GQBE: Querying Entity-Relationship Graphs by Example Tuples

Nandish Jayaram[†] Mahesh Gupta[†] Arijit Khan[§] Chengkai Li[†] Xifeng Yan[§] Ramez Elmasri[†]
[†]University of Texas at Arlington, [§]University of California, Santa Barbara

ABSTRACT

We witness in many domains an unprecedented proliferation of entity-relationship graphs that capture entities and their relationships. Such data is difficult to use. The challenges lie in the gap between large and complex graphs and non-expert users. If writing structured queries over "simple" tables is difficult, complex graphs are only harder to query. As an initial step toward better usability of query systems over entity-relationship graphs, we propose to query such data by example entity tuples, without requiring users to form complex graph queries. To the best of our knowledge, there was no such proposal in the past. Our system, GQBE, tackles the challenges in supporting this query approach. It derives a weighted hidden maximal query graph based on input query tuples, to capture a user's query intent. It efficiently finds the top approximate answer tuples that are ranked by how well they match the query tuple. Its top-*k* query algorithm guarantees to evaluate only the query graphs that are necessary for obtaining top-*k* answers. We conducted experiments and user studies on the large Freebase and DBpedia datasets to evaluate the accuracy and efficiency of GQBE.

1. INTRODUCTION

As our society enters the era of big data, we witness in many domains an unprecedented proliferation of *entity-relationship graphs* that capture entities (e.g., persons, products, organizations) and their relationships. Figure 1 is an excerpt of an entity-relationship 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. Real-world instances of such graphs include knowledge bases (e.g., DBpedia [2], YAGO [24], Freebase [4]), social graphs, drug and disease databases, gene and protein databases, and program analysis graphs, to name just a few.

Users and developers are trying hard to tap into entity-relationship graphs for numerous applications. Google's Knowledge Graph, for instance, enhances users' search experience using data from Freebase and other sources. It echoes the shift of Web users' interest towards entity-related information, evidenced by the estimation that 71% of Microsoft Bing search queries contain named entities [32]; Hunch.com's "taste graph" makes personalized recommendations of things based on people's social relationships and shared tastes; Given a DNA sequence and a protein in a biology entity graph, a biologist may want to find out if the protein is involved in an interaction with another protein encoded by the DNA sequence [3].

Both users and application developers are often overwhelmed by the daunting task of understanding and using entity-relationship 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. While a number of graph database systems, triplestores, and RDF stores have emerged in recent years (e.g., Neo4j, Pregel [20], GBase [15], Trinity [23], AllegroGraph, and many others), *usability* has not been the focus of innovation. 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 is challenging. It requires extensive experiences in query language and data model and good understanding of particular datasets. For this reason, database usability has received considerable attention lately (cf. an overview in [12]). Graph data is not "easier" than relational data in either query language or data model. The fact that it is schema-less makes it even more intangible to understand such data and express queries on it. *If querying "simple" tables is difficult, aren't complex graphs harder to query?*

We aim to tackle the challenges in building query systems with good usability for entity-relationship graphs. As an initial step toward the objective, we build GQBE (Graph Query by Example), a system that queries entity-relationship graphs by example tuples, without requiring users to write complex graph queries. Specifically, given a data graph and a query tuple consisting of entities, GQBE finds similar answer tuples. Consider the data graph in Figure 1. Given a 2-entity query tuple $\langle$Jerry Yang, Yahoo!$\rangle$, the answer tuples can be $\langle$Steve Wozniak, Apple Inc.$\rangle$, $\langle$Sergey Brin, Google$\rangle$ and $\langle$Bill Gates, Microsoft$\rangle$, which are all founder-company pairs. If the query tuple consists of 3 or more entities (e.g., $\langle$Jerry Yang, Yahoo!, San Jose$\rangle$), the answers will be similar tuples of the same cardinality (e.g., $\langle$Steve Wozniak, Apple Inc., San Jose$\rangle$, since apart from the founder-company relationship, both Jerry Yang and Steve Wozniak have the relationship *places_lived* with San Jose).

The paradigm of *query-by-example* (QBE) has a long history in relational databases [34], in which the idea is to express queries by filling example tables with constants and shared variables in multiple tables, which correspond to selection and join conditions, respectively. Its simplicity and improved user productivity make QBE an influential database query language. By proposing to query entity-relationship graphs by example tuples, which was not seen before, our conjecture is that the QBE paradigm will enjoy similar advantages on graph data. The technical challenges and approaches

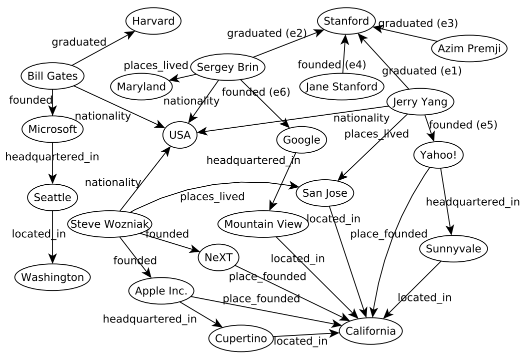


Figure 1: An Excerpt of an Entity-Relationship Graph

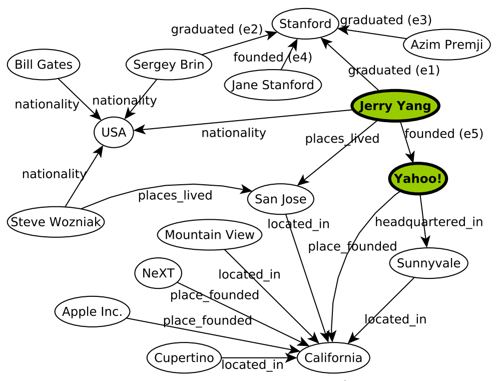


Figure 2: Neighborhood Graph for ⟨Jerry Yang, Yahoo!⟩

are vastly different, due to the fundamentally different data models. Note that query graphs or patterns are often used in the literature to graphically present queries over graphs. Underlyingly they are formed by using structured query languages or other query mechanisms such as keyword query [22, 31], interactive and visual query formulation [8, 13, 6, 14], and potentially the QBE approach proposed in this paper. Therefore they are not what we refer to as query-by-example.

There are several challenges in building GQBE. (1) To start with, the query input to GQBE is an entity tuple instead of a query graph. The system needs to figure out what are important "features" of the query tuple to be matched in answer tuples. More concretely, the entities in the query tuple are vertices in a data graph. These entities and other entities in their neighborhood are related to each other in certain ways. For an answer tuple, the corresponding answer entities and their neighboring entities are also expected to be related in such ways. Hence, with regard to *query semantics*, GQBE must derive a hidden query graph based on the query tuple, to capture a user's query intent. The *query graph discovery* component (Section 3) of GQBE fulfills this requirement and the derived graph is termed a *maximal query graph* (MQG). Edges in the MQG are weighted by several frequency-based and distance-based heuristics, in order to capture the relative importance of different relationships in the neighborhood of the query tuple entities. Answer graphs matching the MQG are projected to answer tuples, which consist of entities corresponding to the query tuple entities. We further facilitate multi-tuple queries that help to discover an MQG that better captures the user intent (Section 6).

(2) It is unlikely to find answer graphs exactly matching the MQG. Therefore query answers have to be approximate. In the aforementioned example, ⟨Steve Wozniak, Apple Inc.⟩ might be a more accurate answer tuple than others, because both Steve Wozniak and Jerry Yang have lived in San Jose, and both Apple Inc. and Yahoo! were founded and are headquartered in California. ⟨Sergey Brin, Google⟩ is arguably a better answer than ⟨Bill Gates, Microsoft⟩ because Google is also headquartered in California, and both Jerry Yang and Sergey Brin are graduates of Stanford. With regard to the relationships between query entities, there can be multiple direct and indirect paths between them. For instance, Jerry Yang and Yahoo! are connected through San Jose and California. The entities in query and answer tuples only need to have some paths in common. In summary, with regard to *answer space modeling* (Section 4), there can be a large space of approximate answer tuples. We must formalize the space and rank query answers by how well they match the query tuple.

In tackling this challenge, GQBE models the space of answer tuples by a *query lattice*. In more details, every query graph is a subgraph of the MQG that contains all query tuple entities. The query lattice is formed by the subsumption relation between all possible query graphs. Given a query graph, its corresponding answer graphs are also subgraphs of the data graph and are isomorphic to the query graph. Given an answer graph, its entities corresponding to the query tuple entities form an answer tuple, according to the bijection between the entity sets of the query graph and the answer graph. Thus the answer tuples are essentially approximate answers to the MQG. For ranking answer tuples, their scores are calculated based on the weights of edges in their query graphs and the match between vertices 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 thus be prohibitively expensive. Therefore, with regard to *query processing* (Section 5), it is imperative for GQBE to find the top ranked answers efficiently. We propose a top-k lattice exploration algorithm that only partially evaluates the lattice nodes. Without delving into details, the gist of the idea is to explore the lattice in the order of the query graphs' upper-bound scores.

We conducted extensive experiments and user study on the large Freebase and DBpedia datasets to evaluate the accuracy and efficiency of GQBE (Section 7). We also compared GQBE with a state-of-the-art graph querying framework NESS [18], and observed that the accuracy of GQBE is twice as better as NESS and it outperforms NESS on efficiency in most of the queries.

2. DATA MODEL, QUERY MODEL, AND PROBLEM STATEMENT

Definition 1 An ***entity-relationship graph***, or simply 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 $id(v)$. [1] Each edge $e{=}(v_i, v_j){\in}E(G)$ denotes a directed relationship from entity v_i to entity v_j. Each edge has a label, denoted as $label(e)$. Multiple edges can have the same label. ■

Definition 2 An ***n-tuple*** $t{=}\langle v_1, v_2, \ldots, v_n\rangle$ is an ordered list of n entities, where each entity is a node in G. ■

The user input and output to GQBE are both n-tuples and are thus called *query tuples* and *answer tuples*, respectively. Their entities are called *query (answer) tuple entities* or simply *query (answer) entities*. Our goal is to find the top-*k* most "similar" answer tuples, as stated below.

Problem Statement Given an entity-relationship graph G and a query tuple t, find the k answer tuples t' with the highest similarity scores $\mathsf{score}_t(t')$.

We define $\mathsf{score}_t(t')$, the score of an answer tuple t' with respect to a query tuple t, by matching the relationships between the

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

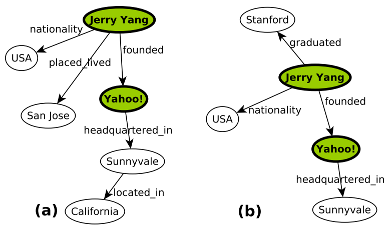


Figure 3: Query Graphs in Figure 2

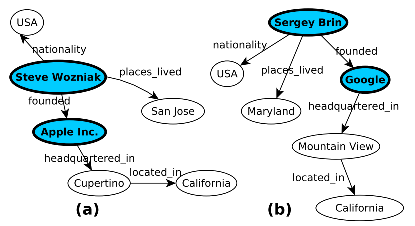


Figure 4: Answer Graphs for Figure 3(a)

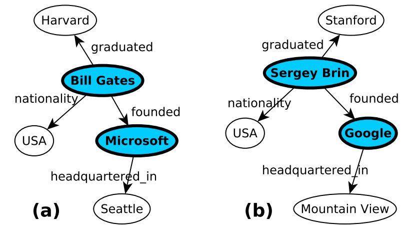


Figure 5: Answer Graphs for Figure 3(b)

entities in the two tuples. Observe that entities have relationships between each other not only directly but also indirectly through other entities. Therefore, the essence of the problem is the matching between two graphs, constructed from t and t', respectively. In this context, we define the *neighborhood graph* for an n-tuple, which is based on the concept of undirected path.

An *undirected path* in a directed graph is a path whose edges are not necessarily oriented in the same direction. Unless otherwise stated, we will refer to undirected path simply as "path". We consider undirected path instead of directed path because both incoming edges and outgoing edges of a node can represent important relationships with other nodes. More formally, we define a path p as a sequence of edges $e_1, e_2, \ldots, e_n$ and we say each edge $e_i \in p$. The path connects two nodes v_0 and v_n through intermediate nodes $v_1, v_2, \ldots, v_{n-1}$, 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 length of the path, $len(p)$, is n and the endpoints of the path, $ends(p)$, are $\{v_0, v_n\}$. Note that there is no undirected cycle in a path, i.e., the entities $v_0, v_1, \ldots, v_n$ are all distinct.

Definition 3 The ***neighborhood graph*** of query tuple t, denoted H_t, is the *weakly connected subgraph*[2] of G that consists of all undirected paths in G of length d or smaller, with at least one endpoint of each such path being a query entity in t. 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 Figure 1, the neighborhood graph for query tuple ⟨Jerry Yang, Yahoo!⟩ with path length threshold d=2 is shown in Figure 2. The nodes in dark color are the query entities. Other nodes are those to which there exists an undirected path of length at most 2 from either Jerry Yang, or Yahoo!, or both in the data graph. ■

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. It is unlikely that we will find query answers that exactly match the neighborhood graph. It is however possible to find exact matches to some subgraphs of the neighborhood graph. Such subgraphs are all query graphs and their exact matches are approximate answers that match the neighborhood graph to different extents.

Definition 4 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, Figure 3 shows two query graphs for the neighborhood graph in Figure 2.

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

Echoing the intuition behind neighborhood graph, the definitions of answer graph and answer 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 5 An ***answer graph*** A to a query graph Q is a weakly connected subgraph of G that is 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 an edge $e = (f^{-1}(u_i), f^{-1}(u_j)) \in E(Q)$ such that $label(e) = label(e')$.

The ***answer tuple*** in A is t_A=$\langle f(v_1), f(v_2), \ldots, f(v_n)\rangle$. 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) Figures 4 and 5 each show two answer graphs for query graphs (a) and (b) in Figure 3, respectively. The answer tuples in Figure 4 are ⟨Steve Wozniak, Apple Inc.⟩ and ⟨Sergey Brin, Google⟩. The answer tuples in Figure 5 are ⟨Bill Gates, Microsoft⟩ and ⟨Sergey Brin, Google⟩. ■

The same answer tuple may be projected from multiple answer graphs, which in turn can match different query graphs. For instance, Figures 4(b) and 5(b), which are answer graphs for different query graphs, have the same projection—⟨Sergey Brin, Google⟩.

Our objective is to return only distinct answer tuples. Given a query tuple t, if an answer tuple t' can be projected from multiple answer graphs, the highest score attained by the answer graphs is assigned as the score of t', capturing how well t' matches t.

Definition 6 The set of answer tuples for query tuple t are $\{t_A | A \in \mathcal{A}_Q, Q \in \mathcal{Q}_t\}$. The ***answer tuple 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 the similarity between A and the query graph Q. Its exact equation is given in Section 4. Without delving into details, we note that $\mathsf{score}_Q(A)$ sums up two values—the total weight of edges in Q and the extra credits given to identical matching nodes in A and Q. How edge weights are derived is discussed in Section 3.

3. QUERY GRAPH DISCOVERY

The concept of neighborhood graph H_t (Definition 3) 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.

The query graph discovery component of GQBE constructs a weighted *maximal query graph* (MQG) from the neighborhood graph

H_t. The resulting graph is expected to be drastically smaller than H_t and capture only important features of the query tuple. It is constructed by several heuristics. First, clearly unimportant edges are removed from H_t. Then, remaining edges are weighted by both their global frequencies in the data graph and local frequencies in the reduced graph. We use a greedy method to find a connected subgraph of the reduced graph with high total edge weight, under a target edge cardinality. Finally, edge weights in the resulting graph are revised, by considering distances to query entities, in addition to the aforementioned frequencies. This section explains these ideas.

3.1 Reduced Neighborhood Graph

Unimportant Edges Consider the neighborhood graph H_t in Figure 2, based on the data graph excerpt in Figure 1. Edge e_1=(Jerry Yang, Stanford) and $label(e_1)$=*graduated*. Two other edges labeled *graduated*, 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 mutually exclusive subsets—the important edges $IE(v)$, the unimportant edges $UE(v)$, and the rest. They are defined as follows:

$IE(v)$=$\{e{\in}E(v) \mid \exists v_i{\in}t, p$ s.t. $e{\in}p, ends(p)$=$\{v, v_i\}, len(p){\leq}d\}$;
$UE(v)$=$\{e{\in} E(v) \mid e{\notin}IE(v), \exists e'{\in}IE(v)$ 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 no more than the path length threshold d. For example, edge e_1 in Figure 2 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 Figure 2, 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. [3] It can be proved that one of the components contains all query entities in t. In other words, it is the largest weakly connected subgraph of H_t containing all query entities and no unimportant edges. It is called the *reduced neighborhood graph*, denoted H'_t.

Theorem 1 Given the neighborhood graph H_t for a query tuple t, the reduced neighborhood graph H'_t always exists. ■

PROOF. The proofs of this and most other theorems are omitted due to space limitations. ■

3.2 Maximal Query Graph

The reduced neighborhood graph H'_t may still be large. It is thus critical to differentiate the importance of the remaining edges. We therefore weight these edges by the following measures:

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

$$\mathsf{ief}(e) = \log \frac{|E(G)|}{\#label(e)} \tag{2}$$

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 the data graph G that share the same label and at least one of the two end nodes of e. Formally,

$$\mathsf{p}(e) = \mid \{e'{=}(u', v') \mid label(e){=}label(e'), u'{=}u \vee v'{=}v\} \mid \tag{3}$$

While $\mathsf{ief}(e)$ captures the global frequencies of edge labels, $\mathsf{p}(e)$ measures their local frequencies—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 edges representing the *employment* relationship between a company and many of its employees and edges for the *board member* relationship between the company and its board members. The latter edges are more significant, because most likely a company has much more employees than board members.

The weight $\mathsf{w}_1(e)$ of an edge e in H'_t is proportional to $\mathsf{ief}(e)$ and inversely proportional to $\mathsf{p}(e)$, given by the following equation:

$$\mathsf{w}_1(e) = \frac{\mathsf{ief}(e)}{\mathsf{p}(e)} \tag{4}$$

We now define maximal query graph based on $\mathsf{w}_1(e)$.

Definition 7 The ***maximal query graph*** MQG_t, given a parameter m, is a weakly connected subgraph of the reduced neighborhood graph H'_t that maximizes total edge weight $\sum_e \mathsf{w}_1(e)$ while satisfying (1) it contains all query entities in t and (2) it has m edges. ■

There are two challenges in finding MQG_t by directly going after the above definition. First, a weakly connected subgraph of H'_t with exactly m edges, for an arbitrarily given m, may not exist. A trivial value of m that guarantees the existence of the corresponding MQG_t is $|E(H'_t)|$, because H'_t itself 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 2.

Theorem 2 The decision version of finding the maximal 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 [19]. Given an undirected connected graph G_1=(V, E) with non-negative weight $w(e)$ for every $e{\in}E$, a subset $V_n{\subset}V$, and a positive integer m, the CSN problem is to find a connected subgraph G'=(V', E') with the smallest total edge weight, where $V_n{\subseteq}V'$ and $|E'|$=m. The polynomial-time reduction from the CSN problem to the MQG problem is by transforming G_1 to G_2, where each edge e is given an arbitrary direction and a new weight $w'(e)$=$W{-}w(e)$, where W=$\sum_{e{\in}E} w(e)$. Let V_n be the query tuple. The maximal query graph MQG_{V_n} found from G_2 provides a CSN in G_1, by ignoring edge direction. This completes the proof. ■

Based on the theoretical analysis, we first design a greedy method (Algorithm 1) to find a plausible sub-optimal graph of edge cardinality *close* to a given value m. The value of m is empirically chosen and is much smaller than $|E(H'_t)|$. Consider edges of H'_t in descending order by their weights $\mathsf{w}_1(e)$. 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 of G_s containing all query entities in t, if it exists. Our method finds the smallest s such that $|E(M_s)|$=m

[3] A weakly connected component of a directed graph is a maximal subgraph where an undirected path exists for every pair of vertices.

Algorithm 1: Discovering the Maximal Query Graph

```
Input: reduced neighborhood graph $H'_t$, query tuple $t$, an integer $r$
Output: maximal 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      $s \leftarrow \min\{i \mid |E(M_i)|= m\}$; //$E(M_i)$ is the weakly connected
       component found from the top-$i$ edges of $G$
9      if $s$ does not exist then
10         $s \leftarrow \max\{i \mid |E(M_i)|< m\}$;
11         if $s$ does not exist then $s \leftarrow \min\{i \mid |E(M_i)|> m\}$;
12     $V(MQG_t) \leftarrow V(MQG_t) \cup V(M_s)$;
13     $E(MQG_t) \leftarrow E(MQG_t) \cup E(M_s)$;
```

(Line 8). If such an M_s does not exist, the method finds the largest s such that $|E(M_s)|<m$ (Line 10). If that still does not exist, it chooses the smallest s such that $|E(M_s)|>m$ (Line 11), whose existence is guaranteed because $|E(H'_t)|>m$. The implementation of the method works by setting s to m initially and increasing s by 1 in each iteration until $|E(M_s)|=m$ (then it terminates) or $|E(M_s)|>m$. If $|E(M_s)|>m$, it then decreases s by 1 in each iteration until $|E(M_s)|<m$ or s=0. In each iteration, 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.

The M_s found by this method might not be a balanced graph, in which each query entity has a fair number of associated edges. Query entities with more neighbors in H'_t will likely have more prominent representation in the resulting M_s. 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 any pair of 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 2-6 of Algorithm 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.

Complexity $\mathsf{ief}(e)$ and $\mathsf{p}(e)$ of an edge e in Equation 4 are query independent and thus precomputed offline. 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.

In constructing MQG_t, the weighting scheme in Equation 4 considers the importance of an edge independent of its 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. By empirical observations, we find it imperative to further 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. Intuitively, edges closer to the query entities convey more meaningful relationships than those farther away. This is captured by *edge depth*:

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

$$d(e) = \min_{v_i \in t} \min_{u,v} \{\mathsf{dist}(u, v_i), \mathsf{dist}(v, v_i)\} \quad (5)$$

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

Finally, we define $\mathsf{w}_2(e)$, the weight of any edge e in MQG_t, by combining $\mathsf{ief}(e)$, $\mathsf{p}(e)$ and $\mathsf{d}(e)$, as follows. Section 4 scores answer graphs based on this measure.

$$\mathsf{w}_2(e) = \frac{\mathsf{ief}(e)}{\mathsf{p}(e) \times \mathsf{d}^2(e)} \quad (6)$$

4. ANSWER SPACE MODELING

Given the maximal query graph MQG_t for a tuple t, we model the space of possible query graphs by a lattice. We further discuss the scoring of answer graphs by how well they match query graphs.

(A) Query Lattice

Definition 8 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 set of query graphs (Definition 4) 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 Hasse diagram, the poset is a directed acyclic graph, in which each node corresponds to a 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.

Definition 9 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 3 (Query Lattice and Minimal Query Tree) Figure 6(a) shows a maximal query graph MQG_t, which contains two query entities in shaded circles and five edges A, B, C, D, and E. Its corresponding query lattice $\mathcal{L}$ is in Figure 6(b). The root node of $\mathcal{L}$, denoted $ABCDE$, represents MQG_t itself. The two nodes at the bottom, denoted A and BC, are the two minimal query trees.

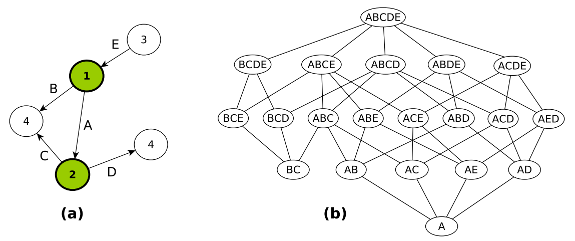


Figure 6: Maximal Query Graph and Query Lattice

Each lattice node is a distinct subgraph of MQG_t. For example, the node ABD represents a query graph with only edges A, B and D. Note that there is no lattice node for BDE, which is not a valid query graph since it is not connected. ■

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 (Section 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 bottom nodes, i.e., the minimal query trees, are generated as follows.

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, which is described in Section 3.2. To generate all minimal query trees, our method enumerates all distinct spanning trees of M_s by the technique in [10] and then trim 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 can be expensive since its time complexity is a function of the number of spanning trees. However, in our experiments on the Freebase dataset, the MQG_t discovered by the approach in Section 3 mostly contains less than 15 edges. Hence, the M_s from the core graph is also empirically small. Specifically, it typically has only 4 edges, for which the cost of enumerating all spanning trees is negligible.

(B) Answer Graph Scoring Function

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

$$\mathsf{score}(Q) = \sum_{e \in E(Q)} \mathsf{w}_2(e)$$

$$\mathsf{score}_Q(A) = \mathsf{score}(Q) + \sum_{\substack{e=(u,v)\in E(Q) \\ e'=(f(u),f(v))\in E(A)}} \mathsf{match}(e,e') \qquad (7)$$

$\mathsf{score}_Q(A)$ sums up the total edge weight of Q ($\mathsf{score}(Q)$) and the extra credits given to identical matching nodes in A and Q ($\mathsf{match}(e,e')$). For instance, the identical matching nodes in Figures 3(a) and 4(a) are USA, San Jose, and California. Note that f is the bijection between $V(Q)$ and $V(A)$ as in Definition 5 and the edge weight $\mathsf{w}_2(e)$ is given by Equation 6. The function $\mathsf{match}(e,e')$ is defined below. It does not award an identical matching node excessively. Instead, only a fraction of $\mathsf{w}_2(e)$ 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 are also identical matching nodes.

$$\mathsf{match}(e,e') = \begin{cases} \frac{\mathsf{w}_2(e)}{|E(u)|} & \text{if } u{=}f(u) \\ \frac{\mathsf{w}_2(e)}{|E(v)|} & \text{if } v{=}f(v) \\ \frac{\mathsf{w}_2(e)}{min(|E(u)|,|E(v)|)} & \text{if } u{=}f(u), v{=}f(v) \\ 0 & \text{otherwise} \end{cases} \qquad (8)$$

5. QUERY PROCESSING

The query processing component of GQBE takes the maximal query graph MQG_t (Section 3) and the query lattice (Section 4) and finds answer graphs matching the query graphs in the lattice. Before we discuss how the lattice is evaluated, we introduce the storage model and query plan for processing one query graph.

5.1 Processing One Query Graph

The abstract data model of entity-relationship 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 proposed in [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 Figure 6(a) corresponds to SELECT A.subj, A.obj FROM A,B,C,D,E WHERE A.subj=B.subj AND A.obj=C.subj AND A.obj=D.subj AND A.subj=E.obj AND B.obj=C.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 $E \bowtie (A \bowtie (D \bowtie (C \bowtie B)))$. With regard to its topmost join operator, the left operand is E's hash table that uses $E.obj$ as the hash key, and the right operand is $(A \bowtie (D \bowtie (C \bowtie B)))$. The hash-join operator iterates through tuples from the probe relation, finds matching tuples from the build relation, and joins them together to produce 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 Algorithm 2, which allows sharing of computation. The algorithm explores the query lattice in a *bottom-up* way. It starts with the minimal query trees, i.e., the bottom-most 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, Algorithm 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 node 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 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). The rest of this section details the algorithm.

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 $\mathcal{LF} \leftarrow$ leaf nodes of $\mathcal{L}$, Terminate $\leftarrow$ false;
2  while not Terminate do
3      $Q_{best} \leftarrow$ node with the highest upper-bound score in $\mathcal{LF}$;
4      $\mathcal{A}_{Q_{best}} \leftarrow$ evaluate $Q_{best}$; (Section 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 [Th. 3] then Terminate $\leftarrow$ true ;
```

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

To choose Q_{best} from $\mathcal{LF}$, the algorithm exploits two important properties, dictated by the way the query lattice is structured.

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}. ■

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. Therefore, if evaluating Q_1 returns answer graph A_1 and corresponding t_{A_1}, the same answer tuple may be found in the ancestor nodes of Q_1.

Property 2 If $Q_1 \prec Q_2$, $A_1 \in \mathcal{A}_{Q_1}$, $A_2 \in \mathcal{A}_{Q_2}$ and $A_1 \prec A_2$ (thus t_{A_1}=t_{A_2}), then $\mathsf{score}_{Q_1}(A_1) < \mathsf{score}_{Q_2}(A_2)$. ■

Property 2 further says that, if an answer graph A_1 to a lattice node Q_1 is grown into an answer graph A_2 to an ancestor node Q_2, A_2 has a higher score. This can be proved by referring to the scoring function in Equation 7, since $A_1 \prec A_2$ and $Q_1 \prec Q_2$. Since A_1 and A_2 are projected to the same answer tuple, the answer tuple always gets the better score from the ancestor's answer graph A_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 an answer tuple is obtained from Q, it has the potential to be an answer to an ancestor node, which will give it a higher score. Hence, its upper-bound score depends on Q's ancestors that do not have any parents. Such nodes form the *upper-frontier* ($\mathcal{UF}$).
$\mathcal{UF} = \{Q|\ Q$ is not pruned, $\nexists Q'$ s.t. $Q \prec Q'$ and Q' is not pruned$\}$.

We further define the concept of *upper boundary* based on $\mathcal{UF}$:

Definition 10 The ***upper boundary*** of a node Q in $\mathcal{LF}$ is the set of nodes Q' such that Q' subsumes or equals to Q and Q' is in $\mathcal{UF}$.

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

The upper-bound score of Q is given by the best possible score Q's answer tuples can attain if they are also answers to nodes in $\mathcal{UB}(Q)$. Deriving the upper-bound score based on Equation 7 leads to loose upper-bound. In Equation 7, the score of an answer graph is the summation of two components. The first component only depends on the query graph itself, while the second component captures the matching nodes in answer and query graphs. Without evaluating a query graph, we can only assume a perfect match for $\mathsf{match}(e, e')$, which is clearly an over-optimism. Under such a loose upper-bound, an early termination of lattice evaluation can be hard. Therefore we define the upper-bound score by only using the aforementioned first component.

Definition 11 The ***upper-bound score*** of a node Q is the maximum score of any query graph in its upper boundary. Formally,

$$U(Q) = \max_{Q' \in \mathcal{UB}(Q)} \mathsf{score}(Q') \qquad (9)$$

(B) 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 and thus are pruned by Algorithm 2, based on Property 3 which follows directly from Property 1. The pruning of null nodes changes the upper boundaries and thus upper-bound scores for some lower-frontier nodes. The algorithm recomputes these variables by an efficient method discussed in Section 5.3.

Property 3 (Upward Closure) If $\mathcal{A}_{Q_1}$=$\emptyset$, then $\forall Q_2 \succ Q_1$, $\mathcal{A}_{Q_2}$=$\emptyset$. ■

(C) Termination After Q_{best} is evaluated, its answer tuples are $\{t_A | A \in \mathcal{A}_{Q_{best}}\}$. For an answer tuple t_A projected from answer graph A, the score assigned by Q_{best} to t_A is $\mathsf{score}_{Q_{best}}(A)$, which is given by Equation 7. If t_A was also projected from already evaluated nodes, it has a current score. By Definition 6, the final score of t_A will be from its best answer graph. Hence, if $\mathsf{score}_{Q_{best}}(A)$ 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 the following Theorem 3.

Theorem 3 Terminating the lattice evaluation at the aforementioned condition guarantees that the current top-k answer tuples have scores higher than $\mathsf{score}(Q)$ for any unevaluated query graph Q. ■

Note that, however, the upper-bound score in Definition 9 cannot guarantee to bound the best possible score an answer tuple to Q can get. Instead, in contrast to using the full Equation 7, which is over-optimistic, this definition is more realistic. Hence, when the algorithm terminates, it does not guarantee the top-k answers returned are true top-k answers with regard to Equation 7. Our approach in alleviating it is to first find the top-k' answers (k'>k) based on Definition 9. The algorithm then computes the scores

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 $\mathcal{LF}$
1  foreach $Q \in \mathcal{LF}$ do
2      $\mathcal{NB} \leftarrow \phi$; // set of new upper boundary candidates of $Q$.
3      foreach $Q' \in \mathcal{UB}(Q) \cap \mathcal{UB}(Q_{best})$ do
4          $\mathcal{UB}(Q) \leftarrow \mathcal{UB}(Q) \setminus \{Q'\}$;
5          $\mathcal{UF} \leftarrow \mathcal{UF} \setminus \{Q'\}$;
6          $V(Q'') \leftarrow V(Q')$;
7          foreach $e \in E(Q_{best}) \setminus E(Q)$ do
8              $E(Q'') \leftarrow E(Q') \setminus \{e\}$;
9              find $Q_{sub}$, the weakly-connected component of $Q''$,
               containing all query entities;
10             $\mathcal{NB} \leftarrow \mathcal{NB} \cup \{Q_{sub}\}$;
11     foreach $Q_{sub} \in \mathcal{NB}$ do
12         if $Q_{sub} \not\preceq$ (any node in $\mathcal{UF}$ or $\mathcal{NB}$) then
13             $\mathcal{UB}(Q) \leftarrow \mathcal{UB}(Q) \cup \{Q_{sub}\}, \mathcal{UF} \leftarrow \mathcal{UF} \cup \{Q_{sub}\}$;
14     recompute $U(Q)$ using Eq. 9;
```

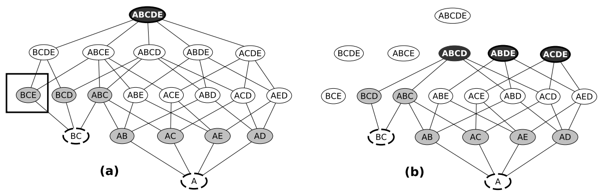


Figure 7: Recomputing Upper Boundary of Dirty Node AE

of these top-k' answers by the full scoring function in Equation 7 to re-rank them and finally return 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.

(D) Complexity 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.

5.3 Recomputation of Upper-frontier and Upper-bound Scores

When Q_{best} is evaluated to be a null node, the algorithm prunes Q_{best} and its ancestors, which changes the upper-frontier $\mathcal{UF}$. It is worth noting here 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 of its upper boundary nodes among the pruned ones, the change of $\mathcal{UF}$ will lead to changes in their upper boundaries and sometimes, their upper-bound scores too. Such nodes are referred to as *dirty nodes*. The rest of this section presents an efficient method (Alogirthm 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 properties of a potential new upper boundary node for Q are that it should (1) be a supergraph of Q, (2) be a subgraph of Q' and (3) not be 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' excepting exactly one of the aforementioned q edges (Line 8 in Alogirthm 3). For each Q'', we find Q_{sub} which is the weakly connected component 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 $\mathcal{L}$. ■

Lemma 2 $Q \preceq Q_{sub}$. ■

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 the upper-bound score of Q (Line 14). Theorem 4 justifies the correctness of the above procedure.

Theorem 4 The aforementioned procedure guarantees to identify all new upper boundary nodes for every dirty node Q. ■

Example 4 (Recomputing Upper Boundary) Consider the lattice in Figure 7(a) where the bold-dashed nodes belong to the evaluated, the lightly shaded nodes belong to $\mathcal{LF}$ and the darkly shaded node belongs to $\mathcal{UF}$. If node *BCE* is the currently evaluated null node Q_{best} and *ABCDE* is Q', let *AE* be the dirty node Q whose upper boundary is to be recomputed. The edges in Q_{best} that are not present in Q are *B* and *C*. A new upper boundary node Q'' contains all edges in Q' excepting exactly either *B* or *C*. This leads to two new upper boundary nodes, *ABDE* and *ACDE*, by removing *C* and *B* from *ABCDE*, respectively. Since *ABDE* and *ACDE* 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 $\mathcal{UF}$. Figure 7(b) shows the modified lattice where the nodes not connected to the lattice are the pruned nodes. *ABCD* is another node in $\mathcal{UF}$ that is discovered using dirty nodes such as *AB* and *BCD*. ■

Complexity 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 $O(|\mathcal{L}_n| \times D_p \times q \times (3 \times |E(MQG_t)|)$. 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.

6. MULTI-TUPLE QUERIES

The query graph discovery component of GQBE essentially 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 Section 7 show that a single-tuple query obtains excellent accuracy in many cases, the results also exhibit that allowing multiple query tuples often helps improve query answer accuracy. The idea is that important relationships commonly associated with

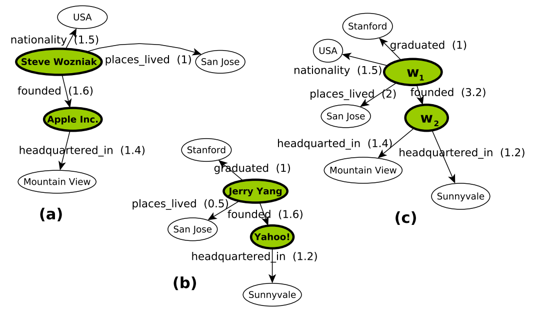


Figure 8: Merging Maximal Query Graphs

multiple query tuples express the user intent more precisely. For instance, suppose a user has provided two query tuples together—⟨Jerry Yang, Yahoo!⟩ and ⟨Steve Wozniak, Apple Inc.⟩. The query entities in both tuples share common properties such as *places_lived* in San Jose and *headquartered_in* a city in California, as shown in Figure 1. This might indicate that the user is interested in finding people from San Jose who founded technology companies in California.

Given a set of n-tuples T, GQBE aims at finding top-k answer tuples similar to T collectively. To accomplish this, one approach is to discover and evaluate the maximal 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 algorithm in Section 5. 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 $label(e)$=$label(e')$.

The merged MQG is denoted MQG_T. It 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. Essentially, the multiple MQGs are overlaid on each other. Formally,

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

The edge cardinality of MQG_T might be larger than the target size r. Thus Algorithm 1 proposed in Section 3 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 maximal weight among all such M'_{t_i}.

Example 5 (Merging Maximal Query Graphs) Let Figures 8(a) and (b) be the M_{t_i} for query tuples ⟨Steve Wozniak, Apple Inc.⟩ and ⟨Jerry Yang, Yahoo!⟩, respectively. Figure 8(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 *graduated*, which appear in only one M_{t_i}, are also present in MQG_T. The number next to each edge is its weight. ■

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.

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

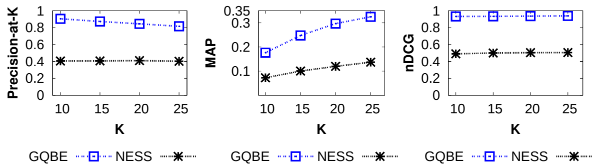
This section presents the results of our experiments on the accuracy and efficiency of GQBE. The experiment setup is as follows.

Datasets The experiments were conducted over two large real-world entity-relationship graphs, the Freebase [4] and DBpedia [2] datasets. We preprocessed the graphs so that the kept nodes are all named entities (e.g., Stanford University) and abstract concepts (e.g., Jewish people). 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 and NESS [18]. NESS is a graph querying framework that finds matches of query graphs with unlabeled nodes which correspond to query entity nodes in MQG. Note that 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. Similar to the best-first method (Section 5),

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

Figure 9: Accuracy of GQBE and NESS on Freebase Queries

Baseline explores a query lattice in a bottom-up manner and prunes ancestors of null nodes. However, differently, it evaluates the lattice by breadth-first traversal instead of in the order of upper-bound scores. There is no early-termination by top-k scores, as Baseline terminates when every node is either evaluated or pruned. We implemented GQBE and Baseline in Java and we obtained the source code of NESS from the authors. The experiments were conducted on a double quad-core 24 GB Memory 2.0 GHz Xeon server.

Queries and Ground Truth Two groups of queries are used on the two datasets, respectively. The Freebase queries F_1– F_{20} are based on Freebase tables such as http://www.freebase.com/view/computer/programming_language_designer?instances, except F_1 and F_6 which are from Wikipedia tables such as http://en.wikipedia.org/wiki/List_of_English_football_club_owners. The DBpedia queries D_1– D_8 are based on DBpedia tables such as the values for property 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} and F_{19}, due to space limitations.

(A) Accuracy Based on Ground Truth

We measured the accuracy of GQBE and NESS by comparing their query results with the ground truth. The accuracy for a set of queries is the average of accuracy on individual queries, which is captured by three different widely-used measures [21], 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 given by 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): The cumulative gain of the top-k results is $\text{DCG}_k = rel_1 + \sum_{i=2}^{k} \frac{rel_i}{\log_2(i)}$. It penalizes the results if a ground truth result is ranked low. DCG_k is normalized by IDCG_k, the cumulative gain for an ideal ranking of the top-k results. Thus $\text{nDCG}_k = \frac{\text{DCG}_k}{\text{IDCG}_k}$.

Figures 9 shows these measures for different values of k on the Freebase queries. GQBE has high accuracy. For instance, its P@25 is over 0.8. The absolute value of MAP is not high, merely because

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

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 between GQBE and Amazon MTurk Workers, k=30

Figure 9(b) only shows the MAP for at most top-25 results, while the ground truth size (i.e., the denominator in calculating MAP) for many queries is much larger. Moreover, GQBE outperforms NESS substantially, as its accuracy in all three measures is almost always twice as better. This is because GQBE gives priority to query entities and important edges in MQG, while NESS gives equal importance to all nodes and edges except the pivot. Furthermore, the way NESS handles edge labels does not explicitly require answer entities to be connected by the same paths between query entities.

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 Study

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 $(\text{E}(XY) - \text{E}(X)\text{E}(Y))/(\sqrt{\text{E}(X^2) - (\text{E}(X))^2}\sqrt{\text{E}(Y^2) - (\text{E}(Y))^2})$. 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 [7]. 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.

(C) Accuracy on Multi-tuple Queries

We investigated the effectiveness of the multi-tuple querying approach (Section 6). In aforementioned single-tuple query experiment (A), GQBE attained perfect P@25 for 13 of the 20 Freebase queries. We thus focused on the remaining 7 queries. For each query, Tuple1 refers to the query tuple in Table 1, while Tuple2 and Tuple3 are two tuples from its ground truth. Table 5 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

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 6: Time for Discovering and Merging MQGs (secs.)

MQGs, which are denoted Combined(1,2) and Combined(1,2,3), respectively. F_4 attained perfect precision after Tuple2 was included. Therefore, Tuple3 was not used for F_4. The results show that, in most cases, Combined(1,2) had better accuracy than individual tuples and Combined(1,2,3) further improved accuracy.

(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. Figure 10 compares the three methods' query processing time, in logarithmic scale. For each edge cardinality of MQG, the figure shows the average time on queries with the same edge cardinality. 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 16 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 a 9-edge MQG and a 10-edge MQG, taking 51 and 552 seconds, 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 Figure 11 which shows the numbers of lattice nodes evaluated, under varying edge cardinality of MQG. GQBE evaluated considerably less nodes in most cases and about 6 times less on queries with 12 edges in MQG. (The value for Baseline is 380, which is off the chart and listed explicitly.)

MQG discovery precedes the query processing step and is shared by all three methods. Column MQG_1 in Table 6 lists the time spent on discovering MQG for each Freebase query. This time component varies across individual queries, depending on the sizes of query tuples' neighborhood graphs. Compared to the values shown in Figures 10, the time taken to discover an MQG in average is comparable to the time spent by GQBE in evaluating it.

Figure 12 shows GQBE's query processing time, in logarithmic scale, on the merged MQGs of 2-tuple queries in Table 5, denoted by Combined(1,2). It also shows the total time for evaluating the two tuples' MQGs individually, denoted Tuple1+Tuple2. The time for Combined(1,2) is 1-3 orders of magnitude less in 8 out of 20 queries and is significantly less in 5 other queries. 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 query tuples, leading to improved answer accuracy as shown in Table 5. Table 6 further shows the time taken to discover individual MQGs (MQG_1 and MQG_2) for the two tuples, along with the time for merging them. The latter is negligible compared to the former.

8. RELATED WORK

Our work is the first to query entity-relationship graphs by example entity tuples. 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 a query pattern. Such is not what we refer to as query-by-example, because underlyingly the query graphs and patterns are formed by using structured query languages or other query mechanisms. In fact, substantial progress has been made on more user-friendly query mechanisms that do not require explicit query graphs or help users construct query graphs. Such mechanisms include keyword search (e.g., [16]), keyword-based query formulation [31], natural language questions [30], interactive and form-based query formulation [8], and visual interface for query graph construction [6, 14].

PathSim [25] 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 [33], given a query graph as input, the system finds structurally isomorphic answer graphs with semantically similar entity nodes. In both works, a user should know the network schema to specify a meta-path or a query graph. In contrast, the query-by-example approach in GQBE only requires a user to provide an entity tuple, without knowing the underlying schema.

The goal of *set expansion* is to grow a set of objects starting from seed objects. Example systems include [29], [11], 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 in web pages such as HTML tables and lists. Furthermore, all these systems except Google Squared and [11] take a set of individual entities as input. GQBE is more general in that each query tuple contains multiple entities.

Several works [27, 17, 9] 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) The graphs in [27] contain nodes of the same type and edges representing the same relationship, e.g., social networks capturing friendship between people. The graphs in GQBE and others [17, 9] have many different types of entities and relationships. (2) The paths discovered by their techniques only connect the input nodes. REX [9] has the further limitation of allowing only two input entities. Differently the maximal query graph in GQBE includes edges incident on individual query tuple entities. (3) 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, such as G-Ray [28], TALE [26], and NESS [18]. The query processing component of GQBE is different from them on several aspects. First, GQBE requires to match only edge labels, but 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 graph querying methods (e.g., NESS [18]) allow unlabeled query nodes. Second, in GQBE, the top-*k* query algorithm is centered around query tuple entities. More specifically, the weighting function gives more importance to the 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.

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**	N/A	N/A	N/A	N/A	N/A	N/A
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**

Table 5: Accuracy of GQBE on Multi-tuple Queries, k=25

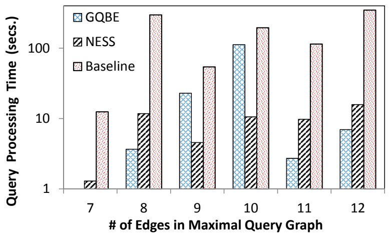


Figure 10: Query Processing Time

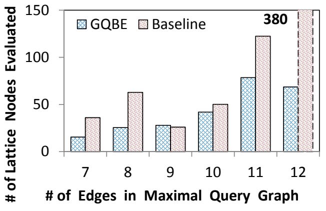


Figure 11: Number of Lattice Nodes Evaluated

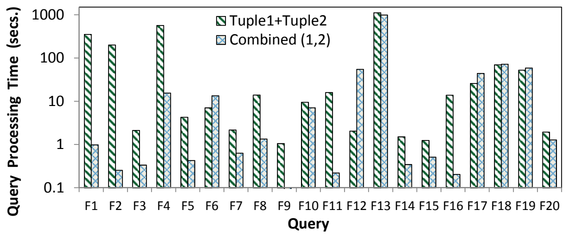


Figure 12: Query Processing Time of 2-tuple Queries

9. CONCLUSION

We introduce GQBE, a system that queries entity-relationship 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. To the best of our knowledge, there has been no such proposal in the past. 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 system NESS on both accuracy and efficiency.

10. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.
[2] 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.
[3] A. Birkl and G. Yona. Biozon: a hub of heterogeneous biological data. *Nucleic Acids Research*, 34, 2006.
[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] D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad. GRAPHITE: A visual query system for large graphs. In *ICDM Workshops*, pages 963–966, 2008.
[7] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.
[8] E. Demidova, X. Zhou, and W. Nejdl. FreeQ: an interactive query interface for Freebase. In *WWW*, demo paper, 2012.
[9] L. Fang, A. D. Sarma, C. Yu, and P. Bohannon. REX: explaining relationships between entity pairs. In *PVLDB*, pages 241–252, 2011.
[10] H. N. Gabow and E. W. Myers. Finding all spanning trees of directed and undirected graphs. *SIAM J. Comput.*, 7(3):280–287, 1978.
[11] R. Gupta and S. Sarawagi. Answering table augmentation queries from unstructured lists on the web. *Proc. VLDB Endow.*
[12] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, pages 13–24, 2007.
[13] M. Jarrar and M. D. Dikaiakos. A query formulation language for the data web. *TKDE*, 24:783–798, 2012.
[14] C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi. GBLENDER: Towards blending visual query formulation and query processing in graph databases. In *SIGMOD*, pages 111–122, 2010.
[15] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: a scalable and general graph management system. In *KDD*, 2011.
[16] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, pages 681–692, 2011.
[17] G. Kasneci, S. Elbassuoni, and G. Weikum. MING: mining informative entity relationship subgraphs. In *CIKM*, 2009.
[18] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, pages 901–912, 2011.
[19] 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.
[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
[21] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, NY, USA, 2008.
[22] J. Pound, I. F. Ilyas, and G. E. Weddell. Expressive and flexible access to web-extracted data: a keyword-based structured query language. In *SIGMOD*, pages 423–434, 2010.
[23] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.
[24] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge unifying WordNet and Wikipedia. In *WWW*, pages 697–706, 2007.
[25] 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.
[26] Y. Tian and J. M. Patel. TALE: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.
[27] H. Tong and C. Faloutsos. Center-piece subgraphs: Problem definition and fast solutions. In *KDD*, pages 404–413, 2006.
[28] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. *KDD*, 2007.
[29] R. C. Wang and W. W. Cohen. Language-independent set expansion of named entities using the web. In *ICDM*, pages 342–350, 2007.
[30] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum. Deep answers for naturally asked questions on the web of data. In *WWW*, demo paper, pages 445–449, 2012.
[31] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword query reformulation on structured data. *ICDE*, pages 953–964, 2012.
[32] X. Yin and S. Shah. Building taxonomy of web search intents for name entity queries. In *WWW*, pages 1001–1010, 2010.
[33] X. Yu, Y. Sun, P. Zhao, and J. Han. Query-driven discovery of semantically similar substructures in heterogeneous networks. In *KDD*, pages 1500–1503, 2012.
[34] M. M. Zloof. Query by example. In *AFIPS*, 1975.