TOWARDS BETTER USABILITY OF QUERY SYSTEMS FOR MASSIVE ULTRA-HETEROGENEOUS GRAPHS: NOVEL APPROACHES OF QUERY FORMULATION AND QUERY SPECIFICATION

by

NANDISH JAYARAM

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2016

To my

Mother: for all her sacrifices, and always wishing I'd be a "doctor".

Father: for being my greatest teacher.

ACKNOWLEDGEMENTS

I would like to acknowledge and express my deepest appreciation to my supervising professor and committee chair, Dr. Chengkai Li. The research in this dissertation would not have been possible without his constant support and mentoring. I have learned tremendous amount from him during this journey, starting from problem formulation, designing rigorous solutions, to presenting them to an audience. He has always believed in working on real-world problems and building practical systems useful to people, and that has left a huge positive impact on me. His "treat words like gold while writing a research paper" advice is certainly a tip for life. I will forever be indebted to him for always standing by me, and I cannot thank him enough for his constant support and encouraging words when I needed them the most. He has been instrumental in nurturing my research acumen.

I would like to thank my co-supervising professor, Dr. Ramez Elmasri. He has been extremely kind to me and has encouraged me tremendously. I have been his teaching assistant for several semesters, and that has been a great learning experience for me. I would also like to thank my other committee members: Dr. Christoph Csallner, Dr. Gautam Das, and Dr. Xifeng Yan. Dr. Csallner's candid feedback and positive criticisms during important PhD milestones have been extremely helpful in shaping this dissertation. I have always been amazed by Dr. Das' ability to explain complex ideas with great ease. His courses on algorithms laid an important foundation that has helped me greatly in my research projects. Dr. Yan of UCSB has been an invaluable mentor and collaborator. He has always believed in my research direction, and his advice on presenting the core ideas of a problem in a research paper has helped me write better. I thank these five incredible mentors for teaching the most important lesson by example, to exhibit kindness and humility.

I would also like to extend my gratitude to the department of Computer Science and Engineering at the University of Texas at Arlington, and Dr. Chengkai Li for providing me with financial supports during my entire graduate studies. Special thanks to Mahesh, Sidharth and Rohit for helping me build demos of the systems that form an integral part of this dissertation. I also thank my lab mates Afroza, Fatma, Gensheng, Naffi, Ning and Sona. Many thanks to my friends Saravanan, Mahesh, Jijo, Mayank, Praveen, Manimala, Mahashweta, Ramesh, Azade and Rasool for making my Ph.D. memorable. I will especially cherish having long funny conversations with Mahesh and Jijo. I have always come out knowing more after my discussions with Saravanan, and his incredible thirst for knowledge is something I wish for.

I would like to convey my heartfelt gratitude to my parents and sister. I still remember my mother carrying me around for a year when I had broken my leg as a 9-year old kid. Her love and sacrifices have made me who I am today. My father is one of the smartest, ethical and hard working people I have known. His determination and ability to develop expertise in diverse areas such as electronics, structural chemistry and aerospace never ceases to amaze me! I thank my parents for instilling the importance of knowledge in me. Finally, I would like to thank my wife Kruthi. She is my best friend and I am extremely lucky to have her in my life. Her patience in dealing with me is truly a gift. She has made me a better and a happier person, and her joyfulness is contagious! I thank her parents for raising such a loving person. This journey would be incomplete without her.

February 12, 2016

ABSTRACT

TOWARDS BETTER USABILITY OF QUERY SYSTEMS FOR MASSIVE ULTRA-HETEROGENEOUS GRAPHS: NOVEL APPROACHES OF QUERY FORMULATION AND QUERY SPECIFICATION

NANDISH JAYARAM, Ph.D.

The University of Texas at Arlington, 2016

Supervising Professor: Chengkai Li, Ramez Elmasri

There is a pressing need to tackle the usability challenges in querying massive, ultra-heterogeneous entity graphs which use thousands of node and edge types in recording millions to billions of entities (persons, products, organizations) and their relationships. Widely known instances of such graphs include Freebase, DBpedia and YAGO. Applications in a variety of domains are tapping into such graphs for richer semantics and better intelligence. Both data workers and application developers are often overwhelmed by the daunting task of understanding and querying these data, due to their sheer size and complexity. To retrieve data from graph databases, the norm is to use structured query languages such as SQL, SPARQL, and those alike. However, writing structured queries requires extensive experience in query language, data model and the datasets themselves. In this dissertation, as an initial step toward improving the usability of query systems for large graphs, we present two novel and first-of-its-kind systems: Orion and GQBE.

The database community has long recognized the importance of graphical query interface to the usability of data management systems. Yet, relatively little has been done.

Existing visual query builders allow users to build queries by drawing query graphs, but do not offer suggestions to users regarding what nodes and edges to include. At every step of query formulation, a user would be inundated with possibly hundreds of or even more options. We present Orion, a visual query interface that iteratively assists users in query graph construction by making suggestions using machine learning methods. In its active mode, Orion suggests top-k edges to be added to a query graph, without being triggered by any user action. In its passive mode, the user adds a new edge manually, and Orion suggests a ranked list of labels for the edge. Orion's edge ranking algorithm, Random Decision Paths (RDP), makes use of a query log to rank candidate edges by how likely they are predicted to match users' query intent. Extensive user studies using Freebase demonstrated that Orion users have a 70% success rate in constructing complex query graphs, a significant improvement over the 58% success rate by users of a baseline system that resembles existing visual query builders. Furthermore, using active mode only, the RDP algorithm was compared with several methods adapting other machine learning algorithms such as random forests and naive Bayes classifier, as well as recommendation systems based on singular value decomposition and class association rules. On average, RDP required only 40 suggestions to correctly reach a target query graph while other methods required 1.5-4 times as many suggestions.

We also propose to query large graphs by example entity tuples, without requiring users to form complex graph queries. Our system, GQBE (Graph Query By Example), provides a complementary approach to the existing keyword-based methods, facilitating user-friendly graph querying. GQBE 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. GQBE also lets users provide multiple example tuples as input, and efficiently uses them to better capture the user's query intent. User studies with Freebase demonstrated that GQBE's

ranked answer tuple list has a strong positive correlation with the users' ranking preferences. Other extensive experiments showed that GQBE has a significantly better accuracy than other state-of-the-art systems. GQBE was also faster than NESS (one of the compared systems) for 17 of the 20 queries used in the experiments, and was 3 times faster for 10 of them.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

There is an unprecedented proliferation of large ultra-heterogeneous graph data in our society today. Graphs are increasingly used to represent complex relationships in schema-less data such as DBpedia [1], YAGO [2], Freebase [3] and Probase [4]. Given a large ultra-heterogeneous graph that represents such ubiquitous linked data, being able to use it by easily querying it is a fundamental problem and a critical task for many graph applications. Examples of such large graphs include *knowledge graphs*, that record millions of entities (e.g., persons, products, organizations) and their relationships. Figure 1.1 is an excerpt of a heterogeneous 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.

Users and developers are tapping into such large 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 these 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

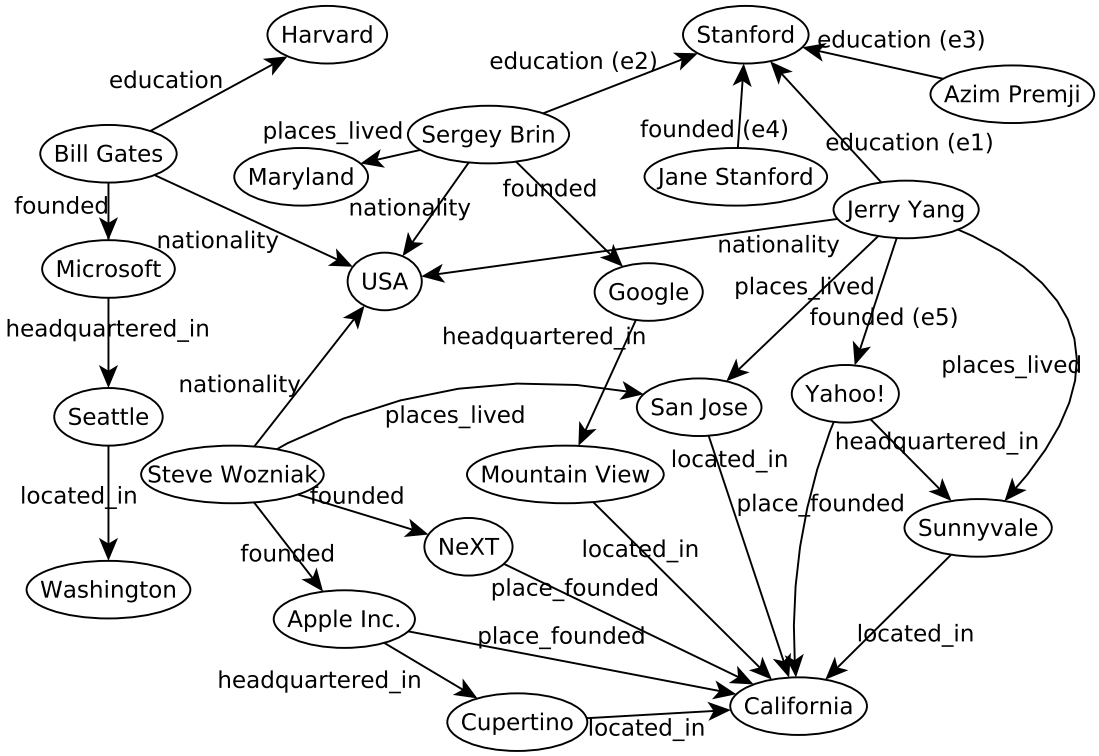


Figure 1.1: An Excerpt of a Heterogeneous Graph

model, and a good understanding of particular datasets [5]. If querying "simple" tables is difficult, aren't complex graphs harder to query?

Example 1 (Expressing Query Intent) *Consider the scenario where a Silicon Valley analyst is interested in finding various software companies head-quartered in the Silicon Valley, that were founded by American citizens. Figure 1.2 represents a query graph to capture the query intent, while the SPARQL query to capture the same query intent is:* `SELECT ?company ?founder WHERE { :?founder dbo:founded :?company. :?founder dbo:nationality db:USA. :?company dbprop:headquartered_in db:Silicon Valley.}`*, where dbo, dbprop and db are various namespaces used. Clearly, specifying a query even for such a simple query intent requires users to understand the schema and the data well.*

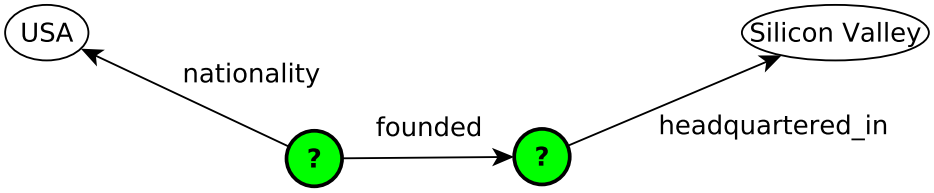


Figure 1.2: Query Graph for Example 1

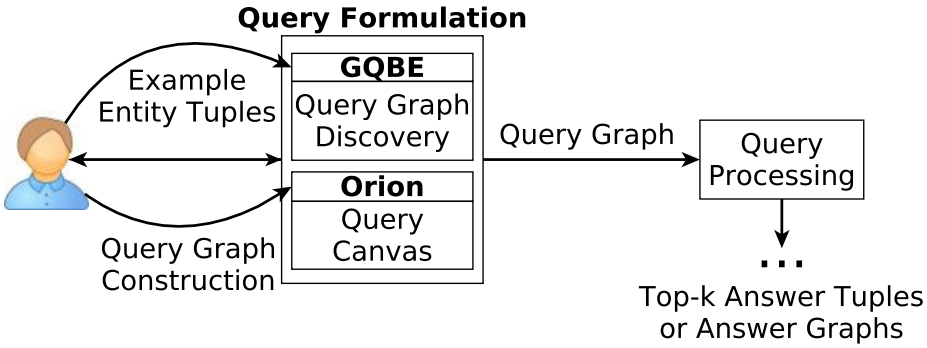


Figure 1.3: Framework for Querying Heterogeneous Graphs

1.2 Contributions

Motivated by the aforementioned usability challenges, in this dissertation we focus on addressing the problem of improving the query formulation capability of query systems for large heterogeneous graphs. Figure 1.3 shows the architecture of the proposed framework. More specifically, we present two different techniques: 1) Orion, a system that helps schema-agnostic users formulate query graphs specifying their exact query intent. Orion helps users in the query formulation process by automatically making suggestions that are ranked by how likely they are predicted to match the user's query intent. The *query canvas* component of Orion shown in Figure 1.3, provides an interactive interface for users to formulate their query graph in, and 2) GQBE (**G**raph **Q**uery **B**y **E**xample), a system that supports a new querying paradigm that queries graphs by example entity tuples, instead of query graphs. GQBE lets schema-agnostic users provide example tuples

as input to obtain similar answer tuples as output. The *query graph discovery* component shown in Figure 1.3 automatically discovers a hidden query graph that tries to capture the query intent behind the example query tuples. Demonstration systems of Orion and GQBE that help users query the real-world Freebase data graph can be found at `http://idir.uta.edu/orion` and `http://idir.uta.edu/gqbe` respectively. Demonstration videos of Orion and GQBE can be found at `https://www.youtube.com/watch?v=80iU5EFTVAk` and `https://www.youtube.com/watch?v=-uja23CgOrA` respectively.

Orion helps users *easily* formulate *exact* query graphs. Orion provides a visual interface that enables users to easily construct query graph components. To help schema-agnostic users specify their exact query intent, Orion operates in *active* and *passive* modes. By default Orion operates in active mode. Based on the partially constructed query graph, the system automatically suggests top-k new edges that may be relevant to the user's query intent, without being triggered by any user actions. The passive mode is triggered when the user adds new nodes or edges to the partial query graph. For a newly added edge, the suggested edge labels are ranked based on the likelihood of their relevance to the user's query intent. The graph in Figure 1.2 can be constructed iteratively with the help of suggestions made by Orion. To the best of our knowledge, Orion is the first visual query formulation system that makes ranked suggestions to help users construct exact query graphs.

GQBE [6, 7] is among the first to query ultra-heterogeneous graphs by example entity tuples. Given a data graph and one or more example query tuples consisting of entities, GQBE finds similar answer tuples. Suppose the Silicon Valley analyst in Example 1 knows an example query tuple such as $\langle$ Jerry Yang, Yahoo! $\rangle$ that satisfies her query intent. The answer tuples can be $\langle$ Sergey Brin, Google$\rangle$ and $\langle$ Mark Zuckerberg, Facebook $\rangle$, which are company-founder pairs. The user need not specify *how* various entities in the example tuple are related.

Instead, the system discovers a query graph that tries to capture relationships that may be relevant to the query intent.

Once a query graph is formed, the *query processing* component shown in Figure 1.3 evaluates the query graph to find the top-k approximately matching answer graphs. The answer tuples for GQBE are projected from the answer graphs.

1.3 Outline

The overall contribution of this dissertation is to improve the usability of query systems for large ultra-heterogeneous graphs. The rest of the dissertation elucidates our approaches towards achieving this goal, and is structured as follows:

- In Chapter 2, we present Orion, an interactive visual query interface that helps schema-agnostic users construct query graphs, by automatically suggesting new edges to add to the query graph.
- In Chapter 3, we present GQBE, a system that lets users query large graphs by example entitiy tuples. Given an example input tuple of what a user is looking for, the system finds an underlying query graph to capture the user's query intent and finds similar answer tuples.
- In Chapter 4, we discuss the system design and implementation details of Orion and GQBE. We also discuss some of the lessons learned by us while creating these systems capable of dealing with large real-world graphs.
- In Chapter 5, we present a literature survey relevant to this dissertation.
- In Chapter 6, we present an overview of some future directions to continue improving the usability of query systems for massive ultra-heterogeneous graphs, and finally conclude the contributions made in this dissertation.

CHAPTER 2

AUTO-SUGGESTION BASED VISUAL INTERFACE FOR INTERACTIVE QUERY CONSTRUCTION

2.1 Introduction

The database community has long recognized the importance of graphical query interfaces to the usability of data management systems [8]. Yet, relatively less has been done and there remains a pressing need for investigation in this area [5, 9]. Nevertheless, a few important ideas (e.g., Query-By-Example [10]) and systems (e.g., Microsoft SQL Query Builder) have been developed for querying relational databases [11], web services [12] and XML [13, 14].

For querying graph data, existing systems [15, 16, 17, 18, 19, 20] allow users to build queries by visually drawing nodes and edges of query graphs, which can then be translated into underlying representations such as SPARQL and SQL queries. While focusing on blending query processing with query formulation [16, 17, 18, 19, 20], existing visual query builders do not offer suggestions to users regarding what nodes/edges to include into query graphs. At every step of visual query formulation, after adding a new node or a new edge into the query graph, a user would need to choose from a list of candidate *labels*—names and types for a node or types for an edge. The user, when knowing what label to use, can search the list of labels by keywords or sift through alphabetically sorted options using binary search. But, oftentimes the user does not know the label due to lack of knowledge of the data and the schema. In such a scenario, the user may need to sequentially comb the option list. Furthermore, the user may not have a clear label in mind due to her vague query intent.

The lack of query suggestion presents a substantial usability challenge when the graph data require a long list of options, i.e., many different types and instances of nodes and edges. The aforementioned systems [15, 16, 17, 18, 19, 20] were all deployed on relatively small graphs. The crisis is exacerbated by the proliferation of *ultra-heterogeneous graphs* which have thousands of node/edge types and millions of node/edge instances. Widely-known ultra-heterogeneous graphs include Freebase [3], DBpedia [1], YAGO [2], Probase [4], and the various RDF datasets in the "linked open data" [1]. Users would be better served, if graph query builders provided suggestions during query formulation. In fact, query suggestion has been identified as an important feature-to-have among the desiderata of next-generation visual query interfaces [21].

This chapter presents Orion, a visual query builder that provides suggestions, iteratively, to assist users formulate queries on ultra-heterogeneous graphs. Orion's graphical user interface allows users to construct query graphs by drawing nodes and edges onto a canvas using simple mouse actions. To allow schema-agnostic users to specify their exact query intent, Orion suggests candidate edge types by ranking them on how likely they will be of interest to the user, according to their relevance to the existing edges in the partially constructed query graph. The relevance is based on the correlation of edge occurrences exhibited in a query log. To the best of our knowledge, Orion is the first visual query formulation system that automatically makes ranked suggestions to help users construct query graphs. The demonstration proposal for an early prototype of Orion [22] was based on a subset of the ideas in this chapter.

Orion supports both an *active* and a *passive* operation mode. (1) If the canvas contains a partially constructed query graph, Orion operates in the active mode by default. The system automatically recommends top-k new edges that may be relevant to the user's query

[1] Linking open data. `http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`.

intent, without being triggered by any user actions. Figure 2.2(a) shows the snapshot of a partially constructed query graph, with nodes and edges suggested in the active mode. The white nodes and the edges incident on them are newly suggested. The user can select some of the suggested edges by clicking on them, and a mouse click on the canvas adds the selected edges to the partial query graph, and ignores the unselected edges. (2) The passive mode is triggered when the user adds new nodes or edges to the partial query graph using simple mouse actions. For a newly added node, labels are suggested for its type, the domain of its type, and its name if the node is to be matched with a specific entity. The suggested labels are displayed in a pop-up box, as shown in Figure 2.2(b), where type PERSON is chosen as the label for the node. For a newly added edge, the suggested edge types are ranked based on their relevance to the user's query intent. Figure 2.2(c) shows the ranked suggestions for the newly added edge between the two nodes of types PERSON and FILM, displayed in a pop-up box.

The query construction process of a user can be summarized as a query session, consisting of positive and negative edges that correspond to edge suggestions accepted and ignored by the user, respectively. At every step of the iterative process, based on the partially constructed query graph so far and the corresponding query session, Orion's edge ranking algorithm—Random Decision Paths (RDP)—ranks candidate edges using a query log of past query sessions. RDP ranks the candidate edges by how likely they will be of interest to the user, according to their correlation with the current query session's edges. RDP constructs multiple decision paths using different random subsets of edges in the query session. This idea is inspired by the ensemble learning method of random forests, which uses multiple decision trees. Entries in the query log that subsume the edges of a decision path are used to find the "support" score of each candidate edge. For each candidate, its support scores over all random decision paths are aggregated into its final score. Section 2.3.2.2 describes this ranking method in detail. We also implemented several other

edge ranking methods by adapting machine learning algorithms such as random forests (RF) and naïve Bayes classifier (NB), as well as class association rules (CAR) and recommendation systems based on singular value decomposition (SVD). Section 2.3.1 describes these techniques in detail.

To the best of our knowledge, there exists no publicly available real-world graph query log in the aforementioned form. Existing visual query builders, possibly due to lack of users, do not have publicly available logs from their usage either. The DBpedia SPARQL query benchmark [23] records queries posed by real users through the SPARQL query interface on DBpedia. This can represent the positive edges in query sessions. However, this query log may offer little help to Orion, due to two limitations: 1) It is applicable to DBpedia only and no other data graph, and 2) Only a third of the edge types present in DBpedia are used in the query log. Hence, in addition to experimenting with this query log, we also simulated query logs for both Freebase and DBpedia data graphs using Wikipedia. The premise is that the various relationships between entities, implied in the sentences of Wikipedia articles, represent co-occurring properties that simulate the positive edges in a query session. Section 2.4 describes various ways of finding such positive edges and injecting negative edges, in order to simulate query logs. Once Orion is in use, query sessions collected by it would result in a real-world query log that might be useful to the community in this line of research.

We conducted extensive user studies over the Freebase data graph, using 30 graduate students from the authors' institution, to compare Orion with a baseline system resembling existing visual query builders. 15 students worked on Orion, and the other 15 on the baseline system. A total of 105 query tasks were performed by users of each system. It was observed that Orion users had a 70% success rate in constructing complex query graphs, significantly better than the 58% success rate of the baseline system's users. We also conducted experiments on both Freebase and DBpedia data graphs to compare RDP with other

edge ranking methods—RF, NB, CAR and SVD. The experiments were executed on the computing resources of the Texas Advanced Computing Center (TACC), [2] to accommodate memory-intensive methods such as RF, SVD and CAR, which required between 40 GB to 100 GB of memory. On average, the other methods required 1.5-4 times more suggestions to complete a query graph, compared to RDP's 40 suggestions. The wall-clock time required to complete query graphs by RDP was mostly comparable with that of RF and NB, and significantly less than that of SVD and CAR. We also performed experiments to study the effectiveness of the various query logs simulated. RDP attained higher efficiency with the Wikipedia based query log compared to the query logs simulated using other ways discussed in Section 2.4.

We summarize the contributions of this chapter as follows:

- We present Orion, a visual query builder that helps schema-agnostic users construct query graphs by making automatic edge suggestions. To the best of our knowledge, none of the existing visual query builders for graphs offers suggestions.
- To help users quickly construct query graphs, Orion uses a novel edge ranking algorithm, Random Decision Paths (RDP), which ranks candidate edges by how likely they are to be relevant to the user's query intent. RDP is trained using a query log containing past query sessions.
- There exists no such real-world query logs publicly available. We thus designed several ways of simulating query logs. Once Orion is in use, the real-world query log collected by it will become a valuable resource to the community.
- We conducted user studies on the Freebase data graph to compare Orion with a baseline system resembling existing visual query builders. Orion had a 70% success rate of constructing complex query graphs, significantly better than the baseline system's 58%.

[2] `http://www.tacc.utexas.edu`.

- We also performed extensive experiments comparing RDP with several other machine learning based methods, on the Freebase and DBpedia data graphs. Other methods required 1.5–4 times more suggestions than RDP, in order to complete query graphs.

2.2 System Overview

2.2.1 Data Model and Query Model

An ultra-heterogeneous graph G_d, also called the data graph, is a connected, directed multi-graph with node set $V(G_d)$ and edge set $E(G_d)$. A node is an entity [3] and an edge represents a relationship between two entities. The nodes and edges belong to a set of *node types* T_V and a set of *edge types* T_E, respectively. Each node (edge) type has a number of node (edge) instances. Each node $v \in V(G_d)$ has an unique identifier, a name, [4] and one or more node types $\mathrm{vtype}(v) \subseteq T_V$. Each edge $e = (v_i, v_j) \in E(G_d)$, denoting a relationship from node v_i to node v_j, belongs to a single *edge type* $\mathrm{etype}(e) \in T_E$.

For example, Will Smith and Tom Cruise are instances of node type FILM ACTOR. They are also instances of node type PERSON. There exist an edge (Tom Cruise, Top Gun) and another edge (Will Smith, Men in Black) which are both edges of type *starring*.

The type of an edge constraints the types of the edge's two end nodes. For instance, given any edge $e = (v_i, v_j)$ of edge type STARRING, it is implied that v_i is an instance of node type FILM ACTOR and v_j is an instance of node type FILM. In other words, FILM ACTOR$\in \mathrm{vtype}(v_i)$ and FILM $\in \mathrm{vtype}(v_j)$.

Given a data graph, users can specify their query intent through query graphs. The concept of query graph is in Definition 8. The nodes in a query graph are labeled by either names of specific nodes or node types. Each answer graph to the query graph is a subgraph

[3] Atomic values such as integers are not supported in the current version of the system.

[4] Without loss of generality, we use a node's name as its identifier in presenting examples, assuming the names are unique.

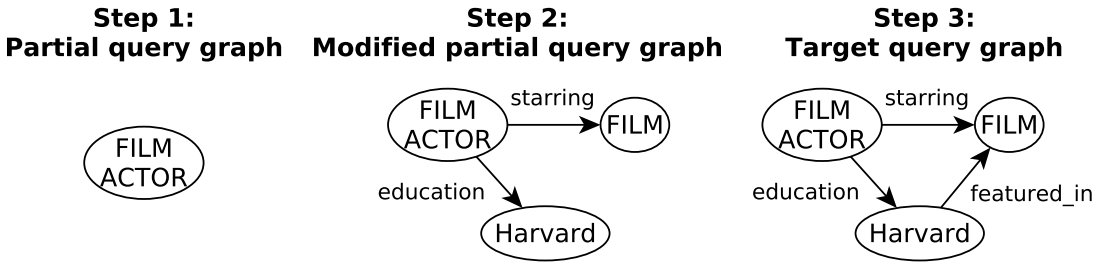


Figure 2.1: Example Partial and Target Query Graphs

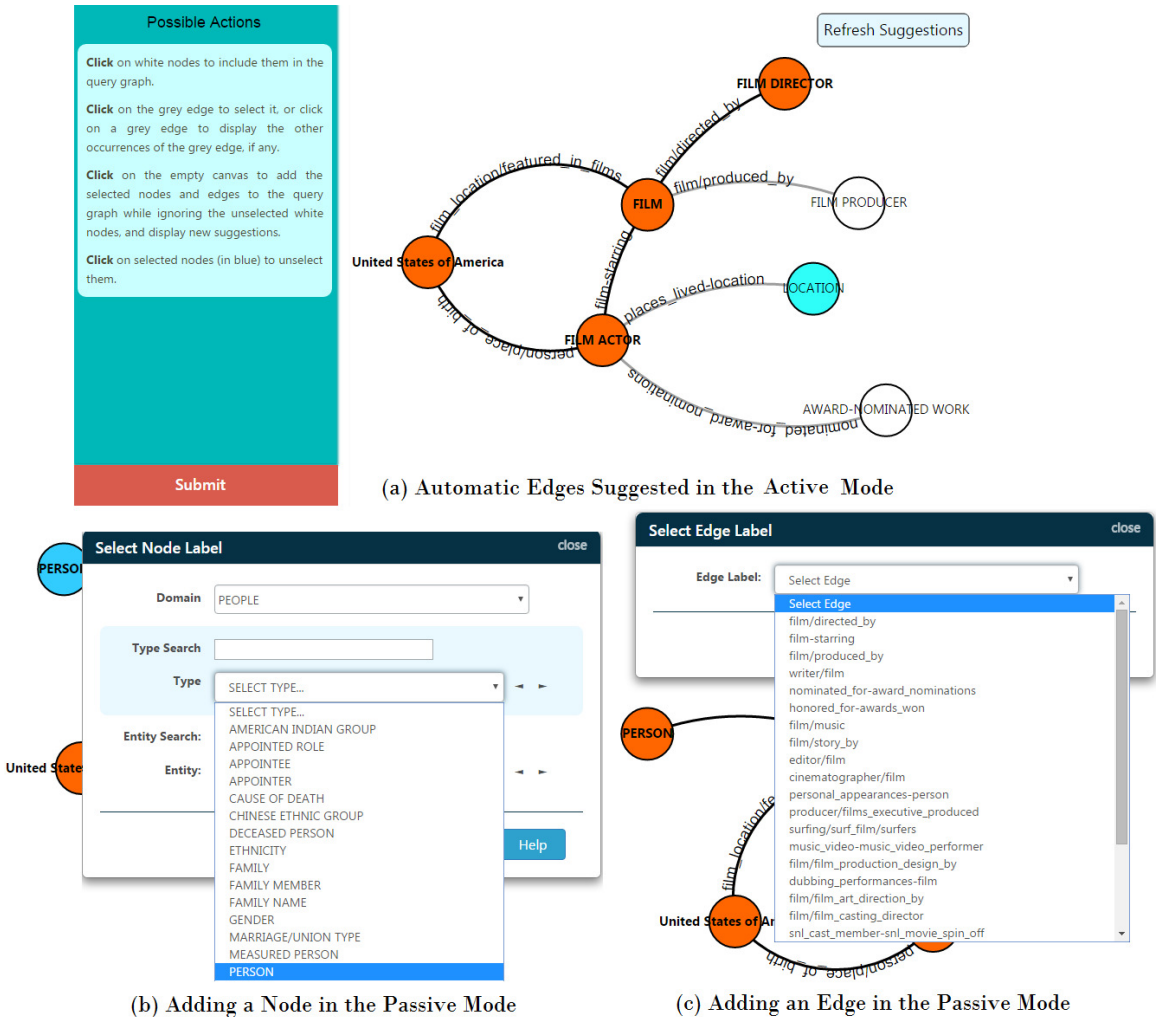
of the data graph and is edge-isomorphic to the query graph. In the answer graph, a node of the query graph is matched by a node of the specified name or any node of the specified type. For instance, the query graph in Step 3 of Figure 2.1 finds all Harvard educated film actors who starred in films featuring Harvard. In Figure 2.1 and other query graphs in this chapter, the all-capitalized node labels represent node types, while others represent node names.

Definition 1 (Query Graph) *A query graph G_q is a connected, directed multi-graph with node set $V(G_q)$ that may consist of both names and types, and edge set $E(G_q)$, such that:*

- $V(G_q) \subseteq T_V \cup V(G_d)$.
- $\forall e \in E(G_q), \text{etype}(e) \in T_E$.

2.2.2 User Interface for Providing Suggestions

Orion helps users interactively and iteratively grow a partial query graph G_p to a target query graph G_t. It suggests edges to a user and solicit the user's response on the edges' relevance, in order to obtain a G_t that satisfies the user's query intent. The query session ends when either the user is satisfied by the constructed query graph or the user aborts the process. The goal is to minimize the number of suggestions required to construct the target query graph.



(a) Automatic Edges Suggested in the Active Mode

(b) Adding a Node in the Passive Mode

(c) Adding an Edge in the Passive Mode

Figure 2.2: User Interface of Orion

Figure 2.1 shows an example sequence of steps to construct a query graph. The user starts by forming the initial partial query graph G_p consisting of a single node. Step 1 in Figure 2.1 shows one such G_p with a node of type FILM ACTOR. New edges are then suggested to the user, who can choose to accept some of the suggestions. For instance, step 2 in Figure 2.1 shows the modified partial query graph obtained after adding two edges (together with two new nodes incident on the edges). Without taking the suggested edges, the user can also directly add a new node or a new edge. The system provides a ranked list of suggestions on the label of the new node/edge, for the user to choose from. Step 3 in

Figure 2.1 shows the example target query graph obtained after adding the edge *featured_in* between Harvard and FILM. In general, to arrive at the target query graph G_t, the user continues the aforementioned process iteratively. Figure 2.2(a) shows the user interface of Orion. It consists of a query canvas where the query graph is constructed. In its active mode, Orion automatically suggests and displays top-k new edges to add to the partial query graph. In its passive mode, users use simple mouse actions on the query canvas to add new nodes and new edges. Orion ranks candidate node and edge labels and displays them using drop-down lists in pop-up windows as shown in Figures 2.2(b) and (c). Orion also offers dynamic tips which list all allowable user actions at any given moment of the query construction process, as shown in Figure 2.2(a).

Active Mode: An Orion user begins the query construction process by adding a single node into the empty canvas. Once the canvas contains a partial query graph consisting of at least a node, Orion automatically operates in its active mode and suggests top-k new edges. Each suggested new edge is between two existing nodes or between an existing node and a new node. Figure 2.2(a) shows a partial query graph comprised of the four dark nodes and the edges between them. The system suggests top-3 new edges, of which each is between an existing node (dark color) and a new node (white or light color). The user can click on some white nodes (which then become light colored, e.g., LOCATION in Figure 2.2(a)) to add them to the query graph, and ignore others. The unselected white nodes are removed from display with a mouse click on the canvas, and the next set of new suggestions are automatically displayed. If the user does not want to select any white nodes, a new set of suggestions can be manually triggered by clicking the "Refresh Suggestions" button on the query canvas.

Passive Mode: At any moment in the query construction process, a user can add a node or an edge using simple mouse actions, which triggers Orion to suggest labels for the newly added node/edge, i.e. it operates in the passive mode. **1)** To add a new edge between

two existing nodes in the partial query graph, the user clicks on one node and drags their mouse to the destination node. The possible edge types for the newly added edge are displayed using a drop-down list in a pop-up suggestion panel, as shown in Figure 2.2(c). The edge types are ranked by their relevance to the query intent. **2)** To add a new node, the user can click on any empty part of the canvas. A suggestion panel pops up, as shown in Figure 2.2(b). It assists the user to select either a name or a type for the node. The options in the two drop-down lists in Figure 2.2(b), one for selecting names and the other for types, are sorted alphabetically. [5] To help the user find the desired node name or type, the suggestion panel is organized in a 3-level hierarchy. Node types are grouped into domains. The user can choose a domain first, followed by a node type in the domain and, if desired, the name of a specific node belonging to the chosen type. The panel also allows the user to search for desired node name or type using keywords. Right after the new node is added, it is not connected to the rest of the partial query graph. Orion makes sure the partial query graph is connected all the time, except for such a moment. Hence, no other operation is allowed, until the user adds an edge connecting the newly added node with some existing node, by using the aforementioned step 1).

2.2.3 Candidate Edges

Orion assists users in query construction by suggesting edge types to add to the partial query graph G_p, in both active and passive modes. In its passive mode, a new edge is drawn between nodes v and v' by clicking the mouse on one node and dragging it to the other. The set of candidate edges in the passive mode, C_P, consists of all possible edge types between v and v'. The set of candidate edges in the active mode, C_A, consists of any edge that can be incident on any node in $V(G_p)$, subject to the schema of the underlying data graph. A

[5] Orion currently ranks suggested edges by their relevance to users' query intent, in both active and passive modes. How to rank node names/types based on query intent is an interesting future direction.

candidate edge can be either between two existing nodes in G_p, or between a node in G_p and a new node automatically suggested along with the edge.

Definition 2 (Incident Edges) *Given a data graph G_d, the incident edges $\mathrm{IE}(v)$ of a node $v \in V(G_d)$, is the set of types of the edges in $E(G_d)$ that are incident on node v. I.e., $\mathrm{IE}(v) = \{\mathrm{etype}(e) | e = (v, v_i)$ or $e = (v_i, v), e \in E(G_d)\}$.*

Definition 3 (Neighboring Candidate Edges) *Given a partial query graph G_p, the neighboring candidate edges $\mathrm{NE}(v)$ of any node $v \in V(G_p)$, is the set of edge types defined as follows, depending on if v is a specific node name or a node type (cf. Definition 8):*

1) if $v \in V(G_d), \mathrm{NE}(v) = \mathrm{IE}(v)$;

2) if $v \in T_V, \mathrm{NE}(v) = \bigcup\{\mathrm{IE}(v') | v' \in V(G_d), v \in \mathrm{vtype}(v')\}$.

When a new edge is added between two nodes v and v' in passive mode, $C_P = \mathrm{NE}(v) \cap \mathrm{NE}(v')$, and the set of candidate edges in active mode is $C_A = \bigcup_{v \in V(G_p)} \{e | e \in \mathrm{NE}(v)\}$.

Definition 4 (Candidate Edges) *Candidate edges C is the set of possible edges that can be added to the partial query graph G_p at any given moment in the query construction process.*

$$C = \begin{cases} C_P & \textit{in passive mode} \\ C_A & \textit{in active mode} \end{cases} \quad (2.1)$$

In Section 2.3 we discuss how to rank candidate edges and thus make suggestions to users in the query construction process.

2.3 Ranking Candidate Edges

A simple method to rank candidate edges is to order them alphabetically. A more sophisticated method is to rank them by using statistics such as frequency in the data graph. Such a method ignores information regarding users' intent. A query log naturally captures

different users' query intent. It contains past query sessions which indicate what edges have been used together by users. Such co-occurrence information gives evidence useful to rank candidate edges by their relevance to the user's query intent.

In a user's query session, edges found relevant, accepted and added to the query graph by the user are called *positive* edges. In Orion's active mode, suggested edges that are not accepted by the user are called *negative* edges. Both positive and negative edges play an important role in gauging the user's query intent, as evidenced by our experiments. At any given moment in the query formulation process, the set of all positive and negative edges hitherto forms a query session.

Definition 5 (Query Log and Query Session) *A query log W is a set of query sessions. A query session Q is defined as a set of positive and negative edges. T_E (cf. Section 3.3) is the set of all possible positive edges for a data graph G_d. The set of all possible negative edges, denoted $\overline{T_E}$, is defined as $\overline{T_E} = \cup_{e \in T_E}\{\overline{e}\}$. If an edge $e \in T_E$ appears as a negative edge in a query session, it is represented as $\overline{e}$. Let $T = T_E \cup \overline{T_E}$. A query session $Q \in \mathcal{P}(T)$, where $\mathcal{P}(T)$ is the power set of T.*

Table 2.1 shows an example query log containing 8 query sessions, one per line. For instance, w_4 is a query session where the suggested edges $\overline{\textit{artist}}$ and $\overline{\textit{title}}$ were not accepted by the user, while edges *writer* and *director* were accepted.

Problem Statement: Given a query log W, an ongoing query session Q and a set of candidate edges C (cf. Equation 2.1), the problem is to rank the edges in C by a scoring function that captures the likelihood that the user would find them relevant.

In Section 2.3.1, we describe several baseline methods to rank candidate edges using query logs. In Section 2.3.2 we propose a novel method inspired by random forests. Section 2.4 discusses several ways of obtaining a query log.

2.3.1 Baseline Methods

Several machine learning algorithms can be adapted to rank candidate edges. For instance, it can be seen as a recommendation problem. One can also use a naïve Bayes classifier or a random forest based classifier to find the probability that an edge e is the *class* associated with the ongoing query session Q, given by $P(e|Q)$. The query log W can be used to learn such models off-line. We implemented several baseline methods by adapting random forests (RF) and nave Bayes classifier (NB), as well as class association rules (CAR) [24] and recommendation systems based on singular value decomposition (SVD) [25]. Below we provide a brief sketch of these methods.

For RF and NB, we used a modified version of the query log W as the training data. A query session with t positive edges and t' negative edges was converted to t training instances, with a different positive edge as the class of each training instance containing $t-1+t'$ attributes. For instance, w_1 in Table 2.1 was converted to $\langle(\textit{education}, \overline{\textit{nationality}}), (\textit{founder})\rangle$ and $\langle(\textit{founder}, \overline{\textit{nationality}}), (\textit{education})\rangle$, where *founder* is the class of the first instance and *education* the class for the second instance. Multi-class classification models were learnt for RF and NB, wherein the number of classes equals the number of distinct positive edge types found in W.

For CAR, W was modified to generate multiple rules. The query sessions in W are itemsets. For a query session with t positive edges and t' negative edges, we generated t association rules. The antecedent (left hand side) of each rule contains $t-1+t'$ attributes, while the consequent (right hand side) contains exactly one positive edge. For instance, w_1 in Table 2.1 was converted to rules $\langle \textit{education}, \overline{\textit{nationality}} \rightarrow \textit{founder}\rangle$ and $\langle \textit{founder}, \overline{\textit{nationality}} \rightarrow \textit{education}\rangle$. If the antecedent of a rule and the ongoing session Q overlap, the rule's consequent can be suggested to the user, weighted by the degree of overlap together with the commonly used measures of support and confidence in association rule mining.

Id	Query Session
w_1	*education*, *founder*, $\overline{\textit{nationality}}$
w_2	*starring*, $\overline{\textit{music}}$, *director*
w_3	*nationality*, $\overline{\textit{education}}$, *music*, $\overline{\textit{starring}}$
w_4	$\overline{\textit{artist}}$, $\overline{\textit{title}}$, *writer*, *director*
w_5	$\overline{\textit{director}}$, *founder*, *producer*
w_6	*writer*, $\overline{\textit{editor}}$, *genre*
w_7	$\overline{\textit{award}}$, *movie*, *director*, $\overline{\textit{genre}}$
w_8	*education*, *founder*, $\overline{\textit{nationality}}$

Table 2.1: Example Query Log W

For SVD, W was converted to a $|W|$ rows $\times$ $|T|$ columns matrix. Each element in the matrix was assigned a value of 0 or 1, based on their occurrence in the corresponding query session. For example, for query log W in Table 2.1, in the first row of the matrix, the columns corresponding to *education*, *founder* and $\overline{\textit{nationality}}$ were set to 1, while the rest were set to 0.

2.3.2 **Random Decision Paths (RDP)**

Here we describe random decision paths (RDP), a novel method for measuring the relevance of a candidate edge. The RDP formulation is motivated by random forests [26]. However, RDP has important differences from the standard definition and application of random forests, and significantly outperforms standard random forests in our experiments.

2.3.2.1 Motivation: from Random Forests to Random Decision Paths

To better understand the similarities and differences between RDP and random forests, it is useful to briefly review decision trees and random forests. In a general classification setting, a decision tree D defines a probability function $P_D(y|x)$, where x is a pattern, and y is the class of that pattern. The decision tree D can also be seen as a classifier that maps patterns to classes: $D(x) = \arg\max_y P(y|x)$. The output of tree D on a pattern x is computed by applying to x a test defined at the root of D, and using the result of the test to direct x to one of the children of the root. Each child of the root is a decision tree in itself,

and thus x moves recursively along a path from the root to a leaf, based on results of tests applied at each node. A leaf node L stores precomputed probabilities $P_L(y)$ for each class y. If pattern x ends up on a leaf L of D, then the tree outputs $P_D(y|x) = P_L(y)$.

A random forest F is a set of decision trees. A forest F defines a probability $P_F(y|x)$, as the average $P_D(y|x)$ over all trees $D \in F$. To construct a random forest, each tree is built by choosing a random feature to test at each node, until reaching a predetermined number of trees. The probability values stored at the leaves of each tree are computed using a set of training patterns, for each of which the true class is known.

Random forests can be applied to our problem, but have certain undesirable properties. Each pattern is a query session, consisting typically of a few (or a few tens of) positive and negative edges. The total number of edge types can reach thousands (it equals 5253 in one of our experimental datasets). The test applied at each node of a decision tree simply checks if a certain edge (positive or negative) is present in the query session. Since query sessions contain relatively few edges compared to the number of edge types, for most tests the vast majority of results is a "no", meaning that the query session does not contain the edge specified in the test. This leads to highly unbalanced trees, where the path corresponding to all "no" results gets the majority of training examples, and paths corresponding to more than 1-2 "yes" results frequently receive no training examples. At classification time, the input pattern x ends up at the all-no path most of the times, and thus the class probabilities $P_D(y|x)$ do not vary much from the priors $P(y)$ averaged over all training examples.

Our solution to this problem is mathematically equivalent to constructing a random forest on the fly, given a query session Q to classify. This random forest is explicitly constructed to classify Q, and is discarded afterwards; a new forest is built for every Q. The tests that we use for tree nodes in that forest consider exclusively edges that appear in Q. This way, the probabilities stored at leaf nodes are computed from training examples

that are similar to Q in a sense, as they share at least some edges with Q. This is why we expect these probabilities to be more accurate compared to the probabilities obtained from a random forest constructed offline, without knowledge of Q. This expectation is validated in the experimental results.

At the same time, since we know Q, constructing full random forests is not necessary, and we can save significant computational time by exploiting that fact. The key idea is that, for any decision tree D that we may build, since we know Q, we know the path that Q is going to take within that tree. Computing the output for any other paths of D is useless, since D is constructed for the sole purpose of being applied to Q. Therefore, out of every tree in the random forest, we only need to compute and store a single path. Consequently, our random forest is reduced to a set of decision paths, and this set is what we call "random decision paths" (RDP).

2.3.2.2 Formulation of Random Decision Paths

We measure the relevance of a candidate edge e to query session Q, by aggregating the relevance of e to several different subsets of edges in Q. We estimate the relevance of an edge e to each such subset of Q using the query log W. We define a support function $\text{supp}(e, Q_i, W)$ to estimate the relevance of an edge e to $Q_i \subseteq Q$:

$$\text{supp}(e, Q_i, W) = \frac{|\{w | w \in W,\, Q_i \cup \{e\} \subseteq w\}|}{|\{w | w \in W,\, Q_i \subseteq w\}|} \tag{2.2}$$

The intuition behind using multiple subsets of Q to measure the relevance of an edge e to the query session Q, instead of using the entire query session Q alone is the following: if Q is long, i.e.the query session contains a large number of positive and negative edges, $\text{supp}(e, Q_i, W)$ might be equal to 0 for every candidate edge e. This is because it is unlikely to find any query session in the query log that is a super-set of Q.

If $\mathcal{P}(Q)$ is the power set of query session Q, we propose to build a set of random decision paths $\Re$, that is: 1) a set of decision paths based only on the edges in query session Q, and 2) a subset of $\mathcal{P}(Q)$ such that $|\Re| \ll |\mathcal{P}(Q)|$. We do not attempt to pre-learn a set of decision paths using query log W that are used to rank edges for any arbitrary query session (like learning a decision tree or rules for a classification model). Instead, given a query session Q, we only build random decision paths specific to Q, that measure the correlation of a candidate edge e with different random subsets of edges in Q. In other words, we assume the presence of a virtual space of all possible decision paths, but only instantiate and use a few random paths specific to Q.

Definition 6 (Decision Path) *A decision path $\overrightarrow{O}$ is an ordered sequence of edges, for a set of edges O.*

The positive and negative edges in a query session Q reflect the relevance and irrelevance of the edges to the user's query intent. An example order for the decision path $\overrightarrow{Q}$ corresponding to query session Q is the order of the edge suggestion sequence. There can be several such ordered sequences for a query session. For any query session $O \in \mathcal{P}(T)'$, the number of possible orders are equal to the total number of permutations of O, which is equal to $|O|!$. Given the set of all query sessions $\mathcal{P}(T)'$, we define $\overrightarrow{\mathcal{P}(T)'}$ as the set of all possible decision paths. $\overrightarrow{\mathcal{P}(T)'} = \bigcup_{O \in \mathcal{P}(T)'} \{\overrightarrow{O_i} | \forall i, 1 \leq i \leq |O|!\}$, and $|\overrightarrow{\mathcal{P}(T)'}|$ is prohibitively large in practice.

A decision path $\overrightarrow{O}$ has a prefix path associated with it. For instance, the prefix of a decision path $\overrightarrow{O}$, denoted by $\text{prefix}(\overrightarrow{O})$, is the path before adding the last edge that formed $\overrightarrow{O}$. If $\overrightarrow{O} = \{e_1, e_2, \ldots, e_{k-1}, e_k\}$, then $\text{prefix}(\overrightarrow{O}) = \{e_1, e_2, \ldots, e_{k-1}\}$. The support for a decision path $\overrightarrow{O}$ is given by $\text{count}(\overrightarrow{O})$, defined as

$$W_{\overrightarrow{O}} = \{w | w \in W, O \subseteq w\}, \text{count}(\overrightarrow{O}) = |W_{\overrightarrow{O}}| \tag{2.3}$$

For a single edged query session, i.e.if $|O|= 1$, the support of the corresponding prefix path $\text{count}(\text{prefix}(\overrightarrow{O})) = |W|$.

Given the query session Q, we define $\mathcal{Q} \subseteq \overrightarrow{\mathcal{P}(T)'}$, the set of all decision paths that can be formed using subsets of edges in Q, whose support is no more than a threshold τ. More formally,

$$\mathcal{Q} = \{\overrightarrow{Q_i} | Q_i \subseteq Q, \text{count}(\overrightarrow{Q_i}) \leq \tau, \text{count}(\text{prefix}(\overrightarrow{Q_i})) > \tau\} \tag{2.4}$$

We propose to build a random set of decision paths $\Re \subseteq \mathcal{Q}$, such that $|\Re|= N$, consisting of only decision paths that are based on the current query session Q, and whose support is no more than τ. A random decision path $\overrightarrow{Q_i}$ is grown using edges in Q until either $\text{count}(\overrightarrow{Q_i}) \leq \tau$, or all the edges in Q are exhausted, whichever comes first. Note that in case all edges in Q are exhausted before we obtain a path $\overrightarrow{Q_i} \in \mathcal{Q}$, then $\mathcal{Q} = \phi$. The final score of an edge $e \in C$ for query session Q is given by

$$\text{score}(e) = \frac{1}{|\Re|} \times \sum_{\overrightarrow{Q_i} \in \Re} \text{supp}(e, Q_i, W) \tag{2.5}$$

Algorithm 1 explains the random decision paths based edge ranking algorithm in detail. Given a set of candidate edges C and a query session Q, we instantiate N random decision paths (line 2). The next edge of the path is chosen uniformly at random without replacement from Q (line 7). The new edge chosen in the path is used to obtain a subset of entries from the query log W. Only those entries in W that contain all the positive and negative edges in the decision path $\overrightarrow{Q_i}$ are chosen to be present in W_{Q_i} (line 6). A decision path $\overrightarrow{Q_i}$ is grown until W_{Q_i} contains no more than τ entries in it (or there are no more edges to be randomly chosen from in Q). The support for each candidate edge $e \in C$ is computed for each decision path (line 15). The support for each candidate edge is averaged across all the decision paths and the edges are ranked based on the final score obtained using Equation 2.5 (line 20).

Algorithm 1: Random Decision Paths Based Edge Suggestion

Input: Data graph G_d, Query Log W, candidate edges C, query session Q, number of random decision paths N, query log subset threshold τ

Output: Ranked list of candidate edges

$E_{sugg} \leftarrow \phi$, $i \leftarrow 0$;
while $i < N$ **do**
  $\overrightarrow{Q_i} \leftarrow \phi$;
  $s_i \leftarrow 0$;
  $W_{\overrightarrow{Q_i}} \leftarrow W$;
  while $s_i < |Q|$ **do**
    $e_{rand} \leftarrow$ sample_without_replacement(Q);
    $\overrightarrow{Q_i} \leftarrow \overrightarrow{Q_i} \cup \{e_{rand}\}$;
    foreach $w \in W_{\overrightarrow{Q_i}}$ **do**
      if $e_{rand} \notin w$ **then**
        $W_{\overrightarrow{Q_i}} \leftarrow W_{\overrightarrow{Q_i}} \setminus \{w\}$;
    if $|W_{\overrightarrow{Q_i}}| \leq \tau$ **then**
      break;
    $s_i \leftarrow s_i + 1$;
  foreach $e \in C$ **do**
    $\mathrm{supp}(e, Q_i, W) \leftarrow$ Equation 2.2;
    $E_{sugg} \leftarrow E_{sugg} \cup \{(e, \mathrm{supp}(e, Q_i, W))\}$;
  $i \leftarrow i + 1$;
foreach $e \in C$ **do**
  $\mathrm{score}(e) \leftarrow$ Equation 2.5;
/* Return candidate edges by decreasing order of $\mathrm{score}(.)$;*/

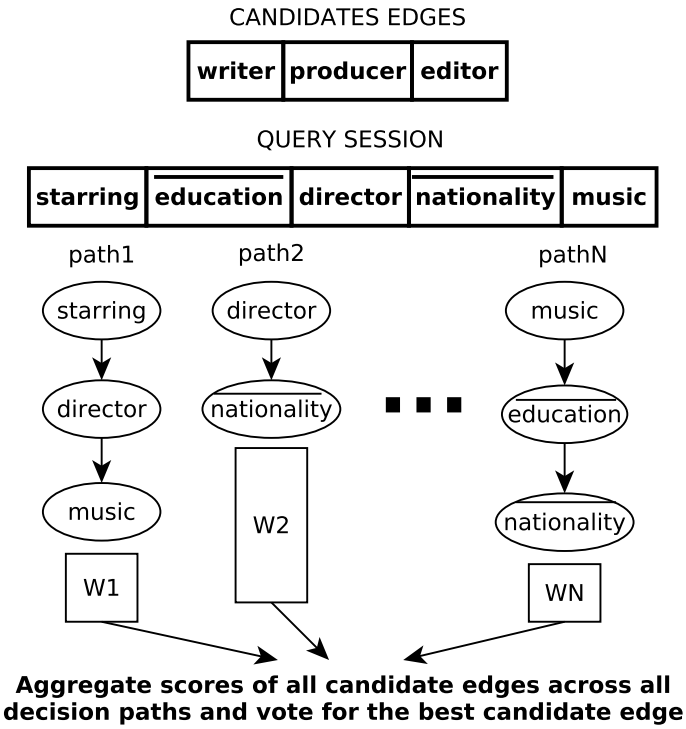


Figure 2.3: Random Decision Paths Based Edge Selection

Figure 2.3 shows an example of using random decision paths to rank the candidate edges. If the set of candidate edges is C = {*writer*, *producer*, *editor*} and query session Q contains edges *starring*, $\overline{\textit{education}}$, *director*, $\overline{\textit{nationality}}$, and *music*, $\overrightarrow{path_1}$ through $\overrightarrow{path_N}$ are examples of various random decision paths. For instance, decision path $\overrightarrow{path_2}$ consists of edges *director* and $\overline{\textit{nationality}}$, which lead to query log subset W_{path_2} where $|W_{path_2}| \leq \tau$. In a decision path $\overrightarrow{path_i}$, the support for each candidate edge $e \in C$ with entry $\bar{e}$ in W_{path_i} is computed. The support for each candidate across all the decision paths is aggregated to rank edges in C.

2.4 Simulating Query Logs

All the baseline methods and the random decision paths rely on a query log. But, to the best of our knowledge, a query log for large graphs is not publicly available, except

for a SPARQL query log [23], which is applicable only for the DBpedia data graph. We thus simulate and bootstrap a query log. We first find correlated positive edges, using three different methods: 1) using Wikipedia and the data graph, 2) using only the data graph, and 3) using the aforementioned SPARQL query log. Then negative edges, which indicate edge suggestions that were not accepted by the user, are injected into the simulated query sessions. If positive edges e_1 and e_2 are in query session Q_i, and another query session Q_j contains e_1 but not e_2, then e_2 is injected into Q_j as a negative edge.

Positive edges using Wikipedia and data graph (WikiPos): Each Wikipedia article describes an entity in detail and refers to other Wikiepdia entities by wikilinks. Given a sentence in a Wikipedia article (or a window of consecutive sentences), the multiple entities mentioned in it can be considered related in some way. We discover the pairwise relationships between these entities. Our premise is that these co-occurring relationships simulate the positive edges of a query session. The intuition is that such consecutive sentences describe closely related facts, and an Orion user may also have such closely related facts as their query intent.

To find co-occurring positive edges, we map entities mentioned in Wikipedia articles to nodes in the data graph. Data graphs such as Freebase and DBpedia provide a straightforward mapping of their nodes to Wikipedia entities. Given a sentence window, all edges found in the data graph between the mapped entities are approximated to the co-occurring positive edges of a query session in W. We consider all edges between the mapped entities in the data graph, while only a subset of these might actually be mentioned in the corresponding Wikipedia article. Thus, the co-occurring positive edges identified using this method might be noisy. We filter out co-occurring positive edges with less support. Every session in the query log is viewed as an itemset. We use the Apriori algorithm to generate frequent itemsets, subject to a support ρ_w. The resulting frequent itemsets thus form query sessions with only positive edges.

Positive edges using the data graph (DataPos): Another way of finding co-occurring positive edges is to use statistics based on the data graph G_d alone. For every node $v \in V(G_d)$, an itemset is created which includes all edges incident on v in G_d. This way we converted the graph G_d to $|V(G_d)|$ itemsets. Here too, we apply the Apriori algorithm to find all frequent itemsets using support ρ_d.

Positive edges using SPARQL query log (SparqlPos): The DBpedia SPARQL query log [23] contains benchmark queries posed by users on DBpedia through its SPARQL query interface. We extract co-occurring positive edges using the properties specified in the WHERE clause of the queries. Since this is a real query log, every set of positive edges found in each WHERE clause is used as is, without applying any pruning as in WikiPos and DataPos.

Injecting negative edges to query log (InjectNeg): The aforementioned methods only generate query sessions with positive edges. But it is crucial to simulate edges that were not accepted by users, since we must rank candidate edges that are correlated with both accepted and ignored edges in a query session. A simple, but effective strategy is used to introduce negative edges into the query logs. Consider a query log which has only positive edges, as produced by the aforementioned methods. For a query session $w \in W$, $T(w)$ is defined as the set of node types of end nodes of all edges in w. I.e., $T(w) = \{t | t \in T_V, \exists e{=}(u,v) \in E(G_d), \text{etype}(e) \in w \text{ s.t. } t \in \text{vtype}(u) \text{ or } t \in \text{vtype}(v)\}$. The set of negative edges added to w, denoted $\overline{w}$, is the set of all edges incident on the node types in $T(w)$. I.e., $\overline{w} = \{\overline{e} | e{=}(u,v) \in E(G_d), \text{vtype}(u) \in T(w) \text{ or } \text{vtype}(v) \in T(w), \text{etype}(e) \notin w\}$. The new entry for every $w \in W$ consists of $w \cup \overline{w}$, which is then used as the final query log by the various candidate edge ranking methods in Section 2.3.

Query Log	Components Used in Query Log Simulation			
	Freebase	**DBpedia**	**Wikipedia**	**SPARQL** [23]
Wiki-FB	Yes	-	Yes	-
Data-FB	Yes	-	-	-
Wiki-DB	-	Yes	Yes	-
Data-DB	-	Yes	-	-
QLog-DB	-	-	-	Yes

Table 2.2: Query Logs Simulated

2.5 Experiments

2.5.1 Setup

We conducted user studies on a double quad-core 24 GB memory 2.0 GHz Xeon server. Furthermore, RDP was compared with other edge ranking algorithms (RF, NB, CAR and SVD) on the Lonestar Linux cluster of TACC, [6] which consists of five Dell PowerEdge R910 server nodes, with four Intel Xeon E7540 2.0GHz 6-core processors on each node, and a total of 1TB memory.

Datasets: We used two large real-world data graphs: the 2011 version of Freebase [3], and the 2015 version of DBpedia [1]. We pre-processed the graphs to keep only nodes that are named entities (e.g.Brad Pitt), while pruning out nodes corresponding to constant values such as integers and strings among others. In the original Freebase dataset, every relationship has an inverse relationship in the opposite direction. For instance, the relationship *director* has *directed by* in the opposite direction. All such edges in the opposite direction were deleted, since they are redundant. The resulting Freebase graph contains 30 million nodes, 33 million edges, and 5253 edge types. After similar pre-processing, the DBpedia graph obtained contains 4 million nodes, 12 million edges and 647 edge types.

Query Logs: Table 2.2 lists the various query logs simulated using the techniques described in Section 2.4. One can find positive edges of a query session using different methods, and inject negative edges into them using the method InjectNeg in Section 2.4. We simulated two different query logs for Freebase: Wiki-FB and Data-FB. The positive

[6] `https://portal.tacc.utexas.edu/user-guides/lonestar`.

Query Type	Query Task
Easy	Find all Basketball players in Chicago Bulls.
Medium	Find all award winning films directed by Steven Spielberg.
Hard	Find all film-actor pairs such that the actor was born in Israel and studied in Harvard University.

Table 2.3: Sample Query Tasks From User Studies

edges for Wiki-FB were simulated using both Wikipedia (September 2014 version) and the Freebase data graph, and the positive edges for Data-DB were simulated using only the Freebase data graph, by methods WikiPos and DataPos in Section 2.4, respectively. We simulated three different query logs for DBpedia: Wiki-DB, Data-DB and QLog-DB. Wiki-DB and Data-DB were simulated via the same approach for Wiki-FB and Data-FB, except that DBpedia (instead of Freebase) was the data graph. For QLog-DB, the positive edges were simulated by SparqlPos in Section 2.4.

Systems Compared in User Studies: To verify if Orion indeed makes it easier for users to formulate query graphs, we conducted user studies with two different user interfaces: Orion, and Naive. Orion operates in both passive and active modes (cf. Section 2.2.2). Naive on the other hand does not make any automatic suggestions and only lets users manually add nodes and edges on the canvas. The various candidate edges are sorted alphabetically and presented to the user in a drop down list. This mimics the query formulation support offered in existing visual query systems such as [20].

Methods Compared for Ranking Candidate Edges: We compared the effectiveness of Orion's candidate edge ranking algorithm (RDP) with the baseline methods described in Section 2.3.1, including RF, NB, CAR and SVD.

2.5.2 User Studies

User Study Set-up: We conducted an extensive user study with 30 graduate students in the authors' institution. The students neither had any expertise with graph query

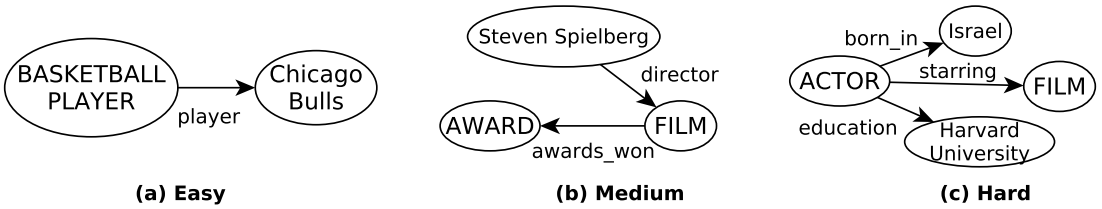


Figure 2.4: Target Query Graphs of Tasks in Table 2.3

formulation, nor did they have exposure to the data graphs. None of these students were exposed to this research in any way other than participating in the user study. We conducted A/B testing using the two interfaces, Orion and Naive. The underlying data graph for both systems was Freebase, and were hosted online on the aforementioned Xeon server. We arbitrarily chose 15 students to work with Orion, and the other 15 students worked with Naive. The users of Orion were not exposed to Naive, and vice versa. We created a pool of 21 query tasks, which consisted of three levels of difficulty. 9 queries were *easy*, 6 queries were *medium* and 6 queries were *hard*. The target query graphs for each easy and medium query tasks had exactly one and two edges, respectively. The target query graphs for hard query tasks had at least three and at most 5 edges. Table 2.3 lists one sample query for each of the three categories. Figures 2.4(a), (b) and (c) depict the target query graphs for the query tasks listed in Table 2.3.

We created 15 different query sheets, where each consisted of 3 easy, 2 medium and 2 hard query tasks, chosen from the pool of 21 queries designed. Each Orion and Naive user was given a query sheet as the task set to complete which ensured that users of both systems worked on the same query tasks. Each user was given an initial 15-minute introduction by the moderators regarding the data graphs, graph query formulation, and the user interface. The users then spent 45 minutes working on their respective query sheets. The users were allowed to ask any clarification questions regarding the tasks during the user study. Each user was awarded a gift card worth $\$15.00$ for their participation in the user study. Since

Likert Scale Score	1	2	3	4	5
Question 1 (Q1): How well do you think the query graph formulated by you captures the required query intent?	Very Poorly	Poorly	Adequately	Well	Very Well
Question 2 (Q2): How easy was it to use the interface for formulating this query?	Very Hard	Hard	Neither Easy Nor Hard	Easy	Very Easy
Question 3 (Q3): How satisfactory was the overall experience?	Unacceptable	Poor	Satisfactory	Good	Excellent
Question 4 (Q4): The interface provided features necessary for easily formulating query graphs.	Strongly Disagree	Disagree	Uncertain	Agree	Strongly Agree

Table 2.4: Survey Questions and Options

System	Queries	Sample Size	Conversion Rate (c)	z-value	p-value
Orion	All	105	c_O=0.74	0.92	0.1788
Naive			c_N=0.68		
Orion	Medium + Hard	60	c_O=0.70	**1.36**	**0.0869**
Naive			c_N=0.58		

Table 2.5: Conversion Rates of Naive and Orion

15 users worked on 7 queries each, we obtained a total of 105 responses for both Orion and Naive.

Survey Form: The users were requested to fill an online survey form at the end of each query task, thus resulting in 105 different survey form responses for each user interface. The survey form had four questions: $Q1$, $Q2$, $Q3$ and $Q4$, as listed in Table 2.4. Each question had five options, specifying the level of agreement a user could have with the particular aspect of the interface measured by the question. We assign a score for every option in each question based on the Likert scale shown in Table 2.4. The least favourable experience with respect to each question is assigned a score of 1, and the most favoured experience is assigned a score of 5.

2.5.2.1 Efficiency Based on Conversion Rate

Measure: One of the popular metrics used to measure the effectiveness of the systems compared in A/B testing is conversion rate c, which is the percentage of tasks completed successfully by users. The conversion rate is defined over a set of Tasks as:

$$c = \frac{\sum_{\mathrm{task} \in \mathrm{Tasks}} \mathrm{sim}(G_u, G_t)}{|\mathrm{Tasks}|} \tag{2.6}$$

where task is a query task assigned to the user, G_u is the corresponding query graph constructed by the user, and G_t is the actual target query graph corresponding to task. The similarity measure $\mathrm{sim}(G_u, G_t)$ captures the notion of success, based on how similar G_u is to G_t. Since we designed the query tasks, the target query graph for each query task was known to us apriori. The query graph constructed by each user was recorded by the interface during the user study. Intuitively, the similarity between G_u and G_t is based on the edge-preserving subgraph isomorphic match between the two graphs. More formally, $sim(G_u, G_t)$ is defined as:

$$\mathrm{sim}(G_u, G_t) = \frac{\max_f \sum_{\substack{e=(u,v) \in E(G_u) \\ e'=(f(u),f(v)) \in E(G_t)}} \mathrm{match}(e, e')}{|E(G_t)|} \tag{2.7}$$

where $f : V(G_u) \rightarrow V(G_t)$ is a bijection, and $\mathrm{match}(e, e')$ is a matching function defined as:

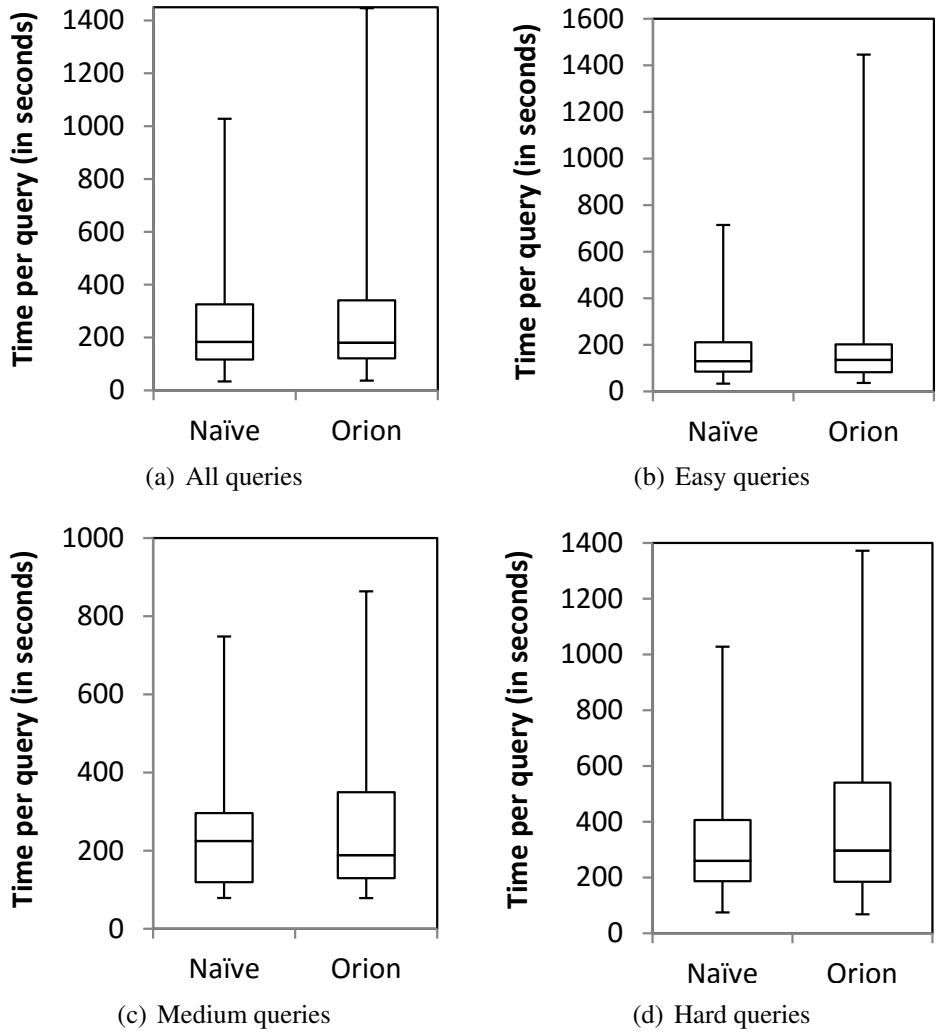
$$\mathrm{match}(e, e') = \begin{cases} 1 & \text{if } u{=}f(u), v{=}f(v), etype(e) = etype(e') \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

Results: Table 2.5 summarizes the conversion rates of Orion and Naive over the set of all query tasks (easy, medium and hard query tasks), and also over only the medium and hard query tasks. We observe that Orion has a better conversion rate than Naive in both scenarios. But, on performing a two sample Z-test with significance level α=0.1, only the observation that Orion has a better conversion rate than Naive for medium and hard queries

is statistically significant. We next describe the hypothesis testing of the two scenarios in detail.

The conversion rate of Orion, c_O, over all the 105 query tasks is 0.74, and the conversion rate of Naive, c_N, for the same set of tasks is 0.68. On average, Orion users had a higher chance of formulating the correct query graph compared to the Naive users. We assume that constructing a query graph follows a Bernoulli trial, with the probability of successfully constructing the target query graph on Orion and Naive as $p_O = c_O$ and $p_N = c_N$ respectively. Our hypothesis, H_{A1}, is that Orion has a better conversion rate than Naive: H_{A1}: $p_O > p_N$. The null hypothesis H_{01} is given by H_{01}: $p_O \leq p_N$. For the aforementioned conversion rates of Orion and Naive, and a sample size of 105, $z = 0.92$. This results in a p-value of 0.1788. Since the p-value $> \alpha$, the null hypothesis cannot be rejected as the data does not significantly support our hypothesis.

We dive in deeper to investigate if there are scenarios where Orion does perform better than Naive. The conversion rate of only medium and hard query tasks (which is equal to a total of 60 query tasks) for Orion is 0.70, and is equal to 0.58 for Naive, i.e.$c_O = p_O = 0.70$ and $c_N = p_N = 0.58$. This indicates that Orion users have a better chance of successfully constructing query graphs with two or more edges, compared to Naive users. Our new hypothesis, H_{A2}, is that Orion has a better conversion rate than Naive for medium and hard queries: H_{A2}: $p_O > p_N$. The null hypothesis H_{02} is given by H_{02}: $p_O \leq p_N$. For the aforementioned conversion rates of Orion and Naive, and a sample size of 60, $z = 1.36$, resulting in a p-value of 0.0869. Since the p-value $< \alpha$, the data significantly supports our claim that Orion users have a higher chance of successfully constructing complex query graphs containing two or more edges.



(a) All queries (b) Easy queries

(c) Medium queries (d) Hard queries

Figure 2.5: User Studies Efficiency Based on Time: Naive and Orion

2.5.2.2 Efficiency Based on Time

We next measure the time taken by a user to construct the query graph for a given query task: the time elapsed between the first time a user clicks on the query canvas for a new query task, to the time the user clicks on the ”Submit” button of the interface. This was recorded in the background during the user study. Figure 2.5(a) shows the distribution of the time taken to complete a query task. We observe that half of the 105 query tasks were completed within 180 seconds by Orion users, while Naive users completed the

same number of query tasks within 183.2 seconds. Around 26 query tasks were completed between 180 to 340.5 seconds, and between 183.24 to 325.7 seconds by Orion and Naive users respectively. Although, there were a few query tasks that took a long time to be completed, with a maximum of 1446.3 seconds for Orion users and 1027.8 seconds for Naive users. We further study the distribution of the time taken to complete query tasks based on the level of difficulty of the tasks. Figure 2.5(b) compares the time taken for easy query tasks. We observe that around 23 of the 45 easy queries are completed within 135.5 and 130.3 seconds by Orion and Naive users respectively. Another 12 queries were completed between 135.5 to 202.3 seconds by Orion users, and between 130.3 to 211.3 seconds by Naive users. Figure 2.5(c) compares the time taken for medium query tasks. We observe that around 15 of the 30 medium queries are completed within 188.2 and 224.6 seconds by Orion and Naive users respectively. Another 7 queries were completed between 188.2 to 349.6 seconds by Orion users, and between 224.6 to 296.2 seconds by Naive users. Finally, Figure 2.5(d) compares the time taken for hard query tasks. We observe that around 15 of the 30 hard queries are completed within 296.1 and 259.6 seconds by Orion and Naive users respectively. Another 7 queries were completed between 296.1 to 540.4 seconds by Orion users, and between 259.6 to 406.4 seconds by Naive users. We observe that despite the steeper learning curve of Orion due to the superior number of features in it, the time taken to complete a majority of the query tasks is comparable with that of Naive.

2.5.2.3 Efficiency Based on Number of Iterations

We next measure the effectiveness of Orion using the number of iterations involved in the query construction process: the number of times a ranked list of edges is presented to the user. The number of iterations is incremented in one of three ways: 1) the user selects one or more of the automatically suggested edges in active mode, and clicks on the canvas to get the next set of suggestions, 2) the user ignores all the suggestions made in active mode

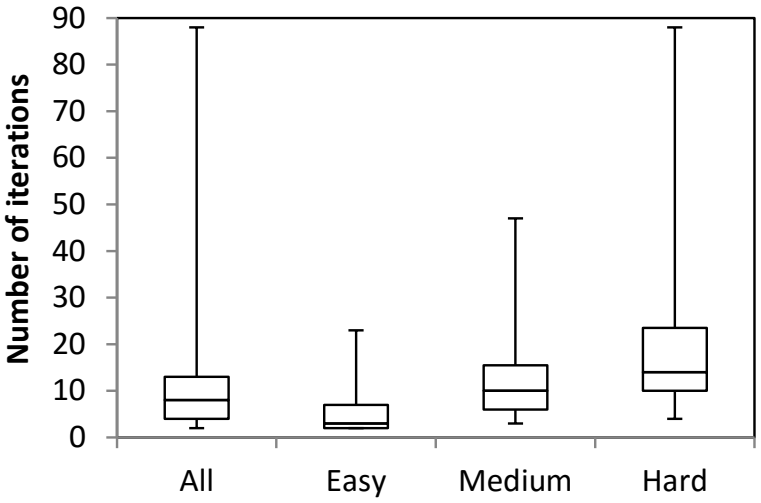
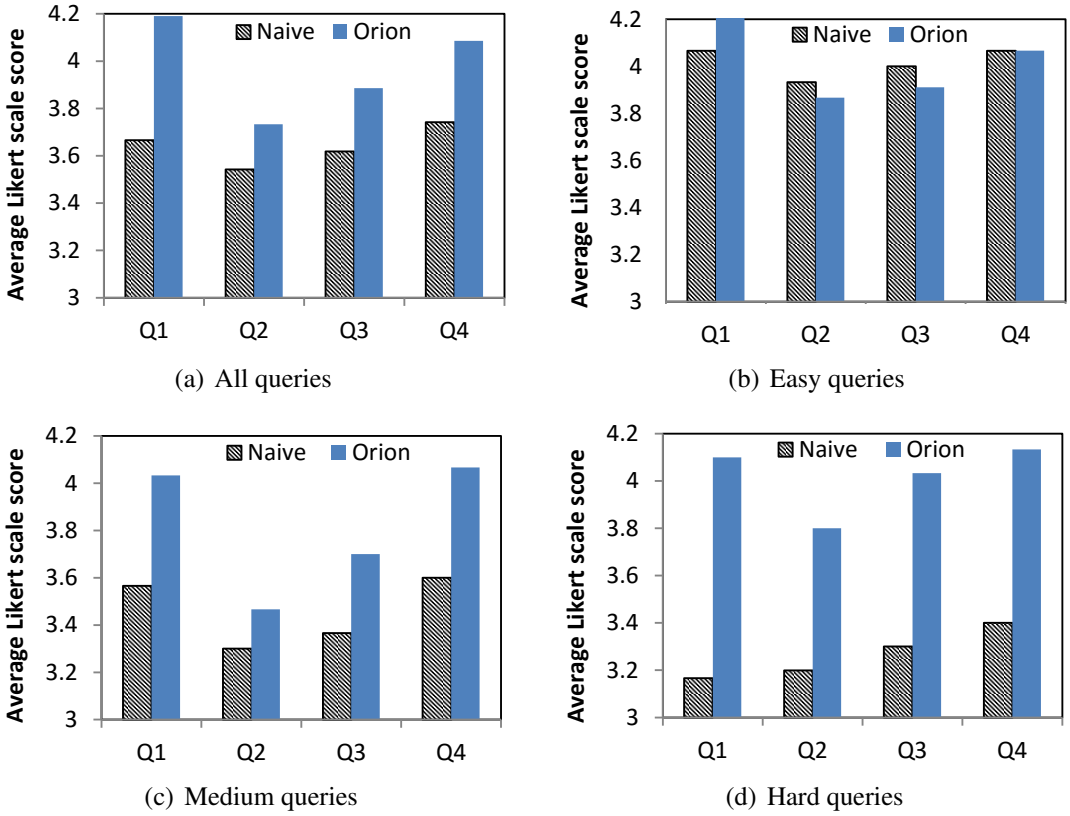


Figure 2.6: User Studies Efficiency Based on Iterations: Orion

and clicks on ”Refresh Suggestions” to get a new set of automatic suggestions, and 3) the user draws a new edge in passive mode. We do not measure this for Naive since there are no automatic ranked suggestions made in it. Figure 2.6 shows the distribution of the number of iterations required to construct query graphs. Overall, Orion users needed no more than only 13 iterations to complete around 79 of the 105 queries. Half of the easy, medium and hard queries required no more than 3, 10 and 14 iterations respectively. Another 11 easy queries required between 3 to 7 iterations, while 7 medium and hard queries each required between 10 to 15.5 and 14 to 23.5 iterations respectively. This indicates that the features offered by Orion helped users formulate query graphs with few interactions with the interface.

2.5.2.4 User Experience Results

The user experience results is based on the answers to all the questions in the survey form by all the users. The overall user experience for each question of an interface is measured by averaging the score obtained for that question across all the users working on that interface. Figure 2.7(a) shows the overall user response of all the questions, across all



(a) All queries

(b) Easy queries

(c) Medium queries

(d) Hard queries

Figure 2.7: User Experience Based on Survey Responses

the 105 users for both Orion and Naive. We observe that Orion users report an improvement of 0.5 for $Q1$, 0.2 for $Q2$, 0.25 for $Q3$ and 0.3 for $Q4$ on Likert scale, when compared to the Naive users.

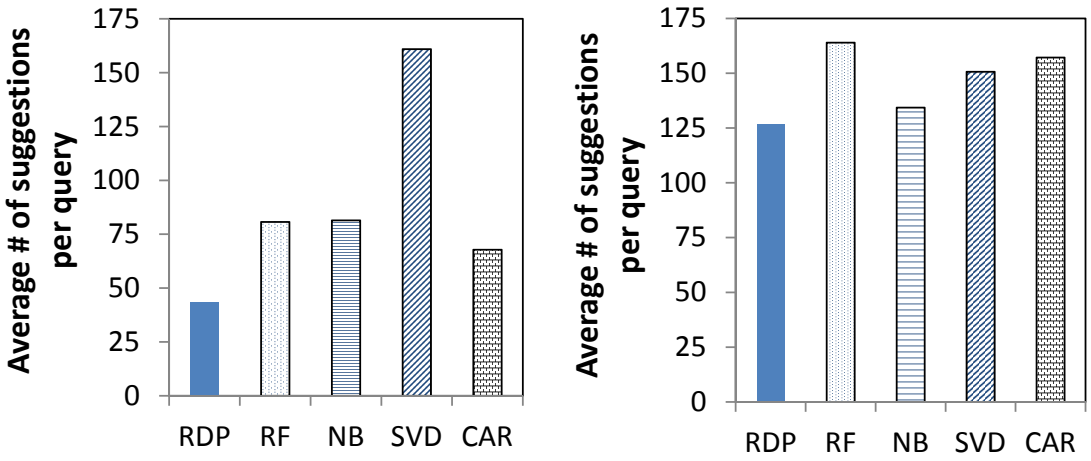
We further break down the average score over each question based on the difficulty level of the query task to study the difference in user experience between Orion and Naive in detail. Figure 2.7(b) shows the average score over only the easy query tasks (a total of 45 query tasks each for both Orion and Naive), which shows that Orion users had a better experience than the Naive users w.r.t $Q1$, while the Naive users had a slightly better experience than Orion users w.r.t $Q2$ and $Q3$. Both the sets of users had similar experience w.r.t $Q4$. Figure 2.7(c) shows the average score over only the medium query tasks (a total of 30 query tasks each for both Orion and Naive), which shows that Orion users had an

improvement of 0.4 on Likert scale w.r.t $Q1$ and $Q4$ compared to the Naive users. They also had an improvement close to 0.1 on Likert scale w.r.t both $Q2$ and $Q3$. Finally, Figure 2.7(d) shows the average score over only the hard query tasks (a total of 30 query tasks each for both Orion and Naive), which shows that Orion users felt a significant improvement in the user experience across all four questions. Orion users had an improvement of around 1.0 w.r.t $Q1$, 0.6 w.r.t $Q2$, and 0.7 w.r.t both $Q3$ and $Q4$. We thus observe that as the difficulty level of the query graph being constructed increases, the usability of Orion seems significantly better than Naive's. Naive users find the system uncomfortable to use when the target query graph contains two or more edges.

2.5.3 **Comparing Candidate Edge Ranking Methods**

We next compare the performance of RDP, Orion's edge ranking algorithm, with other machine learning algorithms: RF, NB, SVD and CAR. We compared the performance of these algorithms over two widely used real-world data graphs: Freebase and DBpedia. We used the Wiki-FB and Wiki-DB query logs for Freebase and DBpedia respectively. We had to perform these experiments on the TACC machine, because RF has high memory requirements. For instance, generating a random forest model with 80 trees, using a query log containing around 100,000 query sessions, requires 55 GB of RAM.

We created multiple target query graphs for each dataset, conforming with the schema of the underlying data graph. For a given target query graph, the input to each of the algorithms was an initial partial query graph containing exactly one edge in it. The task of each algorithm was to iteratively suggest exactly one edge at a time, given the partial query graph. If the edge suggested was present in the target query graph, it was added into the partial query graph, and recorded as a positive edge. If not, the edge was ignored, and recorded as a negative edge. The process was stopped either when the partial query graph was grown completely into the target query graph, or if 200 suggestions were up. For each



(a) Freebase (b) DBpedia

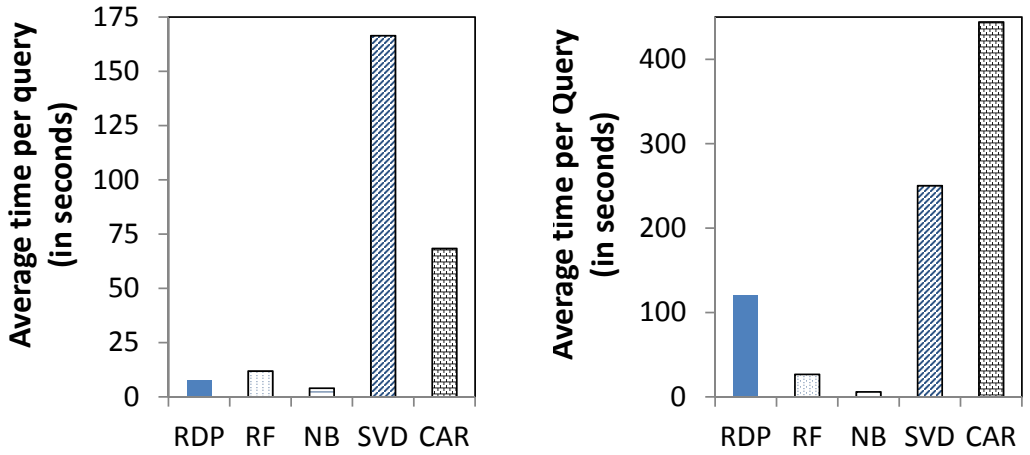
Figure 2.8: Efficiency of All Methods: Number of Suggestions

target query graph G_t containing $E(G_t)$ number of edges, we internally converted it into $E(G_t)$ different instances of target query graphs, each starting with a different-edged initial partial query graph as input to the algorithms.

We created 43 target query graphs for Freebase, consisting of 6 two-edged query graphs, 10 three-edged query graphs, 9 four-edged query graphs, 17 five-edged query graphs and 1 six-edged query graph. These 43 target query graphs were thus converted to 167 different input instances, creating a query set called *Freebase-Queries*. We created 33 target query graphs for DBpedia, consisting of 2 three-edged query graphs, 29 four-edged query graphs, and 2 five-edged query graphs. These 33 target query graphs were converted to 130 different input instances, creating a query set called *DBpedia-Queries*.

2.5.3.1 Efficiency Based on Number of Suggestions

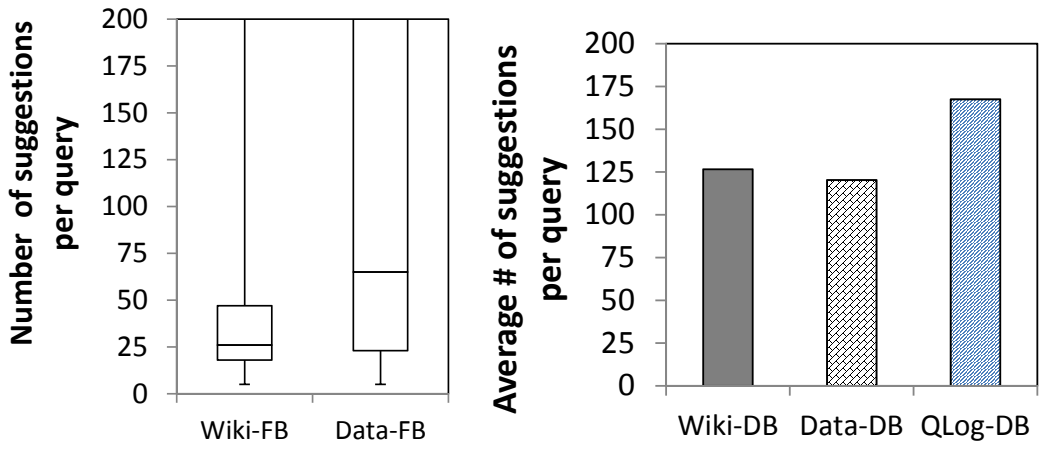
For a query graph completion system, we believe an important measure of its efficiency is the number of suggestions required to successfully grow a partial query graph to its corresponding target query graph. This is because, if a system can help users construct



(a) Freebase (b) DBpedia

Figure 2.9: Efficiency of All Methods: Time

the target query graph with fewer number of suggestions, it indicates that the suggestions made indeed captured the user's query intent. Figure 2.8(a) shows the average number of suggestions required to complete each of the 167 input instances for Freebase. We observe that RDP significantly outperforms the other methods. RDP requires only 43.5 suggestions per query graph on average, nearly half the number of suggestions required to complete a query graph using RF and NB. It also requires only a quarter of the number of suggestions required to complete a query graph using SVD, while CAR requires 67.8 suggestions. Figure 2.8(b) shows the average number of suggestions required to complete each of the 167 input instances for DBpedia. We observe that RDP requires 126.6 suggestions on average to complete a query graph, performing slightly better than NB which requires 134.3 suggestions. RDP also comfortably outperforms RF, SVD and CAR which on average require 164, 150.7 and 157.9 suggestions per query graph respectively.



(a) Freebase (b) DBpedia

Figure 2.10: Effectiveness of Query Logs

2.5.3.2 Efficiency Based on Time

We next compare the efficiency of the various methods over the time required to grow the initial partial query graph to its corresponding target query graph. Figure 2.9(a) compares the average time required to complete a query task by each of the algorithms over Freebase. RDP, NB and RF significantly outperform SVD and CAR. RDP requires 7.7 seconds, slightly higher than NB's 3.9 seconds, and better than RF's 11.8 seconds per query, which is commendable especially since both random forest and Bayesian classifiers are extremely efficient once the models are learnt. Figure 2.9(b) compares the average time required to complete a query task by each of the algorithms over DBpedia. SVD and CAR are inefficient requiring 250.2 and 444.2 seconds per query respectively. NB requires 5.9 seconds, which is faster than both RF and RDP that require 26.7 and 119.7 seconds per query respectively.

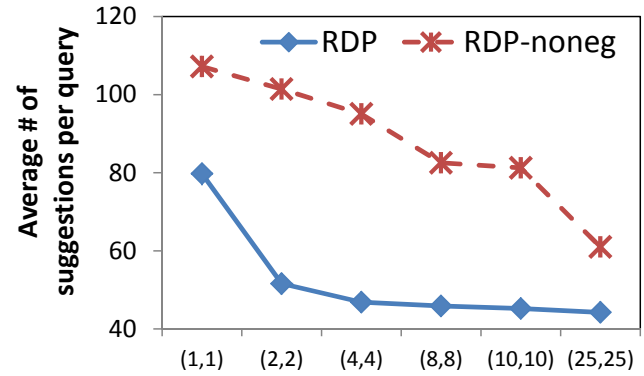
2.5.4 **Effectiveness of Query Logs**

We compare the effectiveness of the various query logs listed in Table 2.2. We use RDP as the algorithm for edge suggestion, and the number of suggestions required to grow

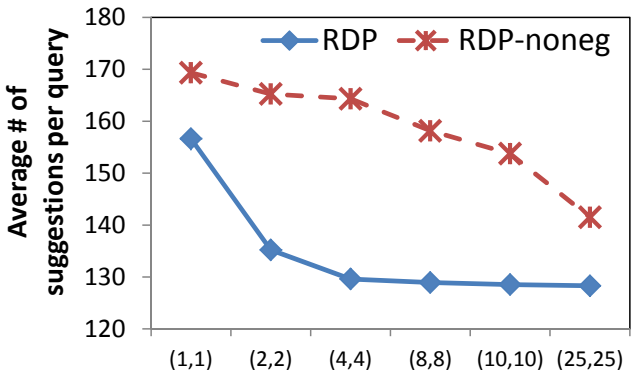
the initial partial query graph to the target query as the measure of effectiveness of the query logs. Freebase-Queries and DBpedia-Queries, described in Section 2.5.3, were the sets of queries used to compare the various Freebase and DBpedia query logs respectively.

Query Logs for Freebase: Figure 2.10(a) shows the distribution of the number of suggestions required to complete a query task using Wiki-FB and Data-FB query logs. We observe that 83 of the 167 input instances needed no more than 26 edge suggestions with the Wiki-FB query log, while it required at most 65 edge suggestions to complete the same number of queries using the Data-FB query log. Around 42 more input instances required between 26 to 47 suggestions with Wiki-FB, while it required between 65 to 200 suggestions with Data-FB. This indicates that the query log simulated using Wikipedia and the Freebase data graph using WikiPos described in Section 2.4 is of superior quality compared to the one simulated using only the Freebase data graph. This suggests that positive edges established based on the context of human usage of the relationships is better than the positive edges established using only the data graph.

Query Logs for DBpedia: Figure 2.10(b) shows the average number of edge suggestions required to process the 130 different DBpedia input instances, using each of the three aforementioned query logs for DBpedia. We first observe that QLog-DB performs poorly compared to the other two query logs. This is because the DBpedia SPARQL query log is not comprehensive enough and is limited in the variety of relationships captured, making it ineffective. The second interesting observation we make is the algorithm requires 120.3 suggestions on average using Data-DB, while it requires 126.6 suggestions with Wiki-DB. Data-DB performs slightly better than Wiki-DB due to the fact that DBpedia is a high quality data graph generated using the info-boxes in Wikipedia pages. The sets of positive edges in Wiki-DB are simulated using the text in Wikipedia and the DBpedia data graph. The two query logs are thus highly similar to each other, unlike the case in



(a) Freebase



(b) DBpedia

Figure 2.11: Effect of Parameters on RDP (N, τ)

Freebase where we could see a significant difference between the performance of Wiki-FB and Data-FB.

2.5.5 **Parameter Tuning for RDP**

We finally study a variation of RDP, and the effect of N and τ, the two parameters used in RDP. As described in Section 2.3.2.2, given a query session Q, RDP builds N different random decision paths. Each random decision path is grown incrementally, until either the support for the path is no more than a threshold τ, or if all edges in Q are exhausted. While building a random decision path, RDP considers both the positive and negative edges. To study if considering the negative edges indeed helps in better identify-

ing the user's query intent, we create a variation of RDP, called RDP-noneg, which does not include any negative edges in the random decision paths. Figures 2.11(a) and 2.11(b) compare the average number of suggestions required to complete each query graph with different values of N and τ, for Freebase and DBpedia queries respectively. In both the cases, we observe that the average number of suggestions required per query decreases as we increase the number of random decision paths, and the threshold τ. It saturates after we reach around 10 for both N and τ in RDP. Figures 2.11(a) and 2.11(b) also compare the average number of suggestions required to complete the query graphs using RDP and RDP-noneg. With the best parameter values of $N = 25$ and $\tau = 25$, RDP requires 44.2 suggestions while RDP-noneg requires 60.9 suggestions in Freebase. RDP also requires fewer suggestions in DBpedia with 128.5 suggestions compared to 141.5 suggestions required by RDP-noneg. We observe that RDP significantly outperforms its variation RDP-noneg, indicating that considering negative edges in query sessions is indeed helpful.

CHAPTER 3

GRAPH QUERY BY EXAMPLE

3.1 Introduction

Large ultra-heterogeneous knowledge graphs are ubiquitous and content-rich. But the sheer size and complexity of these graphs make it difficult for users to query them. In this chapter we present GQBE (Graph Query by Example), a system that queries knowledge graphs by example entity tuples instead of graph queries or structured query languages. Given a data graph and a query tuple consisting of entities, GQBE finds similar answer tuples. Consider the data graph in Figure 1.1 and a 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. 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⟩).

GQBE is among the first to query knowledge graphs by example entity tuples. 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 Figure 3.1.

(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 (Section 3.4) 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* (Section 3.6), 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* (Section 3.7), 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 chapter as follows:

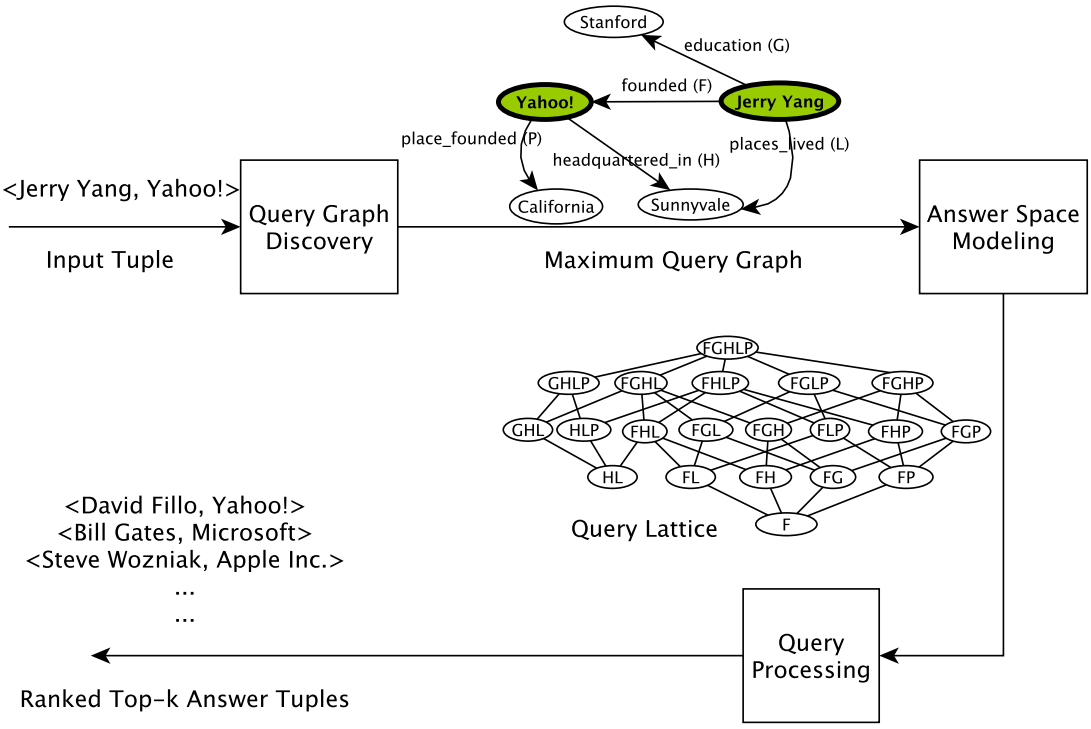


Figure 3.1: The Architecture and Components of GQBE

- 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 (Section 3.9). The comparison with a state-of-the-art graph querying framework NESS[27] and an exemplar query sys-

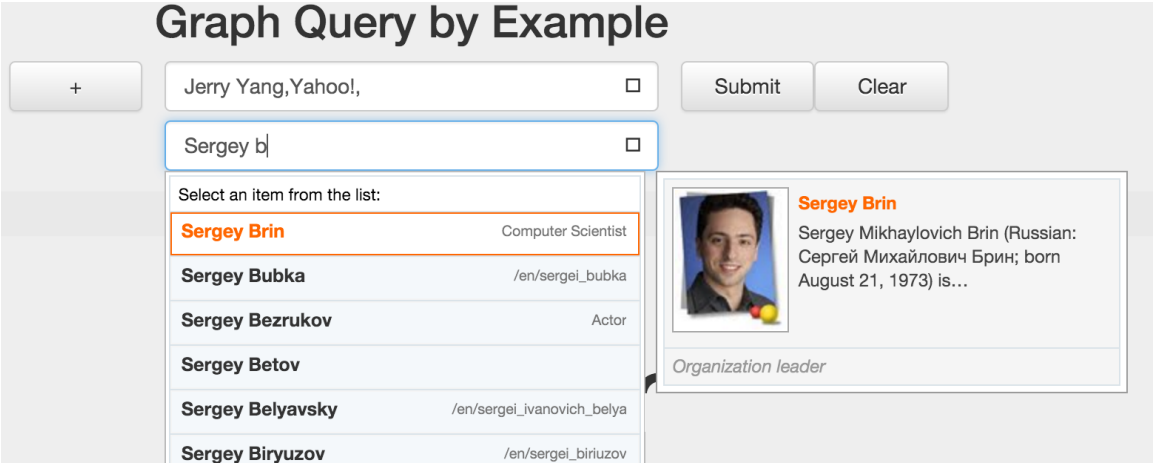


Figure 3.2: GQBE's Input Interface

tem EQ [28] shows that GQBE is over twice as accurate as NESS and EQ. GQBE also outperforms NESS on efficiency in most of the queries.

3.2 User Interface and Functionality

GQBE provides several features that aid in convenient query experience: 1) a simple search box for entering example entity tuple, 2) auto completion of entity names that helps a user find the exact entity she is typing for, 3) provision to provide multiple example tuples, 4) display of the query graph discovered by the system for capturing user intent, and 5) display of a ranked list of answer tuples with their corresponding answer graphs that justify the ranking. The rest of this section provides the details.

GQBE features a simple keyword-based input interface (Figure 3.2), in which a user enters example tuples of entities known to her. For instance, in Figure 3.2, the user is in the middle of typing Sergey Brin. GQBE offers auto completion, powered by Freebase API. Specifically, when the user partially enters the name of an entity ("Sergey b" in Figure 3.2), GQBE shows a list of suggested entities whose names match the keywords. Hovering the mouse pointer over one suggested entity (Sergey Brin in Figure 3.2) will bring out a summary

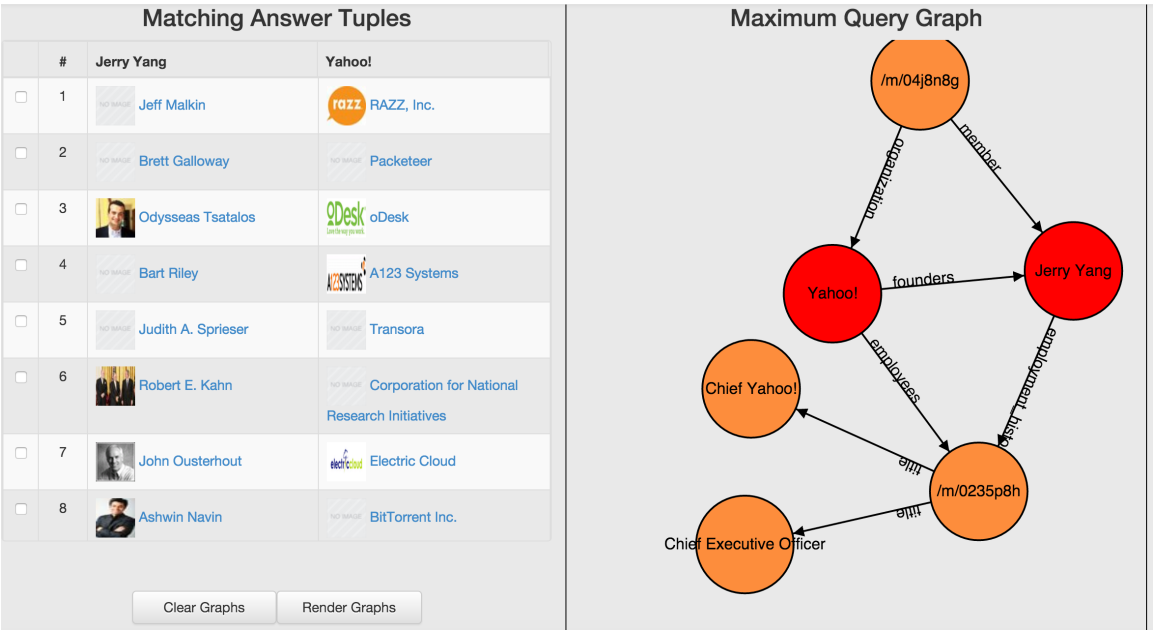


Figure 3.3: Interface Displaying Answer Tuples

of the entity from its corresponding Freebase page. This summary can be used to resolve ambiguity among multiple entities with similar names.

Deriving user intent based on a single example entity tuple is a hard task. GQBE allows multiple example tuples to better capture the user intent. As shown in Figure 3.2, the user has entered the first tuple $\langle$Jerry Yang, Yahoo!$\rangle$ and is in the middle of entering the second tuple $\langle$Sergey Brin, Google$\rangle$. More example tuples can be entered by clicking the '+' sign preceding the first entered tuple. Entered tuples can also be altered by directly changing the keywords in the corresponding search boxes.

Once the user provides example tuples and clicks the "Submit" button, GQBE's back-end query processor kicks in. It discovers the MQG, a hidden weighted query graph, to capture the user's query intent. GQBE evaluates the MQG to find similar answer graphs and corresponding answer tuples, and ranks them by how well they match the input tuples (details in Section 3.7). Figure 3.3 shows the result interface displaying the ranked answer tuples. The user can further explore the entities in the answer tuples by clicking on them

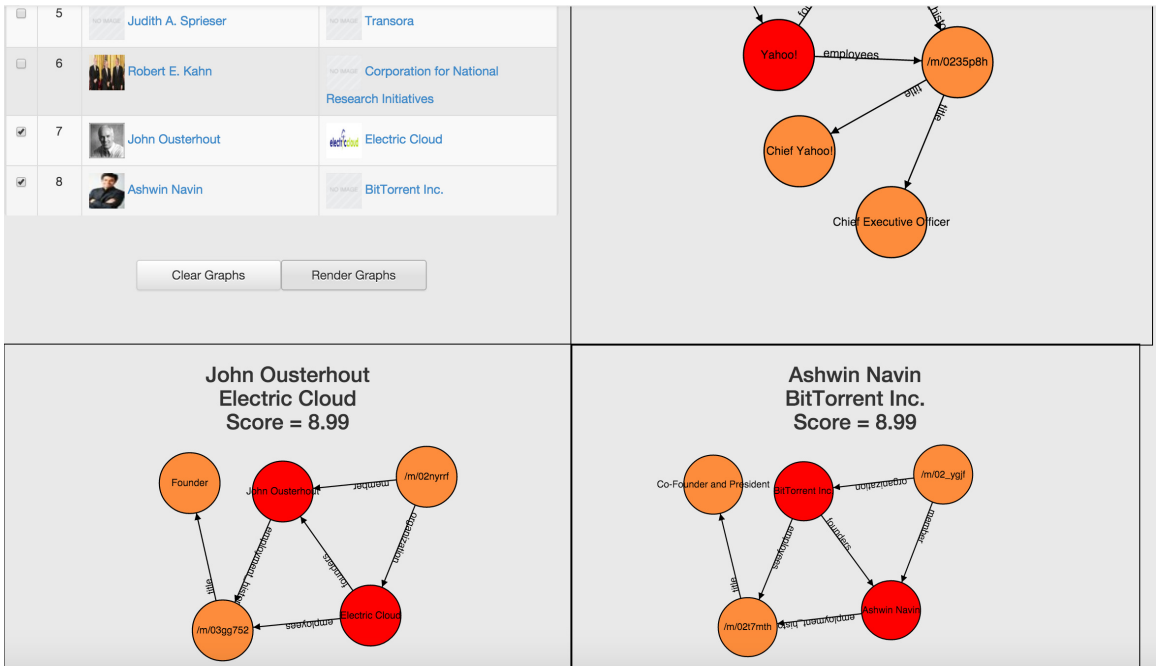


Figure 3.4: Interface Displaying Answer Graphs

which opens their corresponding Freebase pages in new Web browser windows. GQBE assists the user in understanding the rationale behind the answer tuples and their ranking. To this end, the MQG is displayed on the right-hand side of the screen that helps the user find out if her query intent was captured or not, as shown in Figure 3.3. One can also view the answer graphs corresponding to the ranked answer tuples by selecting the check-box before the answer tuples of interest, and clicking on the "Render Graphs" button as shown in Figure 3.4. A score indicating how well the answer tuple matches the example query tuple is also displayed with the answer graphs. For instance, Figure 3.4 shows the answer graphs and the corresponding scores of ⟨John Ousterhout, Electric Cloud⟩ and ⟨Ashwin Navin, BitTorrent Inc.⟩ answer tuples.

3.3 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 $id(v)$. [1] 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 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

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

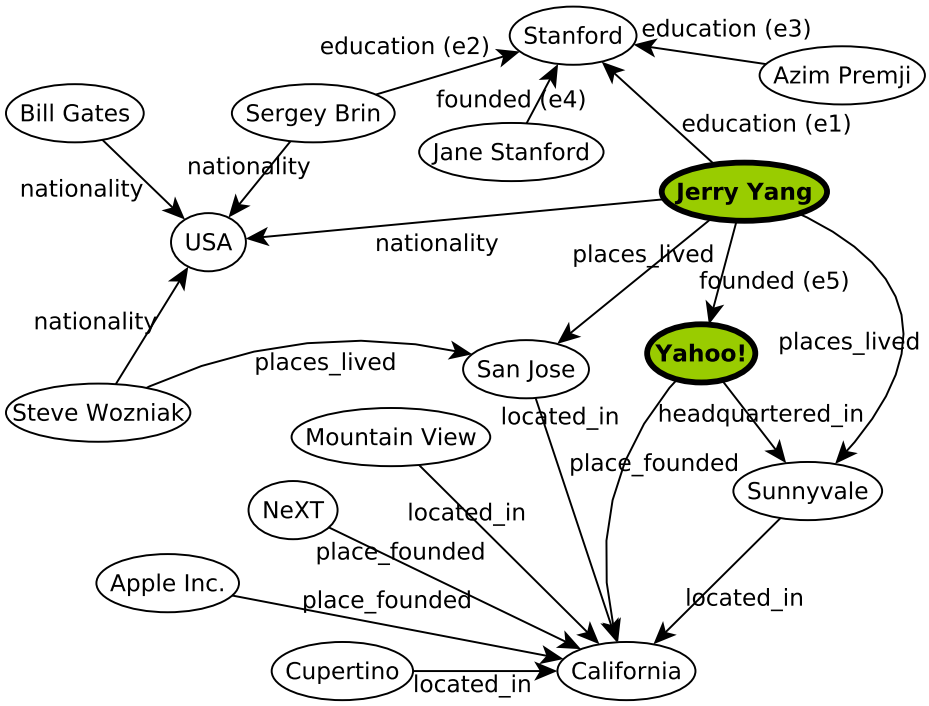


Figure 3.5: Neighborhood Graph for ⟨Jerry Yang, Yahoo!⟩

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 7 *The* **neighborhood graph** *of query tuple t, denoted H_t, is the* weakly connected subgraph[2] *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* $e \in p\}$.

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

Example 2 (Neighborhood Graph) *Given the data graph in Figure 1.1, Figure 3.5 shows the neighborhood graph for query tuple* $\langle$*Jerry Yang, Yahoo!*$\rangle$ *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.

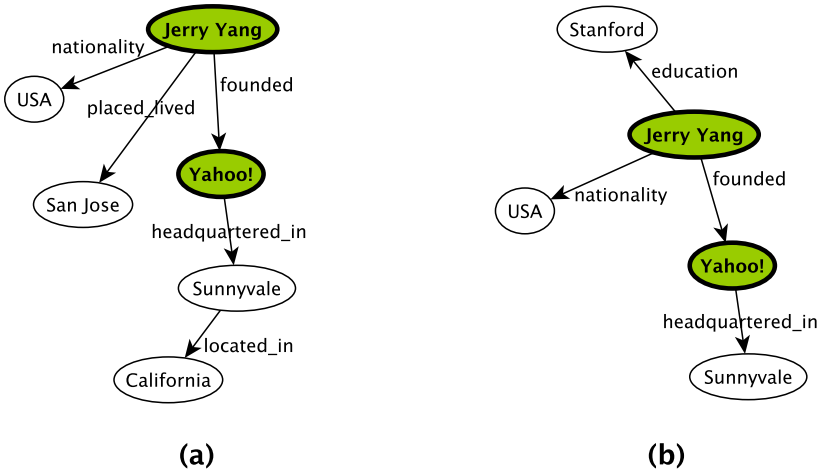


Figure 3.6: Two Query Graphs in Figure 3.5

Definition 8 *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.* $\forall v \in t$, $v \in V(Q)\}$.

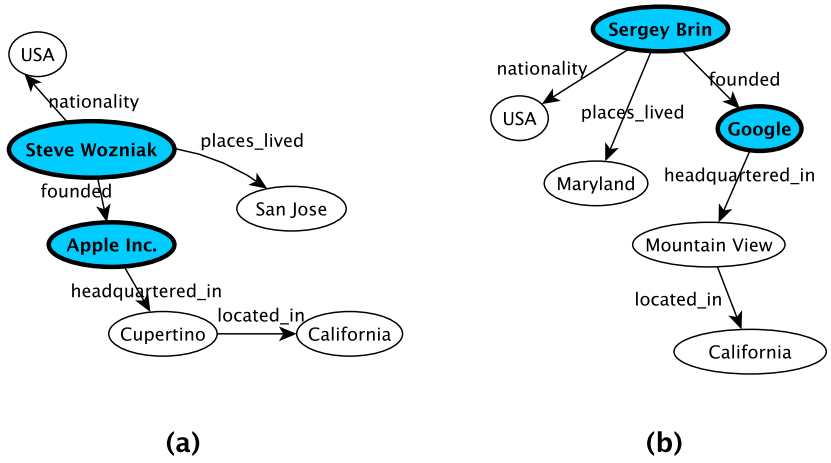


Figure 3.7: Two Answer Graphs for Figure 3.6(a)

Continuing the running example, Figure 3.6 shows two query graphs for the neighborhood graph in Figure 3.5.

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 9 *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*.*

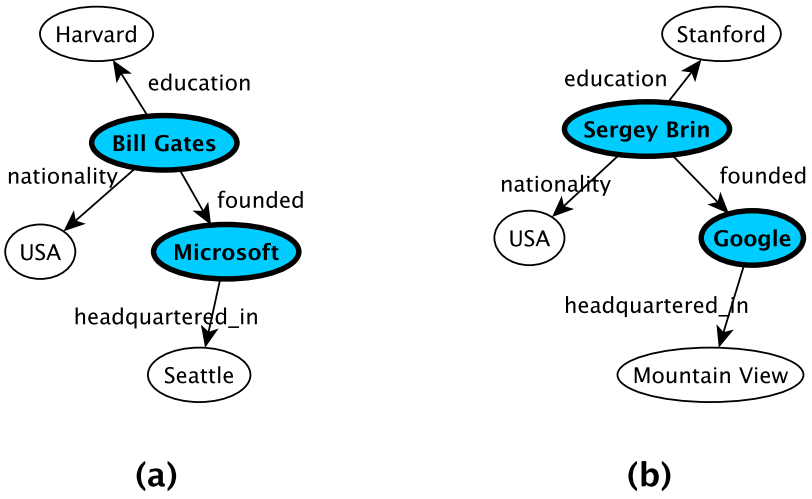


Figure 3.8: Two Answer Graphs for Figure 3.6(b)

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 3 (Answer Graph and Answer Tuple) *Figure 3.7 and Figure 3.8 each show two answer graphs for query graphs Figure 3.6(a) and Figure 3.6(b), respectively. The answer tuples in Figure 3.7 are* $\langle$*Steve Wozniak, Apple Inc.*$\rangle$ *and* $\langle$*Sergey Brin, Google*$\rangle$*. The answer tuples in Figure 3.8 are* $\langle$*Bill Gates, Microsoft*$\rangle$ *and* $\langle$*Sergey Brin, Google*$\rangle$*.*

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:

$$\mathsf{score}_t(t') = \max_{A \in \mathcal{A}_Q, Q \in \mathcal{Q}_t} \{\mathsf{score}_Q(A) | t' = t_A\} \tag{3.1}$$

The score of an answer graph A ($\mathsf{score}_Q(A)$) captures A's similarity to query graph Q. Its equation is given in Section 3.6.2.

The same answer tuple t' may be projected from multiple answer graphs, which can match different query graphs. For instance, Figures 3.7(b) and 3.8(b), which are answers to different query graphs, have the same projection—$\langle$Sergey Brin, Google$\rangle$. By Equation (3.1),

the highest score attained by the answer graphs is assigned as the score of t', capturing how well t' matches t.

3.4 Query Graph Discovery

3.4.1 Maximum Query Graph

The concept of neighborhood graph H_t (Def. 7) 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 $\langle$Jerry Yang, Yahoo!$\rangle$ 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 10 *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 Section 3.8.*

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 arbi-

trary 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 [29]. 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.*

Based on the theoretical analysis, we present a greedy method (Alg. 2) 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

Algorithm 2: Discovering the Maximum Query Graph

Input: neighborhood graph H_t, query tuple t, an integer r

Output: maximum query graph MQG_t

$m \leftarrow \frac{r}{|t|+1}$; $V(MQG_t) \leftarrow \phi$; $E(MQG_t) \leftarrow \phi$; $\mathcal{G} \leftarrow \phi$;
foreach $v_i \in t$ **do**
    $G_{v_i} \leftarrow$ use DFS to obtain the subgraph containing vertices (and their incident edges) that connect to other v_j in t only through v_i;
    $\mathcal{G} \leftarrow \mathcal{G} \cup \{G_{v_i}\}$;
$G_{core} \leftarrow$ use DFS to obtain the subgraph containing vertices and edges on undirected paths between query entities;
$\mathcal{G} \leftarrow \mathcal{G} \cup \{G_{core}\}$;
foreach $G \in \mathcal{G}$ **do**
    $step \leftarrow 1$; $s_1 \leftarrow 0$; $s \leftarrow m$;
    while $s > 0$ **do**
        $M_s \leftarrow$ the weakly connected component found from the top-s edges of G that contains all of G's query entities;
        if M_s *exists* **then**
            if $|E(M_s)| = m$ **then break**;
            if $|E(M_s)| < m$ **then**
                $s_1 \leftarrow s$;
                if $step = -1$ **then break**;
            if $|E(M_s)| > m$ **then**
                if $s_1 > 0$ **then**
                    $s \leftarrow s_1$; **break**;
                $s_2 \leftarrow s$; $step \leftarrow -1$;
        $s \leftarrow s + step$;
    if $s = 0$ **then** $s \leftarrow s_2$;
    $V(MQG_t) \leftarrow V(MQG_t) \cup V(M_s)$;
    $E(MQG_t) \leftarrow E(MQG_t) \cup E(M_s)$;

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. 2). 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. 2 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. 2 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×$(m$+$c))$. For all n+1 subgraphs, the total time required to find the final MQG_t is $O((n$+$1) \times (r' \log r'$+c×$(m$+$c)))$. 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.

3.5 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 Section 3.9 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 Figure 1.1 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 $label(e)$=$label(e')$.

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. 2 (Section 3.4.1) is also used to trim MQG_T to a size close to r. In MQG_T, the weight of an

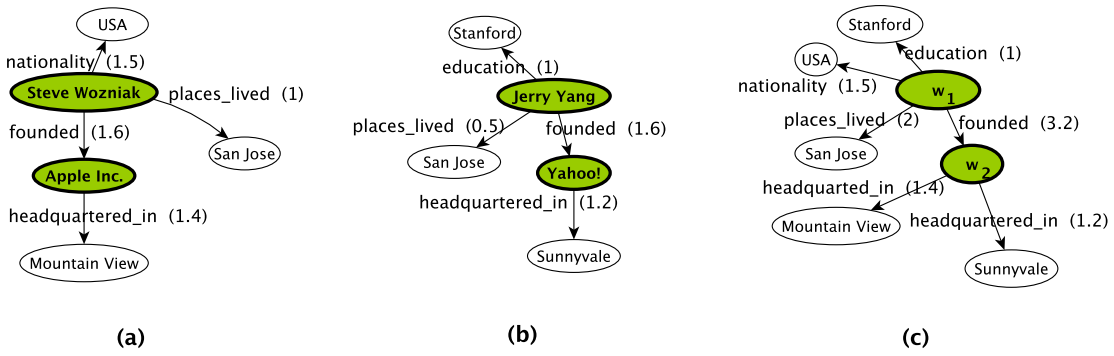


Figure 3.9: Merging Maximum Query Graphs

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}.

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.

Example 4 (Merging Maximum Query Graphs) *Let Figures 3.9 (a) and (b) be the M_{t_i} for query tuples $\langle$Steve Wozniak, Apple Inc.$\rangle$ and $\langle$Jerry Yang, Yahoo!$\rangle$, respectively. Figure 3.9(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.*

3.6 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.

3.6.1 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. 8) 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 5 (Query Lattice and Minimal Query Tree) *Figure 3.10(a) shows a maximum query graph MQG_t, which contains two query entities in shaded circles and five edges*

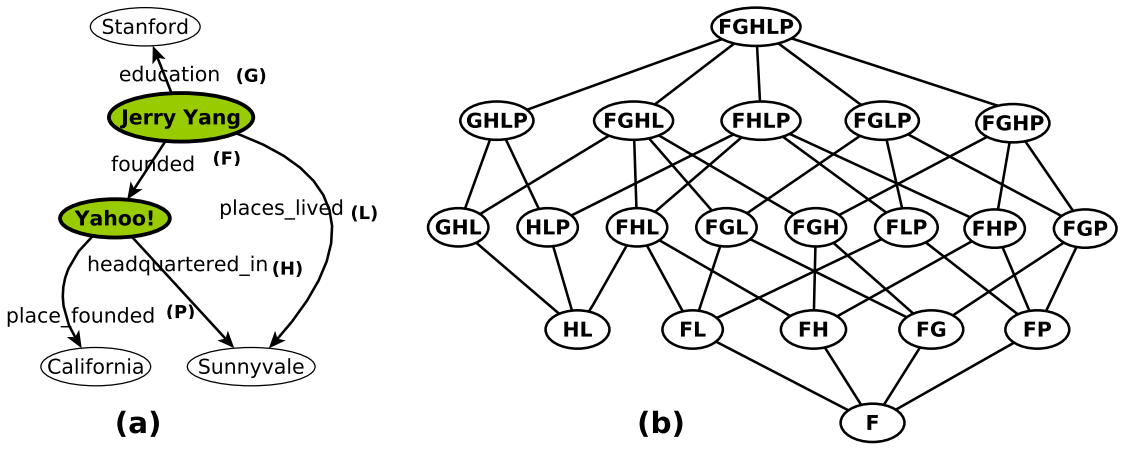


Figure 3.10: Maximum Query Graph and Query Lattice

F, G, H, L, and P. Its corresponding query lattice $\mathcal{L}$ is in Figure 3.10(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 (Section 3.7) 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 Section 3.4.1. To generate all minimal query trees, our method enumerates all distinct spanning trees of M_s by the technique in [30] 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 Section 3.4 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.

3.6.2 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 Equation (3.1) for defining answer tuple score.

$$\begin{aligned} \mathsf{score}_Q(A) &= \mathsf{s\_score}(Q) + \mathsf{c\_score}_Q(A) \\ \mathsf{s\_score}(Q) &= \sum_{e \in E(Q)} \mathsf{w}(e) \\ \mathsf{c\_score}_Q(A) &= \sum_{\substack{e=(u,v) \in E(Q) \\ e'=(f(u),f(v)) \in E(A)}} \mathsf{match}(e, e') \end{aligned} \tag{3.2}$$

In Equation (3.2), $\mathsf{score}_Q(A)$ 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. 9. For instance, among the 6 pairs of matching nodes between Figure 3.6(a) and Figure 3.7(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.

$$\mathsf{match}(e, e')= \begin{cases} \frac{\mathsf{w}(e)}{|E(u)|} & \text{if } u{=}f(u) \\ \frac{\mathsf{w}(e)}{|E(v)|} & \text{if } v{=}f(v) \\ \frac{\mathsf{w}(e)}{min(|E(u)|,|E(v)|)} & \text{if } u{=}f(u), v{=}f(v) \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

3.7 Query Processing

GQBE's query processing component takes MQG_t (Section 3.4) and the query lattice $\mathcal{L}$ (Section 3.6) and finds answer graphs matching the query graphs in $\mathcal{L}$. Before we discuss how $\mathcal{L}$ is evaluated (Section 3.7.2), we introduce the storage model and query plan for processing one query graph (Section 3.7.1).

3.7.1 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 [31]. 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 3.10(a) corresponds to SELECT F.subj, F.obj FROM F,G,H,L,P WHERE F.subj=G.subj 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 $G.subj$ 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.

3.7.2 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.

Algorithm 3: 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 LF ← leaf nodes of L; Terminate ← false;
2  while not Terminate do
3      Q_best ← node with the highest upper-bound score in LF;
4      A_{Q_best} ← evaluate Q_best; (Section 3.7.1)
5      if A_{Q_best} =∅ then
6          prune Q_best and all its ancestors from L;
7          recompute upper-bound scores of nodes in LF; (Alg. 4)
8      else
9          insert Parents(Q_best) into LF;
10     if top-k answer tuples found [Theorem 3] then Terminate←true ;
```

We propose Alg. 3, 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. 3 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).

Complexity Analysis of Alg. 3 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.

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 Equation (3.2) 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 Equation (3.2), 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 Equation (3.2) 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.

3.7.3 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. 3). 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 9, 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 Equation (3.2).*

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 Equation (3.2).

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. 3 adopts as mentioned in Section 3.7.2. 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:

$\mathcal{UF}$=$\{Q |\ Q$ is not pruned, $\nexists Q' \succ Q$ s.t. Q' is not pruned$\}$.

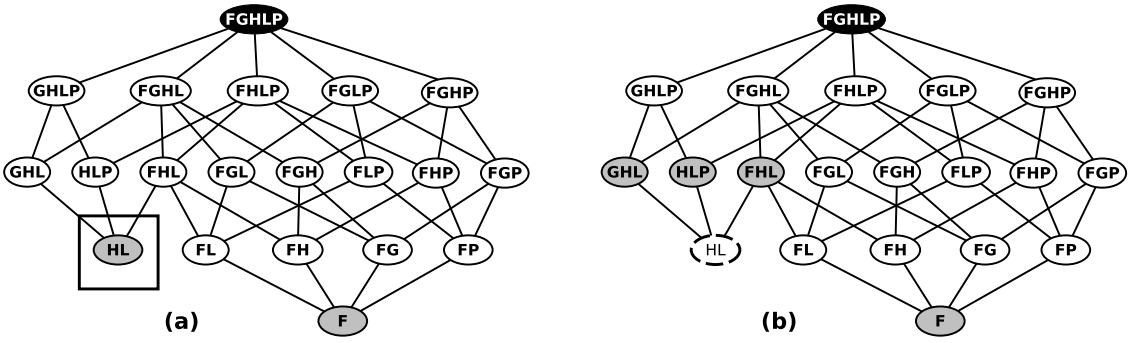


Figure 3.11: Evaluating Lattice in Figure 3.10 (b)

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

$$U(Q) = \max_{Q' \in \mathcal{UB}(Q)} \mathsf{s\_score}(Q') \tag{3.4}$$

Example 6 (Lattice Evaluation) *Consider the lattice in Figure 3.11(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 Figure 3.11(b). The evaluated node* HL *is represented in bold dashed node.*

(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.*

Algorithm 4: 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 ∈ LF do
    NB ← φ; // set of new upper boundary candidates of Q.
    foreach Q' ∈ UB(Q) ∩ UB(Q_best) do
        UB(Q) ← UB(Q) \ {Q'};
        UF ← UF \ {Q'};
        V(Q'') ← V(Q');
        foreach e ∈ E(Q_best) \ E(Q) do
            E(Q'') ← E(Q') \ {e};
            find Q_sub, the weakly-connected component of Q'', containing all query entities;
            NB ← NB ∪ {Q_sub};
    foreach Q_sub ∈ NB do
        if Q_sub ⊀ (any node in UF or NB) then
            UB(Q) ← UB(Q) ∪ {Q_sub}, UF ← UF ∪ {Q_sub};
    recompute U(Q) using Equation (3.4);
```

Based on Property 3, when Q_{best} is evaluated to be a null node, Alg. 3 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. 4) 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. 4). 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. 4, 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. 4 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.*

Example 7 (Recomputing Upper Boundary) *Consider the lattice in Figure 3.12(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 $\mathcal{UF}$. Figure 3.12(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*.*

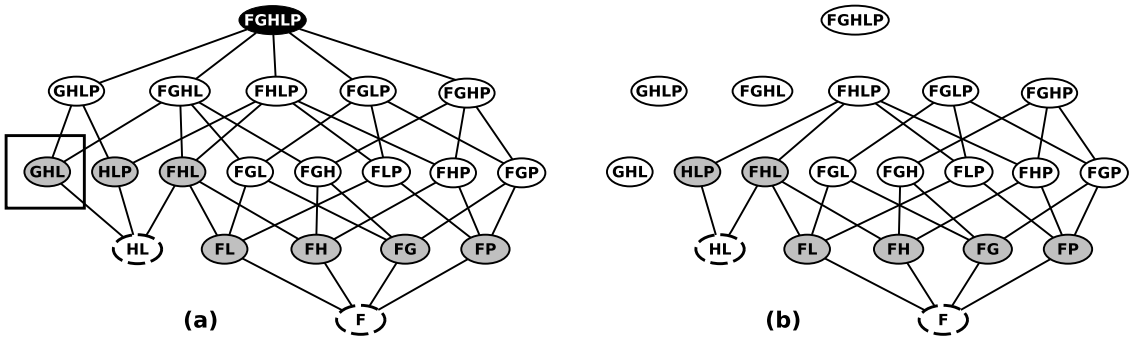


Figure 3.12: Recomputing Upper Boundary of Dirty Node FG

(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. 3. If t_A was also projected from already evaluated nodes, it has a current score. By Equation 3.1, 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 $\mathsf{score}_t(t_k) > U(Q_{best})$.*

Complexity Analysis of Alg. 4

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.

3.8 Edge Weighting Function

The definition of MQG_t (Def. 10) 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

$$\mathsf{w}(e) = \mathsf{ief}(e) \ / \ \mathsf{p}(e) \tag{3.5}$$

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*.

$$\mathsf{ief}(e) = \log\left(|E(G)| \;/\; \#label(e)\right) \tag{3.6}$$

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,

$$\mathsf{p}(e) = |\ \{e'{=}(u', v') \mid label(e){=}label(e'), u'{=}u \vee v'{=}v\}\ | \tag{3.7}$$

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. 2, the weights of edges in H_t are defined by Equation (3.5) 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 Equation (3.2) and Equation (3.3), 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.

$$\mathsf{d}(e) = \min_{v_i \in t} \min_{u,v} \{\mathsf{dist}(u, v_i), \mathsf{dist}(v, v_i)\} \tag{3.8}$$

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

In summary, GQBE uses Equation (3.5) as the definition of $\mathsf{w}(e)$ in weighting edges in H_t. After MQG_t is discovered from H_t by Alg. 2, it uses the following Equation (3.9) as the definition of $\mathsf{w}(e)$ in weighting edges in MQG_t. Equation (3.9) incorporates $\mathsf{d}(e)$ into Equation (3.5). The answer graph scoring functions Equation (3.2) and Equation (3.3) are based on Equation (3.9).

$$\mathsf{w}(e) = \mathsf{ief}(e) \ / \ (\mathsf{p}(e) \times \mathsf{d}^2(e)) \tag{3.9}$$

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. We next discuss a heuristic to prune edges that are deemed unimportant.

3.8.1 Preprocessing: Reduced Neighborhood Graph

Alg. 2 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. 2. The reduced size of H_t not only makes the execution of Alg. 2 more efficient but also helps prevent clearly unimportant edges from getting into MQG_t.

Consider the neighborhood graph H_t in Figure 3.5, based on the data graph excerpt in Figure 1.1. 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:

$IE(v)$=

$\{e \in E(v) \mid \exists v_i \in t, p$ 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 at most d. For example, edge e_1 in Figure 3.5 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 3.5, 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. 2 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 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. 7, 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.*

3.9 Experiments

3.9.1 Setup

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 [3] and DBpedia [1]. 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 labeled *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 [27] and exemplar queries [28] (EQ). We implemented all the methods except EQ. For EQ, queries used in our experiments were provided to the authors of [28] 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* [28] 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 [28], 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 (Section 3.7). 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 3.1. They cover diverse domains, including people, companies, movies, sports, awards, religions, universities and automobiles.

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 3.1: Queries and Ground Truth Table Size

Sample Answers: Table 3.2 only lists the top-3 results found by GQBE for 3 queries (F_1, F_{18}, F_{19}).

3.9.2 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 [32], 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.

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 3.2: Case Study: Top-3 Results for Selected 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.

Figure 3.13 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 Figure 3.13(a) and nDCG at top-25 is over 0.9 as shown in Figure 3.13(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 Figure 3.13(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.

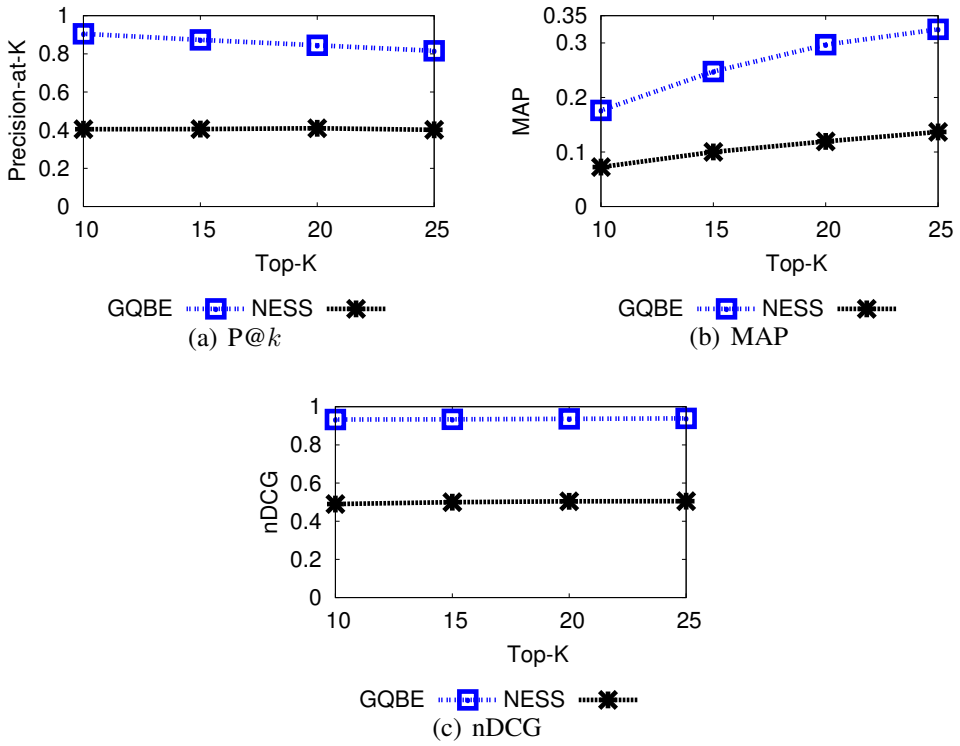


Figure 3.13: Accuracy of GQBE and NESS over all Freebase Queries

Figure 3.14 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 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

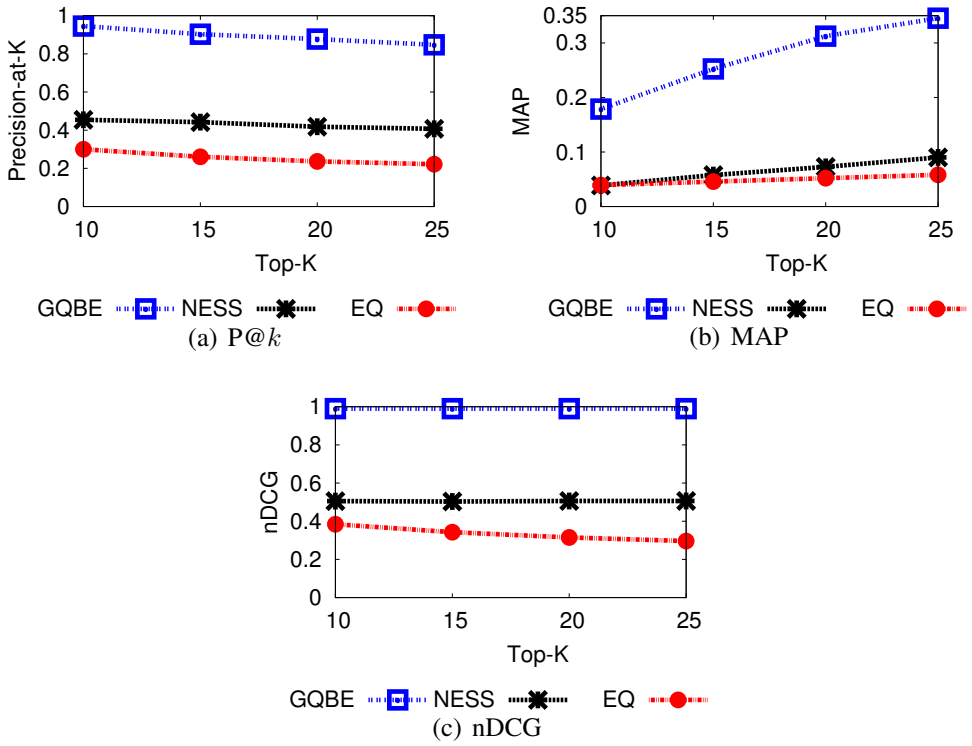


Figure 3.14: Accuracy of GQBE, NESS and EQ over 11 Freebase Queries

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 3.3: Accuracy of GQBE on DBpedia Queries, k=10

the answer tuples. As mentioned in Section 3.3, 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 3.3 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.

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

3.9.3 **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 [33]. PCC is undefined, by definition, when X and/or Y contain all equal values.

Table 3.4 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}

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

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.

3.9.4 Accuracy on Multi-tuple Queries

We investigated the effectiveness of the multi-tuple querying approach (Section 3.5). We experimented with up to three example tuples for each query: Tuple1 refers to the query tuple in Table 3.1, while Tuple2 and Tuple3 are two tuples from its ground truth. Table 3.5 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 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. 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.

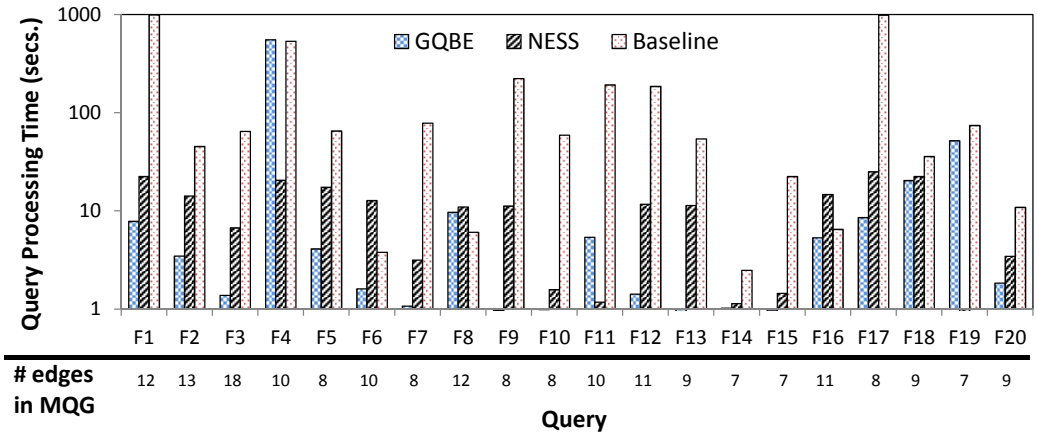


Figure 3.15: Query Processing Time

3.9.5 **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 [28] executed the queries was different from ours. Figure 3.15 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

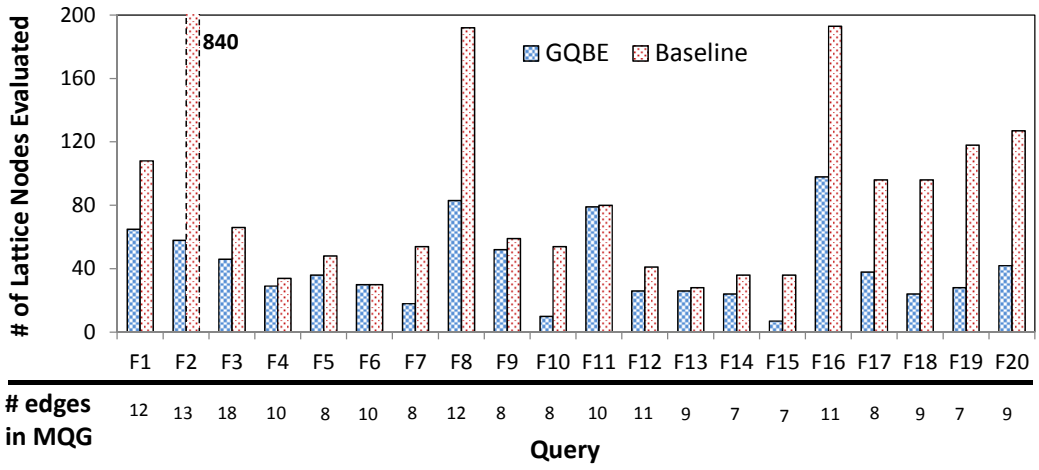


Figure 3.16: Lattice Nodes Evaluated

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 Figure 3.16 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 3.6 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 Figure 3.15, the time taken to discover an MQG in average is comparable to the time spent in evaluating it.

Figure 3.17 shows the distribution of GQBE's query processing time, in logarithmic scale, on the merged MQGs of 2-tuple queries in Figure 3.5, 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

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

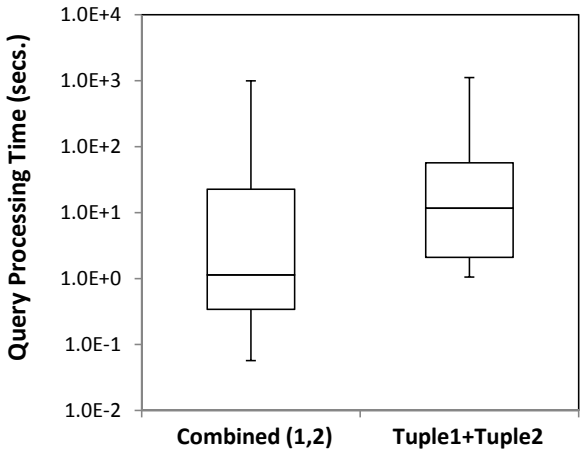


Figure 3.17: Query Processing Time of 2-tuple Queries

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 Figure 3.5. Table 3.6 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.

CHAPTER 4

SYSTEMS DESIGN AND IMPLEMENTATION

We describe the design and implementation of both Orion and GQBE in this chapter. We discuss about the programming languages used for the implementation of both the front-end and back-end of the two systems, along with the design decisions made to help scale the systems to work with the large real-world Freebase data graph.

4.1 Orion Design and Implementation

As described in Chapter 2, Orion has an interactive graphical user interface. Orion's GUI consists of a query canvas that the user constructs the query graph in, as shown in Figure 2.2. Orion offers several features to help schema-agnostic users query large heterogeneous graphs, such as: 1) hierarchically displaying node labels as domains, types and entities (as shown in Figure 2.2 (b)), 2) efficient keyword search to help users search for a node label, and 3) editing a node's assigned value. As mentioned in Section 2.2.2, new candidate edges based on active or passive events, are ranked and presented to the user on the interactive query canvas, as shown in Figure 2.2 (a) and (c) respectively.

Figure 4.1 shows the overall architecture of Orion's implementation. The query canvas is created using Scalable Vector Graphics (SVG), which is an XML-based vector image format ideal for interactive graphics. The rest of the GUI is implemented using Javascript, which interacts with the back-end server using RESTful [34] APIs. The Javascript also captures information regarding the partially constructed query graph, along with the rejected edges in the background. GUI requests corresponding to selecting/editing node values are neither active nor passive events, but are features to improve the usability of Orion. These

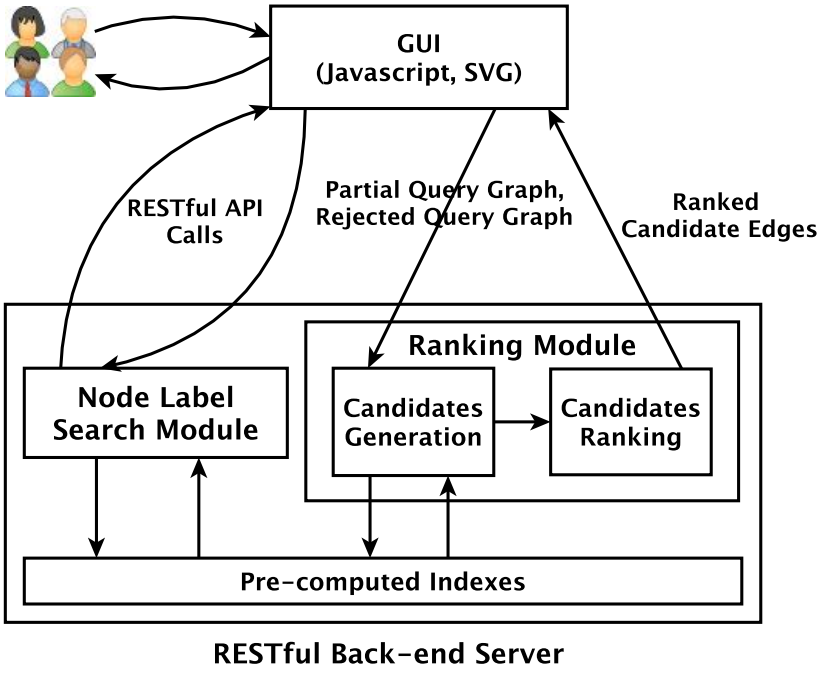


Figure 4.1: Orion System Components

requests are directed to the Node Label Search Module of the back-end. GUI requests corresponding to active or passive events send the partially constructed graph and the rejected graph as input parameters to the Ranking Module of the back-end. The back-end is implemented in core Java. Both the node label search, and the ranking modules use pre-computed indexes to efficiently cater to multiple client requests occurring simultaneously.

Indexes for Node Label Search: File based indexes are utilized to help users efficiently select labels for a node. The pop-up box in Figure 2.2 (b) contains three drop-down lists, one each for domain, type and entity. If a domain value is selected, users can view all types and entities of the selected domain in the type drop-down list and entity drop-down list respectively. If no domain value is selected, all types and entities can be viewed in their corresponding drop-down lists. Similarly, if a type is selected under the type drop-down list, all entities of the selected type can be viewed under the entity drop-down list. One

can also perform a keyword search to find types and entities using the Type Search and Entity Search boxes respectively. The keyword search is also based on the aforementioned filter-based mechanism, i.e., if a domain value is selected, then type keyword-search and entity keyword-search are contained to those specific to the selected domain only, and if a type value is selected, the entity keyword-search is contained to the entities of the selected type only.

In order to help users navigate through the node labels hierarchically using the drop-down lists, we create several index files: 1) a domain-specific comma-separated file which stores a domain ID and type string in each row, such that the type belongs to the corresponding domain, 2) a domain-specific comma-separated file that stores a domain ID and entity string, such that the entity belongs to the corresponding domain, 3) a type-specific comma-separated file that stores the type ID followed by an entity string, such that the entity belongs to the corresponding type. All these files are first sorted by the first ID column, followed by the second string column. We perform binary search on these files over the first column, to list all sorted strings stored in the second column.

We create several inverted index files for efficient keyword search: 1) a type-specific comma-separated file that stores the type string followed by the corresponding type ID, used to search a type based on the keyword specified in the Type Search box, 2) an entity-specific comma-separated file that stores the entity string followed by the corresponding entity ID, used to search an entity based on the keyword specified in the Entity Search box. We maintain five copies of the entity-specific inverted index file. The first file is sorted on the first word of the entity string, the second file sorted on the second word, third file sorted on the third word, fourth file sorted on the fourth word, and the fifth file sorted on the fifth word of the entity string. Each of these files contains only those rows that have sufficient number of words in the corresponding entity string. White spaces are considered to be the delimiter between words. These auxiliary files are created to find an entity label even if the

keyword search string is not at the beginning of the entity label. The keyword search is first performed on the first file, followed by the second, third, fourth and the fifth files, using binary search on the corresponding sorted word column of the entity string. This order of search ensures that the row that matches the keyword search string at the beginning of the label is ranked higher than the match that appears at a later part of the string. The IDs stored in the second column of each of the five files is also used to ensure the same entity string is not matched multiple times when searching across the five files.

We create file-based indexes instead of in-memory indexes for the node label search feature to handle the scale of the underlying Freebase data graph. Storing the string values of around 30 million entities, 5253 types, hundreds of domains, and their corresponding integer IDs easily amounts to several tens of giga-bytes. With less than 30 GB of memory at our disposal, we use these file-based indexes, which work efficiently due to the binary search that is performed to search these files.

Indexes for Ranking Candidate Edges: We create an in-memory index to efficiently find all potential candidate edges given a partially constructed query graph. We pre-process the data graph to find the set of neighboring edges defined in Def. 3, for every vertex type in Freebase data graph. We use a in-memory hash-map to store all neighboring edges corresponding to each vertex type. The vertex types form the keys of the hash-map, and the value, which represents a set of edges, is stored in a hash-set. This helps us to efficiently find all candidate edges for a given vertex type. This is also useful in passive mode, where the user draws an edge between two specific nodes. If the two nodes are vertex types, an intersection between their corresponding hash-sets of neighboring edges results in the potential candidate edges to be ranked.

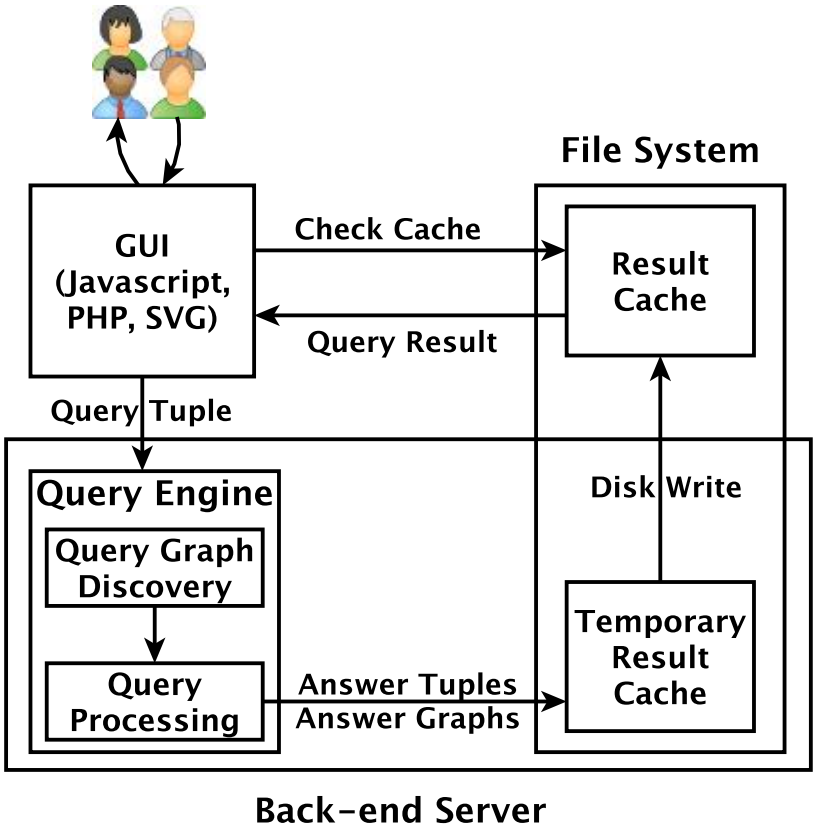


Figure 4.2: GQBE System Components

4.2 GQBE Design and Implementation

As described in Chapter 3, GQBE users can query large heterogeneous graphs using example tuples. The input interface of GQBE is shown in Figure 3.2, where a user enters the example query tuple in the provided search box. Users can also provide multiple example tuples as input to the system, using the '+' button before the search box. The results are displayed as a table in the left, and the maximum query graph automatically discovered by GQBE is displayed on the right, as shown in Figure 3.4. Answer graph corresponding to an answer tuple can also be viewed using the Render Graphs button.

Figure 4.2 shows the overall architecture of GQBE's implementation. The GUI is implemented using Javascript and PHP, while the MQG and answer graphs are rendered

using SVG. The keyword auto-complete functionality in GQBE's search box is powered by Freebase API, and the back-end server is implemented in core Java. Given an input query (one or more example query tuples), the PHP code validates the input, and first checks the Result Cache for the result. If the result is found in the cache, it fetches it directly from there and displays it. If not, the query tuple is passed as a parameter to the Query Engine in the back-end server, which processes the query request.

Query Engine: The query engine has two main components: the Query Graph Discovery and the Query Processing modules. We store the cleaned Freebase data graph considered in GQBE in an in-memory two-level hash-map that requires 18 GB of memory. The query graph discovery module finds the MQG as described in Section 3.4, and the approximately matching answer graphs are found using the techniques described in Section 3.7. The query processing module evaluates a query lattice to find the top-k answer tuples. In order to evaluate the lattice efficiently, we maintain two in-memory hash-maps for each edge type in Freebase. For all instances $e = (subj, obj)$ of an edge type, of its two corresponding hash-maps, one stores *subj* and *obj* as the keys and values respectively, while the other hash-map stores *obj* and *subj* as the keys and values respectively. These are used to perform quick joins while evaluating the query lattice, as described in Section 3.7.1. Every time a lattice node is evaluated, the result corresponding to that lattice node is materialized into a file for future use. We create temporary binary files to materialize these intermediate results for efficient disk-based reading and writing.

The answer tuples and their corresponding answer graphs obtained after the lattice evaluation are all stored in a temporary file. These files are then read by the PHP code of the GUI, which creates a JSON object that is returned back to the GUI's Javascript code for rendering the results. The result files are then copied to the Result Cache, which is used to return results quickly, if the same query is made by another user.

CHAPTER 5

RELATED WORK

5.1 Query Specification

Substantial progress has been made on query mechanisms that help users construct query graphs or even do not require explicit query graphs. Such mechanisms include keyword search (e.g., [35]), keyword-based query formulation [36], natural language questions [37], interactive and form-based query formulation [38, 39], and visual interface for query graph construction [16, 17]. Little has been done on comparison across these graph query mechanisms. While a usability comparison of these mechanisms and the querying paradigms proposed in Orion and GQBE is beyond the scope of this dissertation, we note that they all have pros and cons and thus complement each other.

The paradigm of *query-by-example* (QBE) has a long history in relational databases [10]. Its simplicity and improved user productivity make QBE an influential database query language. By proposing to query knowledge graphs by example tuples in GQBE, 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.

Keyword-based methods are attractive mainly due to the success they have enjoyed over textual data. But using keyword-based methods over graph data is less intuitive. For instance, a Silicon Valley business analyst interested in finding entrepreneurs who founded technology companies head-quartered in California has to articulate query keywords, "technology companies head-quartered in California and their founders". Not only may the analyst find it challenging to clearly articulate the query, but also a query system might not

return accurate answers, since it is non-trivial to precisely separate these keywords and correctly match them with entities, entity types and relationships. This has been verified through our own experience on a keyword-based system adapted from SPARK [40]. In contrast, a GQBE user only needs to know the names of some entities in example tuples, without having to specify how exactly the entities are related. On the other hand, keyword-based querying is more adequate when a user does not know a few sample answers with respect to her query.

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 [41] 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 [42], 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. [43] 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. [44] provides a theoretical aspect of the example-driven query specification problem. Users are provided with answer tuples, and feedback on the relevance of each answer tuple is used to refine the query. This work only deals with a special class of quantified boolean queries called *qhorn*.

The goal of *set expansion* is to grow a set of objects starting from seed objects. Example systems include [45, 46, 47, 48], and the now defunct Google Sets and Squared services (`http://en.wikipedia.org/wiki/List_of_Google_products`). Chang et al. [49] 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 [49], do not operate on data

graphs. Instead, they find existing answers within structures such as HTML tables and lists. Further, except Google Squared and [48], they all take a set of individual entities as input. Wang and Cohen developed the SEAL system that uses random walk to rank the candidate entities for set expansion [45, 46, 47]. He et. al. proposed SEISA, a system that ranks the candidate set as a whole based on relevance and coherence [50]. Ghahramani et al. used the Bayesian inference for set expansion [51]. Later, Jindal et. al. proposed an inference method that also considers negative examples [52]. The problem of finding the top-k correlated terms given a set of homogeneous entities in an information network was studied in [49]. GQBE is different from these set expansion works, since we allow the users to enter multiple heterogeneous entities in a seed tuple. We then capture the relationships between the entities in a seed tuple, and identify a ranked list of other tuples with similar entities and relationships. 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 [53, 54] 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 [53, 54] have many different types of entities and relationships, but the paths discovered by their techniques only connect the input nodes. REX [54] has the further limitation of allowing only two query entities. Differently, the maximum query graph 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.

The concept of *exemplar queries* proposed in [28] is similar to the querying paradigm proposed in GQBE. However, these two studies are different in fundamental ways. *First*,

[28] does not provide a definitive way of discovering query graph given an exemplar query tuple, while we define a heuristic-based approach to identify a maximum query graph that tries to capture the user intent. Our detailed experimental analyses attest that the query graph has a direct impact on the quality of the results found. *Second*, the query processing algorithm in [28] mandates all answer graphs to be edge preserving isomorphic matches to the query graph for the example tuple. This precludes their system from finding approximate answers to the query graph, which is evident in our experiments where [28] suffers when compared to GQBE. We also observe that in several cases where query graphs are large, there might not be any edge preserving isomorphic matches. *Lastly*, GQBE allows multiple example tuples which helps users to better communicate their intent, while [28] provides no such provision.

Recently, Yu et. al. proposed a query-driven graph querying method [42]. However, they assume the existence of a fixed schema, which is not always available for entity-relationship graphs. The absence of schema makes the problem addressed by GQBE very challenging.

5.2 Visual Query Formulation

The unprecedented proliferation of linked data and large, heterogeneous graphs has sparked extensive interest in building knowledge-intensive applications. The usability challenges in building such applications is widely recognized—declarative query languages such as SPARQL present a steep learning curve, as forming queries requires expertise in these languages and knowledge of data schema. To tackle the challenges, a number of alternate querying paradigms for graph data have been proposed recently, including keyword search [55, 56], query-by-example [6, 7, 43, 57], natural language query [58], and faceted browsing [59, 60, 61].

Visual query builders [16, 17, 18, 19, 20] provide an intuitive and simple approach to query formulation. Most of these systems deal with querying a graph database and not a single large graph, except [20, 16]. Firstly, it is unclear how to directly apply the techniques proposed by systems that deal with graph databases to a single large graph. This is because, their solutions work best on a data model with many small graphs, rather than a single large graph. Secondly, these systems do not assist the user in query formulation by automatically suggesting the new top-k relevant edges.

QUBLE [20] and GRAPHITE [16] provide visual query interfaces for querying a single large graph. But, they focus on efficient query processing, and only facilitate query graph formulation by giving options to quickly draw various components of the query graph. Instead of recommending query components that a user might be interested in, they alphabetically list all possible options for node labels (which may be extended to edge labels similarly). They also deal with smaller data graphs. For instance, the graph considered by QUBLE contains only around 10 thousand nodes with 300 distinct node labels, and they do not consider edge labels. Orion, on the other hand, considers large graphs such as Freebase, which has over 30 million distinct node labels and 5 thousand distinct edge types. With such large graphs, it is impractical to expect users to browse through all options alphabetically to select the most appropriate edge to add to a query graph. Ranking these edges by their relevance to the user's query intent is a necessity, a feature offered in Orion.

5.3 Query Graph Processing

There are many studies on approximate/inexact subgraph matching in large graphs, e.g., G-Ray [62], TALE [63] and NESS [27]. 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 [27]) 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 [64] is similar to GQBE's query lattice, the scoring mechanism of their relaxed queries is different and depends on XML-based relaxations.

The subgraph matching problem identifies all the occurrences of a query graph in the target graph. In bio-informatics, exact and approximate *subgraph matching* have been extensively studied, e.g., PathBlast [65], SAGA [66], NetAlign [67], IsoRank [68]. There have been significant studies on inexact subgraph matching in large graphs. Tong et al. [62] proposed the best-effort pattern matching, which aims to maintain the shape of the query. Tian et al. [63] proposed an approximate subgraph matching tool, called TALE, with efficient indexing. There are other works on inexact subgraph matching. An incomplete list (see [69] for surveys) includes homomorphism based subgraph matching [70], edge-edit-distance based subgraph indexing [71], subgraph matching in billion node graphs [72], regular expression based graph pattern matching [73], unbalanced ontology matching [74], sample-driven schema matching [75], bisimulation-based graph pattern matching [76], and neighborhood similarity based graph querying [27]. The approximate query processing module of GQBE is different from all these works, because we require to match only the query edge labels, but query node labels need not be matched. This is equivalent to matching of a query graph with all unlabeled nodes, and thereby increases the complexity of the problem significantly.

CHAPTER 6

FUTURE DIRECTIONS AND CONCLUSIONS

6.1 Future Directions

In this dissertation we presented a framework to improve the usability of query systems for large ultra-heterogeneous graphs. We presented two novel methods to help schema-agnostic users specify query intents easily. We also presented efficient ways to find approximately matching answer graphs to a given query graph. Orion and GQBE are capable of helping users query Freebase, a large real-world knowledge graph. They are designed to help users query other such knowledge graphs too, so long as the data graph is preprocessed to follow the specified data model. Nevertheless, we believe several future directions of research can be pursued to improve the effectiveness of this framework.

Figure 6.1 shows an extension of the dissertation framework shown in Figure 1.3. Finding a user's query intent with as little information as possible is a difficult problem. One possible way to alleviate this problem is to seek feedback from the user regarding the relevance of the answers presented, using the *Feedback Module* shown in Figure 6.1. For Orion, user feedback can be obtained for the answer graphs of the query graph constructed. Since the user is unaware of the underlying schema, the user may not be completely certain of the exact edges and nodes to add in the query graph. The user's feedback regarding the relevance of answer graphs may be useful in finding and ranking candidate edges better. For GQBE, a user can mark the relevance of the top-k answer tuples obtained, which can then be used to refine the MQG that was automatically discovered. The user feedback may be useful in better capturing the user's query intent.

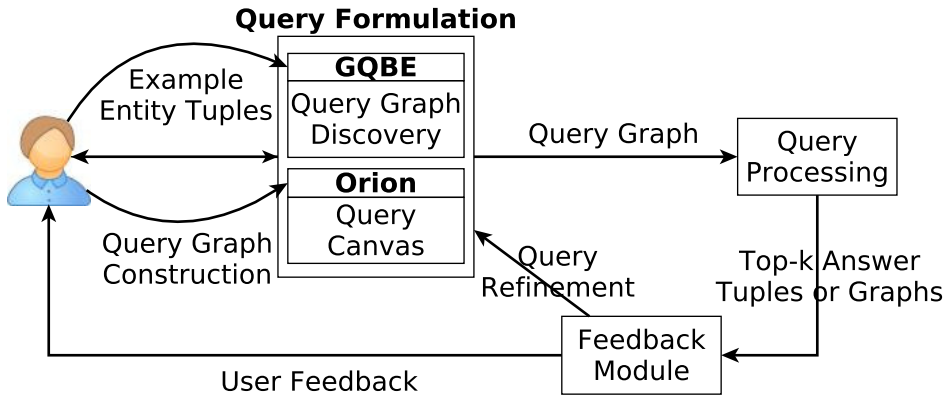


Figure 6.1: Framework for Querying Heterogeneous Graphs, with Future Directions

The current version of Orion is not integrated with the query processing module of Figure 6.1. This is an important problem to address to make Orion available for the public. Another interesting problem to address with regard to efficient query processing in Orion, is to find exactly matching answer graphs to the partial query graph at every intermediate step of the query construction process. This would not only help schema-agnostic users better understand the query graph they are constructing, but can also be used by Orion's edge suggestion algorithm to better rank the candidate edges.

GQBE has two important components, and potential bottlenecks. The first is to capture the users' query intent correctly in the MQG. This is especially difficult since the only context information GQBE receives is the entities of an example tuple. The other component is to efficiently process the query graph to find the top-k answer tuples. One of the avenues to better capture an user's query intent is the aforementioned method of obtaining user feedback regarding the relevance of answer tuples. The other avenue is to weight edges based on query logs that capture users' interests, rather than only the data graph statistics as used in GQBE. GQBE can be further improved by having a more efficient query processing module. GQBE's current query processing module does not use a distributed environment to evaluate the query lattice. The edge labels form an integral part of finding good matches

in GQBE, which has to be taken into consideration while designing a distributed algorithm for GQBE's approximate query processing problem. We believe that a distributed algorithm to find approximate matches, based on edge-preserving structural isomorphism, is a challenging and promising direction to pursue to improve the system's response time.

6.2 Conclusions

The principal contribution of this dissertation is improving the usability of query systems for large ultra-heterogeneous graphs. Querying such graphs is a difficult task, since existing graph query systems and paradigms are either difficult to use or cannot be used to formulate exact queries. In this dissertation, as an initial step towards achieving our goal, we propose two novel, first-of-its-kind systems: 1) Orion, an interactive visual interface that helps users construct query graphs by automatically suggesting relevant edges to add in active mode, or by ranking labels for explicitly added edge types in passive mode, and 2) GQBE, that supports a new querying paradigm that queries such graphs by example entity tuples, without the user having to form query graphs.

The importance of ultra-heterogeneous graphs cannot be stressed enough in the future, since they are ubiquitous, content-rich and can be an invaluable source of information. We believe that the techniques presented in this dissertation is a successful initial step towards helping schema-agnostic users easily query large ultra-heterogeneous graphs. We also hope that this dissertation would inspire new threads of research in the area of query systems for large graphs, leading to more sophisticated systems that can better capture users' query intents.

REFERENCES

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A Nucleus for a Web of Open Data," in *Proceedings of the 6th International The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference (ISWC/ASWC)*, 2007, pp. 722–735.

[2] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A Core of Semantic Knowledge," in *Proceedings of the 16th International Conference on World Wide Web (WWW)*, 2007, pp. 697–706.

[3] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008, pp. 1247–1250.

[4] W. Wu, H. Li, H. Wang, and K. Q. Zhu, "Probase: A Probabilistic Taxonomy for Text Understanding," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012, pp. 481–492.

[5] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu, "Making Database Systems Usable," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007, pp. 13–24.

[6] N. Jayaram, M. Gupta, A. Khan, C. Li, X. Yan, and R. Elmasri, "GQBE: Querying Knowledge Graphs by Example Entity Tuples," in *IEEE 30th International Conference on Data Engineering (ICDE)*, 2014, pp. 1250–1253.

[7] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri, "Querying Knowledge Graphs by Example Entity Tuples," *In IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 27, no. 10, pp. 2797–2811, 2015.

[8] P. A. Bernstein *et al.*, "Future Directions in DBMS Research - The Laguna Beach Participants," *SIGMOD Record*, pp. 17–26, 1989.

[9] D. Abadi *et al.*, "The Beckman Report on Database Research," *SIGMOD Record*, pp. 61–70, 2014.

[10] M. M. Zloof, "Query-by-example: The Invocation and Definition of Tables and Forms," in *Proceedings of the 1st International Conference on Very Large Data Bases (PVLDB)*, 1975, pp. 1–24.

[11] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, "A Graphical Query Language Supporting Recursion," *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, vol. 16, no. 3, pp. 323–330, 1987.

[12] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou, "Interactive Query Formulation over Web Service-accessed Sources," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006, pp. 253–264.

[13] D. Braga, A. Campi, and S. Ceri, "XQBE (XQuery By Example): A Visual Interface to the Standard XML Query Language," *Proceedings of the ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 398–443, 2005.

[14] M. Petropoulos, Y. Papakonstantinou, and V. Vassalos, "Graphical Query Interfaces for Semistructured Data: The QURSED System," *Proceedings of the ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 2, pp. 390–438, 2005.

[15] H. Blau, N. Immerman, and D. Jensen, "A Visual Query Language for Relational Knowledge Discovery," Tech. Rep., 2001.

[16] 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 *Workshops Proceedings*

of the 8th IEEE International Conference on Data Mining (ICDM), 2008, pp. 963–966.

[17] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010, pp. 111–122.

[18] H. H. Hung, S. S Bhowmick, B. Q. Truong, B. Choi, and S. Zhou, "QUBLE: Blending Visual Subgraph Query Formulation with Query Processing on Large Networks," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013, pp. 1097–1100.

[19] S. S. Bhowmick, B. Choi, and S. Zhou, "VOGUE: Towards A Visual Interaction-aware Graph Query Processing Framework," in *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[20] H. H. Hung, S. S. Bhowmick, B. Q. Truong, B. Choi, and S. Zhou, "QUBLE: Towards Blending Interactive Visual Subgraph Search Queries on Large Networks," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 23, no. 3, pp. 401–426, 2014.

[21] S. S. Bhowmick, "DB $\bowtie$ HCI: Towards Bridging the Chasm between Graph Data Management and HCI," in *Proceedings of the 25th International Conference on Database and Expert Systems Applications, (DEXA)*, 2014, pp. 1–11.

[22] N. Jayaram, S. Goyal, and C. Li, "VIIQ: Auto-suggestion Enabled Visual Interface for Interactive Graph Query Formulation," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1940–1943, 2015.

[23] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "DBpedia SPARQL Benchmark: Performance Assessment with Real Queries on Real Data," in *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I (ISWC)*, 2011, pp. 454–469.

[24] B. Liu, W. Hsu, and Y. Ma, "Integrating Classification and Association Rule Mining," in *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD)*, 1998, pp. 80–86.

[25] X. Su and T. M. Khoshgoftaar, "A Survey of Collaborative Filtering Techniques," *Advances in Artificial Intelligence (AAI)*, vol. 2009, pp. 421 425:1–421 425:19, 2009.

[26] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[27] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao, "Neighborhood Based Fast Graph Search in Large Networks," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011, pp. 901–912.

[28] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar Queries: Give Me an Example of What You Need," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 5, pp. 365–376, 2014.

[29] Z. Li, S. Zhang, X. Zhang, and L. Chen, "Exploring the Constrained Maximum Edge-weight Connected Graph Problem," *Acta Mathematicae Applicatae Sinica*, vol. 25, pp. 697–708, 2009.

[30] H. N. Gabow and E. W. Myers, "Finding All Spanning Trees of Directed and Undirected Graphs," *SIAM Journal of Computing (SICOMP)*, vol. 7, no. 3, pp. 280–287, 1978.

[31] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," in *Proceedings of the 33rd International Conference on Very Large Data Bases (PVLDB)*, 2007, pp. 411–422.

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

[33] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 1988.

[34] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, 2000.

[35] M. Kargar and A. An, "Keyword Search in Graphs: Finding R-cliques," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, no. 10, pp. 681–692, 2011.

[36] J. Pound, I. F. Ilyas, and G. Weddell, "Expressive and Flexible Access to Web-extracted Data: A Keyword-based Structured Query Language," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010, pp. 423–434.

[37] 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 *Proceedings of the 21st International Conference on World Wide Web (WWW)*, 2012, pp. 445–449.

[38] E. Demidova, X. Zhou, and W. Nejdl, "FreeQ: An Interactive Query Interface for Freebase," in *Proceedings of the 21st International Conference on World Wide Web (WWW)*, 2012, pp. 325–328.

[39] M. Jarrar and M. D. Dikaiakos, "A Query Formulation Language for the Data Web," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, no. 5, pp. 783–798, 2012.

[40] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: Top-k Keyword Query in Relational Databases," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007, pp. 115–126.

[41] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu, "PathSim: Meta Path-Based Top-K Similarity Search in Heterogeneous Information Networks," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, pp. 992–1003, 2011.

[42] X. Yu, Y. Sun, P. Zhao, and J. Han, "Query-driven Discovery of Semantically Similar Substructures in Heterogeneous Networks," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2012, pp. 1500–1503.

[43] L. Lim, H. Wang, and M. Wang, "Semantic Queries by Example," in *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, 2013, pp. 347–358.

[44] A. Abouzied, D. Angluin, C. Papadimitriou, J. M. Hellerstein, and A. Silberschatz, "Learning and Verifying Quantified Boolean Queries by Example," in *Proceedings of the 32nd symposium on Principles of database systems (PODS)*, 2013, pp. 49–60.

[45] R. C. Wang and W. W. Cohen, "Language-Independent Set Expansion of Named Entities Using the Web," in *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining (ICDM)*, 2007, pp. 342–350.

[46] ——, "Iterative Set Expansion of Named Entities Using the Web," in *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining (ICDM)*, 2008, pp. 1091–1096.

[47] ——, "Character-level Analysis of Semi-Structured Documents for Set Expansion," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, (EMNLP)*, 2009, pp. 1503–1512.

[48] R. Gupta and S. Sarawagi, "Answering Table Augmentation Queries from Unstructured Lists on the Web," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, pp. 289–300, 2009.

[49] L. Chang, J. X. Yu, L. Qin, Y. Zhu, and H. Wang, "Finding Information Nebula over Large Networks," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM)*, 2011, pp. 1465–1474.

[50] Y. He and D. Xin, "SEISA: Set Expansion by Iterative Similarity Aggregation," in *Proceedings of the 20th International Conference on World Wide Web (WWW)*, 2011, pp. 427–436.

[51] S. Verma and E. R. H. Jr., "Coupled Bayesian Sets Algorithm for Semi-supervised Learning and Information Extraction," in *European Conference on Machine Learning*

and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), vol. 7524, 2012, pp. 307–322.

[52] P. Jindal and D. Roth, "Learning from Negative Examples in Set-Expansion," in *11th IEEE International Conference on Data Mining (ICDM)*, 2011, pp. 1110–1115.

[53] G. Kasneci, S. Elbassuoni, and G. Weikum, "MING: Mining Informative Entity Relationship Subgraphs," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM)*, 2009, pp. 1653–1656.

[54] L. Fang, A. D. Sarma, C. Yu, and P. Bohannon, "REX: Explaining Relationships Between Entity Pairs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 3, pp. 241–252, 2011.

[55] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: Ranked Keyword Searches on Graphs," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007, pp. 305–316.

[56] E. Demidova, X. Zhou, and W. Nejdl, "Efficient Query Construction for Large Scale Data," in *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2013, pp. 573–582.

[57] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar Queries: Give Me an Example of What You Need," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 7, no. 5, pp. 365–376, 2014.

[58] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum, "Natural Language Questions for the Web of Data," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2012, pp. 379–390.

[59] M. Arenas, B. Cuenca Grau, E. Kharlamov, S. Marciuska, and D. Zheleznyakov, "Faceted Search over Ontology-Enhanced RDF Data," in *Proceedings of the 23rd*

ACM International Conference on Conference on Information and Knowledge Management (CIKM), 2014, pp. 939–948.

[60] E. Oren, R. Delbru, and S. Decker, "Extending Faceted Navigation for RDF Data," in *5th International Semantic Web Conference, Athens, GA, USA, November 5-9, 2006*, 2006.

[61] M. Hildebrand, J. van Ossenbruggen, and L. Hardman, "/facet: A Browser for Heterogeneous Semantic Web Repositories," in *Proceedings of the 5th International Semantic Web Conference, (ISWC)*, 2006, pp. 272–285.

[62] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast Best-effort Pattern Matching in Large Attributed Graphs," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2007, pp. 737–746.

[63] Y. Tian and J. M. Patel, "TALE: A Tool for Approximate Large Graph Matching," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE)*, 2008, pp. 963–972.

[64] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman, "Structure and Content Scoring for XML," in *Proceedings of the 31st International Conference on Very Large Data Bases (PVLDB)*, 2005, pp. 361–372.

[65] B. P. Kelley, B. Yuan, F. Lewitter, R. Sharan, B. R. Stockwell, and T. Ideker, "PathBLAST: A Tool for Alignment of Protein Interaction Networks," *Nucleic Acids Research*, vol. 32, pp. 83–88, 2004.

[66] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel, "SAGA: A Subgraph Matching Tool for Biological Graphs," *Bioinformatics/computer Applications in The Biosciences*, vol. 23, pp. 232–239, 2007.

[67] Z. Liang, M. Xu, M. Teng, and L. Niu, "NetAlign: A Web-based Tool for Comparison of Protein Interaction Networks," *Bioinformatics*, vol. 22, no. 17, pp. 2175–2177, 2006.

[68] R. Singh, J. Xu, and B. Berger, "Global Alignment of Multiple Protein Interaction Networks with Application to Functional Orthology Detection," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 105, no. 35, pp. 12 763–12 768, 2008.

[69] B. Gallagher, "Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching," *Association for the Advancement of Artificial Intelligence (AAAI)*, 2006.

[70] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, "Graph Homomorphism Revisited for Graph Matching," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, pp. 1161–1172, 2010.

[71] S. Zhang, J. Yang, and W. Jin, "SAPPER: Subgraph Indexing and Approximate Matching in Large Graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 1185–1194, 2010.

[72] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient Subgraph Matching on Billion Node Graphs," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 9, pp. 788–799, 2012.

[73] P. Barceló, L. Libkin, and J. L. Reutter, "Querying Graph Patterns," in *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2011, pp. 199–210.

[74] Q. Zhong, H. Li, J. Li, G. Xie, J. Tang, L. Zhou, and Y. Pan, "A Gauss Function Based Approach for Unbalanced Ontology Matching," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009, pp. 669–680.

[75] L. Qian, M. J. Cafarella, and H. V. Jagadish, "Sample-driven Schema Mapping," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2012, pp. 73–84.

[76] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing Topology in Graph Pattern Matching," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 4, pp. 310–321, 2011.

BIOGRAPHICAL STATEMENT

Nandish Jayaram was born in Bengaluru (Bangalore), India. He received his Bachelor's degree in Computer Science and Engineering from Visveswaraiah Technological University, India, in 2005. He then received his Masters' degree in Information Technology from the International Institute of Information Technology, Bangalore, in 2007. He worked as a senior software engineer for three years at Novell, before starting his doctoral research at the University of Texas at Arlington in 2010. His current research interests include searching and querying large graphs, graph query formulation, graph query processing, and machine learning. During his doctoral research, he visited IBM Research Lab as a research intern in summer 2013. He also interned at HP Labs during the summer and fall of 2014. He has also served as a Graduate Teaching Assistant and Graduate Research Assistant in the department of Computer Science and Engineering at the University of Texas at Arlington from 2010 to 2016. He is the recipient of VLDB 2015 Prof. Ram Kumar Memorial Fellowship. He is also the recipient of the STEM Fellowship from 2013 till 2016, and the Extended GTA Fellowship from 2010 till 2013 at the University of Texas at Arlington. Following the completion of his Ph.D., Nandish Jayaram will begin working at Pivotal as a Member of Technical Staff 3 in Palo Alto, USA.