Orion: Enabling Suggestions in a Visual Query Builder for Ultra-Heterogeneous Graphs

Nandish Jayaram Rohit Bhoopalam Chengkai Li Vassilis Athitsos
University of Texas at Arlington

ABSTRACT

The database community has long recognized the importance of graphical query interface to the usability of data management systems. Yet, relatively less has been done. We present Orion, a visual interface for querying ultra-heterogeneous graphs. It iteratively assists users in query graph construction by making suggestions via machine learning methods. In its active mode, Orion automatically suggests top-k edges to be added to a query graph. 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 will match the user's 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 the 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 class association rules and recommendation systems based on singular value decomposition. On average, RDP required 40 suggestions to correctly reach a target query graph (using only its active mode of suggestion) while other methods required 1.5–4 times as many suggestions.

1. INTRODUCTION

The database community has long recognized the importance of graphical query interfaces to the usability of data management systems [5]. Yet, relatively less has been done and there remains a pressing need for investigation in this area [17, 1]. Nevertheless, a few important ideas (e.g., Query-By-Example [35]) and systems (e.g., Microsoft SQL Query Builder) have been developed for querying relational databases [4], web services [28] and XML [10, 29].

For querying graph data, existing systems [8, 12, 22, 21, 7, 16] 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 [12, 22, 21, 7, 16], 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 [8, 12, 22, 21, 7, 16] 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 [9], DBpedia [3], YAGO [32], Probase [33], 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 [6].

This paper 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 [18] was based on a subset of the ideas in this paper.

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 intent, without being triggered by any user actions. Figure 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

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

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 edge, the suggested edge types are ranked based on their relevance to the user's query intent. Figure 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. 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(b), where type PERSON is chosen as the label for the node.

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 4.2.2 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 4.1 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 [25] 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 5 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 5.

We summarize the contributions of this paper 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%.
- 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. RELATED WORK

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 are 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 [14, 13], query-by-example [19, 20, 23, 26], natural language query [34], and faceted browsing [2, 27, 15].

Visual query builders [12, 30, 22, 21, 7, 16] 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 [16, 12, 30]. 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 [16], GRAPHITE [12] and [30] 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

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

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 types, and they do not consider edge types. Orion, on the other hand, considers large graphs such as Freebase, which has over 30 million distinct node types 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, for which Orion is designed.

3. SYSTEM OVERVIEW

3.1 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 $\text{vtype}(v) \subseteq T_V$. 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* $\text{etype}(e) \in T_E$.

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$\in$ $\text{vtype}(v_i)$ and FILM $\in \text{vtype}(v_j)$.

Given a data graph, users can specify their query intent through query graphs. The concept of query graph is in Definition 1. 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 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 1 finds all Harvard educated film actors who starred in films featuring Harvard. In Figure 1 and other query graphs in this paper, 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:

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

3.2 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.

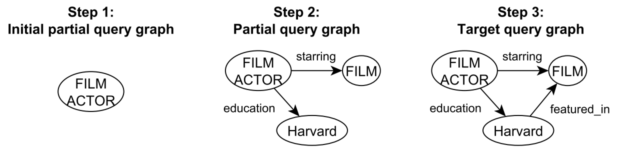


Figure 1: Example Partial and Target Query Graphs

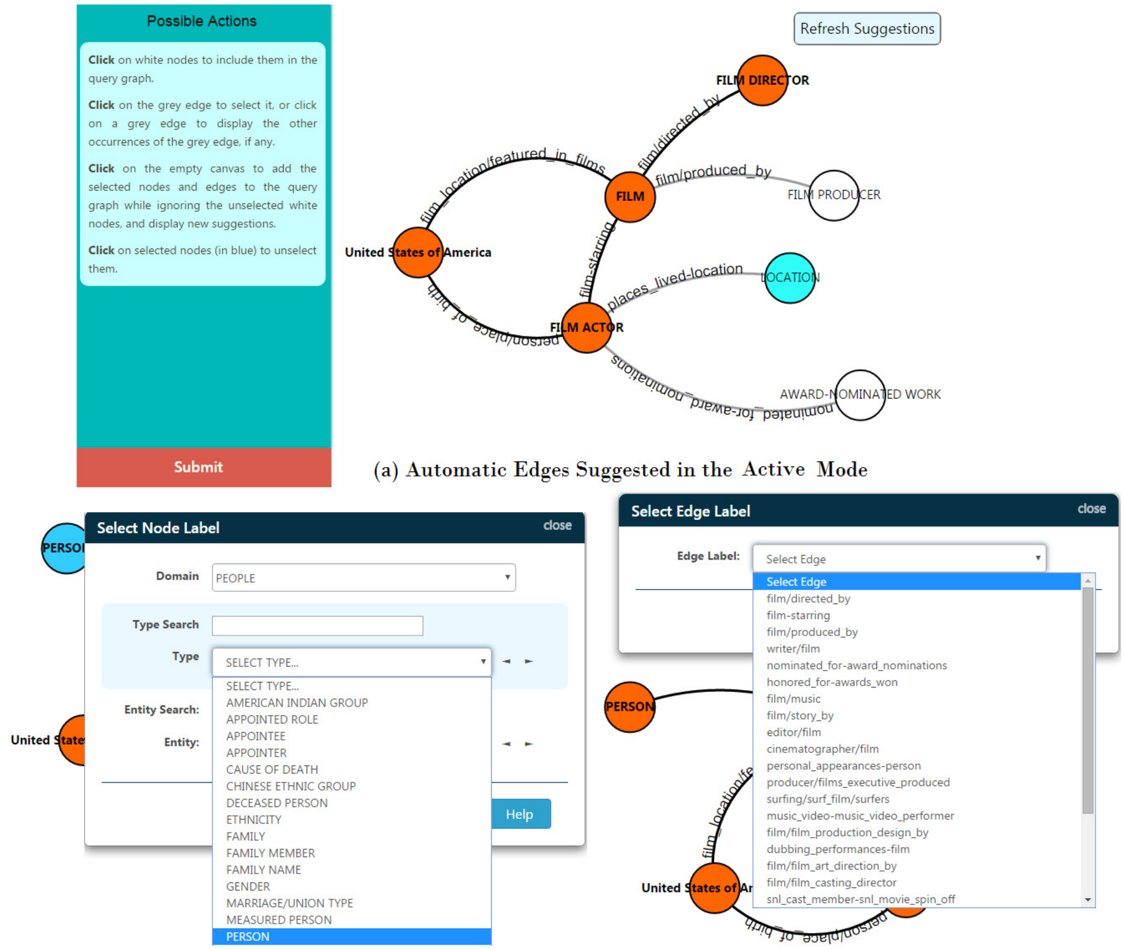
Figure 1 shows an example sequence of steps to construct a query graph. The user starts by forming the initial partial query graph G_p consisting of a single node. Step 1 in Figure 1 shows one such G_p with a node of type FILM ACTOR. New edges are then suggested to the user, who can choose to accept some of the suggestions. For instance, step 2 in Figure 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 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(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(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(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(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(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(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(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(b), one for selecting

[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.


Possible Actions

Click on white nodes to include them in the query graph.

Click on the grey edge to select it, or click on a grey edge to display the other occurrences of the grey edge, if any.

Click on the empty canvas to add the selected nodes and edges to the query graph while ignoring the unselected white nodes, and display new suggestions.

Click on selected nodes (in blue) to unselect them.

(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: User Interface of Orion

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).

3.3 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 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., $\mathrm{IE}(v) = \{\mathrm{etype}(e) | e = (v, v_i) \text{ or } e = (v_i, v), e \in E(G_d)\}$. ■

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 1):
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.

$$C = \begin{cases} C_P & \text{in passive mode} \\ C_A & \text{in active mode} \end{cases} \quad (1)$$

In Section 4 we discuss how to rank candidate edges and thus make suggestions to users in the query construction process.

[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.

4. 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 3.1) 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 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 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 4.1, we describe several baseline methods to rank candidate edges using query logs. In Section 4.2 we propose a novel method inspired by random forests. Section 5 discusses several ways of obtaining a query log.

4.1 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) [31]. 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 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 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 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 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.

4.2 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 [11]. However, RDP has important differences from the standard definition and application of random forests, and significantly outperforms standard random forests in our experiments.

4.2.1 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).

4.2.2 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 $Q_i \subseteq Q$:

$$\text{supp}(e, Q_i, W) = \frac{|\{w|w \in W, Q_i \cup \{e\} \subseteq w\}|}{|\{w|w \in W, Q_i \subseteq w\}|} \qquad (2)$$

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. $\overrightarrow{\mathcal{P}(T)'} = \bigcup_{O \in \mathcal{P}(T)'} \{\overrightarrow{O_i} | \forall i, 1 \le i \le |O|!\}$, 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 $\overrightarrow{O} = \{e_1, e_2, \dots, e_{k-1}, e_k\}$, then $\text{prefix}(\overrightarrow{O}) = \{e_1, e_2, \dots, e_{k-1}\}$. The support for a decision path $\overrightarrow{O}$ is given by $\text{count}(\overrightarrow{O})$, defined as

$$W_{\overrightarrow{O}} = \{w|w \in W, O \subseteq w\}, \text{count}(\overrightarrow{O}) = |W_{\overrightarrow{O}}| \qquad (3)$$

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,

$$\mathcal{Q} = \{\overrightarrow{Q_i} | Q_i \subseteq Q, \text{count}(\overrightarrow{Q_i}) \le \tau, \text{count}(\text{prefix}(\overrightarrow{Q_i})) > \tau\} \qquad (4)$$

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 $\text{count}(\overrightarrow{Q_i}) \le \tau$, 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

$$\text{score}(e) = \frac{1}{|\Re|} \times \sum_{\overrightarrow{Q_i} \in \Re} \text{supp}(e, Q_i, W) \qquad (5)$$

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 5 (line 20).

Figure 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 $|W_{path_2}| \le \tau$. In a decision path $\overrightarrow{path_i}$, the support for each candidate edge $e \in C$ with entry $\overline{e}$ in W_{path_i} is computed. The support for each candidate across all the decision paths is aggregated to rank edges in C.

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

```
1  $E_{sugg} \leftarrow \phi, i \leftarrow 0$;
2  while $i < N$ do
3      $\overrightarrow{Q_i} \leftarrow \phi$;
4      $s_i \leftarrow 0$;
5      $W_{\overrightarrow{Q_i}} \leftarrow W$;
6      while $s_i < |Q|$ do
7          $e_{rand} \leftarrow$ sample_without_replacement($Q$);
8          $\overrightarrow{Q_i} \leftarrow \overrightarrow{Q_i} \cup \{e_{rand}\}$;
9          foreach $w \in W_{\overrightarrow{Q_i}}$ do
10             if $e_{rand} \notin w$ then
11                 $W_{\overrightarrow{Q_i}} \leftarrow W_{\overrightarrow{Q_i}} \setminus \{w\}$;
12         if $|W_{\overrightarrow{Q_i}}| \leq \tau$ then
13             break;
14         $s_i \leftarrow s_i + 1$;
15     foreach $e \in C$ do
16         supp$(e, Q_i, W) \leftarrow$ Equation 2;
17         $E_{sugg} \leftarrow E_{sugg} \cup \{(e, \text{supp}(e, Q_i, W))\}$;
18     $i \leftarrow i + 1$;
19 foreach $e \in C$ do
20     score$(e) \leftarrow$ Equation 5;
21 /* Return candidate edges by decreasing order of score(.);*/
```

5. 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 [25], 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 straight-forward 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

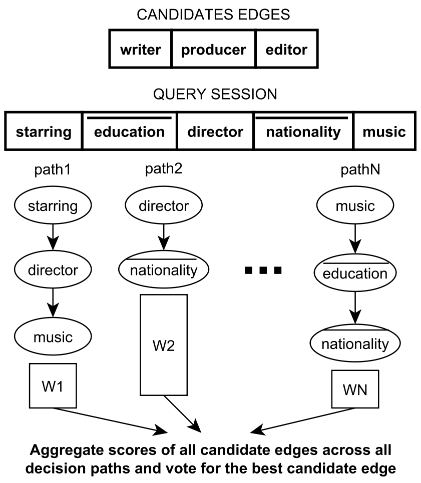


Figure 3: Random Decision Paths Based Edge Selection

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 [25] 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 4.

6. EXPERIMENTS

6.1 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.

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

Query Log	Components Used in Query Log Simulation			
	Freebase	**DBpedia**	**Wikipedia**	**SPARQL** [25]
Wiki-FB	Yes	-	Yes	-
Data-FB	Yes	-	-	-
Wiki-DB	-	Yes	Yes	-
Data-DB	-	Yes	-	-
QLog-DB	-	-	-	Yes

Table 2: Query Logs Simulated

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 3: Sample Query Tasks From User Studies

Datasets: We used two large real-world data graphs: the 2011 version of Freebase [9], and the 2015 version of DBpedia [3]. 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 lists the various query logs simulated using the techniques described in Section 5. One can find positive edges of a query session using different methods, and inject negative edges into them using the method InjectNeg in Section 5. We simulated two different query logs for Freebase: Wiki-FB and Data-FB. The positive 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 5, 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 5.

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 3.2). 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 [16].

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 4.1, including RF, NB, CAR and SVD.

6.2 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 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.

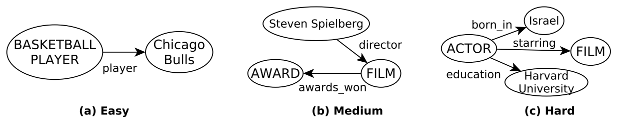


Figure 4: Target Query Graphs of Tasks in Table 3

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 3 lists one sample query for each of the three categories. Figures 4(a), (b) and (c) depict the target query graphs for the query tasks listed in Table 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 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 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 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.

6.2.1 *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:

$$c = \frac{\sum_{\text{task}\in\text{Tasks}} \text{sim}(G_u, G_t)}{|\text{Tasks}|} \qquad (6)$$

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 $\text{sim}(G_u, G_t)$ 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 sub-graph isomorphic match between the two graphs. More formally, $sim(G_u, G_t)$ is defined as:

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

Table 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 5: Conversion Rates of Naive and Orion

$$\mathrm{sim}(G_u, G_t) = \frac{\sum_{\substack{e=(u,v)\in E(G_u)\\ e'=(f(u),f(v))\in E(G_t)}} \mathrm{match}(e, e')}{|E(G_t)|} \quad (7)$$

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

$$\mathrm{match}(e, e')=\begin{cases} 1 & \text{if } u{=}f(u), v{=}f(v), etype(e) = etype(e') \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

Results: Table 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.

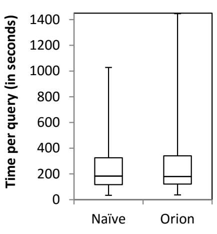
We dive in deeper to investigate if there are scenarios where Orion does perform better than Naive. The conversion rate of only medium and hard query tasks (which is equal to a total of 60 query tasks) for Orion is 0.70, and is equal to 0.58 for Naive, i.e., $c_O = p_O = 0.70$ and $c_N = p_N = 0.58$. 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.

6.2.2 *Efficiency Based on Time*

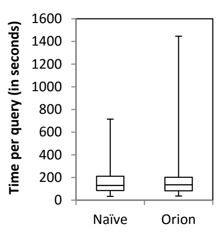
We next measure the time taken by a user to construct the query graph for a given query task: the time elapsed between the first time a user clicks on the query canvas for a new query task, to the time the user clicks on the "Submit" button of the interface. This was recorded in the background during the user study. Figure 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 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 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 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.

6.2.3 *Efficiency Based on Number of Iterations*

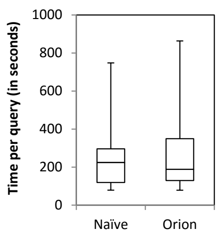
We next measure the effectiveness of Orion using the number of iterations involved in the query construction process: the number of times a ranked list of edges is presented to the user. The number of iterations is incremented in one of three ways: 1) the user selects one or more of the automatically 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 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 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



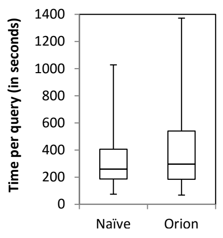
(a) All queries



(b) Easy queries



(c) Medium queries



(d) Hard queries

Figure 5: User Studies Efficiency Based on Time: Naive **and** Orion

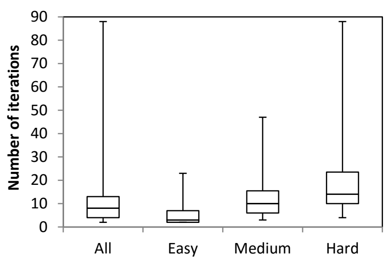


Figure 6: User Studies Efficiency Based on Iterations: Orion

iterations respectively. This indicates that the features offered by Orion helped users formulate query graphs with few interactions with the interface.

6.2.4 *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 7(a) shows the overall user response of all the questions, across all 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 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 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 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.

6.3 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.

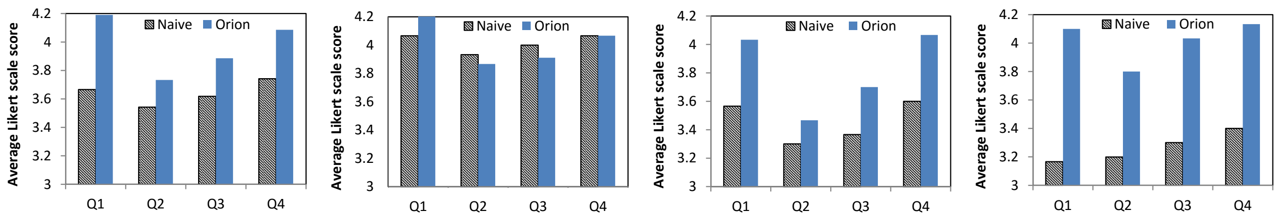
We created multiple target query graphs for each dataset, conforming with the schema of the underlying data graph. For a given target query graph, the input to each of the algorithms was an initial partial query graph containing exactly one edge in it. The task of each algorithm was to iteratively suggest 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 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*.

6.3.1 *Efficiency Based on Number of Suggestions*

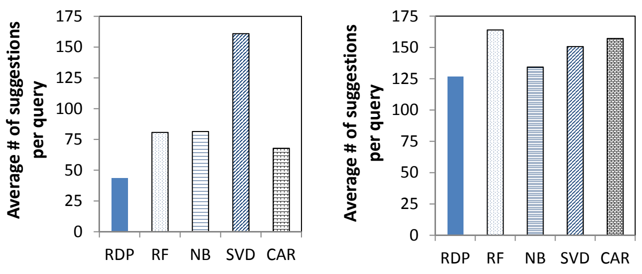
For a query graph completion system, we believe an important measure of its efficiency is the number of suggestions required to successfully grow a partial query graph to its corresponding target query graph. This is because, if a system can help users construct the target query graph with fewer number of suggestions, it indicates that the suggestions made indeed captured the user's query intent. Figure 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 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.

6.3.2 *Efficiency Based on Time*



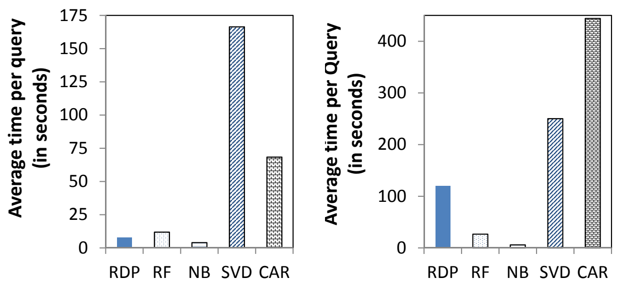
(a) All queries (b) Easy queries (c) Medium queries (d) Hard queries

Figure 7: User Experience Based on Survey Responses



(a) Freebase (b) DBpedia

Figure 8: Efficiency of All Methods: Number of Suggestions



(a) Freebase (b) DBpedia

Figure 9: Efficiency of All Methods: 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 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 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.

6.4 Effectiveness of Query Logs

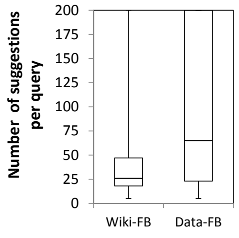
We compare the effectiveness of the various query logs listed in Table 2. We use RDP as the algorithm for edge suggestion, and the number of suggestions required to grow the initial partial query graph to the target query as the measure of effectiveness of the query logs. Freebase-Queries and DBpedia-Queries, described in Section 6.3, were the sets of queries used to compare the various Freebase and DBpedia query logs respectively.

Query Logs for Freebase: Figure 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 5 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.

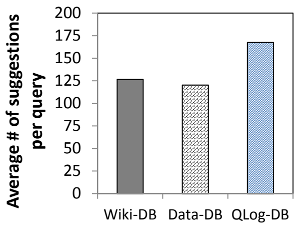
Query Logs for DBpedia: Figure 10(b) shows the average number of edge suggestions required to process the 130 different DBpedia input instances, using each of the three aforementioned query logs for DBpedia. We first observe that QLog-DB performs poorly compared to the other two query logs. This is because 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 Freebase where we could see a significant difference between the performance of Wiki-FB and Data-FB.

6.5 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 4.2.2, 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 identifying 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 11(a) and 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 11(a) and 11(b) also compare

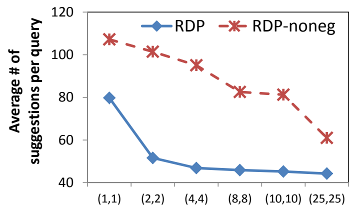


(a) Freebase

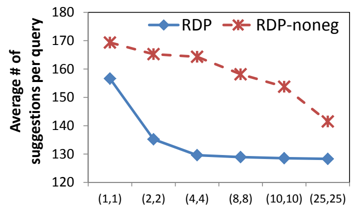


(b) DBpedia

Figure 10: Effectiveness of Query Logs



(a) Freebase



(b) DBpedia

Figure 11: Effect of Parameters on RDP (N, τ)

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.

7. CONCLUSIONS

We introduce Orion, a visual query builder that helps schema-agnostic users construct complex query graphs by automatically suggesting new edges to add to the query graph. Orion's edge ranking algorithm RDP, ranks candidate edges by how likely they will be of interest to the user, using a query log. Since there are no real-world query logs, we propose several ways of simulating a query log. User studies show that Orion has a 70% success rate of building complex query graphs, significantly better than a baseline system resembling existing visual query builders, that has a 58% success rate. We also compare RDP with several methods based on other machine learning algorithms and observe that, on average, those other methods require 1.5-4 more suggestions to complete query graphs.

8. REFERENCES

[1] D. Abadi et al. The beckman report on database research. *SIGMOD Rec.*, pages 61–70, 2014.
[2] M. Arenas, B. Cuenca Grau, E. Kharlamov, S. Marciuska, and D. Zheleznyakov. Faceted search over ontology-enhanced RDF data. CIKM, pages 939–948, 2014.
[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a Web of open data. ISWC, 2007.
[4] N. H. Balkir, G. zsoyoglu, and Z. M. zsoyoglu. A Graphical Query Language: VISUAL and Its Query Processing. *IEEE TKDE*, 2002.
[5] P. A. Bernstein et al. Future directions in DBMS research - the laguna beach participants. *SIGMOD Rec.*, pages 17–26, 1989.
[6] S. S. Bhowmick. DB ⋈ HCI: towards bridging the chasm between graph data management and HCI. DEXA, pages 1–11, 2014.
[7] S. S. Bhowmick, B. Choi, and S. Zhou. VOGUE: Towards A visual interaction-aware graph query processing framework. CIDR, 2013.
[8] H. Blau, N. Immerman, and D. D. Jensen. A visual language for relational knowledge discovery. Technical Report UM-CS-2002-37, Department of Computer Science, University of Massachusetts, 2002.
[9] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. SIGMOD, pages 1247–1250, 2008.
[10] D. Braga, A. Campi, and S. Ceri. XQBE (xquery by example): A visual interface to the standard XML query language. *ACM Trans. Database Syst.*, pages 398–443, 2005.
[11] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32.
[12] D. H. Chau, C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, and T. Eliassi-Rad. GRAPHITE: A visual query system for large graphs. ICDMW, pages 963–966, 2008.
[13] E. Demidova, X. Zhou, and W. Nejdl. Efficient query construction for large scale data. SIGIR, pages 573–582, 2013.
[14] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked keyword searches on graphs. SIGMOD, pages 305–316, 2007.
[15] M. Hildebrand, J. van Ossenbruggen, and L. Hardman. /facet: A browser for heterogeneous semantic web repositories. ISWC, 2006.
[16] 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. SIGMOD, pages 1097–1100, 2013.
[17] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. SIGMOD, pages 13–24, 2007.
[18] N. Jayaram, S. Goyal, and C. Li. VIIQ: Auto-suggestion enabled visual interface for interactive graph query formulation. *VLDB*, pages 1940–1943, 2015.
[19] N. Jayaram, M. Gupta, A. Khan, C. Li, X. Yan, and R. Elmasri. GQBE: querying knowledge graphs by example entity tuples. ICDE, pages 1250–1253, 2014.
[20] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. *IEEE TKDE*, 27(10):2797–2811, 2015.
[21] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. PRAGUE: A practical framework for blending visual subgraph query formulation and query processing. ICDE, pages 222–233, 2012.
[22] C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi. GBLENDER: Towards blending visual query formulation and query processing in graph databases. SIGMOD, pages 111–122, 2010.
[23] L. Lim, H. Wang, and M. Wang. Semantic queries by example. EDBT, pages 347–358, 2013.
[24] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. KDD, pages 80–86, 1998.
[25] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo. DBpedia SPARQL benchmark: Performance assessment with real queries on real data. ISWC, pages 454–469, 2011.
[26] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *VLDB*, 2014.
[27] E. Oren, R. Delbru, and S. Decker. Extending faceted navigation for RDF data. ISWC, pages 559–572, 2006.
[28] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. Interactive query formulation over web service-accessed sources. SIGMOD, pages 253–264, 2006.
[29] M. Petropoulos, Y. Papakonstantinou, and V. Vassalos. Graphical query interfaces for semistructured data: The QURSED system. *TOIT*, pages 390–438, 2005.
[30] R. Pienta, A. Tamersoy, H. Tong, A. Endert, and D. H. P. Chau. Interactive querying over large network data: Scalability, visualization, and interaction design. IUI, pages 61–64, 2015.
[31] X. Su and T. M. Khoshgoftaar. A survey of collaborative filtering techniques. *AAI*, 2009.
[32] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge unifying WordNet and Wikipedia. WWW, 2007.
[33] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. SIGMOD, pages 481–492, 2012.
[34] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum. Natural language questions for the web of data. EMNLP-CoNLL, pages 379–390, 2012.
[35] M. M. Zloof. Query by example. AFIPS, pages 1–24, 1975.