an efficient method (Alg. 3) to recompute the upper boundaries, and if changed, the upper-bound scores of the dirty nodes.

Consider all the pairs $\langle Q, Q' \rangle$ such that Q is a dirty node in $\mathcal{LF}$, and Q' is one of its pruned upper boundary nodes. Three necessary conditions for a new candidate upper boundary node of Q are that it is (1) a supergraph of Q, (2) a subgraph of Q' and (3) not a supergraph of Q_{best}. If there are q edges in

Q_{best} but not in Q, we create a set of q distinct graphs Q''. Each Q'' contains all edges in Q' except exactly one of the aforementioned q edges (Line 8 in Alg. 3). For each Q'', we find Q_{sub} which is the weakly connected component of Q'' containing all the query entities (Lines 9-10). Lemma 1 and 2 show that Q_{sub} must be one of the unevaluated nodes after pruning the ancestor nodes of Q_{best} from $\mathcal{L}$.

Lemma 1 Q_{sub} is a *query graph* and it does not belong to the pruned nodes of lattice $\mathcal{L}$.

Proof: Q_{sub} is a query graph because it is weakly connected and it contains all the query entities. Suppose Q_{sub} is a newly generated candidate upper boundary node from pair $\langle Q, Q' \rangle$ and Q_{sub} belongs to the pruned nodes of lattice $\mathcal{L}$. This can happen only if: 1) it is a supergraph of the current null node Q_{best} or 2) it is an already pruned node. The former cannot happen since the construction mechanism of Q_{sub} proposed ensures that it is not a supergraph of Q_{best}. the latter implies that Q_{sub} was the supergraph of an previously evaluated null node (or Q_{sub} itself was a null node). In this case, since $Q_{sub} \prec Q'$, Q' would also have been pruned and thus could not have been part of the upper-boundary. Hence $\langle Q, Q' \rangle$ cannot be a valid pair for recomputing the upper boundary if Q_{sub} is pruned. This completes the proof. □

Lemma 2 $Q \preceq Q_{sub}$.

Proof: Based on Alg. 3, Q'' is the result of deleting one edge from Q' and that edge does not belong to Q. Therefore, Q is subsumed by Q''. By the same algorithm, Q_{sub} is the weakly connected component of Q'' that contains all the query entities. Since Q already is weakly connected and contains all the query entities, Q_{sub} must be a supergraph of Q. □

If Q_{sub} (a candidate new upper boundary node of Q) is not subsumed by any node in the upper-froniter or other candidate nodes, we add Q_{sub} to $\mathcal{UB}(Q)$ and $\mathcal{UF}$ (Lines 11-13). Finally, we recompute Q's upper-bound score (Line 14). Theorem 2 justifies the correctness of the above procedure.

Theorem 2 If $\mathcal{A}_{Q_{best}} = \emptyset$, then Alg.3 identifies all new upper boundary nodes for every dirty node Q.

Proof: For any dirty node Q, its original upper boundary $\mathcal{UB}(Q)$ consists of two sets of nodes: (1) nodes that are not supergraphs of Q_{best} and thus remain in the lattice, (2) nodes that are supergraphs of Q_{best} and thus pruned. By definition of upper boundary node, no upper boundary node of Q can be a subgraph of any node in set (1). So any new upper boundary node of Q must be a subgraph of a node Q' in set (2). For every pruned upper boundary node Q' in set (2), the algorithm enumerates all (specifically q) possible children of Q' that are not supergraphs of Q_{best} but are supergraphs of Q. For each enumerated graph Q'', the algorithm finds Q_{sub}—the weakly connected component of Q'' containing all query entities. Thus all new upper boundary nodes of Q are identified. □

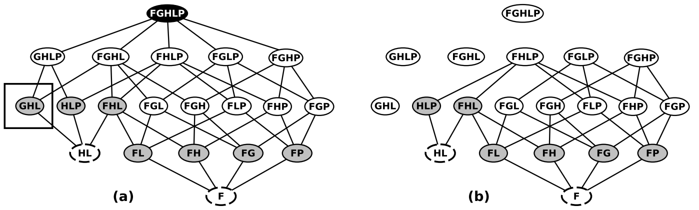


Fig. 10: Recomputing Upper Boundary of Dirty Node FG

Example 6 (Recomputing Upper Boundary) Consider the lattice in Fig.10(a) where nodes *HL* and *F* are the evaluated nodes and the lightly shaded nodes belong to the new $\mathcal{LF}$. If node *GHL* is the currently evaluated null node Q_{best} and *FGHLP* is Q', let *FG* be the dirty node Q whose upper boundary is to be recomputed. The edges in Q_{best} that are not present in Q are *H* and *L*. A new upper boundary node Q'' contains all edges in Q' excepting exactly either *H* or *L*. This leads to two new upper boundary nodes, *FGHP* and *FGLP*, by removing *L* and *H* from *FGHLP*, respectively. Since *FGHP* and *FGLP* do not subsume each other and are not subgraphs of any other upper-frontier node, they are now part of $\mathcal{UB}(Q)$ and the new $\mathcal{UF}$. Fig.10(b) shows the modified lattice where the pruned nodes are disconnected. *FHLP* is another node in $\mathcal{UF}$ that is discovered using dirty nodes such as *FL* and *HLP*.

(3) Termination

After Q_{best} is evaluated, its answer tuples are $\{t_A | A \in \mathcal{A}_{Q_{best}}\}$. For a t_A projected from answer graph A, the score assigned by Q_{best} to A (and thus t_A) is $\mathsf{s\_score}(Q_{best})$, based on $\mathsf{score}_Q(A) = \mathsf{s\_score}(Q)$—the simplified scoring function adopted by Alg.2. If t_A was also projected from already evaluated nodes, it has a current score. By Eq. 1, the final score of t_A will be from its best answer graph. Hence, if $\mathsf{s\_score}(Q_{best})$ is higher than its current score, then its score is updated. In this way, all found answer tuples so far are kept and their current scores are maintained to be the highest scores they have received. 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, by Theorem 3.

Theorem 3 Suppose t_k is the current k^{th} best answer tuple and $\mathsf{score}_t(t_k) > U(Q_{best})$. If lattice evaluation is terminated, then $\mathsf{score}_t(t_k) > \mathsf{s\_score}(Q)$ for any unevaluated query graph Q.

Proof: Suppose, upon termination, there is an unevaluated query graph Q such that $\mathsf{score}_t(t_k) \leq \mathsf{s\_score}(Q)$. This implies that there exists some node in the lower-frontier $\mathcal{LF}$, whose upper-bound score is at least $\mathsf{s\_score}(Q)$ and is thus greater than $\mathsf{score}_t(t_k)$. This is a contradiction to the termination condition $\mathsf{score}_t(t_k) > U(Q_{best})$. □

6 Edge Weighting Function

The definition of MQG_t (Def.4) depends on edge weights. There can be various plausible weighting schemes. Any edge weighting function that reflects the importance of edges can be used and our system is capable of adopting any such function. We next present the weighting function used in our implementation, which is based on several heuristic ideas.

The weight of an edge e in the neighborhood graph H_t, $\mathsf{w}(e)$, is proportional to its inverse edge label frequency ($\mathsf{ief}(e)$) and inversely proportional to its participation degree ($\mathsf{p}(e)$), given by

$$\mathsf{w}(e) = \mathsf{ief}(e) \ / \ \mathsf{p}(e) \quad (5)$$

Inverse Edge Label Frequency Edge labels that appear frequently in the entire data graph G are often less important. For example, edges labeled *founded* (for a company's founders) can be rare and more important than edges labeled *nationality*. We capture this by the *inverse edge label frequency*.

$$\mathsf{ief}(e) = \log\left(|E(G)| \;/\; \#label(e)\right) \tag{6}$$

where $|E(G)|$ is the number of edges in G, and $\#label(e)$ is the number of edges in G with the same label as e.

Participation Degree The *participation degree* $p(e)$ of an edge $e{=}(u, v)$ is the number of edges in G that share the same label and one of e's end nodes. Formally,

$$\mathsf{p}(e) = |\; \{e'{=}(u', v') \mid label(e){=}label(e'), u'{=}u \vee v'{=}v\} \;| \tag{7}$$

Participation degree $\mathsf{p}(e)$ measures the local frequencies of edge labels—an edge is less important if there are other edges incident on the same node with the same label. For instance, *employment* might be a relatively rare edge globally but not necessarily locally to a company. Specifically, consider the edges representing the *employment* relationship between a company and its *many* employees and the edges for the *board member* relationship between the company and its *few* board members. The latter edges are more significant.

Note that $\mathsf{ief}(e)$ and $\mathsf{p}(e)$ are precomputed offline, since they are query-independent and only rely on the data graph G.

In discovering MQG_t from H_t by Alg.1, the weights of edges in H_t are defined by Eq. (5) which does not consider an edge's distance from the query tuple. The rationale behind the design is to obtain a balanced MQG_t which includes not only edges incident on query entities but also those in the larger neighborhood. For scoring answers by Eq. (2) and Eq. (3), however, our empirical observations show it is imperative to differentiate the importance of edges in MQG_t with respect to query entities, in order to capture how well an answer graph matches MQG_t. Edges closer to query entities convey more meaningful relationships than those farther away. Hence, we define edge depth ($\mathsf{d}(e)$) as follows. The larger $\mathsf{d}(e)$ is, the less important e is.

Edge Depth The depth $\mathsf{d}(e)$ of an edge $e{=}(u, v)$ is its smallest distance to any query entity $v_i \in t$, i.e.,

$$\mathsf{d}(e) = \min_{v_i \in t} \min_{u,v} \{\mathsf{dist}(u, v_i), \mathsf{dist}(v, v_i)\} \tag{8}$$

Here, $\mathsf{dist}(.,.)$ is the shortest length of all undirected paths in MQG_t between the two nodes.

In summary, GQBE uses Eq. (5) as the definition of $\mathsf{w}(e)$ in weighting edges in H_t. After MQG_t is discovered from H_t by Alg.1, it uses the following Eq. (9) as the definition of $\mathsf{w}(e)$ in weighting edges in MQG_t. Eq. (9) incorporates $\mathsf{d}(e)$ into Eq. (5). The answer graph scoring functions Eq. (2) and Eq. (3) are based on Eq. (9).

$$\mathsf{w}(e) = \mathsf{ief}(e) \;/\; (\mathsf{p}(e) \times \mathsf{d}^2(e)) \tag{9}$$

Several other factors can be considered for the weighting function. For instance, one can leverage a query log, if available, to give higher weights to edges that are used more often by other users. A comprehensive comparison of various weighting functions is an interesting future study to pursue. Nevertheless, given a better weighting function, the proposed algorithms can better capture the user intent. Various edges can also be pruned, i.e., they are given zero weight. Interested readers can refer to the Appendix in the online supplemental material for details of a heuristic-based preprocessing mechanism used in GQBE for pruning edges.

Query	Query Tuple	Table Size
F_1	⟨Donald Knuth, Stanford University, Turing Award⟩	18
F_2	⟨Ford Motor, Lincoln, Lincoln MKS⟩	25
F_3	⟨Nike, Tiger Woods⟩	20
F_4	⟨Michael Phelps, Sportsman of the Year⟩	55
F_5	⟨Gautam Buddha, Buddhism⟩	621
F_6	⟨Manchester United, Malcolm Glazer⟩	40
F_7	⟨Boeing, Boeing C-22⟩	89
F_8	⟨David Beckham, A. C. Milan⟩	94
F_9	⟨Beijing, 2008 Summer Olympics⟩	41
F_{10}	⟨Microsoft, Microsoft Office⟩	200
F_{11}	⟨Jack Kirby, Ironman⟩	25
F_{12}	⟨Apple Inc, Sequoia Capital⟩	300
F_{13}	⟨Beethoven, Symphony No. 5⟩	600
F_{14}	⟨Uranium, Uranium-238⟩	26
F_{15}	⟨Microsoft Office, C++⟩	300
F_{16}	⟨Dennis Ritchie, C⟩	163
F_{17}	⟨Steven Spielberg, Minority Report⟩	40
F_{18}	⟨Jerry Yang, Yahoo!⟩	8349
F_{19}	⟨C⟩	1240
F_{20}	⟨TomKat⟩	16
D_1	⟨Alan Turing, Computer Scientist⟩	52
D_2	⟨David Beckham, Manchester United⟩	273
D_3	⟨Microsoft, Microsoft Excel⟩	300
D_4	⟨Steven Spielberg, Catch Me If You Can⟩	37
D_5	⟨Boeing C-40 Clipper, Boeing⟩	118
D_6	⟨Arnold Palmer, Sportsman of the year⟩	251
D_7	⟨Manchester City FC, Mansour bin Zayed Al Nahyan⟩	40
D_8	⟨Bjarne Stroustrup, C++⟩	964

TABLE 1: Queries and Ground Truth Table Size

7 EXPERIMENTS

This section presents our 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.

Datasets We used two large real-world knowledge graphs—the 2011 versions of Freebase [4] and DBpedia [3]. We preprocessed the graphs so that the kept nodes are all named entities (e.g., Stanford University) and abstract concepts (e.g., Jewish people). In the Freebase graph, every edge is associated with an redundant back edge in the opposite direction. For instance, the back edge of *founded* is labelled *founded_by*. All back edges were removed. We also removed administrative edges such as *created_by* and those nodes having constant or numerical values. The resulting Freebase graph contains 28M nodes, 47M edges, and $5,428$ distinct edge labels. The DBpedia graph contains 759K nodes, 2.6M edges and $9,110$ distinct edge labels.

Methods Compared GQBE was compared with a Baseline, NESS [14] and exemplar queries [18] (EQ). We implemented all the methods except EQ. For EQ, queries used in our experiments were provided to the authors of [18] who executed them on their system and shared the results with us.

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. For each node v in the query graph, a set of candidate nodes in the data graph are identified. 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. The score of a candidate v' is the similarity between the neighborhoods of v and v', represented in the form of vectors, and further refined using an iterative process. Finally, one unlabeled query node is chosen as the pivot p. The top-k candidates for multiple unlabeled query nodes are put together to form answer tuples, if they are within the neighborhood of p's top-k candidates.

EQ proposes the concept of *exemplar queries* [18] which is similar to the paradigm of GQBE. However, EQ does not provide a definitive way of discovering query graph given an exemplar query tuple. Therefore, we provided the MQG discovered by GQBE as the query graph to the authors of [18], who then executed the MQG on EQ and shared the evaluation results with us. Similar to NESS, EQ also captures the neighborhood information of each node in the data graph and indexes it. It iteratively picks nodes from the query graph and finds all similar candidate nodes in the data graph, while keeping only those candidates of each query node that also preserve the edges in the query graph with other nodes' candidates. It mandates all answer graphs to be edge preserving isomorphic matches to the query graph for the query tuple. This precludes their system from finding approximate answers to the query graph. These answer graphs are then ranked by the similarity of the nodes in the query graph and their corresponding nodes in the answer graphs.

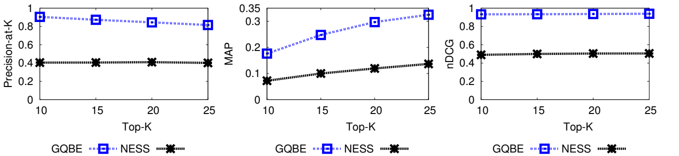
Baseline explores a query lattice in a bottom-up manner and prunes ancestors of null nodes, similar to the best-first method (Sec.5). Baseline . However, differently, it evaluates the lattice by breadth-first traversal instead of in the order of upper-bound scores. There is no early-termination by top-k scores, as Baseline terminates when every node is either evaluated or pruned.

Queries and Ground Truth Two groups of queries are used on the two datasets, respectively. The Freebase queries F_1 and F_6 are from Wikipedia tables such as http://en.wikipedia.org/wiki/List_of_English_football_club_owners. The remaining Freebase queries are based on tables obtained as a result of either constructing structured queries over Freebase, or pre-defined Freebase tables such as http://www.freebase.com/view/computer/programming_language_designer?instances. The DBpedia queries D_1– D_8 are based on DBpedia tables such as the values for property is dbpedia-owl:author of on page http://dbpedia.org/page/Microsoft. 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. All the 28 queries and their corresponding table sizes are summarized in Table 1. They cover diverse domains, including people, companies, movies, sports, awards, religions, universities and automobiles.

Sample Answers Table 2 only lists the top-3 results found by GQBE for 3 queries (F_1, F_{18}, F_{19}), due to space limitations.

Query Tuple	Top-3 Answer Tuples
⟨Donald Knuth, Stanford, Turing Award⟩	⟨D. Knuth, Stanford, V. Neumann Medal⟩ ⟨J. McCarthy, Stanford, Turing Award⟩ ⟨N. Wirth, Stanford, Turing Award⟩
⟨Jerry Yang, Yahoo!⟩	⟨David Filo, Yahoo!⟩ ⟨Bill Gates, Microsoft⟩ ⟨Steve Wozniak, Apple Inc.⟩
⟨C⟩	⟨Java⟩ ⟨C++⟩ ⟨C Sharp⟩

TABLE 2: Case Study: Top-3 Results for Selected Queries



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

Fig. 11: Accuracy of GQBE and NESS over all Freebase Queries

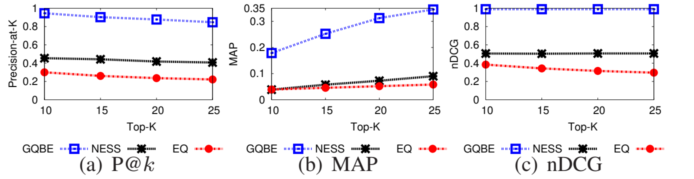
(A) Accuracy Based on Ground Truth

We measured the accuracy of GQBE and NESS based on the ground truth. The accuracy of a system is its average accuracy on a set of queries. The accuracy on a single query is captured by three widely-used measures [17], as follows.

- Precision-at-k (P@k): the percentage of the top-k results that belong to the ground truth.
- Mean Average Precision (MAP): The average precision of the top-k results is AvgP=$\frac{\sum_{i=1}^{k} \text{P@}i \times rel_i}{\text{size of ground truth}}$, where rel_i equals 1 if the result at rank i is in the ground truth and 0 otherwise. MAP is the mean of AvgP for a set of queries.
- Normalized Discounted Cumulative Gain (nDCG): $\text{nDCG}_k = \frac{\text{DCG}_k}{\text{IDCG}_k}$, where DCG_k is the cumulative gain of the top-k results, and IDCG_k is the cumulative gain for an ideal ranking of the top-k results. $\text{DCG}_k = rel_1 + \sum_{i=2}^{k} \frac{rel_i}{\log_2(i)}$, i.e., it penalizes a system if a ground truth result is ranked low.

Fig.11 shows these measures for different values of k over all Freebase queries for GQBE and NESS. GQBE has high accuracy. For instance, its P@25 is over 0.8 as evident in Fig. 11(a) and nDCG at top-25 is over 0.9 as shown in Fig. 11(c). For 13 of the 20 queries, either the P@25 was 1, or when the ground-truth size was less than 25, the AvgP was 1 (indicating that all answers in the ground-truth were ranked higher than any other answer). The absolute value of MAP is not high, merely because Fig.11(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 finds approximate matches to the query graph while giving priority to query entities and important edges in the MQG. NESS on the other hand 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.

Fig. 12 compares the measures for GQBE, NESS and EQ, on different values of k. Only 11 of the 20 Freebase queries (F_3, F_5, F_6, F_7, F_{10}, F_{11}, F_{14}, F_{15}, F_{16}, F_{17} and F_{18}) were considered in this experiment, since the authors of EQ were unable to produce answer tuples to other query graphs we provided. EQ performs weakly on these 11 queries. Furthermore, on 7 of the 11 queries, EQ was unable to return more than 5 answer tuples. This is because EQ finds



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

Fig. 12: Accuracy of GQBE, NESS and EQ over 11 Freebase Queries

Query	P@k	nDCG	AvgP	Query	P@k	nDCG	AvgP
D_1	1.00	1.00	0.20	D_2	1.00	1.00	0.04
D_3	1.00	1.00	0.03	D_4	0.80	0.94	0.19
D_5	0.90	1.00	0.08	D_6	1.00	1.00	0.04
D_7	0.90	0.98	0.22	D_8	1.00	1.00	0.01

TABLE 3: Accuracy of GQBE on DBpedia Queries, k=10

answer graphs that are exact matches to the query graph structure, and as query graphs get bigger, finding such edge-preserving isomorphic answer graphs becomes less likely. On the contrary, GQBE finds approximate matches too and thus has a better recall and accuracy than EQ. This also highlights the fact that the initial query graph provided to EQ plays a crucial role in its accuracy. Both NESS and EQ rely on finding the best matches for individual entities in the query tuple, and then integrating them to form the answer tuples. As mentioned in Section 2, best matches for individual entities may not form the best match for the query tuple as a whole. This is attested by the results we present here.

Table 3 further shows the accuracy of GQBE on individual DBpedia queries at k=10. It exhibits high accuracy on all queries, including perfect precision in several cases.

(B) Accuracy Based on User Studies

We conducted an extensive user study through Amazon Mechanical Turk (MTurk, https://www.mturk.com/mturk/) to evaluate GQBE's accuracy on Freebase queries, measured by Pearson Correlation Coefficient (PCC). For each of the 20 queries, we obtained the top-30 answers from GQBE and generated 50 random pairs of these answers. We presented each pair to 20 MTurk workers and asked for their preference between the two answers in the pair. Hence, in total, $20,000$ opinions were obtained. We then constructed two value lists per query, X and Y, which represent GQBE and MTurk workers' opinions, respectively. Each list has 50 values, for the 50 pairs. For each pair, the value in X is the difference between the two answers' ranks given by GQBE, and the value in Y is the difference between the numbers of workers favoring the two answers. The PCC value for a query is $(\mathrm{E}(XY) - \mathrm{E}(X)\mathrm{E}(Y))/(\sqrt{\mathrm{E}(X^2) - (\mathrm{E}(X))^2}\sqrt{\mathrm{E}(Y^2) - (\mathrm{E}(Y))^2})$. The value indicates the degree of correlation between the pairwise ranking orders produced by GQBE and the pairwise preferences given by MTurk workers. The value range is from -1 to 1. A PCC value in the ranges of [0.5,1.0], [0.3,0.5) and [0.1,0.3) indicates a strong, medium and small positive correlation, respectively [6]. PCC is undefined, by definition, when X and/or Y contain all equal values.

Table 4 shows the PCC values for F_1-F_{20}. Out of the 20 queries, GQBE attained strong, medium and small positive correlation with MTurk workers on 9, 5 and 3 queries, respectively. Only query F_7 shows no correlation. Note that PCC is undefined for F_{12} and F_{13}, because all the top-30 answer tuples have the same score and thus the same rank, resulting in all zero values in X, i.e., GQBE's list.

Query	PCC	Query	PCC	Query	PCC	Query	PCC
F_1	0.79	F_2	0.78	F_3	0.60	F_4	0.80
F_5	0.34	F_6	0.27	F_7	0.06	F_8	0.26
F_9	0.33	F_{10}	0.77	F_{11}	0.58	F_{12}	undefined
F_{13}	undefined	F_{14}	0.62	F_{15}	0.43	F_{16}	0.29
F_{17}	0.64	F_{18}	0.30	F_{19}	0.40	F_{20}	0.65

TABLE 4: Pearson Correlation Coefficient (PCC) between GQBE and Amazon MTurk Workers, k=30

(C) Accuracy on Multi-tuple Queries

We investigated the effectiveness of the multi-tuple querying approach (Sec.3.2). We experimented with up to three example tuples for each query: Tuple1 refers to the query tuple in Table 1, while Tuple2 and Tuple3 are two tuples from its ground truth. Fig. 13 shows the accuracy of top-25 GQBE answers for the three tuples individually, as well as for the first two and three tuples together by merged MQGs, which are denoted Combined(1,2) and Combined(1,2,3), respectively. The results show that, in most cases, Combined(1,2) had better accuracy than individual tuples and Combined(1,2,3) further improved the accuracy. In the aforementioned single-tuple query experiment (A), 13 of the 20 queries attained perfect precision. Due to space constraints, we present in Fig. 13 the results of only the remaining 7 queries. The results of all queries can be found in Table 6 of the Appendix that appears in the online supplemental material.

(D) Efficiency Results

We compared the efficiency of GQBE, NESS and Baseline on Freebase queries. The total run time for a query tuple is spent on two components—query graph discovery and query processing. We did not include EQ in this comparison since the system configuration on which the authors of [18] executed the queries was different from ours. Fig.15 compares the three methods' query processing time for each Freebase query, in logarithmic scale. The edge cardinality of the MQG for each query 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. It finished within 10 seconds on 17 queries. However, it performed very poorly on F4 and F19, which have 10 and 7 edges respectively. This indicates that the edges in the two MQGs lead to poor join selectivity. Baseline clearly suffered, due to its inferior pruning power compared to the best-first exploration employed by GQBE. This is evident in Fig.16 which shows the numbers of lattice nodes evaluated for each query. GQBE evaluated considerably less nodes in most cases and at least 2 times less on 11 of the 20 queries.

MQG discovery precedes lattice evaluation and is shared by all three methods. Column MQG_1 in Table 5 lists the time spent on discovering MQG for each Freebase query. The time varies across individual queries, depending on the sizes of query tuples' neighborhood graphs. Compared to the values shown in Fig.15, the time taken to discover an MQG in average

Query	Tuple1			Tuple2			Combined (1,2)			Tuple3			Combined (1,2,3)		
	P@k	nDCG	AvgP	P@k	nDCG	AvgP	P@k	nDCG	AvgP	P@k	nDCG	AvgP	P@k	nDCG	AvgP
F_1	**0.36**	0.76	0.32	**0.36**	**1.00**	**0.50**	0.12	0.38	0.02	**0.36**	0.73	0.22	0.12	0.49	0.02
F_2	0.76	**1.00**	0.79	0.00	0.00	0.00	**0.80**	**1.00**	0.80	0.12	0.70	0.05	**0.80**	**1.00**	**0.91**
F_4	0.32	0.73	0.09	0.40	0.65	0.08	**1.00**	**1.00**	0.45	**1.00**	**1.00**	0.04	**1.00**	**1.00**	**0.48**
F_6	0.24	0.89	0.16	0.28	0.89	0.18	**0.40**	0.87	0.16	0.36	0.98	**0.22**	0.12	**0.94**	0.07
F_8	0.92	0.79	0.20	**1.00**	**1.00**	**0.27**	0.96	0.98	0.24	0.48	0.86	0.08	**1.00**	**1.00**	**0.27**
F_9	0.68	0.72	0.23	0.56	0.66	0.17	0.80	0.86	0.35	**1.00**	**1.00**	0.62	**1.00**	**1.00**	**0.66**
F_{17}	0.32	**1.00**	0.33	0.64	0.83	0.25	0.32	**1.00**	0.32	0.56	0.84	0.23	**0.68**	**1.00**	**0.46**

Fig. 13: Accuracy of GQBE on Multi-tuple Queries, k=25

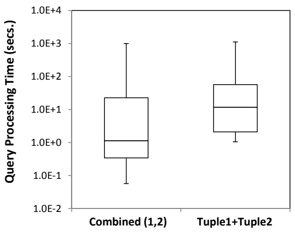


Fig. 14: Query Processing Time of 2-tuple Queries

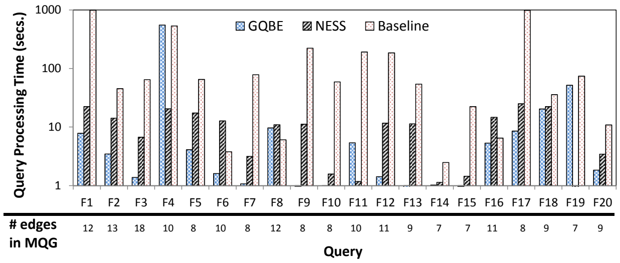


Fig. 15: Query Processing Time

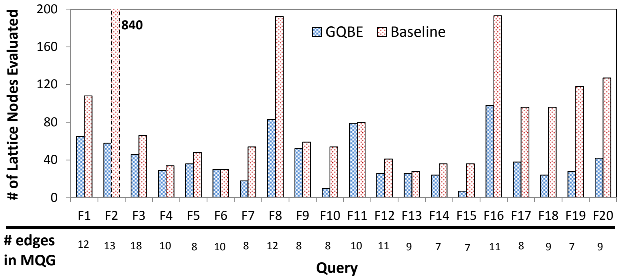


Fig. 16: Lattice Nodes Evaluated

Query	MQG_1	MQG_2	Merge	Query	MQG_1	MQG_2	Merge
F_1	73.141	73.676	0.034	F_2	0.049	0.029	0.006
F_3	12.566	4.414	0.024	F_4	5.731	7.083	0.024
F_5	9.982	2.522	0.079	F_6	6.082	4.654	0.039
F_7	0.152	0.107	0.007	F_8	10.272	2.689	0.032
F_9	62.285	2.384	0.041	F_{10}	2.910	5.933	0.030
F_{11}	59.541	65.863	0.032	F_{12}	1.977	0.021	0.006
F_{13}	9.481	5.624	0.034	F_{14}	0.038	0.015	0.004
F_{15}	0.154	5.143	0.021	F_{16}	54.870	6.928	0.057
F_{17}	60.582	69.961	0.041	F_{18}	58.807	75.128	0.053
F_{19}	0.224	0.076	0.003	F_{20}	0.025	0.017	0.002

TABLE 5: Time for Discovering and Merging MQGs (secs.)

is comparable to the time spent in evaluating it.

Fig.14 shows the distribution of GQBE's query processing time, in logarithmic scale, on the merged MQGs of 2-tuple queries in Fig. 13, denoted by Combined(1,2). It also shows the distribution of the total time for evaluating the two tuples' MQGs individually, denoted Tuple1+Tuple2. Combined(1,2) processes 10 of the 20 queries in less than a second while the fastest query for Tuple1+Tuple2 takes a second. This suggests that the merged MQGs gave higher weights to more selective edges, resulting in faster lattice evaluation. Meanwhile, these selective edges are also more important edges common to the two tuples, leading to improved answer accuracy shown in Fig. 13. Table 5 further shows the time taken to discover MQG_1 and MQG_2, along with the time for merging them. The latter is negligible compared to the former.

8 Related Work

The paradigm of *query-by-example* (QBE) has a long history in relational databases [26]. Its simplicity and improved user productivity make QBE an influential database query language. By proposing to query knowledge graphs by example tuples, our premise is that the QBE paradigm will enjoy similar advantages on graph data. The technical challenges and approaches are vastly different, due to the fundamentally different data models.

In the literature on graph query, the input to a query system in most cases is a structured query, which is often graphically presented as a query graph or pattern. The query graphs and patterns are formed by using structured query languages. For instance, PathSim [20] finds the top-k similar entities that are connected to a query entity, based on a user-defined meta-path semantics in a heterogeneous network. In [25], given a query graph as input, the system finds structurally isomorphic answer graphs with semantically similar entity nodes. In contrast, GQBE only requires a user to provide an entity tuple, without knowing the underlying schema.

Lim et al. [16] use example tuples to find similar tuples in database tables that are coupled with ontologies. They do not deal with graph data and example entity tuples. The goal of *set expansion* is to grow a set of objects starting from seed objects. Example systems include [23] and the now defunct Google Sets and Squared services (http://en.wikipedia.org/wiki/List_of_Google_products). Chang et al. [5] identify top-k correlated keyword terms from an information network given a set of terms, where each term can be an entity. These systems, except [5], do not operate on data graphs. Instead, they find existing answers within structures such as HTML tables and lists. Further, except Google Squared, they all take a set of individual entities as input. GQBE is more general in that each query tuple contains multiple entities. It is unrealistic to find web tables that can cover all possible queries, especially for queries involving multiple entities. Moreover, knowledge graphs and web tables complement each other in content. One does not subsume the other.

Several works [13], [8], [7] identify the best subgraphs/paths in a data graph to describe how several input nodes are related. The query graph discovery component of GQBE is different in important ways– (1) Although the graphs in [13], [8], [7] have many different types of entities and relationships, they cannot discover hidden attributes that are not in the path between input nodes. REX [8] and [7] have the further limitation of allowing only two query entities. Differently, the MQG in GQBE allows multiple query entities and also includes edges incident on individual query entities. (2) GQBE uses the discovered query graph to find answer graphs and answer tuples, which is not within the focus of the aforementioned works.

There are many studies on approximate/inexact subgraph matching in large graphs, e.g., G-Ray [22], TALE [21] and NESS [14]. GQBE's query processing component is different from them on several aspects. (1) GQBE only requires to match edge labels and matching node identifiers is not mandatory. This is equivalent to matching a query graph with all unlabeled nodes and thereby significantly increases the problem complexity. Only a few previous methods (e.g., NESS [14]) allow unlabeled query nodes. (2) In GQBE, the top-k query algorithm centers around query entities—the weighting function gives more importance to edges closer 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. Our empirical results show that this difference makes NESS produce less accurate answers than GQBE. (3) 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.

9 Conclusion

We introduce GQBE, a system that queries 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. Its query graph discovery component derives a hidden query graph based on example tuples. The query lattice based on this hidden graph may contain a large number of query graphs. GQBE's query algorithm only partially evaluates query graphs for obtaining the top-k answers. Experiments on Freebase and DBpedia datasets show that GQBE outperforms the state-of-the-art systems NESS and EQ on both accuracy and efficiency.

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] L. Chang, J. X. Yu, L. Qin, Y. Zhu, and H. Wang. Finding information nebula over large networks. In *CIKM*, 2011.

[6] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[7] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *SIGKDD*, pages 118–127, 2004.

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

[9] H. N. Gabow and E. W. Myers. Finding all spanning trees of directed and undirected graphs. *SIAM J. Comput.*, 7(3):280–287, 1978.

[10] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, 2007.

[11] N. Jayaram, M. Gupta, A. Khan, C. Li, X. Yan, and R. Elmasri. GQBE: Querying knowledge graphs by example entity tuples. In *ICDE'14*.

[12] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Towards a query-by-example system for knowledge graphs. In *GRADES*, 2014.

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

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

[15] Z. Li, S. Zhang, X. Zhang, and L. Chen. Exploring the constrained maximum edge-weight connected graph problem. *Acta Mathematicae Applicatae Sinica*, 25:697–708, 2009.

[16] L. Lim, H. Wang, and M. Wang. Semantic queries by example. In *EDBT*, 2013.

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

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

[19] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge unifying WordNet and Wikipedia. In *WWW*, 2007.

[20] Y. Sun, J. Han, X. Yan, P. S. Yu, , and T. Wu. PathSim: Meta path-based top-k similarity search in heterogeneous information networks. *VLDB*, 2011.

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

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

[23] R. C. Wang and W. W. Cohen. Language-independent set expansion of named entities using the web. In *ICDM*, pages 342–350, 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] X. Yu, Y. Sun, P. Zhao, and J. Han. Query-driven discovery of semantically similar substructures in heterogeneous networks. In *KDD'12*.

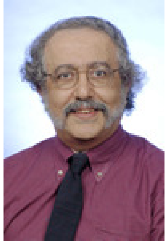
[26] M. M. Zloof. Query by example. In *AFIPS*, 1975.

Nandish Jayaram received his B.E. degree in computer science from VTU and M.S. degree in information technology from International Institute of Information Technology, Bangalore, India. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Texas at Arlington. His research interests include query formulation and processing in large knowledge graphs.

Arijit Khan is a post-doctorate researcher in the systems group at ETH Zurich. His research interests span in the area of big-data, big-graphs, and graph systems. He completed his PhD from the Department of Computer Science, University of California at Santa Barbara in 2013. He is the recipient of the prestigious IBM PhD fellowship award in 2012-2013. Arijit Khan co-presented a tutorial on emerging queries over linked data at ICDE 2012.

Chengkai Li is an Associate Professor in the Department of Computer Science and Engineering at the University of Texas at Arlington. His research interests are in big data management and mining. He received his Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 2007, and an M.E. and a B.S. degree in Computer Science from Nanjing University, China, in 2000 and 1997, respectively.

Xifeng Yan is an Associate Professor at the University of California at Santa Barbara. He holds the Venkatesh Narayanamurti Chair of Computer Science. He received his Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 2006. He has been working on modeling, managing, and mining graphs in information networks, computer systems, social media and bioinformatics.



Ramez Elmasri received his Ph.D. in computer science from Stanford University in 1980, and also received his M.S. from Stanford. He received his B.S. in electrical engineering from Alexandria University, Egypt, in 1972. His current research interests are in applying database techniques to new technologies such as bioinformatics, mobile systems, and sensor networks.

Supplemental Material to “Querying Knowledge Graphs by Example Entity Tuples”

A Appendix

Complexity Analysis of Alg.1

In the aforementioned divide-and-conquer method, if on average there are $r'=\frac{|E(H_t)|}{n+1}$ edges in each subgraph, finding the subgraph by DFS and sorting its r' edges takes $O(r' \log r')$ time. Given the top-s edges of a subgraph, checking if the weakly connected component M_s exists using DFS requires $O(s)$ time. Suppose on average c iterations are required to find the appropriate s. Let $m=\frac{r}{n+1}$ be the average target edge cardinality of each subgraph. Since the method initializes s with m, the largest value s can attain is $m+c$. So the time for discovering M_s for each subgraph is $O(r' \log r' + c \times (m+c))$. For all $n+1$ subgraphs, the total time required to find the final MQG_t is $O((n+1) \times (r' \log r' + c \times (m+c)))$. For the queries used in our experiments on Freebase, given an empirically chosen small r=15, $s \ll |E(H_t)|$ and on average c=22.

Complexity Analysis of Merging Multiple MQGs

In comparison to evaluating a single-tuple query, the extra overhead in handling a multi-tuple query includes creating multiple MQGs, which is $|T|$ times the average cost of discovering an individual MQG, and merging them, which is linear in the total edge cardinality of all MQGs.

Complexity Analysis of Alg. 2

Joins are used to evaluate the lattice nodes. Minimal query trees might require multiple joins and other lattice nodes require a single join each. In evaluating the latter, if on average, the number of answer graphs for a lattice node is j, the time to evaluate a node by joining the answers of its child node and the new edge added to form the node is $O(j)$. If $|\mathcal{L}_e|$ is the actual number of lattice nodes evaluated, the worst case scenario of query processing is $O(|\mathcal{L}_e| \times j)$. In practice, due to the pruning power of the best-first exploration technique, $|\mathcal{L}_e| \ll |\mathcal{L}|$. For the queries used in our experiments on Freebase, on average only 8% of $|\mathcal{L}|$ is evaluated. The average number of answers to a lattice node, j, is 6500. Thus, the time to evaluate a single lattice node has a significant role in the total query processing time. Therefore, the query processing time is not only dependent on the size of MQG_t, but also on the join cardinality involving the edges.

Complexity Analysis of Alg.3

The query graphs corresponding to lattice nodes are represented using bit vectors since we exactly know the edges involved in all the query graphs. The bit corresponding to an edge is set if its present in the query graph. Identifying the dirty nodes, null upper boundary nodes and building a new potential upper boundary node using a pair of nodes $\langle Q, Q' \rangle$, can be accomplished using bit operations and each step incurs $O(|E(MQG_t)|)$ time. Finding the weakly connected component of a potential upper boundary using DFS takes $O(|E(Q')|)$ time. If $\mathcal{L}_n$ is the set of all null nodes encountered in the lattice and there are D_p such pairs for every null node and q is the average number of potential new upper boundary nodes created per pair, the worst case time complexity of recomputing the upper-frontier is $O(|\mathcal{L}_n| \times D_p \times q \times |E(MQG_t)|)$. Our experimental results show low average values of $|\mathcal{L}_n|$, D_p and q with $|\mathcal{L}_n|$ being only 1% of $|\mathcal{L}|$, D_p around 8 and q around 9. In practice, our upper-frontier recomputation algorithm quickly computes the dynamically changing lattice.

Preprocessing: Reduced Neighborhood Graph

Alg. 1 focuses on discovering MQG_t from H_t. The neighborhood graph H_t may have clearly unimportant edges. As a preprocessing step, GQBE removes such edges from H_t before applying Alg.1. The reduced size of H_t not only makes the execution of Alg.1 more efficient but also helps prevent clearly unimportant edges from getting into MQG_t.

Consider the neighborhood graph H_t in Fig.3, based on the data graph excerpt in Fig.1. Edge e_1=(Jerry Yang, Stanford) and $label(e_1)$=*education*. Two other edges labeled *education*, e_2 and e_3, are also incident on node Stanford. The neighborhood graph from a complete real-world data graph may contain many such edges for people graduated from Stanford University. Among these edges, e_1 represents an important relationship between Stanford and query entity Jerry Yang, while other edges represent relationships between Stanford and other entities, which are deemed unimportant with respect to the query tuple.

We formalize the definition of *unimportant edges* as follows. Given an edge $e=(u,v) \in E(H_t)$, e is unimportant if it is unimportant from the perspective of its either end, u or v, i.e., if $e \in UE(u)$ or $e \in UE(v)$. Given a node $v \in V(H_t)$, $E(v)$ denotes the edges incident on v in H_t. $E(v)$ is partitioned into three disjoint subsets—the important edges $IE(v)$, the unimportant edges $UE(v)$ and the rest—defined as follows:

$$IE(v)= \\ \{e \in E(v) \mid \exists v_i {\in} t, p \text{ s.t. } e {\in} p, ends(p){=}\{v, v_i\}, len(p) {\leq} d\};$$

$$UE(v)= \\ \{e \in E(v) \mid e {\notin} IE(v), \exists e' {\in} IE(v) \text{ s.t. } label(e){=}label(e'), \\ (e{=}(u,v) \wedge e'{=}(u',v)) \vee (e{=}(v,u) \wedge e'{=}(v,u'))\}.$$

An edge e incident on v belongs to $IE(v)$ if there exists a path between v and any query entity in the query tuple t, through e, with path length at most d. For example, edge e_1 in Fig.3 belongs to IE(Stanford). An edge e belongs to $UE(v)$ if (1) it does not belong to $IE(v)$ (i.e., there exists no such aforementioned path) and (2) there exists $e' \in IE(v)$ such that e and e' have the same label and they are both either incoming into or outgoing from v. By this definition, e_2 and e_3 belong to $UE(v)$ in Fig.3, since e_1 belongs to $IE(v)$. In the same neighborhood graph, e_4 is in neither $IE(v)$ nor $UE(v)$.

All edges deemed unimportant by the above definition are removed from H_t. The resulting graph may not be weakly connected anymore and may have multiple weakly connected components. Theorem 4 states that one of the components—called the *reduced neighborhood graph*, denoted H'_t—contains all query entities in t. In other words, H'_t is the largest weakly connected subgraph of H_t containing all query entities and no unimportant edges. Alg.1 is applied on H'_t to produce MQG_t.

Theorem 4 Given the neighborhood graph H_t for a query tuple t, the reduced neighborhood graph H'_t always exists.

Proof: We prove by contradiction. Suppose that, after removal of all unimportant edges, H_t becomes a disconnected

Query	**Tuple1**			**Tuple2**			**Combined (1,2)**			**Tuple3**			**Combined (1,2,3)**		
	P@k	nDCG	AvgP	P@k	nDCG	AvgP	P@k	nDCG	AvgP	P@k	nDCG	AvgP	P@k	nDCG	AvgP
F_1	**0.36**	0.76	0.32	**0.36**	**1.00**	**0.50**	0.12	0.38	0.02	**0.36**	0.73	0.22	0.12	0.49	0.02
F_2	0.76	**1.00**	0.79	0.00	0.00	0.00	**0.80**	**1.00**	0.80	0.12	0.70	0.05	**0.80**	**1.00**	**0.91**
F_3	**0.76**	**0.85**	**1.00**	**0.76**	**0.85**	**1.00**	0.72	0.82	**1.00**	**0.76**	**0.85**	**1.00**	0.68	0.79	**1.00**
F_4	0.32	0.73	0.09	0.40	0.65	0.08	**1.00**	**1.00**	0.45	**1.00**	**1.00**	0.04	**1.00**	**1.00**	**0.48**
F_5	**1.00**	**1.00**	**0.04**	**1.00**	**1.00**	**0.04**	**1.00**	**1.00**	**0.04**	**1.00**	**1.00**	**0.04**	**1.00**	**1.00**	**0.04**
F_6	0.24	0.89	0.16	0.28	0.89	0.18	**0.40**	0.87	0.16	0.36	**0.98**	**0.22**	0.12	0.94	0.07
F_7	**1.00**	**1.00**	0.28	**1.00**	**1.00**	0.28	**1.00**	**1.00**	0.28	**1.00**	**1.00**	0.28	**1.00**	**1.00**	**0.29**
F_8	0.92	0.79	0.20	**1.00**	**1.00**	**0.27**	0.96	0.98	0.24	0.48	0.86	0.08	**1.00**	**1.00**	**0.27**
F_9	0.68	0.72	0.23	0.56	0.66	0.17	0.80	0.86	0.35	**1.00**	**1.00**	0.62	**1.00**	**1.00**	**0.66**
F_{10}	**1.00**	**1.00**	0.12	**1.00**	**1.00**	0.12	**1.00**	**1.00**	0.12	**1.00**	**1.00**	0.12	**1.00**	**1.00**	**0.13**
F_{11}	**0.96**	**0.97**	**1.00**	0.32	0.50	0.29	0.72	0.82	0.78	0.00	0.00	0.00	0.36	0.55	0.41
F_{12}	**1.00**	**1.00**	**0.08**	**1.00**	**1.00**	**0.08**	0.96	0.88	0.07	0.36	0.39	0.01	0.96	0.88	0.07
F_{13}	**1.00**	**1.00**	**0.04**	**1.00**	**1.00**	**0.04**	**1.00**	**1.00**	**0.04**	0.00	0.00	0.00	**1.00**	**1.00**	**0.04**
F_{14}	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.96	0.97	**1.00**	**1.00**	**1.00**	**1.00**	0.92	0.95	**1.00**
F_{15}	**1.00**	**1.00**	**0.08**	0.56	0.48	0.02	**1.00**	**1.00**	**0.08**	**1.00**	**1.00**	**0.08**	**1.00**	**1.00**	**0.08**
F_{16}	**1.00**	**1.00**	**0.15**	**1.00**	**1.00**	**0.15**	**1.00**	**1.00**	**0.15**	**1.00**	**1.00**	**0.15**	**1.00**	**1.00**	**0.15**
F_{17}	0.32	**1.00**	0.33	0.64	0.83	0.25	0.32	**1.00**	0.32	0.56	0.84	0.23	**0.68**	**1.00**	**0.46**
F_{18}	**1.00**	**1.00**	**0.01**	**1.00**	**1.00**	**0.01**	**1.00**	**1.00**	**0.01**	**1.00**	**1.00**	**0.01**	**1.00**	**1.00**	**0.01**
F_{19}	**1.00**	**1.00**	**0.02**	**1.00**	**1.00**	**0.02**	**1.00**	**1.00**	**0.02**	**1.00**	**1.00**	**0.02**	**1.00**	**1.00**	**0.02**
F_{20}	**0.52**	**0.68**	0.86	**0.52**	**0.68**	0.86	**0.52**	**0.68**	0.92	**0.52**	**0.68**	0.86	**0.52**	**0.68**	**1.00**

TABLE 6: Accuracy of GQBE on all 20 Freebase Multi-tuple Queries, k=25

graph, of which none of the weakly connected components contains all the query entities. The deletion of unimportant edges must have disconnected at least a pair of query entities, say, v_i and v_j. By Def. 1, before removal of unimportant edges, H_t must have at least a path p of length at most d between v_i and v_j. By the definition of unimportant edges, every edge e=(u, v) on p belongs to both $IE(u)$ and $IE(v)$ and thus cannot be an unimportant edge. However, the fact that v_i and v_j become disconnected implies that p consists of at least one unimportant edge which is deleted. This presents a contradiction and completes the proof. □

Accuracy on Multi-tuple Queries

We show the accuracy of multi-tuple queries in Table 6. Tuple1 refers to the query tuple in Table 1, while Tuple2 and Tuple3 are two tuples from its ground truth. Table 6 shows the accuracy of top-25 GQBE answers for the three tuples individually, as well as for the first two and three tuples together by merged MQGs, which are denoted Combined(1,2) and Combined(1,2,3), respectively. The ground truth size of queries F_1, F_2, F_3, F_{11}, F_{14} and F_{20} is less than or equal to 25. Therefore, the P@k and nDCG values of these queries is lesser than 1, in spite of a complete recall. A value of 1 of the AvgP values of the corresponding entries indicates that all the tuples from the ground truth were ranked higher than any other answer tuple.