KGVQL
A knowledge graph visual query language with bidirectional transformations
Liu, Pengkai; Wang, Xin; Fu, Qiang; Yang, Yajun; Li, Yuan-Fang; Zhang, Qingpeng

Published in:
Knowledge-Based Systems

Published: 17/08/2022

Document Version:
Final Published version, also known as Publisher's PDF, Publisher's Final version or Version of Record

License:
CC BY-NC-ND

Publication record in CityU Scholars:
Go to record

Published version (DOI):
10.1016/j.knosys.2022.108870

Publication details:

Citing this paper
Please note that where the full-text provided on CityU Scholars is the Post-print version (also known as Accepted Author Manuscript, Peer-reviewed or Author Final version), it may differ from the Final Published version. When citing, ensure that you check and use the publisher's definitive version for pagination and other details.

General rights
Copyright for the publications made accessible via the CityU Scholars portal is retained by the author(s) and/or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights. Users may not further distribute the material or use it for any profit-making activity or commercial gain.

Publisher permission
Permission for previously published items are in accordance with publisher's copyright policies sourced from the SHERPA RoMEO database. Links to full text versions (either Published or Post-print) are only available if corresponding publishers allow open access.

Take down policy
Contact lbscholars@cityu.edu.hk if you believe that this document breaches copyright and provide us with details. We will remove access to the work immediately and investigate your claim.

Download date: 05/11/2022
KGVQL: A knowledge graph visual query language with bidirectional transformations

Pengkai Liu a, Xin Wang a, Qiang Fu a, Yajun Yang a, Yuan-Fang Li b, Qingpeng Zhang c

a College of Intelligence and Computing, Tianjin University, Tianjin, 300350, China
b Department of Data Science and AI, Monash University, Clayton, Victoria, Australia
c School of Data Science, City University of Hong Kong, Hong Kong, China

A R T I C L E   I N F O

Article history:
Received 29 August 2021
Received in revised form 17 April 2022
Accepted 18 April 2022
Available online 27 April 2022

A B S T R A C T

With the rapid development of artificial intelligence, knowledge graphs have been widely recognized as a critical component in many AI techniques and systems. A complex knowledge graph may contain hundreds of millions of nodes and edges, thus is challenging for end-users to understand and query. In this paper, we present a knowledge graph interactive visual query language, KGVQL, to improve the efficiency of end-users’ understanding and querying of knowledge graphs. Furthermore, KGVQL realizes the novel capability of flexible bidirectional transformations between query graphs and query results, therefore significantly assisting end-users in constructing queries over large and unfamiliar knowledge graphs in an incremental way. We present the visual syntax of KGVQL, discuss our design rationale behind this interactive visual query language, and illustrate a number of case studies. We empirically evaluate the effectiveness of a visual query system based on KGVQL against a number of textual and visual query environments over a large knowledge graph, DBpedia. Our evaluation demonstrates the superiority of KGVQL in effectiveness and accuracy.

© 2022 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

With the rapid development of artificial intelligence, Knowledge Graphs (KGs) have become the cornerstone of artificial intelligence and have been identified as a critical component in diverse AI-based applications. In the meanwhile, recent years have witnessed rapid growth in knowledge graph construction and application. The growth in the scale of knowledge graph data and the complexity of its structure pose a significant challenge for query processing [1]. Thus, designing query languages to support effective and efficient exploration and query answering over knowledge graphs has become a crucial research problem. However, key challenges remain in terms of the generalizability and ease of use for existing query languages.

Ease of use. RDF graphs [2] and property graphs [3] are two mainstream forms of knowledge graphs. Query answering is an important method for users to perform structured queries on the knowledge graphs to meet various complex query requirements [4]. A large number of query languages have been designed to support the query answering over knowledge graphs. These languages, including SPARQL [5] and Cypher [6], encompass complex syntax and rich semantics and are designed for developers, thus may not be friendly to end users who do not have professional knowledge on KG or the query languages.

Generalizability. Moreover, most of these query languages are textual languages, which are difficult for end-users to learn and use. While many interactive visual graph languages have been proposed, most of them are only bound to one specific type of knowledge graph. For example, VISAGE [7] based on Cypher is designed for property graphs and QueryOWL [8], QuelDI [9], and SPARQLVis [10] based on SPARQL are designed for RDF graphs. Users are required to learn different query languages to deal with various knowledge graphs, which inevitably increases the difficulty for users to learn and query knowledge graphs.

Navigation support. In addition, some systems such as VIGOR [11] use visualization techniques (e.g., D3 [12] and Echarts [13]) to return results in graphical, instead of tabular form, which can help users understand results more easily. Some other systems such as SPARQLVis support the exploration of knowledge graphs, which can help users retrieve the information they need in knowledge graphs. However, to the best of our knowledge, the existing query languages cannot support navigation from query results to query graphs when users query knowledge graphs, let alone provide a bidirectional transformations trajectory between query graphs and the corresponding query results. After
navigating the user to the query result, it is also an issue worth investigating how to guide them to understand the query graph corresponding to the result.

An illustrating example of the process from a question to the corresponding query result is shown in Fig. 1. The black arrows in the visual language correspond to the textual language, while red arrows in the visual language correspond to the exploration process. As can be seen from the textual query (in SPARQL) at the textual query part in Fig. 1, it may be difficult for end-users to quickly learn and use such a query language, or to be familiar with a knowledge graph. Visual query languages can help make it easier for users to construct query graphs, as shown in the visual language part of Fig. 1. However, end-users may not understand the correspondence between a query graph and its result (such as those nodes labeled by the same variable name at the visual language and graphical result part of Fig. 1) in the process of users’ exploration of knowledge graphs. The correspondence of results to query graphs allows users to easily modify and extend the current query to build more complex queries, and the KGQL proposed in this paper is able to support the visualization of the query graphs corresponding to the query results for users through bidirectional transformation.

To sum up, existing knowledge graph query languages such as SPARQL for RDF graphs and Cypher for property graphs require professional knowledge from the users, thus making them more difficult to use. Therefore, it is necessary to propose an interactive visual knowledge graph query system to solve the aforementioned problems, which should be independent of any specific knowledge graph or query language, and can be adapted to, for example, SPARQL for RDF graphs and Cypher or Gremlin for property graphs. To this end, we propose a knowledge graph visual query language, called KGQL, which can guide end-users in constructing query graphs and transforming them into query results. Furthermore, through bidirectional transformations, KGQL establishes the correspondence between query graphs and query results as users explore knowledge graphs. In this paper, we present the basic elements and syntax of KGQL, discuss our design rationale, and demonstrate its user-friendliness and effectiveness in a user study.

Our contributions can be summarized as:

- We propose a knowledge graph interactive visual query language, called KGQL, which supports multiple operators (e.g., UNION, OPT, FILTER, and LIMIT), and is independent of a specific low-level graph query language.
- To the best of our knowledge, KGQL is the first work that implements bidirectional transformations between query graphs and query results. It can provide intermediate results and trajectories of query results when users explore knowledge graphs, and effectively guide end-users to construct query graphs step by step.
- We develop an interactive visual query interface based on KGQL, and empirically demonstrate its usefulness in helping end-users query and understand knowledge graphs in both speed and accuracy.

The rest of this paper is organized as follows. Section 2 reviews related works. In Section 3, we introduce the basic elements, visual syntax, and features of KGQL. The case study and user study are presented in Section 4 and Section 5, respectively. Finally, Section 6 concludes the paper.

2. Related work

With the rapid development of knowledge graphs, numerous works have emerged on graph query engines and representation learning models [14–16]. Meanwhile, visualization of knowledge graphs has received extensive attention in recent years. A number of visual graph query systems and graph query languages have been released to help end-users understand and query knowledge graphs.

Graph query languages. Query language is an important tool for processing data. Query By Example (QBE) is a language proposed by Zloof et al. [17] for graphically representing and querying relational data, which allows end-users to construct queries by creating templates from “example queries” rather than writing textual SQL statements.

Recent research works, such as RDF-GL [18], QueryVOWL [8], and V1 [19], have followed the idea of QBE. (1) In order to make it more convenient for users to query over the RDF graphs, Frederik Hogenboom et al. [18] combined the GQL (Graph Query Language) with QBE and proposed a visual query language RDF-GL for RDF graphs. In RDF-GL, boxes, circles, and arrows are used to construct basic query patterns, but it requires the user to construct complex query graphs to support operators such as COUNT and LIMIT, which decreases the usability. (2) Florian et al. proposed QueryVOWL based on the semantics of SPARQL and the visual elements of VOWL [20]. QueDi [9] splits the query phase into the construction of SPARQL query and the operation of data tables. However, QueryVOWL and QueDi display the results in the form of lists and tables, respectively, which cannot indicate the relations between results. (3) In order to show the relations explicitly, Kogan et al. proposed V1, a declarative visual query language for schema-based property graphs, which supports property graphs with hybrid (both directed and undirected) edges and temporal data types. Nevertheless, because of its rich expressiveness, V1 requires the same expert knowledge as textual query languages. Wang et al. [21] proposed a bidirectional transformation visual query language, named KGVIS to address the above issues, however, its theoretical basis has not been discussed.

Our KGQL not only follows the idea of query by example, but also supports multiple operators, e.g. FILTER OPT. Moreover, KGQL is independent of any specific knowledge graph query language.

Visual graph query. In recent years, a large number of visualization methods are used to display query results from knowledge graphs, while these methods cannot meet the needs of users. Structured textual query languages, such as SPARQL, make it difficult for end-users to construct complex queries. Therefore, many researchers focus on visual graph query. In recent years, a wide variety of systems have been proposed to visualize, query, and explore knowledge graphs [22,23].

Jayaram et al. [24] proposed a weighted hidden maximum query graph in GQBE (Graph Query By Example) to capture the query intent of end-users, which can efficiently find and rank the top approximate query graphs based on the inputs. However, GQBE does not support relation-based query, which prevents users from expressing complete query intents. LogCanvas [25] focuses on helping users reconstruct semantic relationship among their search activities using knowledge graphs. Hogan et al. [26]
proposed a faceted browsing query system based on filtering, called Grafa. In Grafa, the filter conditions are pre-queried and stored by the system, so as to ensure that results are not empty. An interactive visual query system, named VISAGE, with a graph auto-complete function was proposed by Pienta et al. [7], which conducts subgraph matching queries. Following the work of VISAGE, Hohman et al. [11] proposed an interactive visual exploration system VIGOR for graph query results. The experiments on network security datasets and the knowledge graph of co-authors on DBLP show that the system is superior to Neo4j. Although VISAGE and VIGOR can interactively construct queries, they represent relations through different types of nodes, which is ambiguous to represent relations between entities. Vargas et al. [27] proposed a language and an associated interface, RDF Explorer, that enables end-users to build and execute graph-pattern queries on an RDF Graph. On the other hand, the above systems separate query graphs from query results, which means that end-users cannot judge whether the query graphs conform to their query intents.

Our work bridges query graphs and query results, and combines them through flexible bidirectional transformation. Accordingly, end-users can ensure that each step of construction conforms to their query intents. Meanwhile, KGVQL can retain the intermediate results on which end-users can explore.

### 3. The KGVQL Language

In this section, five use cases corresponding to common knowledge graph query intents are given. Meanwhile, we describe the visual syntax of KGVQL, the mapping to SPARQL syntax fragments, the general graph that constitute the trajectory between queries and results, and the process of bidirectional transformation.

As listed in Table 1, five common use cases (U1–U5) are identified from the related work [8] and the DBpedia [28] benchmark, which are regarded as practical background for KGVQL design. The combinations of U1–U5 cover various kinds of knowledge graph queries: U1 and U2 query entities by keywords and types, respectively; U3 queries the relationships between entities; U4 and U5 involves complex queries on various operators, i.e., UNION and OPT.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Intention</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>Query by keywords</td>
<td>Finding entities in knowledge graphs by keywords, such as ‘Epicurus’, respectively;</td>
</tr>
<tr>
<td>U2</td>
<td>Query by types</td>
<td>Finding entities in knowledge graphs of the same type, such as ‘Politician’.</td>
</tr>
<tr>
<td>U3</td>
<td>Simple query</td>
<td>Executing queries that satisfy the relationships between entities.</td>
</tr>
<tr>
<td>U4</td>
<td>Simple query with constraints</td>
<td>Executing simple queries that meet certain constraints.</td>
</tr>
<tr>
<td>U5</td>
<td>Complex query</td>
<td>Finding complex associations among entities.</td>
</tr>
</tbody>
</table>

#### 3.1. Basic elements of KGVQL

Without loss of generality, the design of visual syntax of KGVQL is based on an analysis of the definitions of SPARQL query patterns [29]. The basic unit of KGVQL visual query is a query graph pattern consisting of circles connected by directed edges, which is used to represent a general graph matching query on knowledge graphs. On this basis, KGVQL supports various operators by using double circle with different types of operators, and thus extends basic query graph patterns into complex query graph patterns.

As shown in Table 2, several basic elements are used to represent the visual syntax of KGVQL: (1) entity elements (E), solid circles indicating entities (a constant ‘s’ or variable ‘?s’) in a knowledge graph; (2) relation elements (R), double solid arrows indicating relations (a constant ‘r’ or variable ‘?r’) between entities; (3) operator elements (OP), double circles indicating operators (an operation ‘c’, e.g., UNION, LIMIT); (4) parameter value elements (PV), rectangles indicating parameters; and (5) projection entity and projection relation elements (PE and PK), dashed circles and directed arrows indicating the entity or relations to be projected in the query results, respectively.

The knowledge graph is given in the form of a directed graph consisting of entity and relation elements.

#### Definition 1 (Knowledge Graph). A knowledge graph $G = (E, R)$ is a directed graph, where $E$ is a finite set of entity elements and $R$ is a finite set of relation elements connecting the entity elements.

#### 3.2. Visual syntax of KGVQL

The visual syntax of KGVQL is represented by graph patterns consisting of basic elements, also known as query graph patterns, and can be constructed recursively from the basic elements.

#### Definition 2. The KGVQL query graph pattern $P$ is defined recursively as follows.

1. An entity element $s$ is a solid circle.
2. A relation element $r$ is a double solid arrow.
3. An operator element $c$ is a double circle with a parameter.
4. A projection entity $PE$ is a dashed circle.
5. A projection relation $PK$ is a directed edge.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Entity (E, ‘s’ or ‘?s’)</td>
<td>s</td>
<td>Projection Entity (PE, ‘s’ or ‘?s’)</td>
</tr>
<tr>
<td>r</td>
<td>Relation (R, ‘r’ or ‘?r’)</td>
<td>r</td>
<td>Projection Relation (PK, ‘r’ or ‘?r’)</td>
</tr>
<tr>
<td>c</td>
<td>Operator (OP, ‘c’)</td>
<td>param</td>
<td>Parameter Value (PV)</td>
</tr>
</tbody>
</table>
1. **(Unit Query Graph Pattern)** A graph pattern consisting of elements \( s, o \in S_E \) and \( p \in S_P \), with \( p \) graphically connected to \( s \) and \( o \), is a query graph pattern, where \( S_E \) is a finite set of entity elements (E) and projection entity elements (PE), and \( S_P \) is a finite set of relation elements (R) and projection relation elements (PR).

2. **(Basic Query Graph Pattern)** A graph pattern consisting of several query graph patterns, which contain entities in common and are connected by these entities, is a query graph pattern.

3. **(Restricted Query Graph Pattern)** If \( P \) is a query graph pattern, then \( P \) with an operator element \( c \in S_C \) and a parameter value element \( v \in S_V \) is also a query graph pattern, where \( S_C \) is a finite set of operator elements (OP, i.e., LIMIT or FILTER), and \( S_V \) is a finite set of parameter value elements (PV).

4. **(Combined Query Graph Pattern)** A graph pattern consisting of a query graph pattern \( P_1 \), connected by using an operator element \( c \in S_C \) to a query graph pattern \( P_2 \), is a query graph pattern, where \( S_C \) is a finite set of operator elements (OP, i.e., UNION or OPT). The set of projection entities and projection relations of \( P_1 \) should be consistent with that of \( P_2 \).

Intuitively, (1) a unit query graph pattern is the most primitive query graph pattern of KGVQL, and it is equivalent to a triple pattern in knowledge graphs; (2) a basic query graph pattern can be recursively decomposed into several unit query graph patterns based on their common entity elements; (3) a restricted query graph pattern restricts a query result by means of an operator element, i.e., LIMIT or FILTER, and a parameter value element; (4) in a combined query graph pattern, an operator element, i.e., UNION or OPT, combines \( P_1 \) and \( P_2 \), i.e., \( (P_1 \cup P_2) \) and \( (P_1 \cup P_2) \).

To further explain the construction of a query graph pattern \( P \) with basic elements, we consider the visual construction of the four defined query graph patterns.

1. **Unit query graph pattern.** A unit query graph pattern \( P_U \) is constructed by using \( E = \{ s, o \} \) and \( R \subseteq S_E \) and \( R = \{ p \} \wedge R \subseteq S_R \), i.e., \( P_U = (E, R, C, V) \), where \( C = \emptyset \) and \( V = \emptyset \). An example of a unit query graph pattern is represented as follows.

2. **Basic query graph pattern.** A basic query graph pattern \( P_B \) is recursively constructed by using query graph patterns \( P_1 = (E_1, R_1, C_1, V_1) \) and \( P_2 = (E_2, R_2, C_2, V_2) \), i.e., \( P_B = (E, R, C, V) \), where \( E = E_1 \cup E_2 \) and \( E_1 \cap E_2 \neq \emptyset \), \( R = R_1 \cup R_2 \), \( C = C_1 \cup C_2 \), and \( V = V_1 \cup V_2 \). An example of a basic query graph pattern is represented as follows.

3. **Restricted query graph pattern.** A restricted query graph pattern \( P_R \) is recursively constructed by using a query graph pattern \( P_1 = (E_1, R_1, C_1, V_1) \), an operator element \( c \in S_C \), and a parameter value element \( v \in S_V \), i.e., \( P_R = (E, R, C, V) \), where \( E = E_1 \), \( R = R_1 \), \( C = C_1 \cup \{ c \} \), and \( V = V_1 \cup \{ v \} \). An example of a restricted query graph pattern is represented as follows.

4. **Combined query graph pattern.** A combined query graph pattern \( P_C \) is recursively constructed by using query graph patterns \( P_1 = (E_1, R_1, C_1, V_1) \) and \( P_2 = (E_2, R_2, C_2, V_2) \), and an operator element \( c \in S_C \), i.e., \( P_C = (E, R, C, V) \), where \( E = E_1 \cup E_2 \) and \( E_1 \cap E_2 \neq \emptyset \), \( R = R_1 \cup R_2 \), \( C = C_1 \cup C_2 \), and \( V = V_1 \cup V_2 \). An example of a combined query graph pattern is represented as follows.

KGVQL is based on the queries represented by graph patterns, and Table 3 shows the various cases of query graph patterns represented by using the KGVQL syntax. Given a query graph pattern \( P \) and a knowledge graph \( G \), the results of \( P \) on \( G \) are denoted as the query results \( [P]_G \). The operator element (OP) is used to combine query graph patterns or to restrict the query results. The parameter value element (PV) is used as a parameter for operations, such as FILTER or LIMIT. For FILTER, the parameter value is a constraint on edge and node properties, which are Boolean combinations of terms consisting of inequality symbols \((<, >, \leq, \geq)\) and equation symbols \(=\). For LIMIT, the parameter value is a constant.

### 3.3. **KGVQL as a fragment of SPARQL**

In KGVQL, end-users do not need to learn a specific textual query language. By means of QBE, users only need to use the necessary graphical elements and fill in some required fields on the given interface to execute the query and obtain the appropriate results. To demonstrate the generality of KGVQL syntax, in this subsection, we introduce a fragment of SPARQL syntax that is equivalent to the query features supported in KGVQL.

<table>
<thead>
<tr>
<th>Table 3</th>
<th>The examples of visual syntax in KGVQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graph Pattern</strong></td>
<td><strong>KGVQL Syntax</strong></td>
</tr>
<tr>
<td>Unit query graph pattern</td>
<td>$P = (E, R, C, V)$</td>
</tr>
<tr>
<td>Basic query graph pattern</td>
<td>$P = (E, R, C, V)$</td>
</tr>
<tr>
<td>Combined query graph pattern for UNION operator</td>
<td>$P = (E, R, C, V)$</td>
</tr>
<tr>
<td>Combined query graph pattern for OPT operator</td>
<td>$P = (E, R, C, V)$</td>
</tr>
<tr>
<td>Constraint query graph pattern for FILTER operator</td>
<td>$P = (E, R, C, V)$</td>
</tr>
<tr>
<td>Constraint query graph pattern for LIMIT operator</td>
<td>$P = (E, R, C, V)$</td>
</tr>
</tbody>
</table>
and show the mapping between KGVQL and SPARQL, i.e., the implementation of KGVQL in SPARQL, by means of examples. It is worth noting that the term query pattern is used in SPARQL while the term graph pattern is used in KGVQL.

Let the infinite pairwise disjoint sets $I$ and $L$ represent the URIs and literals in the RDF data model of a knowledge graph. $T = (I \cup L)$ is the set of RDF terms and an RDF graph is a finite set of RDF triples $(s, p, o) \in I \times I \times T$, where $s$, $p$, and $o$ represent the subjects, predicates, and objects of facts in a knowledge graph, respectively.

**Definition 3 (SPARQL Syntax Fragment)**. Let $X = \{ ?x_1, ?x_2, \ldots, ?x_n \}$ be a set of variables disjoint from the RDF terms $T$, the SPARQL syntax fragment mapping to a KGVQL query graph pattern is defined over $X$ and $T$. The expressions and patterns of the SPARQL syntax fragment are defined recursively as follows:

1. **(Unit Query Pattern)** A tuple $t \in (I \cup X) \times (I \cup X) \times (I \cup L \cup X)$ is a pattern, i.e., a triple pattern.
2. **(Basic Query Pattern)** If $P_1$ and $P_2$ are patterns, $P_1$ JOIN $P_2$ is a pattern.
3. **(Restricted Query Pattern for FILTER operator)** (a) A variable $x \in X$ is an expression $E$; (b) a term $t \in T$ is an expression $E$; (c) given expressions $E_1$ and $E_2$, $(E_1 \pm E_2)$, $(E_1 \times E_2)$, $(E_1 / E_2)$, $(E_1 < E_2)$, $(E_1 < E_2)$, $(E_1 \vee E_2)$, and $(E_1 \wedge E_2)$ are expressions; (d) given a pattern $P$ and an expression $E$, $\text{Filter}(E, P)$ is a pattern.
4. **(Restricted Query Pattern for LIMIT operator)** Given a query graph pattern $P$ and a constant $c$, $\text{Limit}(c, P)$ is a pattern.
5. **(Combined Query Pattern)** If $P_1$ and $P_2$ are patterns, $P_1$ UNION $P_2$ and $P_1$ OPT $P_2$ are patterns.

Thus, we can implement the mapping from KGVQL to SPARQL. Table 3 describes the mapping from the query graph patterns of KGVQL syntax to the query patterns of SPARQL syntax. For a KGVQL query graph pattern, it can be recursively decomposed into basic elements and a set of examples is provided describing how each primitive element maps to a SPARQL identifier, covering all possible mappings from KGVQL to SPARQL.

In the following, the mappings between KGVQL and SPARQL for different query graph patterns are described in detail.

1. For unit query patterns and basic query patterns, the subject, predicate, and object in the SPARQL triple pattern correspond to the entity elements and relational elements in KGVQL query graph patterns. Multiple unit query graph patterns are joined by commonly contained entities (e.g., city in $P_3$) to indicate JOIN operations, and iteratively form a basic query graph pattern.
2. For combined query patterns, the operators in SPARQL are represented in KGVQL using a double circle. KGVQL differentiates between the left and right operands of various operators by framing the right operand and the operator with a dashed box. For UNION, if more than one alternatives matches, all the possible solutions of the query graph pattern can be found. For OPT, the right operand specifies the optional parts of the query graph pattern and allows them to optionally match the result. It is worth noting that the query graph pattern framed by a dashed box in KGVQL can represent a sub-query, and the nested form of the SPARQL query statement can be represented by the containment relationship of the dashed box.
3. For constrained query patterns, the expression $E$ in SPARQL is represented by a text box in KGVQL, and the constraint of the operator is applied directly to the entire query graph pattern. For FILTER, the content of the text box specifies to which node or subgraph in the query graph pattern it is applied. For LIMIT, the number of nodes returned is constrained.

**Example**. KGVQL can visually express various query semantics, perform the corresponding query features on knowledge graphs, and visualize the query results. Fig. 2 depicts a complex query graph pattern consisting of various types of query graph patterns and their equivalent query statements in SPARQL. The query returns the cities with populations greater than the average population of each city, and the population of each city is compared with the average of the populations of the cities returned by the sub-query by means of the FILTER operator.

![Fig. 2. A complex query graph pattern and the equivalent SPARQL statement.](image)

Typically, the existing visual graph query systems only return the results in the form of graphs based on the queries input by users. Although these systems support the deep exploration of knowledge graphs, they cannot intuitively provide the query graphs for the query results [8,9,18]. Therefore, it is not friendly for end-users when they would like to to know what the query graph, which corresponds to the query result obtained by exploring the knowledge graph, looks like.

In this subsection, we give the formal definition of a trajectory between the query and the result, which is a sequence of general graphs: specifically, the query graph and the query result are two special cases of general graphs. In addition, the hyper node and the Depth-First Traversal Tree (DFT-Tree) are defined to obtain the concise path in the query results to construct the query graph. The implementation of the bidirectional transformation between query graphs and query results is described in detail in Section 3.5.
Definition 4 (Hyper Node). Given a graph $G = (V, E)$, let $\omega, \varphi : V \to \mathbb{P}(\text{Lab})$ be the mappings for obtaining the sets of the labels of outgoing and incoming edges of a node, respectively. If $\exists v_1, \ldots, v_n \in V, (\omega(v_1) \cap \ldots \cap \omega(v_n) \neq \emptyset) \land (\varphi(v_1) \cap \ldots \cap \varphi(v_n) \neq \emptyset)$, then we can transform $v_1, \ldots, v_n$ into a hyper node $h$. Specifically, for each node $v \in V$, it can also be considered as a hyper node itself.

For example, as shown in Fig. 3(b), $v_2$ and $v_7$ have the same successor node $v_5$ and predecessor node $v_4$. Another feature of the hyper node is that nodes in the hyper node have the same paths in the original graph. For example, the same path in Fig. 3(b) is $(7x, e_2, ?y, e_1, ?z, 7x)$, where $?y$ can be replaced with $v_5$, $?z$ can be replaced with $v_4$, and $7z$ can be replaced with $v_2$ or $v_7$.

Definition 5 (General Graph). A general graph $P = (H, E, \rho, \lambda, \sigma)$ consists of (1) a finite set $H$ of hyper nodes; (2) a finite set $E$ of edges and $H \cap E = \emptyset$; (3) a mapping $\rho : E \to H \times H$ such that $\rho(e) = (h_1, h_2)$ denotes a directed edge $e$ from hyper node $h_1$ to $h_2$; (4) a mapping $\lambda : (H \cup E) \to \text{Lab}$, where $\text{Lab}$ is a set of labels, such that $\lambda(h), h \in H$ (resp. $\lambda(e), e \in E$) denotes a label on nodes (or edges); (5) a mapping $\sigma(H \cup E) \times \text{Prop} \to \text{Val}$, where $\text{Prop}$ is a set of properties and $\text{Val}$ is a set of values, such that $\sigma(h, p) = \text{val}$ (resp. $\sigma(e, p) = \text{val}$) denotes the value of property $p$ on hyper node $h$ (or edge $e$). The example of the general graph is shown in Fig. 3(c).

Definition 6 (Query Graph). Given a general graph $P = (H, E, \rho, \lambda, \sigma)$, the general graph $P$ is referred to as a query graph, if $\exists h \in H$ and $\sigma(h, \text{Results}) = \text{Null}$.

Definition 7 (Query Result). Given a general graph $P = (H, E, \rho, \lambda, \sigma)$, the general graph $P$ is referred to as a query result, if $\forall h \in H$ and $\sigma(h, \text{Number}) = 1$.

For example, as shown in Fig. 3(d), nodes with ‘?’ in the label indicate variables, representing the objects that users would like to query, i.e., $\sigma(?, \text{Results}) = \text{Null}$, thus Fig. 3(d) is a query graph. As shown in Fig. 3(a), $\forall v \in V, \sigma(v_1, \text{Results}) = v_1$ and $\sigma(v_1, \text{Number}) = 1$, thus Fig. 3(a) is a query result.

A main feature of KGQL is bidirectional transformations between query graphs and query results. To get the query graph that corresponds to the query result, we need to find the most concise general graph in the query result and store the nodes of the same structure in the query result using hyper nodes. Currently, we use a depth-first algorithm (i.e., [30]) to traverse the graph of the query result.

DFT-Tree. In order to obtain correct and complete query graphs, we record the paths to each visited node based on DFS. Thus, we ensure that the edges in query results are unique in a DFT-tree, which allows duplicate nodes occurring (e.g., copies $v_5.1$ and $v_5.2$ of the node $v_5$ in Fig. 3(b)). Please note that for one DFT-tree only one query result can be generated eventually. For each unvisited incoming edge $e$ of a visited node $v$, $e$ is replaced with an outgoing edge $e'$ of $v$. Therefore, any query result can be transformed into a corresponding DFT-tree. To record the transformation from query results to query graphs using DFT-tree, we introduce the definition of trajectories.

Definition 8 (Trajectory). A trajectory $T = (P_1, P_2, \ldots, P_n)$ from a query result to the corresponding query graph is defined as a sequence of general graphs, where $P_1$ is a general graph, $P_1$ is the query result, and $P_n$ is the query graph. The four subfigures of Fig. 3 show an example of trajectory from a query result (i.e., Fig. 3(a)) to the query graph (i.e., Fig. 3(d)).

3.5. Bidirectional transformation

To the best of our knowledge, the existing knowledge graph query systems only support the unidirectional query process from query graphs to query results. Although some of these systems provide deep exploration feature on knowledge graphs, as shown in Fig. 4, users are not able to retrospect the original query graph that corresponds to the query result. Thus, it is inconvenient for users to explore the same query graph once again on the knowledge graphs. In this subsection, we show how KGQL enables a bidirectional transformation between query graphs and query results.

Query Graph to Query Result. In KGQL, the query graphs that users input (as shown in Fig. 3(d)) can be equivalently transformed into the underlying knowledge graph query languages (e.g., SPARQL) in terms of translating the visual syntax of KGQL into a specified textual knowledge graph query language, which can then be executed against the knowledge graph storage implementation (possibly remote query on the Web), and the users can finally obtain the corresponding visual query results (as shown in Fig. 3(a)).

Query Result to Query Graph. For a query result obtained from the user's exploration, in KGQL, we can transform the query result backward to the query graph using a trajectory in Definition 8. The detailed transformation process is: (1) in KGQL, a node in the query result is randomly chosen as the root node to construct the DFT-tree, and, in order to get the directed path incoming edge $e$ of $v$ is replaced with an outgoing edge $e'$; and then, (2) the nodes of the same structure (i.e., the nodes with the same incoming and outgoing edges) are merged and transformed into hyper nodes to generate a general graph; finally, (3) the hyper nodes are replaced with the variable nodes to obtain the query graph for the query result.

Example. The query result in Fig. 3(a) corresponds to the result obtained by the user's exploration in Fig. 4, and Fig. 3(a)–(d) show the process of transformation from the query result to the query graph. Here, the node $v_5$ in the query result (Fig. 3(a)) is chosen as the root node of the DFT-tree, and the query result is transformed into the DFT-tree using a graph traversal from $v_5$. In the traversal, for each visited node $v$ in the query result, the unvisited incoming edges of $v$ are all replaced with the outgoing edges (e.g., $e'$ for $v_2$ and $v_7$ in Fig. 3(b)). In the DFT-tree, there exist two paths from the root node $v_5$ (in red) to the leaf nodes $v_{5.1}$ and $v_{5.2}$ (in purple, i.e., copies of the root node), respectively, which satisfy the path $(v_5, e_2, v_4, e_1, ?x, e_1', v_5.1, v_{5.2})$ (we allow the variables, here, $?x$, in a path for placeholders of nodes and edges), where ‘∥’ denotes alternative, and two paths from the node $v_2$ (in yellow) to the leaf nodes $v_1$ and $v_3$ (in light blue), respectively, which satisfy the path $(v_2, e_1, ?y)$. Here, we can observe that the DFT-tree preserves all the information of the query result. When the nodes $?x, ?y$ are replaced with $v_2, v_7$, and $v_1, v_3$, respectively, these paths exist in Fig. 3(a). In Fig. 3(c), according to the definition of hyper node (Definition 4), to obtain the general graph, the nodes $v_2, v_7$ can be transformed into the hyper node $h_1$, and the nodes $v_1, v_3$ to the hyper node $h_2$. Finally,
in Fig. 3(d), to restore the original query graph, the hyper nodes \( h_1 \) and \( h_2 \) in Fig. 3(c) are replaced with the variables \( ?x \) and \( ?y \), respectively. Therefore, through the process of Fig. 3(a)–(d), we realize the retroactive transformation from the query result to query graph in bidirectional transformation mechanism. □

**Theorem 1.** Given a query result \( P_i \), the query graph \( P_j \) obtained by the bidirectional transformation mechanism defined in this subsection is correct.

**Proof (Sketch).** From the query result \( P_i \), a node \( v_i \) is selected as the root node to generate the DFT-tree \( T \). The nodes \( v_1, \ldots, v_n \) are identified such that there exists a node \( v \) in \( T \) that satisfies that node \( v \) is connected to \( v_1, \ldots, v_n \) through the same edge \( e \). Nodes \( v_1, \ldots, v_n \) are merged and transformed into hyper nodes \( h_n \), and the transformation operation is repeated until no new hyper node is generated. The query graph \( P_j \) is obtained by replacing the hyper nodes with variable nodes, i.e., the variable nodes in the query graph correspond to nodes with the same incoming and outgoing edges in the query result, so that the result of using \( P_j \) to query over the knowledge graph \( G \) is the same as \( P_i \), i.e., \( [P_j]_G = P_i \). □

Thus far, we have realized the bidirectional transformation mechanism between query graphs and query results. In Section 4, we will conduct a case study to show the transformation from query graphs to query results in detail, which demonstrates the KGVQL prototype system we have implemented.

### 4. Case study

In this section, we present a set of use cases to further demonstrate how the features of KGVQL can overcome the major challenges to explore knowledge graphs in interactive visual query languages and systems. *Use case 1 (keyword and type query)* and *use case 2 (basic graph query)*, show how KGVQL can meet the basic query requirements for knowledge graphs; *use case 3 (deep exploration query)* describes how queries for further exploration can be performed on the obtained query results, while *use case 4 (bidirectional transformation query)* shows bidirectional transformation between query results and query graphs. We have implemented a prototype system for KGVQL to support end-users’ visual query and interactive bidirectional exploration on knowledge graphs.

#### 4.1. The overview of the KGVQL prototype system

Fig. 4 shows the components of the KGVQL prototype system: the query editor and the search panel. At the top of the query editor, there are buttons for additional functions: adding new nodes, selecting a layout, etc. In KGVQL, the query graph is mainly constructed in a drag-and-drop manner, and the operations (e.g., FILTER, OPT, and UNION) for basic elements are in the right-click menu.

The process starts with a blank visual query editor. A user can add a node, i.e., an entity element, as a starting point for a query and select the keywords and/or types of the starting point. For example, as shown in Fig. 4, after the user adds a new node (1) and inputs the keyword “Bob” (2), the suggested entities will be automatically displayed. Once “Bob Black” is chosen, the thumbnail of Bob Black will be shown automatically. Then the user can drag the border of the “Bob Black” entry to create an empty node and an edge that connects these two nodes. After the user double-clicks and inputs the predicate “influencedBy”, the number of query results (people who influenced Bob Black) will be displayed in the label of the empty node (i.e., “List 28”). Further, the user can find people who influenced both Bob Black and the people who influenced Bob Black (marked with red arrows). The user can expand the results (marked with blue arrows), and deep exploration is supported to query more information when the user is interested in certain query results. For example, the user can find the birth year and death year of Peter Kropotkin, or people who influenced Marshall Sahlins (marked with green arrows).

#### 4.2. Use case 1: keyword and type query

As shown in Fig. 5, when the user adds a new node and double-clicks on it, the input box will prompt to enter keywords or type information and the user can select the keyword or type to query. In order to improve the accuracy of the user queries and avoid invalid keywords, KGVQL supports fuzzy matching with more than three characters. When the nodes returned by the query have thumbnails or summaries, KGVQL will feed them back to help the user understand the query result more intuitively. When the user enters the type information of nodes to query, KGVQL will feedback the number \( n \) of the matched query results (i.e., “List \( n \)”). Meanwhile, the user can interactively perform various operations, such as expansion or collapse, by right-clicking.
4.3. Use case 2: basic graph query

As shown in Fig. 6, after the user adds a node as the subject ① and enters "Karl Marx", a thumbnail of "Karl Marx" will be returned to the user, as described in Section 4.2. Then the user can drag the border of the subject "Karl Marx" ② to create a new predicate and an object. Based on the query intent, the user can enter the predicate or the object for the query. In this use case, the user enters "influencedBy" as a predicate ③, and a hyper node with the label "List 26" is returned, indicating that the query matches 26 results. When the user right-clicks on the hyper node to expand it, all the results of the query will be displayed ④.

4.4. Use case 3: deep exploration query

KGVQL supports deep exploration that can do further queries based on the results of previous queries. Fig. 7 illustrates an expanded layout for deep exploration by selecting "Aristotle" ⑤ as the subject for the next query. As shown in the query result ⑥⑦, the user would be gradually guided to deeply explore the knowledge graph to discover the expected answer even if they initially do not know how the complex queries that match their query intent can be written.

4.5. Use case 4: bidirectional transformation query

In this use case, we will introduce how the user can perform more complex queries through bidirectional transformation, such as finding who influenced the people that influenced Epicurus, and the birth years of these people.

As shown in Fig. 8, in the first step, the user adds a new node and enters "Epicurus" as the keyword, then drags the edge of the node "Epicurus" to create a new node and a directed edge. In the second step, the user double-clicks the edge and input "influencedBy" as the predicate, then the number of results for people who influenced Epicurus is two. The user then double-clicks the hyper node to get the intermediate results containing "Aristotle" and "Pyrrho" (as shown in the blue box). In the third step, the user double-clicks the hyper node ("List 2") again to return to the query graph, and continue to drag the edge of the hyper node ("List 2") to generate a new node and a directed edge. After the user input "influencedBy" as a keyword (the edge "influencedBy" between "List 2" and "List 6"), the number of results for who influenced the people that influenced Epicurus will be displayed on the hyper node ("List 6"). When the user double-clicks the hyper node ("List 6"), the results will be shown in the purple box, i.e., "Parmenides", "Empedocles", "Heracleitus", "Xenophanes", "Gotama_Buddha" and "Democritus". In the fourth step, the user does the same operation as the third step and enters "birthyear" as the predicate, the birth years of the results are shown in the green box. Through the hyper node defined, the intermediate results obtained from each step of the query are stored in the underlying system. Expanding the hyper node can transform query graphs to query results, and collapsing the intermediate results can transform query results to query graphs. Therefore, in this way, KGVQL can construct trajectories of bidirectional transformation between query graphs and query results.

5. User study

In this section, the user study is conducted to assess how effective and user-friendly the visual query language KGVQL is in comparison with the current semantic query language SPARQL, the visual query language QueryVOWL [8], and RDF Explorer [27]. We used the same DBpedia SPARQL endpoint 1 to ensure the fairness of the study. Furthermore, KGVQL was considered to compare with interactive visual query approaches, including VAGE [7] and VIGOR [11]. Nevertheless, as they are not designed for online queries, we chose some interactive visual tools for specific graph query languages as comparative tools (e.g., SPARQLVIS [10] and GraphVista [31]) in the user study to prove the advantages of KGVQL in terms of interaction. At the same time, we designed a user questionnaire to record and analyze the feedback of users in various aspects (e.g., "Easier to use?", "Easier to learn?", and "Is it user-friendly?").

5.1. Participants

We recruited a total of 20 participants from our institution’s local mailing lists. They ranged in age from 20 to 30 (avg: 24, 10 female, 10 male). Ten of them were master students and the other 10 were graduate students.

1 http://dbpedia.org/sparql/
other ten were undergraduate students, all of whom were in computing-related majors. The participants were asked to report their familiarity (out of 5) with: graphs (avg: 3.1), query languages (avg: 3.5), graph querying (avg: 2.3), and Virtuoso SPARQL (avg: 1.9). Each study lasted for about 30 minutes.

5.2. Experiment design

Our study takes a within-subject design and uses KGQL and three baselines, i.e., SPARQL, QueryVOWL, and RDF Explorer, to accomplish the tasks that we designed. We randomly divided volunteers into four groups evenly. Each group used a different tool, i.e., Group A + KGQL, Group B + SPARQL, Group C + QueryVOWL, and Group D + RDF Explorer, and we divided the experiment into two parts: (1) the comparison experiment of completion time and accuracy of each tool; (2) the user preferences for each tool. On the basis of the first part of the experiment, the volunteers were asked to use all the other three tools and also required to complete the previously designed questionnaire of the other three tools.

5.3. Tasks

We created tasks for the user study based on an analysis of common query graphs from previous studies of knowledge graph query. Each task, as shown in Table 4, corresponds to the use cases discussed in Table 1. We ranked the difficulty of each task based on the number of nodes, edges, and constraints needed. As can be seen in Table 4, the tasks are related to each other, and the subsequent task can be built incrementally upon the previous tasks. This design simulates the actual workflow in a real-world setting, where users typically start by writing a simple query and gradually make it more complex.

Our assumption is that KGQL, SPARQL, QueryVOWL, and RDF Explorer would achieve comparative performance for simple queries, and KGQL and RDF Explorer would take less time to execute more complex queries. The task completion time is a crucial measure, which is mainly influenced by query construction time.

Before the participants were given the tasks, we provided an introduction to the experimental process briefly. Participants were provided with a description of the tool they would use and the information about the knowledge graph dataset they would explore. We provided documentation for KGQL, while we offered a tutorial on syntax for SPARQL. The participants were welcome to ask questions during these introductory periods.

At the end of the user study, we asked participants to complete a questionnaire for subjective impressions about each tool using Likert scales [32]. The questionnaire contains six questions: “Easier to learn?”, “Easier to use?”, “Is it user-friendly?” , “Is it more accuracy?”, “Is it faster?”, and “Is it more enjoyable?”, each with a maximum score of 10. We divided these questions into two categories: the first three questions were used to evaluate the difficulty between tools and the last three questions were used to evaluate the confidence of the user using each tool to complete the query tasks. In order to show user preferences more intuitively, we used five-point scale when describing difficulty and confidence of different tools. We mapped ten-point scale to five-point scale in order to uniformly show the results. The higher the score for the difficulty, the less user-friendly it is to the user, and the higher the score for the confidence, the more the user likes it.

5.4. Results

After all 20 participants had completed the study, we measured the completion time and the accuracy of each task, and collected the user questionnaires. The main experimental results are shown in Fig. 9 and Fig. 10. As the data is not normally distributed, we used the non-parametric Wilcoxon test to compare
Table 4
Tasks of the user study.

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
<th>Query pattern</th>
<th>Task</th>
<th>Description</th>
<th>Query pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Find philosophers influenced by Bob Black.</td>
<td>?x influencedBy Bob Black</td>
<td>2</td>
<td>Find people influenced Bob Black and Karl Marx.</td>
<td>Bob Black influencedBy ?y Karl Marx influencedBy ?x</td>
</tr>
<tr>
<td>3</td>
<td>Find people who influenced both Bob Black and people influenced by Bob Black.</td>
<td>Bob Black influencedBy ?y Karl Marx influencedBy ?x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5
Summary of average of difficulty and confidence rating.

<table>
<thead>
<tr>
<th>Task</th>
<th>Difficulty KGVQL</th>
<th>Difficulty Others</th>
<th>Confidence KGVQL</th>
<th>Confidence Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task1</td>
<td>1.45</td>
<td>1.65</td>
<td>4.25</td>
<td>3.95</td>
</tr>
<tr>
<td>Task2</td>
<td>2.15</td>
<td>2.40</td>
<td>3.60</td>
<td>3.30</td>
</tr>
<tr>
<td>Task3</td>
<td>2.70</td>
<td>2.80</td>
<td>2.95</td>
<td>2.75</td>
</tr>
<tr>
<td>Task4</td>
<td>2.90</td>
<td>3.40</td>
<td>2.20</td>
<td>1.90</td>
</tr>
<tr>
<td>Task5</td>
<td>3.90</td>
<td>4.10</td>
<td>1.75</td>
<td>1.70</td>
</tr>
</tbody>
</table>

the accuracy of each tool. We discuss these results in more detail below.

Completion Time. Fig. 9(a) depicts the completion time of the tasks by the participants. We found that, for all the tasks, the participants spent less time on KGVQL than on other tools. On easier tasks, i.e., Task 1 and Task 2, the completion time on KGVQL, QueryVOWL, and RDF Explorer are comparable. As the difficulty of the tasks gradually increases, the completion time increased on KGVQL is lower than on other tools. For Task 4 and Task 5, the completion time on KGVQL is almost 50% less than SPARQL, and approximately 15% less than QueryVOWL and RDF Explorer.

Accuracy. Fig. 9(b) shows the details of the average accuracy per task for each tool. We found that participants achieved the highest accuracy on all tasks with KGVQL. The accuracy of QueryVOWL and RDF Explorer is higher than SPARQL. However, the accuracy of KGVQL is generally better than the other three languages. At the same time, as the difficulty of tasks gradually increases, the average accuracy of KGVQL remains above 60%.

Participant Experience. The detailed Likert-scale scores of each tool are shown in Fig. 10. In general, participants ranked the highest scores on KGVQL for all six questions, showing its superiority in user-friendliness. We also found that participants ranked the lowest score on SPARQL, the textual query language, than the other visual tools, which validates our assumption that a visual query interface can improve user experience. As shown in Table 5, the users’ feedback indicates that KGVQL is easier than other tools, and users have more confidence in the query results of KGVQL. Fig. 11 shows the difficulty and confidence rating between KGVQL and RDF Explorer. The Likert-scale scores on user preference for RDF Explorer are higher than other tools.

Summary. From the above results, we can get the following conclusions: (1) among the five tasks, the query completion time of KGVQL are the least; (2) as the tasks become more difficult, the query completion time (resp. accuracy) of SPARQL is gradually increasing (resp. decreasing); however, KGVQL outperforms all the other baselines: SPARQL, QueryVOWL, and RDF Explorer, and the accuracy of KGVQL remains above 60%; (3) the participants felt that KGVQL was better than SPARQL and others for all six aspects of user experience (shown in Fig. 10), and enjoyed using KGVQL more than others.
As shown in Fig. 9(b), KGVQL moderately outperforms QueryVOWL on accuracy for all tasks, and significantly outperforms QueryVOWL for the more complex ones, i.e., Task 4 and Task 5. Compared with RDF Explorer, KGVQL reduces the completion time by about 15% and increases the accuracy by about 10%. In summary, our evaluation demonstrates the superiority of KGVQL in all three aspects: completion time, accuracy, and user experience.

We believe that the main reasons for superiority of KGVQL are bidirectional transformation and deep exploration it supports. With the bidirectional transformation mechanism in KGVQL, users can employ the query results of the previous query graph as the input of the next query graph, which reduces the difficulty and time used for constructing queries. In contrast, the other graph query languages (e.g., SPARQL and QueryVOWL) cannot effectively take advantage of query results. For changes of the query tasks, traditional query languages require users to construct new queries from scratch, which reduces the speed, usability, and user-friendliness. Although QueryVOWL and RDF Explorer, as visual query languages, are also more intuitive than SPARQL, they use lists to display results, which cannot express the relationships between the query and the results, and cannot support the deep exploration.

6. Conclusion

In this work, we propose a knowledge graph visual query language, KGVQL, which can help end-users query information from knowledge graphs even if they do not possess intimate knowledge of the query language and the underlying knowledge graphs. KGVQL can be mapped to syntax fragments of SPARQL and other knowledge graph query languages (such as Cypher or Gremlin) with the same query intent. A major advantage of KGVQL is the flexible bidirectional transformation mechanism between query graphs and query results, which is a useful tool for end-users to gain insights into large-scale knowledge graphs and eliminate the boundary between query graphs and query results. The basic elements that make up the KGVQL visual syntax can guide end-users to construct a simple query graph interactively until the final query results are gradually obtained. Thus, KGVQL can provide the provenance information of query results when end-users explore knowledge graphs, which can help users learn and understand knowledge graphs efficiently. We have experimentally validated the user-friendliness and effectiveness of KGVQL and its associated interface, showing its superiority over other visual query languages for knowledge graphs.

CRediT authorship contribution statement

Pengkai Liu: Conceptualization, Software, Writing – original draft, Investigation. Xin Wang: Writing – review & editing, Supervision, Resources, Validation, Funding acquisition. Qiang Fu: Writing – original draft, Software. Yajun Yang: Formal analysis, Reviewing. Yuan-Fang Li: Software, Methodology. Qingpeng Zhang: Reviewing and editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Key Research and Development Program of China (2019YFE0198600).

References


