# Semantic Guided and Response Times Bounded Top-k Similarity Search over Knowledge Graphs

Yuxiang Wang[1,2], Arijit Khan[2], Tianxing Wu[2], Jiahui Jin[3], Haijiang Yan[1]
[1] *Hangzhou Dianzi University, China* [2] *Nanyang Technological University, Singapore* [3] *Southeast University, China*
{lsswyx,yanhj}@hdu.edu.cn, {arijit.khan,wutianxing}@ntu.edu.sg, jjin@seu.edu.cn

***Abstract*—Recently, graph query is widely adopted for querying knowledge graphs. Given a query graph $G_Q$, the graph query finds subgraphs in a knowledge graph $G$ that exactly or approximately match $G_Q$. We face two challenges on graph query: (1) the structural gap between $G_Q$ and the predefined schema in $G$ causes mismatch with query graph, (2) users cannot view the answers until the graph query terminates, leading to a longer system response time (SRT). In this paper, we propose a semantic-guided and response-time-bounded graph query to return the top-k answers effectively and efficiently. We leverage a knowledge graph embedding model to build the semantic graph $SG_Q$, and we define the path semantic similarity ($pss$) over $SG_Q$ as the metric to evaluate the answer's quality. Then, we propose an A* semantic search on $SG_Q$ to find the top-k answers with the greatest $pss$ via a heuristic $pss$ estimation. Furthermore, we make an approximate optimization on A* semantic search to allow users to trade off the effectiveness for SRT within a user-specific time bound. Extensive experiments over real datasets confirm the effectiveness and efficiency of our solution.**

## I. Introduction

Knowledge graphs (such as DBpedia [1], Yago [2], and Freebase [3]) have been constructed in recent years, managing large-scale and real-world facts as a graph [4]. In such graphs, each node represents an entity with attributes, and each edge denotes a relationship between two entities. Querying knowledge graphs is essential for a wide range of applications, e.g., question answering and semantic search [5]. For example, consider that a user wants to find *all cars produced in Germany*. One can come up with a reasonable graph representation of this query as a query graph $G_Q$, and identify the exact or approximate matches of $G_Q$ in a knowledge graph $G$ using graph query models [6]–[10]. Correct answers can be returned, such as ⟨*BMW_320*, *assembly*, *Germany*⟩. Graph query also acts as a fundamental component for other query forms, such as keyword and natural language query [9]. We can reduce these query forms to a graph query by translating input text to a query graph [11], [12].

To retrieve the information of interest from a knowledge graph $G$, users are often required to have full knowledge of the vocabulary used in $G$ [13], as well as the underlying schemas defined in $G$, which is difficult for ordinary users (even professional users) [14]. Otherwise, the user-built query graph is likely to be structurally different from the predefined schemas, thus fails to return correct answers due to the mismatch with the query graph. Consider the following motivating example.

*Example 1:* Consider the query: *Find all cars that are produced in Germany* (Q117 from QALD-4 benchmark [15]). Figure 1 provides four correct graph matches with different

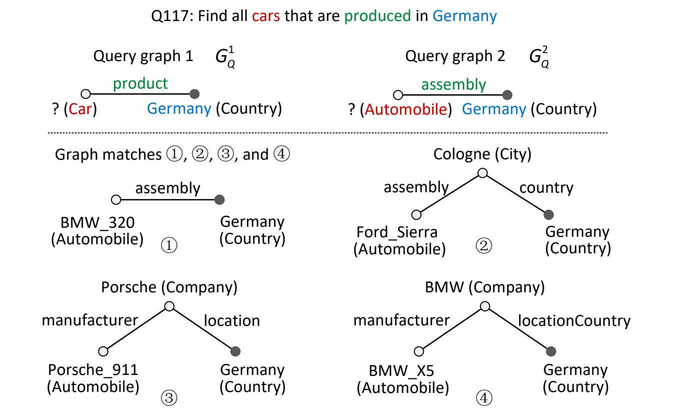


Fig. 1: An example of structural mismatch with query graphs: different users may employ different query graphs (top) to find all cars made in Germany. Four correct graph matches in DBpedia are provided (bottom). Only $G_Q^2$ can retrieve partial correct answers having the same schemas as ①, because the 1-hop edge *assembly* cannot match any $n$-hop ($n > 1$) paths.

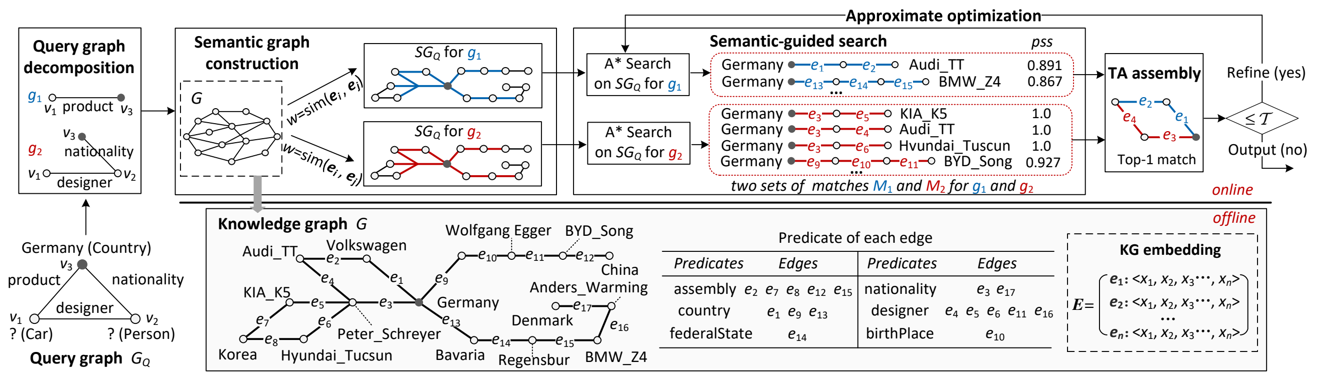
schemas in DBpedia. Each one is represented as an $n$-hop path. An ordinary user may build a query graph $G_Q^1$ based on the phrases from the natural language question [12]. While a professional user may build $G_Q^2$ by using the controlled vocabulary (e.g., *Automobile* is used to represent vehicles in DBpedia) and a schema she already knew. However, both query graphs suffer from the structural mismatch problem.

**Mismatch in query nodes**. In $G_Q^1$, a query node with type *Car* represents the phrase "cars". However, no entity in DBpedia has the same, or even a textually similar type for *Car*, because it is not a term belonging to the controlled vocabulary of DBpedia. Hence, $G_Q^1$ fails to find correct answers.

**Mismatch in query edges**. For $G_Q^2$, the user can retrieve 234 answers that have the same schema as graph match 1. However, more than 200 correct answers are ignored, because a 1-hop edge in $G_Q^2$ cannot be mapped to the semantically similar $n$-hop ($n > 1$) paths (edge-to-path mapping).

If a user has full knowledge about DBpedia, then she can build various query graphs that cover all possible schemas, to obtain all cars of interest. Generally, it is a strong assumption. As an alternative, we aim to provide a graph query system that will be able to support different query graphs without forcing users to use very controlled vocabulary or be knowledgeable about the dataset. This motivates us to fill the structural gap in graph matching by considering the semantics of query graphs.

Another crucial problem involves improving the system response time (SRT) for a graph query. SRT is the amount of time that a user waits before viewing results [16]. A shorter SRT usually indicates a better user experience. To


Approximate optimization
Query graph decomposition
$g_1$ $v_1$ product $v_3$
$g_2$ $v_3$ nationality $v_1$ designer $v_2$
Semantic graph construction
$G$
$w$=sim($e_i$, $e_j$)
$SG_Q$ for $g_1$
$SG_Q$ for $g_2$
A* Search on $SG_Q$ for $g_1$
A* Search on $SG_Q$ for $g_2$
Semantic-guided search
pss
Germany $e_1$ $e_2$ Audi_TT 0.891
Germany $e_{13}$ $e_{14}$ $e_{15}$ BMW_Z4 0.867
Germany $e_3$ $e_5$ KIA_K5 1.0
Germany $e_3$ $e_4$ Audi_TT 1.0
Germany $e_3$ $e_6$ Hvundai_Tuscun 1.0
Germany $e_9$ $e_{10}$ $e_{11}$ BYD_Song 0.927
two sets of matches $M_1$ and $M_2$ for $g_1$ and $g_2$
TA assembly
$e_2$ $e_4$ $e_1$ $e_3$
Top-1 match
Refine (yes)
$\leq \mathcal{T}$
Output (no)
online
offline
Knowledge graph $G$
Audi_TT Volkswagen Wolfgang Egger BYD_Song China Anders_Warming
KIA_K5 Germany Denmark Peter_Schreyer Korea Hyundai_Tucsun Bavaria Regensbur BMW_Z4
Germany (Country)
$v_3$ product nationality designer
$v_1$ ? (Car) $v_2$ ? (Person)
Query graph $G_Q$
KG embedding
$E$= $e_1$: <$x_1$, $x_2$, $x_3$⋯, $x_n$> $e_2$: <$x_1$, $x_2$, $x_3$⋯, $x_n$> $e_n$: <$x_1$, $x_2$, $x_3$⋯, $x_n$>

Predicate of each edge

| *Predicates* | *Edges* | *Predicates* | *Edges* |
|---|---|---|---|
| assembly | $e_2$ $e_7$ $e_8$ $e_{12}$ $e_{15}$ | nationality | $e_3$ $e_{17}$ |
| country | $e_1$ $e_9$ $e_{13}$ | designer | $e_4$ $e_5$ $e_6$ $e_{11}$ $e_{16}$ |
| federalState | $e_{14}$ | birthPlace | $e_{10}$ |

Fig. 2: A running example of our approach, including offline KG embedding (bottom right) and online query processing. Four components of the online part are: (1) Query graph decomposition, (2) Semantic graph construction, (3) Semantic-guided search, and (4) Threshold Algorithm-based assembly. An approximate optimization is applied for response-time-bounded query. All predicates in the example knowledge graph are provided in a table (bottom middle).

the best of our knowledge, no current state-of-the-art work supports response-time-bounded graph query. This motivates us to present an interactive paradigm that allows the user to trade off accuracy for SRT within a user-specific time bound $\mathcal{T}$. As more time is given, better answers can be returned.

In this paper, we blend semantic-guided and response-time-bounded characteristics in one system to support the top-k graph query over a knowledge graph effectively and efficiently.

## A. Our Approach

Many efforts have been made for structural mismatch problem [6], [8], [10], [12], [14], [17]. Among these methods, SLQ [8] is the best one for the query node mismatch via a node transformation library, but it cannot support the edge-to-path mapping. S4 [14] is the most similar work to our paper. It mines the $n$-hop schemas in advance through *string edit distance* and *frequent paths*, by providing semantic instances as the prior knowledge (e.g., given by PATTY [18]). It has two limitations, that are: (1) the string edit distance cannot well represent the real semantics of mined schemas, (2) the accuracy of S4 is sensitive to the quality of prior knowledge.

Unlike S4, we present a semantic-guided search to find the semantically similar paths to query edges without prior knowledge. To the best of our knowledge, we are the first to support semantically edge-to-path mapping in graph query. Moreover, combining our method with SLQ allows us to handle mismatches in query nodes. Figure 2 shows the pipeline of our approach, which has an offline and an online phase.

**Offline phase**. Given a knowledge graph $G$, we leverage a knowledge graph embedding model to represent the predicates of $G$ in a vector space $\boldsymbol{E}$ (**Section IV-A**). Hence, the semantic similarities of predicates can be easily obtained through vector calculation, and this makes it possible to achieve the semantic similarities of a path in $G$ to a query edge in $G_Q$. To be precise, this is essential for semantically edge-to-path mapping.

**Online phase**. In Figure 2, we take a query graph $G_Q$ as an input. $G_Q$ can be of different shapes, for example, the chain, star, tree, cycle, and flower are the commonly used shapes according to [19] . We adopt a decomposition-assembly framework for $G_Q$, which consists of four basic components.

(1) *Query graph decomposition*. We first decompose $G_Q$ into a set of sub-query graphs $\{g_1...g_n\}$ by a dynamic programming algorithm, subject to minimizing the possible query cost. Since this part is not our main focus in this paper, we briefly introduce it in **Section III**, and emphasize more on the querying of the sub-query graphs and assembling their results.

(2) *Semantic graph construction*. To support the semantically edge-to-path mapping, we then construct a semantic graph $SG_Q$ for each sub-query graph $g_i$ by preserving the predicate semantic similarities on the edges of $G$ (**Section IV-B**). In Figure 2, we show example semantic graphs for $g_1$ and $g_2$. For instance, each blue edge has a high semantic similarity to the query edge *product*, e.g., $e_2$ (*assembly*) has 0.98 semantic similarity to *product*. By utilizing these similarities in $SG_Q$, we define the path semantic similarity ($pss$) to measure how semantically similar a path in $G$ is to $g_i$ (**Section IV-C**).

(3) *Semantic-guided search*. We next present an A* semantic search to find the top-k semantically similar matches for each sub-query graph $g_i$ from the semantic graph $SG_Q$ based on the path semantic similarity ($pss$). To improve the efficiency, we propose a well-designed heuristic estimation function of $pss$ that prunes the search space (**Section V-A**). We prove the effectiveness guarantee of our A* semantic search in **Section V-B**, that is, the matches with the greatest $pss$ must be found.

(4) *Threshold Algorithm (TA)-based assembly*. Finally, we assemble the matches of all sub-query graphs based on the threshold algorithm (TA) [20], in order to form the final answers for $G_Q$ in **Section V-C**.

Moreover, we present an approximate optimization on our semantic-guided search to enable a trade-off between accuracy and the system response time (SRT) within a user-specified time bound $\mathcal{T}$ (**Section VI**). As more time is given, more high-quality matches are refined incrementally. We prove that the globally optimal matches for $G_Q$ can be achieved theoretically when sufficient time is given.

## B. Contributions

We summarize our main contributions as follows.

- We present a decomposition-assembly framework for the top-k similarity search over knowledge graphs, which is the first work that considers the semantic-guided and response-time-bounded characteristics in one system.
- We present an A* semantic search to find semantically similar graph matches, that can efficiently prune unpromis-

ing paths through a well-designed path semantic similarity ($pss$) estimation. We prove the effectiveness guarantee of our algorithm.

- We optimize the A* semantic search to enable a trade-off between effectiveness and efficiency with a time bound $\mathcal{T}$. We prove that this method can converge to the globally optimal results as more time is given.
- We evaluate the effectiveness and efficiency of our approach through extensive experiments on three real-world and large-scale knowledge graphs.

## II. Related Work

According to how previous approaches process graph query, we categorize related work as follows.

**Graph pattern matching.** Graph pattern matching is defined in terms of subgraph isomorphism [21], [22], which is NP-complete and is often too restrictive to capture sensible matches. Hence, graph simulation based pattern matching is proposed, such as [17], [23]. These methods cannot be directly deployed to support graph query over knowledge graphs, because they do not consider the semantic constraints on edges even though they can map an edge to an $n$-hop path.

**Graph similarity search.** Many efforts have been made for the graph similarity search based on different similarity metrics: (1) structural similarity [6], [10], [24], (2) graph edit distance [25], [26], and (3) weak semantic similarity [8], [9], [14]. Note that, [6], [10] support edge-to-path mapping (however, do not consider the semantic constraints). Besides, [14] can find $n$-hop paths that are similar to a query edge based on prior knowledge. Unlike [14], we can find better $n$-hop paths (i.e., semantically similar) without external knowledge.

**Query-by-examples.** Query-by-Example (QBE) aims to allow users to express their search intentions with examples. GQBE [27] and Exemplar [28] are proposed for searching matches that are same as their counterparts from the examples. Moreover, [29], [30] are proposed to pose exemplars characterized by tuple patterns, and identify answers close to exemplar. Our approach can extend these QBE methods by returning more semantically similar answers to the given exemplar queries.

**Other methods to query knowledge graph.** The knowledge graph search can also be conducted by the following query forms: (1) keywords search [12], [31], (2) SPARQL search [21], [32], [33], and (3) natural language search [7], [34], [35]. Most of these methods transform the input texts to query graphs for graph searching, so our graph query approach can be used to improve their performance.

## III. Preliminaries

### A. Background

*Definition 1:* **Knowledge graph**. A knowledge graph is defined as a graph $G = (V, E, L)$, with the node set $V$, edge set $E$, and a label function $L$, where (1) each node $u \in V$ represents an entity, (2) $E$ is an ordered subset of $V \times V$, each directed edge $e = u_i u_j \in E$ denotes the relationship between two entities $u_i$ and $u_j$, and (3) $L$ assigns a name and various types on each node, and a predicate on each edge.

Similar to [9], [30], [36], we define the type as a label on each entity, rather than a node connecting to an entity with the predicate *isA*. This assumption can reduce the size ($|E|$) of $G$, it also avoids dealing with irrelevant edges *isA*.

*Example 2:* We assume that each node $u$ in a knowledge graph $G$ has at least one type and a unique name [14], [35], e.g., $L(u).type = \{\textit{Automobile}\}$ and $L(u).name =$*Audi_TT*. For each edge $e$, we assign a predicate as $L(e) =$*assembly*. If the type of a node in $G$ is unknown, we employ a probabilistic model-based entity typing method to assign a type on it [37].

*Definition 2:* **Query graph**. A query graph is defined as a graph $G_Q = (V_Q, E_Q, L_Q)$, with query node set $V_Q$, edge set $E_Q$, and a label function $L_Q$, where (1) $V_Q = V^s \cup V^t$, (2) $V^s = \{v^s\}$ is a set of *specific nodes*, both the type and name of $v^s$ are known, (3) $V^t = \{v^t\}$ refers to *target nodes*, only the type of $v^t$ is known, (4) $\forall e \in E_Q$ has a predicate $L_Q(e)$.

Since we assume that users do not have full knowledge about the dataset, so they are allowed to represent the query nodes without using the controlled vocabulary. For the example query graph in Figure 2, $V^s = \{v_3\}$ (type: *Country* and name: *Germany*), $V^t = \{v_1, v_2\}$ (respective types: *Car*, *Person*), and *Car* is not a term defined in DBpedia's vocabulary.

**Query graph decomposition**. We adopt a decomposition-assembly framework for $G_Q$. We decompose $G_Q$ into sub-query graphs for querying, and then assemble the matches of all sub-query graphs to form the top-k matches of $G_Q$.

*Definition 3:* **Sub-query graph**. We define a sub-query graph of $G_Q$ as a graph $g_i = (V_i, E_i, L_Q)$, with the query node set $V_i$, query edge set $E_i$, and the same label function $L_Q$ as in $G_Q$, where (1) $g_i$ is a path from a specific node $v^s$ to a target node $v^t$, denoted by $\overline{v^s v^t}$, (2) and $V_Q = \cup V_i$ and $E_Q = \cup E_i$ over all sub-query graphs $g_i$ of $G_Q$.

*Example 3:* The query graph in Figure 2 can be decomposed into two sub-query graphs: (1) find automobiles produced in Germany ($g_1$: $\langle v_1$-*product*-$v_3\rangle$), and (2) find automobiles designed by Germans ($g_2$: $\langle v_1$-*designer*-$v_2$-*nationality*-$v_3\rangle$).

In general, all sub-query graphs intersect at a target node (called *pivot node* $v^p$), e.g., $v_1$ in Example 3. Therefore, we can assemble the final matches via a join operation at $v^p$. The objective of query graph decomposition is to derive a number of sub-query graphs with an appropriate pivot node, to minimize the cost of query processing (Eq. 1). We use the possible search space as the cost metric (similar to [9]) and resolve this problem through dynamic programming. The time complexity is $O(|V_Q| \cdot |E_Q|^2)$ according to [9]. We show the impact of query graph decomposition on performance and its scalability in Section VII-B and Section VII-D, respectively.

$$\underset{\{g_1 \cdots g_n\}}{\arg\min} \sum_{i=1}^{n} cost(g_i) \tag{1}$$

According to [19], the chain, star, tree, cycle, and flower are common query graph shapes in knowledge graph search, so we provide an analysis on the effect of these query graph shapes with different sizes in Section VII-D.

**Sub-query graph matching**. For each sub-query graph, we aim to find the semantically similar matches by identifying the candidate node (edge) matches for each query node (edge).

(1) *Node match*. To overcome the mismatch in query nodes, we define a one-to-many relation $\phi$: $V_i \rightarrow V$ considering three cases: *Identical*, *Synonym*, and *Abbreviation*. For each query node $v \in V_i$, $\phi(v) = \{u_1 ... u_n\}$ is a set of candidate matches in $V$, where the type (name) of $v$ is the same, synonymous, or abbreviated for the type (name) of $u$.

Usually, each node $u$ in a knowledge graph $G$ can have multiple types [37], then $u$ is a node match of the query node $v$ if one of $u$'s types satisfies the relation $\phi$. Although we assume that each query node $v$ has a user-specific type in Definition 2, we still need to consider the following cases to enhance the robustness. (1) If a query node $v$ has multiple types, then we separately consider each type of $v$ in the relation $\phi$ for node matching, and then consider union of all of them as the overall node matches for $v$ . (2) If a user provides a query node $v$ without a type, then $v$ is a wildcard query node and can be mapped to the nodes with different types in a knowledge graph. In Section IV-B, we introduce how to implement the relation $\phi$ by building a transformation library.

(2) *Edge match*. To overcome the mismatch in query edges, we support the semantically edge-to-path mapping. Given a sub-query graph $g_i$ and a knowledge graph $G$, a path $\overline{u_i u_j} \in G$ is a match of an edge $v_i v_j \in g_i$, if $u_i \in \phi(v_i)$ and $u_j \in \phi(v_j)$. While considering paths, we ignore edge directionalities. Besides, we expect that the path $\overline{u_i u_j}$ is semantically similar to the edge $v_i v_j$. We elaborate this part in Defintion 6.

Considering the sub-query graph $g_1$ in Figure 2, the edge matches of *product* are the paths from *Germany* to entities with type *Automobile*, e.g., ($e_1$, $e_2$), ($e_9, e_{10}, e_{11}$), etc. We need to identify the most semantically similar path ($e_1$, $e_2$) from other edge matches, which motivates us to define the semantic graph $SG_Q$ for each sub-query graph $g_i$ as follows.

*Definition 4:* **Semantic graph**. Given a sub-query graph $g_i = (V_i, E_i, L_Q)$ and a knowledge graph $G = (V, E, L)$, the semantic graph is a weighted sub-graph of $G$ defined as $SG_Q = (V', E', L, W)$, with the node set $V' \subseteq V$, edge set $E' \subseteq E$, and weight set $W$, where (1) for each node $v \in V_i$, its node match $u \in \phi(v)$ belongs to $V'$, (2) for each edge $e = v_i v_j \in E_i$, its edge match $\overline{u_i u_j} \in SG_Q$ (i.e., $\forall e' \in \overline{u_i u_j}$ belongs to $E'$), (3) each $e'$ has a weight $w \in W$ to represent the semantic similarity between $e'$ and $e$ (Section IV-A).

According to Definition 3, a sub-query graph $g_i$ is a path graph denoted as $\overline{v^s v^t}$. So, we define the match of $g_i$ as a path in $SG_Q$ that is semantically similar to $\overline{v^s v^t}$.

*Definition 5:* **Sub-query graph match**. Given a sub-query graph $g_i = \overline{v^s v^t}$ and a semantic graph $SG_Q$, a path $\overline{u^s u^t} \in SG_Q$ is a match of $g_i$ if (1) $\overline{u^s u^t}$ comprises the edge match of each edge $v_i v_j \in g_i$, (2) the path semantic similarity ($pss$) of $\overline{u^s u^t}$ to $\overline{v^s v^t}$ equals or is greater than a predefined threshold $\tau$, denoted by $\psi(\overline{u^s u^t}, \overline{v^s v^t}) \geq \tau$.

*Definition 6:* **Path semantic similarity** ($pss$). We define the $pss$ $\psi(\overline{u^s u^t}, \overline{v^s v^t})$ as a function $f(w_1 ... w_n)$ of all weights appearing in match $\overline{u^s u^t}$, which measures the semantic similarity of $\overline{u^s u^t}$ to $g_i$. The details are given in Section IV-C.

**Assembly**. For each sub-query graph $g_i$, we can obtain a set of sub-query graph matches, denoted by $M_i = \{\overline{u^s u^t}\}$. The sub-query graph matches from different $M_i$ may intersect at the same pivot node match $u^p \in \phi(v^p)$, so we can assemble them at $u^p$ to form a match for the query graph $G_Q$. Figure 2 shows the assembly procedure. First, we find some matches with the greatest $pss$ for the two sub-query graphs $g_1$ and $g_2$. The match $\langle$*Germany*-$e_1$-$e_2$-*Audi_TT*$\rangle$ from $M_1$ and $\langle$*Germany*-$e_3$-$e_4$-*Audi_TT*$\rangle$ from $M_2$ can be assembled at pivot node match *Audi_TT* to form a match for $G_Q$. We define the match score of a match for $G_Q$ as the sum of $pss$ for all involved sub-query graph matches that intersect at the same $u^p$.

$$S_m(u^p) = \sum_{M_i} \psi(\overline{u^s u^t}, \overline{v^s v^t}) \quad (2)$$
$$s.t. \quad \overline{u^s u^t} \in M_i \text{ that contains the same } u^p$$

The best match of $G_Q$ is the one with the greatest match score, such as the top-1 match involving *Audi_TT* in Figure 2 (with the greatest match score 1.891=0.891+1).

## B. Problems

According to the background above, we derive two major problems that need to be resolved in this paper as follows.

**Problem 1**. *Given a query graph $G_Q = \{g_1 ... g_n\}$ and a knowledge graph G, we find the top-k matches M according to the match score $S_m(u^p)$ as follows.*

$$M = \sigma_{\max(S_m)}(\bowtie_{u^p} M_i) \quad (3)$$
$$s.t. \quad |M| = k, \quad M_i = \{\underset{\overline{u^s u^t}}{\arg\max} \ \psi(\overline{u^s u^t}, \overline{v^s v^t})\}$$

In Eq. 3, we use $\bowtie_{u^p}$ to denote the assembly at $u^p$ and use $\sigma_{\max(S_m)}$ to denote the top-k matches selection based on the match score $S_m(u^p)$. $M_i = \{\overline{u^s u^t}\}$ are the sub-query matches with the greatest $pss$ for each $g_i$. This problem is non-trivial, because (1) we need to find globally optimal $M_i$ for each $g_i$, and (2) the assembly is computationally expensive if $G_Q$ has a large number of $g_i$, and each one has many candidate matches. We solve this problem efficiently in **Section V**.

**Problem 2**. *Given a query graph $G_Q = \{g_1 ... g_n\}$ and a knowledge graph G, we find the approximate top-k matches $\hat{M}$ within a user-specified time bound $\mathcal{T}$ as follows.*

$$\hat{M} = \sigma_{\max(S_m)}(\bowtie_{u^p} \hat{M}_i) \quad (4)$$
$$s.t. \ |\hat{M}| = k, \ \text{ time bound } \mathcal{T}, \ \left(\frac{\hat{M} \cap M}{\hat{M} \cup M}\right)_{\mathcal{T}'} \geq \left(\frac{\hat{M} \cap M}{\hat{M} \cup M}\right)_{\mathcal{T}}$$

We use the Jaccard similarity of $\hat{M}$ and $M$ to measure the degree of approximation. With more time given ($\mathcal{T}' > \mathcal{T}$), $\hat{M}$ can approach $M$. The key of this problem is how to return $\hat{M}$ quickly, and refine it as more time is given. Moreover, we need to prove that the globally optimal results can be returned if sufficient time is given (e.g., $\hat{M} = M$). We deal with this problem in **Section VI**.

## IV. Semantic Graph Construction

In this section, we leverage a knowledge graph embedding model to construct the semantic graph $SG_Q$, then present the predicate semantic similarity ($pss$) based on $SG_Q$.

TABLE I. Transformation library

| ***Synonym* and *abbreviation* records** | **Types and names** |
|---|---|
| *Car, Motorcar, Auto, Vehicle* | type: <*Automobile*> |
| *GER, FRG, Federal Republic of Germany* | name: *Germany* |

## A. Knowledge Graph Embedding

Knowledge graph embedding aims to represent each predicate and entity of a knowledge graph $G$ as an $n$-dimensional semantic vector, such that the original structures and relations in $G$ are preserved in these learned semantic vectors [4]. We summarize the core idea of most existing knowledge graph embedding methods as follows: (1) initialize the vector of each element in triple $\langle h,r,t \rangle$ as $\langle \boldsymbol{h},\boldsymbol{r},\boldsymbol{t} \rangle$, where *h/t* indicates the head/tail entity and *r* denotes the predicate, (2) define a function $g()$ to measure the relation of $\langle \boldsymbol{h},\boldsymbol{r},\boldsymbol{t} \rangle$, and optimize $g()$ to satisfy $\boldsymbol{t} \approx g(\boldsymbol{h}, \boldsymbol{r})$. The predicate semantic space $\boldsymbol{E} = \{\boldsymbol{e_1}...\boldsymbol{e_n}\}$ is an output of a knowledge graph embedding model. The semantic similarity between the two edges can be represented by the similarity between two predicate vectors.

In this paper, we use the cosine similarity to measure the similarity between two predicate vectors. Each weight $w$ in the semantic graph $SG_Q$, denoted by $sim(L_Q(e), L(e'))$, e.g., $sim($*product*, *assembly*$)$, can be calculated as follows.

$$w = sim(L_Q(e), L(e')) = \frac{\mathbf{e} \cdot \mathbf{e'}}{||\mathbf{e}|| \times ||\mathbf{e'}||} \tag{5}$$

We next introduce how to preserve these weights on a knowledge graph to generate semantic graph.

## B. Constructing Semantic Graph

Figure 3(a) shows an example semantic graph $SG_Q$ for the sub-query graph $g_1$ in Figure 2. A straightforward idea to build $SG_Q$ is: (1) find the node matches of each query node, e.g., $\phi(v_1)$=\{*Audi_TT, KIA_K5, BYD_Song, Hyundai_Tucsun, BMW_Z4*\}, and $\phi(v_3)$=*Germany*, (2) find the edge matches of each query edge, e.g., edge matches of *product* include paths $(e_1, e_2)$, $(e_3, e_4)$, $(e_3, e_5)$, $(e_3, e_6)$, $(e_9, e_{10}, e_{11})$, and $(e_{13}, e_{14}, e_{15})$, and (3) assign weights on edges through Eq. 5.

**Analysis**. To find all edge matches for a query edge $v_iv_j \in g_i$, we must enumerate all possible paths between $u_i \in \phi(v_i)$ and $u_j \in \phi(v_j)$. However, the high connectivity of a knowledge graph $G$ makes it computationally expensive.

**A lightweight way**. Figure 3(b) shows an alternative way to construct $SG_Q$ on the fly. We *push down* the semantic graph construction to the query processing stage, which means that $SG_Q$ is partially materialized (not completely constructed in advance). Given a sub-query graph $g_i = \overline{v^s v^t}$, we materialize the $SG_Q$ as follows.

(1) Get node matches of $v^s$. To implement the relation $\phi$ for node matching, we build a *synonym* and *abbreviation* transformation library [8] for all types and names existing in $G$ based on BabelNet (the largest multilingual synonym dictionary [38]). For instance, we invoke the API of BabelNet to collect the synonyms and abbreviations of the input keyword as one row in Table I. For each query node $v^s$, we use this library to find its node matches $\phi(v^s)$ through the *synonym* or *abbreviation* transformation, e.g., a query node with type *Car* is mapped to a set of entities with type *Automobile* in $G$.

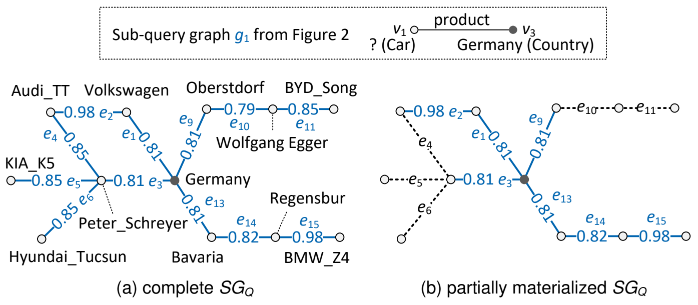


Fig. 3: Semantic graph construction: all predicates in the semantic graph are provided as *assembly*:$\{e_2, e_{15}\}$, *country*:$\{e_1, e_9, e_{13}\}$, *federalState*: $\{e_{14}\}$, *nationality*: $\{e_3\}$, *designer*:$\{e_4, e_5, e_6, e_{11}\}$, and *birthPlace*: $\{e_{10}\}$.

(2) Materialize the 1-hop $SG_Q$ for $u^s \in \phi(v^s)$. Given a node match $u^s \in \phi(v^s)$, we assign the weight $w$ on each edge that connects to $u^s$ based on Eq. 5, generating a 1-hop $SG_Q$ for $u^s$. For example, weighted edges $\{e_1, e_3, e_9, e_{13}\}$ in Figure 3(b) act as the 1-hop $SG_Q$ for the node *Germany*.

(3) Next-hop decision. Given a partially materialized $SG_Q$, we select a next-hop node for further querying. The selected node should be the one with the greatest probability of finding the best match for $g_i$. We show the details in Section V.

(4) Termination check. Starting from the next-hop, we repeat the above steps to materialize $SG_Q$ gradually, until a node match $u^t \in \phi(v^t)$ is detected. The path $\overline{u^s u^t}$ is a match of $g_i$. In Figure 3(b), we can find the best matches from the partially materialized $SG_Q$ (blue lines) excluding the dashed lines.

**Remarks**. To construct a complete $SG_Q$, we need $O(|E'|)$ time to assign weights on all the edges of $SG_Q$. The time complexity is expected to be reduced by constructing $SG_Q$ partially, because a good next-hop selection would reduce the size of $SG_Q$, pruning the search space significantly. Moreover, a good next-hop selection also ensures that the sub-query graph match with the greatest path semantic similarity ($pss$) can be found. We show the details of $pss$ in Section IV-C and show the semantic-guided search based on $pss$ in Section V.

## C. Path Semantic Similarity

We define the path semantic similarity ($pss$) of a sub-query graph match $\overline{u^s u^t}$ to $g_i$ based on the following observations.

- A match $\overline{u^s u^t}$ comprises a set of edges $\{e_1...e_n\}$. Each edge $e_i$ has a weight $w$ that indicates the semantic similarity to one edge $e \in g_i$. Hence, the $pss$ should be a function of all weights appearing at $\overline{u^s u^t}$.
- According to Eq. 5, two edges are semantically similar if their predicate vectors are similar. Thus, the edges with greater $w$ would be more beneficial to the $pss$.
- A smaller $w$ usually indicates that two edges show different semantic meanings. Therefore, the edges with smaller $w$ have a negative effect on the $pss$.

*Example 4:* In Figure 3(a), the paths $(e_1, e_2)$ and $(e_{13}, e_{14}, e_{15})$ are more semantically similar to $g_1$ than others, because the predicate vectors of edges $e_2$ and $e_{15}$ (*assembly*) are more similar to the one of edge *product* (with the greatest $w$=0.98). And other paths containing edges such as $e_4$ (*designer*), $e_{10}$ (*birthPlace*), etc., show the different meanings to $g_1$, because $e_4$ and $e_{10}$ are less semantically similar to *product*.

Based on these intuitions, we calculate the $pss$ of the match $\overline{u^su^t}$ to $\overline{v^sv^t}$, denoted by $\psi(\overline{u^su^t}, \overline{v^sv^t})$, as the geometric mean of all weights appearing at the match $\overline{u^su^t}$.

$$\psi(\overline{u^su^t}, \overline{v^sv^t}) = \sqrt[n]{\prod_{\forall w_j \in \overline{u^su^t}} w_j} \tag{6}$$

## V. Semantic-guided Search

In this section, we first present an A* semantic search to find the top-k matches from $SG_Q$ with the greatest path semantic similarity ($pss$) for each sub-query graph $g_i \in G_Q$. We then assemble all matches for $g_i$ to form the final matches for $G_Q$. The classic A* search [39] finds the shortest path based on a heuristic length estimation. Here, we design the $pss$ estimation based on semantics to find the most semantically similar paths. The basic idea of A* semantic search is that we compute a heuristic $pss$ estimation for a possible match at each detected node, and gradually expand the search space following the guidance of the estimated $pss$ until a match with maximum $pss$ is found. We achieve two benefits from a good $pss$ estimation: (1) we can find the globally optimal matches of $g_i$, and (2) we can prune the search space significantly.

We next introduce the heuristic $pss$ estimation in Section V-A, then discuss A* semantic search based on $pss$ estimation and prove the effectiveness guarantee in Section V-B. Finally, we show the assembling of the matches in Section V-C.

### A. Heuristic Estimation of pss

Given a match $\overline{u^su^t}$ of a sub-query graph $g_i$, it can be divided into an explored partial path $\overline{u^su_i}$ and an unexplored partial path $\overline{u_iu^t}$ at each detected node $u_i$. We compute the upper bound of the exact $pss$ $\psi(\overline{u^su^t}, \overline{v^sv^t})$ at $u_i$ as the estimated $pss$, denoted by $\hat{\psi}(\overline{u^su_i}, \overline{v^sv^t})$ ($\hat{\psi}_i$ for short), by considering the semantic information from both partial paths. Based on the estimated $pss$, we can effectively prune the search space and find the sub-query graph match with the greatest $pss$ due to the following reasons. (1) Suppose that we already find a sub-query graph match, then we can safely prune the potential matches having the smaller $\hat{\psi}_i$ than the explored match's exact $pss$. Only the potential matches with a greater $\hat{\psi}_i$ than the explored match's exact $pss$ would be considered for further searching. (2) Given a predefined $pss$ threshold $\tau$, we can prune the unpromising potential matches that have the $\hat{\psi}_i < \tau$ (we set $\tau = 0.8$ in Section VII).

We next introduce how to obtain the upper bound $\hat{\psi}_i$ of $\psi$ for the $pss$ estimation. And we prove the effectiveness guarantee in Section V-B (Theorem 2).

According to Eq. 6, we need the weight product ($\prod w_j$) and the path length ($n$) of a match $\overline{u^su^t}$ to compute the exact $\psi$. So we first estimate the upper bound of the weight product and path length, in order to estimate the upper bound of $\psi$.

**Upper bound of the weight product**. The weight product of $\overline{u^su^t}$ is divided into two parts at each detected node $u_i$.

- The weight product of partial path $\overline{u^su_i}$, e.g., $\prod_{\forall w_j \in \overline{u^su_i}} w_j$.
- The weight product of partial path $\overline{u_iu^t}$, e.g., $\prod_{\forall w_j \in \overline{u_iu^t}} w_j$.

We can compute the exact weight product of $\overline{u^su_i}$ because $\overline{u^su_i}$ is explored in the partially materialized $SG_Q$. On the other hand, the partial path $\overline{u_iu^t}$ is unexplored, so we use the maximum weight of all adjacent edges of $u_i$ as the upper bound of the weight product of $\overline{u_iu^t}$, denoted as $m(u_i)$.

*Lemma 1:* The maximum weight $m(u_i)$ is the upper bound of the weight product of the partial path $\overline{u_iu^t}$.

*Proof:* Given the weight product $\prod_{w_j \in \overline{u_iu^t}} w_j$ of $\overline{u_iu^t}$, $w_j$ indicates the $j$-th weight in $\overline{u_iu^t}$. Due to the monotonicity of weight product, we have $w_1 \geq \prod_{w_j \in \overline{u_iu^t}} w_j$. Also, $m(u_i) \geq w_1$ because we assume $m(u_i)$ is the max weight of all adjacent edges of $u_i$. Hence, we have $m(u_i) \geq \prod_{w_j \in \overline{u_iu^t}} w_j$. □

**Upper bound of the path length**. Since different matches have different path lengths, it is difficult to get a uniform upper bound of the path length $n$ for all matches. Hence, we relax the upper bound of the exact path length to the upper bound of the user desired path length. If a user wants to find the top-k matches within $\hat{n}$-hop, then only the matches having $n \leq \hat{n}$ ($\hat{n}$-bounded match) will be returned. Hence, $\hat{n}$ is the upper bound of the path length for all the $\hat{n}$-bounded matches.

**Estimated $pss$ of $\hat{n}$-bounded match**. Given the above two upper bounds. We compute the estimated $pss$ $\hat{\psi}_i$ at each node $u_i \neq u^t$ as follows, where $u^t \in \phi(v^t)$ is a target node match. And we set $\hat{\psi}_i$ equals to the exact $pss$ $\psi$ when $u_i = u^t$.

$$\hat{\psi}(\overline{u^su_i}, \overline{v^sv^t}) = \sqrt[\hat{n}]{\prod_{\forall w_j \in \overline{u^su_i}} w_j \cdot m(u_i)} \tag{7}$$

*Theorem 1:* The $pss$ estimation $\hat{\psi}_i$ is the upper bound of the exact $pss$ $\psi$ of the match $\overline{u^su^t}$=$\overline{u^su_i}$+$\overline{u_iu^t}$ with the path length $n \leq \hat{n}$, where $\hat{n}$ is the user desired path length.

*Proof:* We use notation $W_{si}$ ($W_{it}$) to denote the weight product of the partial path $\overline{u^su_i}$ ($\overline{u_iu^t}$). If $u_i{\neq}u^t$, then $\hat{\psi}_i$= $\sqrt[\hat{n}]{W_{si} \cdot m(u_i)}$≥ $\sqrt[n]{W_{si} \cdot m(u_i)}$, because $\hat{n}{\geq}n$ and the $n$-th root $W_{si}{\cdot}m(u_i){\in}$(0,1]. Moreover, we have $m(u_i){\geq}W_{it}$ based on Lemma 1, so that $\sqrt[n]{W_{si} \cdot m(u_i)}$≥ $\sqrt[n]{W_{si} \cdot W_{it}}$=$\psi$. Hence, $\hat{\psi}_i{\geq}\psi$ holds. On the other hand, if $u_i$=$u^t$, then $\hat{\psi}_i$=$\psi$. In summary, $\hat{\psi}_i{\geq}\psi$ holds for all cases. □

**Remarks**. (1) The user desired path length $\hat{n}$ is specified by users before graph querying. (2) Our A* semantic search can find the globally optimal $\hat{n}$-bounded matches (proved later in Theorem 2) based on the above heuristic $pss$ estimation.

### B. A* Semantic Search

In this section, we introduce our A* semantic search based on the above $pss$ estimation, illustrated in Algorithm 1.

**Notations**. We use a max-heap as the match set $M_i$ for a sub-query graph $g_i$, to record each found match and its $pss$, e.g., $\langle\overline{u^su^t}, \psi\rangle$. We use another max-heap as the priority queue $q$ to record each explored partial path $\overline{u^su_i}$ and its estimated $pss$, e.g., $\langle\overline{u^su_i}, \hat{\psi}_i\rangle$. For $u_i = u^s \in \phi(v^s)$, $\overline{u^su_i}$ is the node $u^s$ itself. Each node $u_i$ indicates a next-hop choice for search space expansion. We also use a hash set *visited* to record all visited nodes, avoiding duplicate access. A $pss$ threshold $\tau$ is used to prune the unpromising matches having $\hat{\psi}_i < \tau$.

**Overview**. Given a sub-query graph $g_i$=$\overline{v^sv^t}$, we start with the node match $u^s \in \phi(v^s)$ for A* semantic search (line 1). The main procedures are: (1) *Next-hop selection*. We select

**Algorithm 1:** A* semantic search

```
   Data: sub-query graph $g_i$, number of matches $k$
   Result: match set $M_i$
1  $\forall u^s \in \phi(v^s)$: $q$={$\langle u^s, \hat{\psi}_s \rangle$}, visited={$u_s$}, $M_i$=$\emptyset$;
2  while $q \neq \emptyset$ do
3      $\langle \overline{u^s u_i}, \hat{\psi}_i \rangle$=$q$.pop_max() ;          // Next-hop selection
4      if $u_i \notin \phi(v^t)$ then          // Search space expansion
5          for $\forall u_l \in N(u_i)$ do
6              if !visited.contains($u_l$) then
7                  visited.add($u_l$);
8                  $\overline{u^s u_l}$=$\overline{u^s u_i}$+$u_i u_l$;
9                  $\langle \overline{u^s u_l}, \hat{\psi}_l \rangle$=pssEstimation();
10                 if $\hat{\psi}_l \geq \tau$ then
11                     $q$.push_heap($\langle \overline{u^s u_l}, \hat{\psi}_l \rangle$);
12     else
13         $M_i$.push_heap($\langle \overline{u^s u_i}, \hat{\psi}_i \rangle$);
14         if $|M_i| = k$ then          // Top-k matches check
15             break;
16 return $M_i$;
```

the node $u_i$ with the greatest $\hat{\psi}_i$ as the next-hop for search space expansion, from the priority queue $q$ (line 3). (2) *Search space expansion*. Starting from $u_i$, we expand the search space as $\overline{u^s u_l}$=$\overline{u^s u_i}$+$u_i u_l$ for each neighbour node $u_l$ of $u_i$, and compute $\hat{\psi}_l$ for each new partial path $\overline{u^s u_l}$ (lines 5-9). All these $\langle\overline{u^s u_l}$,$\hat{\psi}_l\rangle$ pairs ($\hat{\psi}_l \geq \tau$) are stored in $q$ for further exploration (lines 10-11). (3) *Top-k matches check*. We repeat (1) and (2) until a match $\overline{u^s u_i}$ is popped from $q$, where $u_i \in \phi(v^t)$. We record it in match set $M_i$ and terminate the search until $k$ matches are found (lines 13-15).

*Example 5:* Figure 4 shows an example for searching the top-1 match from a specific node match $u_1$ to target node matches $\{u_7$,$u_{12}\}$ (we set $\hat{n}$=4). The solid lines indicate the expanded search space, doted lines show the pruned data, and red lines denote the top-1 match. Finally, 38.5% of edges and 25% of nodes are pruned. At the beginning, we expand the search space from $u_1$, all its neighbours are added in the priority queue $q$, e.g., $q$={$\langle\overline{u_1 u_2}$,0.81$\rangle$, $\langle\overline{u_1 u_3}$,0.86$\rangle$, $\langle\overline{u_1 u_4}$,0.73$\rangle$}. We next select $u_3$ in $\overline{u_1 u_3}$ to expand the search space because it has the greatest estimated $pss$ of 0.86, and we add the path $\overline{u_1 u_7}$=$(u_1 u_3, u_3 u_7)$ in the priority queue $q$ as $\langle\overline{u_1 u_7}$,0.74$\rangle$. However, we cannot return $\overline{u_1 u_7}$ as the top-1 match because we still have $\langle\overline{u_1 u_2}, 0.81\rangle$ in $q$. From $\overline{u_1 u_2}$, we may find a better match with $pss$ of 0.81>0.74. Hence, we continue to expand the search space from $u_2$, $u_5$, $u_9$ until $u_{12}$ is detected. Finally, we have $q$={$\langle\overline{u_1 u_{12}}$,0.75$\rangle$, $\langle\overline{u_1 u_7}$,0.74$\rangle$, $\langle\overline{u_1 u_4}$,0.73$\rangle$, $\langle\overline{u_1 u_6}$,0.73$\rangle$}, and $\overline{u_1 u_{12}}$ is the top-1 match, while the potential matches from $\overline{u_1 u_4}$ and $\overline{u_1 u_6}$ can be safely pruned. This is because the upper bound of $pss$ for $\overline{u_1 u_4}$ and $\overline{u_1 u_6}$ are smaller than the $pss$ of the top-1 match.

*Theorem 2:* Our A* semantic search ensures that the sub-query graph match $\overline{u^s u^t}$ with the greatest $pss$ can be found.

*Proof:* Suppose that the returned $\overline{u^s u^t}$ is not the best match, then we have $\psi \leq \psi_{opt}$, where $\psi$ and $\psi_{opt}$ are the $pss$ of $\overline{u^s u^t}$ and the best match, respectively. Since A* semantic search starts from $u^s$ and all its neighbours are considered in the priority queue $q$ for further expansion, then $q$ must contain one partial path $\langle\overline{u^s u_i}, \hat{\psi}_i\rangle$ that belongs to the best match.

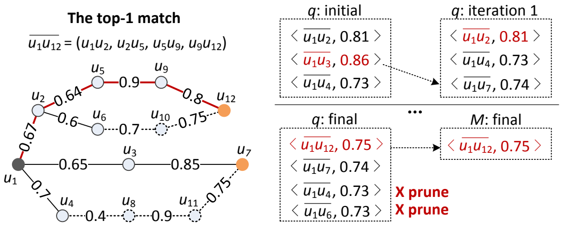

The top-1 match
$\overline{u_1u_{12}}$ = ($u_1u_2$, $u_2u_5$, $u_5u_9$, $u_9u_{12}$)
$u_1$
$u_2$
$u_3$
$u_4$
$u_5$
$u_6$
$u_7$
$u_8$
$u_9$
$u_{10}$
$u_{11}$
$u_{12}$
0.67
0.64
0.9
0.8
0.6
0.7
0.75
0.65
0.85
0.7
0.4
0.9
0.75
$q$: initial
⟨ $\overline{u_1u_2}$, 0.81 ⟩
⟨ $\overline{u_1u_3}$, 0.86 ⟩
⟨ $\overline{u_1u_4}$, 0.73 ⟩
$q$: iteration 1
⟨ $\overline{u_1u_2}$, 0.81 ⟩
⟨ $\overline{u_1u_4}$, 0.73 ⟩
⟨ $\overline{u_1u_7}$, 0.74 ⟩
...
$q$: final
⟨ $\overline{u_1u_{12}}$, 0.75 ⟩
⟨ $\overline{u_1u_7}$, 0.74 ⟩
⟨ $\overline{u_1u_4}$, 0.73 ⟩
⟨ $\overline{u_1u_6}$, 0.73 ⟩
X prune
X prune
$M$: final
⟨ $\overline{u_1u_{12}}$, 0.75 ⟩

Fig. 4: An example of the top-1 match searching

According to Theorem 1, we have $\hat{\psi}_i \geq \psi_{opt}$. Then we have $\hat{\psi}_i \geq \psi_{opt} \geq \psi$. Hence, our A* semantic search will continue to expand the search space from $u_i$ (line 3 in Algorithm 1) rather than returning the non-optimal match $\overline{u^s u^t}$. □

**Complexity**. The time consumption of our A* semantic search is dominated by the search space expansion. Given a partial path $\overline{u^s u_i}$, we expand the search space from the node $u_i$ as follows: (1) construct a new partial path $\overline{u^s u_l}$=$\overline{u^s u_i}$+$u_i u_l$ for each neighbour node $u_l$ of $u_i$ and (2) update the priority queue $q$ with each $\langle\overline{u^s u_l}, \hat{\psi}_l\rangle$ pair. We use $V^*$ to denote all detected nodes in step (1), then the time complexity is $O(|V^*| \log |V^*|)$, where $O(\log |V^*|)$ is the time for the update of max-heap $q$.

**Remarks**. (1) We implement the graph querying in a multi-threaded manner (one thread for each $g_i \in G_Q$). (2) In general, we usually need more than $k$ matches collected for each $g_i$ to ensure that $k$ final matches can be assembled for $G_Q$.

## C. Final Matches Assembly

We employ the Threshold Algorithm (TA) [20] based method to efficiently assemble the top-k matches for $G_Q$.

**Core idea of assembly**. Given the match sets $\{M_i\}$ for all sub-query graphs $\{g_i\}$, the TA-based assembly follows three steps. (1) It accesses all $M_i$ in descending order of the match's $pss$. Since $M_i$ is a max-heap, so we pop the best match in $M_i$ at each access. (2) It joins the detected matches with the same pivot node match $u^p$ to generate a final match $fm(u^p)$, and computes its upper and lower bound on match score, denoted by $\overline{S_m}(u^p)$ and $\underline{S_m}(u^p)$, respectively. (3) It terminates early if $k$ final matches are found, for which the smallest $\underline{S_m}(u^p)$ is larger than other final matches' greatest $\overline{S_m}(u^p)$.

We next introduce how to compute $\overline{S_m}(u^p)$ and $\underline{S_m}(u^p)$ in the TA-based assembly, then we prove that the top-k final matches can be returned through the TA-based assembly. According to Eq. 2, the match score $S_m(u^p)$ of a final match $fm(u^p)$ is computed by aggregating the $pss$ $\psi$ of the matches that contain the same pivot node match $u^p$, from each $M_i$. Intuitively, if we know the $pss$ bounds $\overline{\psi}$ and $\underline{\psi}$ of such a match from each $M_i$, then we can compute the bounds of match score during the TA-based assembly as follows.

$$\overline{S_m}(u^p) = \sum_{M_i} \overline{\psi} \quad \text{and} \quad \underline{S_m}(u^p) = \sum_{M_i} \underline{\psi} \tag{8}$$

where $\overline{\psi}$ ($\underline{\psi}$) is the upper (lower) bound on $pss$ of the match $\overline{u^s u^t} \in M_i$ that contains the same pivot node match $u^p$.

**Upper bound of** $S_m(u^p)$. At each access of the TA-based assembly, if the match that contains the same $u^p$ is not accessed from $M_i$ so far, then we have $\overline{\psi} = \psi_{cur}$, where $\psi_{cur}$

is the $pss$ of the current accessed match in $M_i$. This is because the TA-based assembly accesses each $M_i$ in the descending order of the match's $pss$. All the un-accessed matches from $M_i$ must have $\psi \leq \psi_{cur}$. If we find this match from $M_i$, then we have $\overline{\psi} = \psi$. The upper bound $\overline{\psi}$ is decreased as the TA-based assembly executes. Finally, $\overline{S_m}(u^p)$=$S_m(u^p)$, when the matches containing the same $u^p$ are accessed from each $M_i$.

**Lower bound of $S_m(u^p)$**. At each access of the TA-based assembly, if the match that contains the same $u^p$ is not accessed from $M_i$ so far, then we have $\underline{\psi} = 0$. This is because it is possible that $M_i$ does not contain such a match. If we find this match from $M_i$, then we have $\underline{\psi} = \psi$. The lower bound $\underline{\psi}$ is increased from 0 to $\psi$ as the TA-based assembly executes. Finally, $\underline{S_m}(u^p) = S_m(u^p)$, when the matches containing the same $u^p$ are accessed from each $M_i$.

**Termination check**. We terminate the TA-based assembly if the top-k final matches are found. Specifically, (1) we sort the final matches in descending order of $\underline{S_m}(u^p)$, (2) we select the $k$-th largest $\underline{S_m}(u^p)$ as the lower bound of the top-k match score, denoted as $L$, (3) we select the greatest $\overline{S_m}(u^p)$ among other final matches as their upper bound on match score, denoted as $U$, and (4) we terminate the assembly if $L \geq U$.

*Theorem 3:* The TA-based assembly can obtain the top-k final matches, when $L \geq U$ holds.

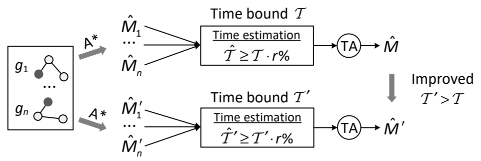
*Proof:* Since the lower bound $\underline{\psi}$ increases from 0 to the exact $pss$ $\psi$ as the TA-based assembly processes, the lower bound of the top-k match score $L$ is also increased (Eq. 8). Similar to $L$, $U$ is decreased because the upper bound $\overline{\psi}$ decreases from $\psi_{cur}$ to the exact $pss$ $\psi$ as the TA-based assembly processes. Hence, if $L \geq U$ holds at the $r$-th access of the TA-based assembly, then it will hold for all $r' > r$ accesses. Therefore, the TA-based assembly can terminate safely and return the top-k final matches when $L \geq U$ holds. □

**Complexity**. In the worst case, all the matches from each match set $M_i$ should be accessed to find the top-k final matches. So, the time complexity of the TA-based assembly in the worst case is $O(\sum_i(|M_i|))$. In Section VII-B, we show the impact of the TA-based assembly on the performance.

## VI. Approximate Optimization

In this section, we introduce an approximate optimization on the semantic-guided search to enable a trade-off between accuracy and the system response time (SRT) within a user-specified time bound $\mathcal{T}$. In the original A* semantic search, we will continuously check the priority queue $q$ until no partial pathes in $q$ having the greater estimated $pss$ than the explored matches' exact $pss$. This operation ensures that the output $k$ matches are globally optimal, but it also increases the SRT because the user cannot view the results before it terminates. Intuitively, if we can output the non-optimal matches earlier (within $\mathcal{T}$), then the SRT could be reduced. As more time is given, the non-optimal matches can be refined incrementally.

**Core idea of the approximate optimization**. Given a user-specific time bound $\mathcal{T}$, the approximate optimization is illustrated in Figure 5. (1) We collect the early explored non-optimal matches of each sub-query graph $g_i$ to generate a non-


$g_1$
$g_n$
A*
A*
$\hat{M}_1$
$\hat{M}_n$
$\hat{M}'_1$
$\hat{M}'_n$
Time bound $\mathcal{T}$
Time estimation
$\hat{\mathcal{T}} \geq \mathcal{T} \cdot r\%$
TA
$\hat{M}$
Improved
$\mathcal{T}' > \mathcal{T}$
Time bound $\mathcal{T}'$
Time estimation
$\hat{\mathcal{T}}' \geq \mathcal{T}' \cdot r\%$
TA
$\hat{M}'$
Fig. 5: An example of approximate optimization

optimal match set $\hat{M}_i$. (2) We estimate the possible overall time of assembling $\{\hat{M}_i\}$ to form the approximate final match set $\hat{M}$, denoted by $\hat{\mathcal{T}}$. (3) We decide to assemble $\hat{M}$, if $\hat{\mathcal{T}} \geq \mathcal{T} \cdot r\%$ is reached. The value of $\mathcal{T} \cdot r\%$ is an alert time threshold to indicate a level beyond which there is a risk of failing to return $\hat{M}$ within $\mathcal{T}$. Moreover, we ensure that $\hat{M}$ can be incrementally improved when more time is given. Two key differences between this optimization and the original A* semantic search (Algorithm 1) are provided as follows.

- We collect each early explored match to the non-optimal match set $\hat{M}_i$ in the step of *search space expansion* (lines 11-12 in Algorithm 2).
- We change the termination condition from *top-k matches check* to *execution time check* (lines 13-15 in Algorithm 2). Specifically, we add a time estimation for the whole graph querying (Algorithm 3) to ensure that the approximate final match set $\hat{M}$ can be returned within $\mathcal{T}$.

**Non-optimal match set $\hat{M}_i$**. In Algorithm 2, we update $\hat{M}_i$ once a match is explored no matter if it is optimal. Obviously, $\hat{M}_i$ will be incrementally refined as more time is given.

*Lemma 2:* The non-optimal match set $\hat{M}_i$ can be incrementally refined and eventually be equal to the globally optimal match set $M_i$, if sufficient time is given.

*Proof:* Suppose that we have $\hat{M}_i \cap M_i = \emptyset$ at time $t$, which means at least $|M_i|$ better matches are not explored in $\hat{M}_i$ so far. Hence, we have at least $|M_i|$ partial paths in the priority queue $q$ that have the greater estimated $pss$ $\hat{\psi}$ than the $pss$ of all matches in $\hat{M}_i$. According to Algorithm 2, we will select the one with the greatest $\hat{\psi}$ from $q$ to expand the search space if $\hat{\mathcal{T}} < \mathcal{T} \cdot r\%$ holds. Once a match that belongs to $M_i$ is explored, we will collect it to $\hat{M}_i$, then we have $|\hat{M}_i \cap M_i| = 1$. Theoretically, if sufficient time $\mathcal{T}$ is given, then Algorithm 2 will keep running until the best $M_i$ matches are explored. □

**Execution time check**. The overall time for querying a query graph $G_Q$ is dominated by the time of A* semantic search ($\mathcal{T}_{A^*}$) for each sub-query graph $g_i$ and the time of TA-based assembly ($\mathcal{T}_{TA}$). Each $g_i$ is processed as an independent thread, so we use $max\{\mathcal{T}_{A^*}\}$ to denote the time of A* semantic search. Given a user-specific time bound $\mathcal{T}$, we want to obtain an approximate final match set $\hat{M}$ with the time $max\{\mathcal{T}_{A^*}\} + \mathcal{T}_{TA} \leq \mathcal{T}$. To this end, we estimate the overall time $\hat{\mathcal{T}}$ of our approximate optimization in Algorithm 3. (1) The A* semantic search of each sub-query graph $g_i$ reports a pair of $\langle \mathcal{T}_{A^*}, |\hat{M}_i| \rangle$ for time estimation (line 1), where $\mathcal{T}_{A^*}$ is the current running time of each $g_i$ and $|\hat{M}_i|$ is the number of explored matches so far. (2) We use all the $|\hat{M}_i|$ to estimate $\mathcal{T}_{TA}$ (line 2). In the worst case, TA-based assembly needs to access all matches in $\hat{M}_i$, so we use $\sum |\hat{M}_i| \cdot t$ as the estimated $\hat{\mathcal{T}}_{TA}$, where $t$ is an empirical time for processing one match of $\hat{M}_i$ in the TA-based assembly. (3) We decide to launch the TA-based assembly if $max\{\mathcal{T}_{A^*}\} + \hat{\mathcal{T}}_{TA} \geq \mathcal{T} \cdot r\%$ (line 3).

**Algorithm 2:** Time bounded A* semantic search

```
Data: sub-query graph $g_i$
Result: non-optimal match set $\hat{M}_i$
1  $\forall u^s \in \phi(v^s)$: $q$={$\langle u^s, \hat{\psi}_s \rangle$}, visited={$u_s$}, $M_i$=$\emptyset$;
2  while $q \neq \emptyset$ do
3      $\langle \overline{u^s u_i}, \hat{\psi}_i \rangle$=q.pop_max() ;        // Next-hop selection
4      for $\forall u_l \in N(u_j)$ do        // Search space expansion
5          if !visited.contains($u_l$) then
6              visited.add($u_l$);
7              $\overline{u^s u_l}$=$\overline{u^s u_j}$+$\overline{u_j u_l}$;
8              $\langle \overline{u^s u_l}, \hat{\psi}_l \rangle$=pssEstimation();
9              if $\hat{\psi}_l \geq \tau$ and $u_l \notin \phi(v^t)$ then
10                 q.push_heap($\langle \overline{u^s u_l}, \hat{\psi}_l \rangle$);
11             if $\hat{\psi}_l \geq \tau$ and $u_l \in \phi(v^t)$ then
12                 $\hat{M}_i$.push_heap($\langle \overline{u^s u_l}, \hat{\psi}_l \rangle$);
13     update($\mathcal{T}_{A*}$);
14     if timeEstimate($\mathcal{T}_{A*}$,$|\hat{M}_i|$) then        // Execution time check
15         break;
16 return $\hat{M}_i$;
```

**Algorithm 3:** timeEstimate($\mathcal{T}_{A*}$,$|\hat{M}_i|$)

```
Data: $\langle \mathcal{T}_{A*}$,$|\hat{M}_i| \rangle$ pair from each $g_i$
1  collect the pair of $\langle \mathcal{T}_{A*}$,$|\hat{M}_i| \rangle$ from each $g_i$;
2  $\hat{\mathcal{T}}_{TA}$=$\sum |\hat{M}_i| \cdot t$, $\hat{\mathcal{T}} = max\{\mathcal{T}_{A*}\} + \hat{\mathcal{T}}_{TA}$;
3  if $\hat{\mathcal{T}} \geq \mathcal{T} \cdot r\%$ then
4      return true;
5  return false;
```

**Approximate $\hat{M}$ assembly**. Given a set of non-optimal match sets $\{\hat{M}_i\}$, we conduct a TA-based assembly to generate the approximate final match set $\hat{M}$. In this paper, we use the Jaccard similarity between $\hat{M}$ and the globally optimal final match set $M$ to quantify their approximation degree as follows,

$$Jad(\hat{M}, M) = \frac{|\hat{M} \cap M|}{|\hat{M} \cup M|} = \frac{k_\cap}{2k - k_\cap} \tag{9}$$

where $k$ is the size of $\hat{M}$, and $k_\cap$ is the size of $|\hat{M} \cap M|$.

*Theorem 4:* Our approximate optimization can incrementally refine the approximate final match set $\hat{M}$ and finally obtain the globally optimal $M$, if sufficient time is given.

*Proof:* Suppose that we have a non-optimal match set $\hat{M}_i$ at time $t$. According to Lemma 2, $\hat{M}_i$ can be refined to $\hat{M}'_i$ at time $t'$ ($t' > t$). Hence, the approximate final match set $\hat{M}'$ assembled from $\{\hat{M}'_i\}$ is better than $\hat{M}$ assembled from $\{\hat{M}_i\}$, which means $k'_\cap \geq k_\cap$. According to Eq. 9, $Jad(\hat{M}', M) \geq Jad(\hat{M}, M)$ holds if $k'_\cap \geq k_\cap$. Moreover, if sufficient time $\mathcal{T}$ is given, then we have $\hat{M}_i = M_i$ (Lemma 2). Hence, the global optimal final match set $M$ can be assembled from $\{\hat{M}_i\}$ when sufficient time $\mathcal{T}$ is given. □

## VII. Experimental Study

We present experiment results to evaluate (1) effectiveness and efficiency, (2) analysis via user-study, (3) impact of query graph shapes and sizes, (4) robustness with noise, (5) scalability of our algorithms, and (6) parameter sensitivity. Due to limitation of space, we refer the reader to full version (https://arxiv.org/pdf/1910.06584.pdf) for the details of (6). The source code of this paper can be obtained from https://github.com/hqf1996/Semantic-guided-search.

TABLE II. Statistics of datasets

| Datasets | # Nodes | # Edges | # Node-Types | # Edge-Predicates |
|---|---|---|---|---|
| DBpedia | 4,521,912 | 15,045,801 | 359 | 676 |
| Freebase | 5,706,539 | 48,724,743 | 11,666 | 5118 |
| YAGO2 | 7,308,372 | 36,624,106 | 6,543 | 101 |

### A. Experimental Setup

**Datasets.** We used three real-world datasets as shown in Table II. (1) ***DBpedia*** [1] is an open-domain knowledge base, which is constructed from Wikipedia. We used the same DBpedia dataset as [14] (authors shared it with us). (2) ***Freebase*** [3] is a knowledge base mainly composed by communities. Since we assume that each entity has a name, we used a Freebase-Wikipedia mapping file [40] to filter 5.7M entities, each entity has a name from Wikipedia. (3) ***YAGO2*** [2] is a knowledge base with information from the Wikipedia, WordNet and GeoNames. In this paper, we only used the CORE portion of Yago (excluding information from GeoName) as our dataset.

**Query workload.** We used four query workloads to construct the query graphs. (1) ***QALD-4*** [15] is a benchmark for *DBpedia*. It provides both SPARQL expression and answers for each query. A SPARQL expression may involve multiple UNION operators, which correspond to different predefined schemas in DBpedia. We selected only one UNION operator to construct the query graph. It is desired to find more answers without considering all UNION operators. (2) ***WebQuestions*** [41] is a benchmark for *Freebase*. It provides a set of questions, denoted by a quadruple $\langle$*qText*, *freebaseKey*, *relPaths*, *answers*$\rangle$. We took the entities and relations from *freebaseKey* and *relPaths* to form query graphs. (3) ***RDF-3x*** [42] contains queries for *YAGO* dataset. It provides SPARQL expressions, but does not provide the answers. To obtain the validation set, we imported YAGO2 to the graph database Neo4j and executed the queries through the sparql-plugin. (4) ***Synthetic graphs*** were generated to evaluate the effect of query graph shapes and sizes on our approach. According to [19], chain, star, tree, cycle, and flower are the commonly used graph shapes, so we generated query graphs with these shapes by extracting subgraphs from our datasets. Similar to [19], we took the number of edges (i.e., triples) in the query graph to measure the query size.

**Metrics.** We adopted two classical metrics to measure the effectiveness. *Precision* ($P$) is the ratio of correctly discovered answers over all discovered top-k answers. *Recall* ($R$) is the ratio of correctly discovered answers over all correct answers. In addition, we also employed *F1-measure* to combine the precision and recall as $F1 = \frac{2}{1/P+1/R}$.

**Comparing methods.** We compared our approach with four recent works on knowledge graph search: S4 [14], $p$-hom [17], GraB [10], and QGA [12]. S4 is a semantic pattern based solution. $p$-hom and GraB support structurally edge-to-path mapping. QGA is a keyword based approach.

There are two versions of our approach: (1) SGQ (semantic-guided query) is the implementation of A* semantic search and TA assembly in Section V. (2) TBQ (time-bounded query) is the approximate optimization in Section VI. We set the default $pss$ threshold $\tau$=0.8 and user desired path length $\hat{n}$=4. We used the TransE [43] embedding model to obtain the predicate semantic space. All the experiments were conducted on a 2.1GHZ, 64GB memory AMD-6272 server with a single core.

TABLE III. Effectiveness and efficiency over DBpedia (top-$k = 20, 40, 100, 200$)

| Methods | Precision | | | | Recall | | | | F1-measure | | | | Response Time (ms) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 |
| SGQ | **0.94** | **0.96** | **0.96** | **0.88** | **0.09** | **0.19** | **0.48** | **0.69** | **0.17** | **0.31** | **0.59** | **0.73** | **65.41** | **73.26** | **102.20** | **136.81** |
| TBQ-0.9 | 0.87 | 0.92 | 0.91 | 0.83 | 0.08 | 0.18 | 0.45 | 0.67 | 0.12 | 0.28 | 0.57 | 0.70 | **54.23** | **69.17** | **93.86** | **122.69** |
| S4 | 0.61 | 0.70 | 0.81 | 0.76 | 0.06 | 0.15 | 0.40 | 0.61 | 0.11 | 0.24 | 0.49 | 0.65 | 185.80 | 224.61 | 352.35 | 385.22 |
| GraB | 0.79 | 0.82 | 0.85 | 0.71 | 0.08 | 0.17 | 0.44 | 0.57 | 0.15 | 0.27 | 0.54 | 0.59 | 382.69 | 386.07 | 470.47 | 651.76 |
| QGA | 0.79 | 0.65 | 0.47 | 0.33 | 0.08 | 0.13 | 0.24 | 0.36 | 0.15 | 0.22 | 0.32 | 0.34 | 787.80 | 821.93 | 1127.41 | 1303.35 |
| $p$-hom | 0.37 | 0.32 | 0.29 | 0.27 | 0.05 | 0.08 | 0.18 | 0.33 | 0.10 | 0.13 | 0.22 | 0.30 | 1214.33 | 1216.83 | 1243.5 | 1269.67 |

TABLE IV. Effectiveness and efficiency over Freebase (top-$k = 20, 40, 100, 200$)

| Methods | Precision | | | | Recall | | | | F1-measure | | | | Response Time (ms) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 |
| SGQ | **0.95** | **0.95** | **0.87** | **0.73** | **0.12** | **0.24** | **0.52** | **0.74** | **0.20** | **0.36** | **0.62** | **0.68** | **115.20** | **118.82** | **147.30** | **212.83** |
| TBQ-0.9 | 0.91 | 0.90 | 0.83 | 0.68 | 0.11 | 0.21 | 0.49 | 0.68 | 0.19 | 0.31 | 0.60 | 0.62 | **100.57** | **113.87** | **133.03** | **191.05** |
| S4 | 0.79 | 0.76 | 0.65 | 0.64 | 0.09 | 0.14 | 0.38 | 0.62 | 0.14 | 0.23 | 0.45 | 0.55 | 185.82 | 212.60 | 271.43 | 342.28 |
| GraB | 0.85 | 0.87 | 0.63 | 0.54 | 0.11 | 0.18 | 0.36 | 0.55 | 0.20 | 0.29 | 0.43 | 0.50 | 345.60 | 324.80 | 357.82 | 564.75 |
| QGA | 0.88 | 0.82 | 0.60 | 0.46 | 0.11 | 0.16 | 0.33 | 0.42 | 0.20 | 0.26 | 0.43 | 0.44 | 811.98 | 1029.66 | 1567.81 | 1606.24 |
| $p$-hom | 0.35 | 0.26 | 0.23 | 0.22 | 0.06 | 0.09 | 0.14 | 0.26 | 0.10 | 0.13 | 0.17 | 0.24 | 996.22 | 1016.22 | 1077.34 | 1125.78 |

TABLE V. Effectiveness and efficiency over YAGO2 (top-$k = 20, 40, 100, 200$)

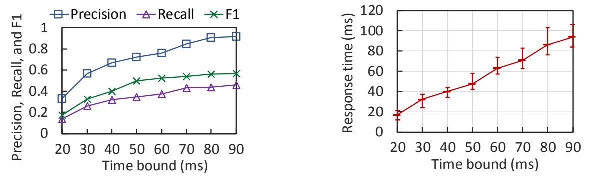
| Methods | Precision | | | | Recall | | | | F1-measure | | | | Response Time (ms) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 | $k$=20 | $k$=40 | $k$=100 | $k$=200 |
| SGQ | **0.75** | **0.76** | **0.73** | **0.69** | **0.04** | **0.07** | **0.17** | **0.33** | **0.07** | **0.13** | **0.28** | **0.45** | **113.41** | **118.80** | **131.04** | **147.42** |
| TBQ-0.9 | 0.72 | 0.74 | 0.71 | 0.67 | 0.03 | 0.07 | 0.17 | 0.32 | 0.07 | 0.13 | 0.27 | 0.43 | **102.06** | **112.46** | **124.80** | **143.20** |
| S4 | 0.64 | 0.67 | 0.65 | 0.63 | 0.03 | 0.06 | 0.15 | 0.30 | 0.06 | 0.12 | 0.25 | 0.41 | 190.64 | 212.73 | 275.11 | 364.38 |
| GraB | 0.60 | 0.64 | 0.62 | 0.59 | 0.03 | 0.06 | 0.15 | 0.28 | 0.05 | 0.11 | 0.24 | 0.38 | 287.17 | 302.25 | 468.01 | 495.33 |
| QGA | 0.65 | 0.60 | 0.57 | 0.57 | 0.03 | 0.06 | 0.14 | 0.28 | 0.06 | 0.11 | 0.23 | 0.37 | 928.84 | 949.32 | 982.68 | 1017.32 |
| $p$-hom | 0.35 | 0.40 | 0.36 | 0.34 | 0.02 | 0.04 | 0.09 | 0.17 | 0.03 | 0.07 | 0.15 | 0.23 | 622.50 | 692.45 | 717.51 | 1065.23 |

TABLE VI. The schemas of returned answers for Q117

| **Answers' schemas of $G_Q$:** ?(Automobile) $\circ\xrightarrow{\textbf{assembly}}\bullet$ Germany(Country) |
|---|
| *Automobile*–**assembly**–*Germany* |
| *Automobile*–**assembly**–*City*–**country**–*Germany* |
| *Automobile*–**manufacturer**–*Company*–**location**–*Germany* |
| *Automobile*–**manufacturer**–*Company*–**locationCountry**–*Germany* |
| *Automobile*–**assembly**–*Company*–**location**–*Germany* |
| *Automobile*–**assembly**–*Company*–**locationCountry**–*Germany* |
| *Automobile*–**designCompany**–*Company*–**location**–*Germany* |

TABLE VII. Average time (ms) of each component (top-$k$=100). C1: Query graph decomposition, C2: Semantic graph construction, C3: Semantic-guided search, and C4: TA-based assembly.

| Dataset | C1 | C2 | | C3 | | C4 |
|---|---|---|---|---|---|---|
| | | partial | complete | prune | w/o prune | |
| DBpedia | 4.30 | **6.59** | 157.33 | **88.11** | 435.67 | 3.20 |
| Freebase | 4.10 | **12.84** | 274.87 | **126.16** | 701.20 | 4.20 |
| Yago2 | 7.05 | **9.46** | 121.80 | **106.91** | 404.30 | 7.62 |



(a) Effectiveness for TBQ (b) Efficiency for TBQ

Fig. 6: Impact of response time bounds (DBpedia, top-$k$=100)

## B. *Effectiveness and Efficiency Evaluation*

**Effectiveness.** In this test, we set the time bound of TBQ as 90% of the execution time of SGQ (TBQ-0.9). Tables III-V (*Precision*, *Recall*, and *F1-measure*) show the effectiveness results over different top-k. For all datasets, our approach outperforms the others. This is because we can find the semantically similar answers following the guidance of the predicate semantics. Table VI shows some schemas of returned answers for the example query in Figure 1 (Q117 from QALD-4). Our approach can find the correct answers (e.g., with the first four schemas). It also finds some reasonable answers not given in the validation set (e.g., with the last three schemas).

**Efficiency.** Tables III-V (*Time*) report that our approach outperforms the other methods because unpromising answers are pruned significantly through the effective $pss$ estimation in runtime. It is natural that delivering more answers (larger $k$) consumes more search time. Moreover, we provide the average time of each component of our approach in Table VII: *query graph decomposition* (C1), *semantic graph construction* (C2), *semantic-guided search* (C3), and *TA-based assembly* (C4). For C2, we show the time of constructing semantic graph partially and completely. While in C3, we show the time of A* semantic search with or without $pss$ estimation. According to the results (bold), we observe that our solutions adopted in C2 and C3 outperform the baselines, which proves that the partially semantic graph construction and $pss$ estimation are very helpful to reduce the search space and improve the efficiency. Among four components, the most time-consuming one is C3. This is because we need to estimate $pss$ for each candidate node match explored in the edge-to-path mapping.

**Response Time-Accuracy Trade-off.** Figure 6 reports the effect of time bounds on TBQ. Because the results over three datasets show the similar trends, we only provide the results over DBpedia for top-$k$=100. We varied the time bound from 20 ms to 90 ms to evaluate the effectiveness and efficiency of TBQ. Figure 6(a) shows that more accurate answers can be returned as more time is given. In Figure 6(b), each bar represents the minimum, maximum, and average response times of queries. Observe that, TBQ can return the answers within a small variation of the actual time bound provided.

## C. *User Study*

Since our approach returns the top-k answers to the user, we want to know if users are satisfied with the high-ranking answers (even though the answers are already in the validation set). We expect that an answer that is more familiar to the user must have a higher rank. Therefore, we conducted a user study through *Baidu Data CrowdSourcing Platform* (https://zhongbao.baidu.com/?language=en) to evaluate the correlation between top-k answers of our approach (SGQ) and user's preference, measured by Pearson Correlation Coefficient (PCC). We did this test according to the steps in [27]. We selected 20 queries (6, 12, and 2 queries from QALD-4, WebQuestions, and RDF-3x, respectively) for this user study. For each query, we generated 30 random pairs of answers (two answers from the same pair have different schemas), and presented each pair

TABLE VIII. An example of answer pairs for user study (Q117)

| Answer1 | SGQ rank | User | Answer2 | SGQ rank | User |
|---|---|---|---|---|---|
| Opel_Super_6 | 62 | | BMW_320 | 10 | ✓ |
| Volkswagen_Passat | 72 | ✓ | 30_PS | 26 | |

TABLE IX. PCC results (DBpedia (D), Freebase (F), YAGO2 (Y))

| Query | PCC | Query | PCC | Query | PCC | Query | PCC |
|---|---|---|---|---|---|---|---|
| D1 | 0.46 | D6 | 0.74 | F5 | 0.69 | F10 | 0.73 |
| D2 | 0.56 | F1 | 0.74 | F6 | 0.37 | F11 | 0.69 |
| D3 | 0.61 | F2 | 0.72 | F7 | 0.41 | F12 | 0.77 |
| D4 | 0.75 | F3 | 0.77 | F8 | 0.71 | Y1 | 0.74 |
| D5 | 0.73 | F4 | 0.72 | F9 | 0.74 | Y2 | 0.45 |

to 10 annotators and asked for their preference (Table VIII for example). If more higher ranked answers (e.g., *BMW_320*) are preferred by more users, then we can say that the top-k answers and users' preferences are positively correlated.

Finally, we obtained 20*30*10=6000 opinions in total. We constructed two lists $X$ and $Y$ for each query based on these opinions. Each list has 30 values for 30 answer pairs. For each pair, the value in $X$ is the difference between the two answers' ranks given by SGQ, and the value in $Y$ is the difference between the numbers of annotators favoring the two answers. Then, we calculated the PCC for each query based on Eq. 10. The PCC value shows the degree of correlation between the preference given by SGQ and annotators. A PCC value in the ranges of [0.5,1.0], [0.3,0.5) and [0.1,0.3) indicates a strong, medium and small positive correlation, respectively [27]. Table IX shows that SGQ achieved strong and medium positive correlations with the annotators on 16 and 4 queries, respectively, which indicates that the users were satisfied with the semantically similar answers identified via our approach.

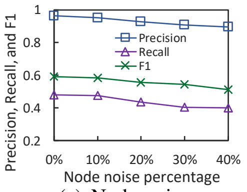
$$PCC = \frac{E(XY) - E(X) \cdot E(Y)}{\sqrt{E(X^2) - E(X)^2} \cdot \sqrt{E(Y^2) - E(Y)^2} \qquad (10)$$

## D. Effect of Query Graph Shapes and Sizes

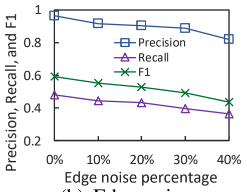
In this experiment, we evaluated the effect of query graph shapes and sizes. (1) ***Graph shapes***. We extracted the subgraphs from the original knowledge graph as the query graphs with different shapes, e.g., chain, star, tree, cycle, and flower (they are very common in knowledge graph search [19]). Figure 8 shows a *flower*-shaped query graph and its definition provided in [19]. (2) ***Graph sizes***. Similar to [19], we use the number of edges (i.e., triples) in a query graph to indicate the size of a query graph, including ***Small*** ($1 \leq |E_Q| \leq 4$) and ***Large*** ($5 \leq |E_Q| \leq 10$). (3) ***Specific nodes***. For each query graph, we randomly select 2 to 4 query nodes as specific nodes and others are target nodes. For each pair of $\langle$shape,$|E_Q|\rangle$, we have 5 query graphs. Table XI shows the experimental results.

**Effect of shapes**. The tree and flower shaped query graphs are more time-consuming than others especially for the large query sizes (e.g., 1467 ms on average for large flower graphs). This is because they have more sub-query graphs than other cases. For example, the larger flower-shaped query graphs have the most sub-query graphs (4.62 on average).

**Effect of sizes**. The large query graphs always take more time than small ones for all shapes. Since the large query graphs have more edges, more candidate node matches need to be considered in the edge-to-path mapping, which increases the time of *semantic graph construction* (C2) and *semantic-guided search* (C3). For *query graph decomposition* (C1), its running time is much smaller compared with C2 and C3. For instance,



(a) Node noise



(b) Edge noise

Fig. 7: Effectiveness vs. Noise (DBpedia, top-$k$=100)

TABLE X. Response time (ms) vs. Noise (DBpedia, top-$k$=100)

| Noise type | 0% | 10% | 20% | 30% | 40% |
|---|---|---|---|---|---|
| node noise | 102.2 | 105.3 | 114.6 | 117.3 | 126.7 |
| edge noise | 102.2 | 116.5 | 126.7 | 145.1 | 168.6 |

C1 takes only 12.47 ms on average for the large flower-shaped query graphs. Even for a quite large flower-shaped query (with 10 edges), it takes only 18.79 ms. Moreover, 90.76% of the query graphs have at most 6 edges [19], so we can say that C1 is scalable to the query size in practice. While for *TA-based assembly* (C4), the time complexity in the worst case is $O(\sum_i |M_i|)$, which is dominated by the number of sub-query graphs and $|M_i|$. If we want to find the top-k matches (e.g., $k$=100), the $|M_i|$ is usually the same order of magnitude as $k$. Besides, the number of sub-query graphs is usually small in practice (e.g., 4.62 on average for the large flower-shaped query graphs). So, the size of $\sum_i |M_i|$ is not large in practice. Hence, we conclude that C4 is also scalable to the query size.

## E. Robustness with respect to Noise

Since we assume that different users may construct different query graphs to represent the similar query intention, we investigate the impact of varying query graphs on the performance of SGQ. To construct the semantically similar but structurally different graphs, we systematically considered node noise and edge noise. (1) ***Node noise***. We added the node noise by changing the node name or type with a randomly selected synonym or abbreviation. (2) ***Edge noise***. We added the edge noise by replacing the predicate with one of its top-10 semantically similar predicates in the predicate semantic space $\boldsymbol{E}$. (3) ***Noise percentage*** is defined as the fraction of query graphs selected to add noise (varied from 10% to 40%). Figure 7 shows the results for DBpedia (top-$k$=100): (1) All effectiveness metrics decrease as the noise ratio increases, (2) SGQ is more sensitive to edge noise. This is because SGQ may misunderstand the query intention if an inappropriate predicate is given. For example, if we use *designer* to replace *assemble* in query Q117, then *Automobiles* designed by Germans would be superior to *Automobiles* assembled in *Germany*. Furthermore, Table X shows that the response time increases slightly with the growth of noise and it is sensitive to edge noise too.

## F. Scalability

This experiment studies the scalability of SGQ. We extracted two subgraphs $G_1$ and $G_2$ from DBpedia, e.g., $G_1$ has 3M nodes and 13.6M edges. Table XII shows the response time of SGQ for top-$k$={80,100,120} and the knowledge graph embedding time and memory usage. Observe that the time of SGQ increases as the graph size increases, but the change is not significant, which means that SGQ is scalable to the data size. This is because our approach can prune the unpromising candidates effectively for the different scale of the dataset.

TABLE XI: Analysis of query graph shapes and complexity (DBpedia, Response time (ms))

| **Metric** | Chain | | Cycle | | Star | | Tree | | Flower | |
|---|---|---|---|---|---|---|---|---|---|---|
| | S | L | S | L | S | L | S | L | S | L |
| Time (C1) | 4.30 | 7.57 | 7.40 | 11.00 | 7.20 | 9.10 | 6.70 | 9.87 | 9.00 | 12.47 |
| Time (C2) | 8.80 | 24.97 | 19.80 | 35.50 | 20.50 | 54.37 | 24.20 | 87.47 | 38.20 | 132.36 |
| Time (C3) | 111.70 | 687.40 | 229.00 | 762.03 | 280.30 | 834.50 | 255.90 | 1116.73 | 359.00 | 1314.90 |
| Time (C4) | 2.80 | 6.40 | 7.90 | 5.47 | 2.90 | 5.90 | 3.40 | 7.77 | 7.40 | 8.10 |
| Total | 127.60 | 726.33 | 264.10 | 814.00 | 310.90 | 903.87 | 290.20 | 1221.83 | 413.60 | 1467.83 |
| Precision | 0.89 | 0.81 | 0.90 | 0.83 | 0.89 | 0.84 | 0.92 | 0.85 | 0.84 | 0.82 |
| #sub-query (avg.) | 1.25 | 2.00 | 2.00 | 2.00 | 3.12 | 3.50 | 3.00 | 3.83 | 3.00 | 4.62 |

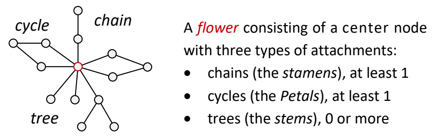

cycle
chain
tree
A *flower* consisting of a center node with three types of attachments:
chains (the *stamens*), at least 1
cycles (the *Petals*), at least 1
trees (the *stems*), 0 or more

Fig. 8: An example of *flower* shape query graph

TABLE XII. Scalability evaluation over DBpedia

| **(#Nodes, #Edges)** | SGQ: *online* (ms) | | | KG embedding: *offline* | |
|---|---|---|---|---|---|
| | $k$=80 | $k$=100 | $k$=120 | time (h) | mem (GB) |
| $G_1$(2M,9.8M) | 71.8 | 85.3 | 118.4 | 2.9 | 3.2 |
| $G_2$(3M,13.6M) | 73.3 | 91.2 | 121.9 | 4.7 | 4.6 |
| $G$(4.5M,15M) | 81.4 | 102.2 | 136.8 | 6.6 | 8.8 |

Moreover, our offline knowledge graph embedding time and memory usage are modest, e.g., within 6.6 hours and 8.8 GB.

## VIII. Conclusions

In this paper, we proposed a semantic-guided and response-time-bounded graph query to search knowledge graphs effectively and efficiently. We leveraged a knowledge graph embedding model to build the semantic graph. Then we presented an A* semantic search to find the top-k semantically similar matches from the semantic graph according to the path semantic similarity. We optimized the A* semantic search to trade off the effectiveness and efficiency within a user-specific time bound, thereby improving the system response time. The experimental results on real datasets confirm the effectiveness and efficiency of our approach.

## Acknowledgment

The second author Arijit Khan is supported by MOE Tier-1 grant 2019-T1-002-059. This work is also supported by the NSFC under grants 61602410 and 61702096; the NSF of Zhejiang under grant LGG19F020017; and the NSF of Jiangsu under grant BK20170689.

## References

[1] P. N. Mendes, M. Jakob, and C. Bizer, "Dbpedia: A multilingual cross-domain knowledge base." in *LREC*, 2012, pp. 1813–1817.
[2] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum, "Yago2: A spatially and temporally enhanced knowledge base from wikipedia," *Artificial Intelligence*, vol. 194, pp. 28–61, 2013.
[3] K. Bollacker, R. Cook, and P. Tufts, "Freebase: A shared database of structured general human knowledge," in *AAAI*, 2007, pp. 1962–1963.
[4] X. Huang, J. Zhang, D. Li, and P. Li, "Knowledge graph embedding based question answering," in *WSDM*, 2019, pp. 105–113.
[5] R. V. Guha, R. McCool, and E. Miller, "Semantic search," in *WWW*, 2003, pp. 700–709.
[6] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan, "Nema: Fast graph search with label similarity," *PVLDB*, vol. 6, no. 3, pp. 181–192, 2013.
[7] L. Zou, R. Huang, H. Wang, J. X. Yu, W. He, and D. Zhao, "Natural language question answering over rdf: a graph data driven approach," in *SIGMOD*, 2014, pp. 313–324.
[8] S. Yang, Y. Wu, H. Sun, and X. Yan, "Schemaless and structureless graph querying," *PVLDB*, vol. 7, no. 7, pp. 565–576, 2014.
[9] S. Yang, F. Han, Y. Wu, and X. Yan, "Fast top-k search in knowledge graphs," in *ICDE*, 2016, pp. 990–1001.
[10] J. Jin, S. Khemarat, and L. Gao, "Querying web scale information networks via bounding matching scores," in *WWW*, 2015, pp. 527–537.
[11] W. Zheng, L. Zou, X. Lian, J. X. Yu, S. Song, and D. Zhao, "How to build templates for rdf question/answering: An uncertain graph similarity join approach," in *SIGMOD*, 2015, pp. 1809–1824.
[12] S. Han, L. Zou, J. X. Yu, and D. Zhao, "Keyword search on rdf graphs-a query graph assembly approach," in *CIKM*, 2017, pp. 227–236.
[13] S. Shekarpour, E. Marx, S. Auer, and A. Sheth, "Rquery: rewriting natural language queries on knowledge graphs to alleviate the vocabulary mismatch problem," in *AAAI*, 2017, pp. 3936–3943.
[14] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao, "Semantic sparql similarity search over rdf knowledge graphs," *PVLDB*, vol. 9, no. 11, pp. 840–851, 2016.
[15] "Qald-4," http://qald.aksw.org/index.php?x=challenge&q=4.
[16] S. S. Bhowmick, B. Choi, and S. Zhou, "Vogue: Towards a visual interaction-aware graph query processing framework," in *CIDR*, 2013.
[17] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, "Graph homomorphism revisited for graph matching," *PVLDB*, vol. 3, pp. 1161–1172, 2010.
[18] N. Nakashole, G. Weikum, and F. Suchanek, "Patty: a taxonomy of relational patterns with semantic types," in *EMNLP-CoNLL*, 2012.
[19] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large SPARQL query logs," *PVLDB*, vol. 11, no. 2, pp. 149–161, 2017.
[20] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001.
[21] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: answering sparql queries via subgraph matching," *PVLDB*, 2011.
[22] J. Cheng, J. X. Yu, B. Ding, S. Y. Philip, and H. Wang, "Fast graph pattern matching," in *ICDE*, 2008, pp. 913–922.
[23] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Strong simulation: Capturing topology in graph pattern matching," *ACM TODS*, 2014.
[24] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao, "Neighborhood based fast graph search in large networks," in *SIGMOD*, 2011, pp. 901–912.
[25] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao, "Graph similarity search with edit distance constraint in large graph databases," in *CIKM*, 2013, pp. 1595–1600.
[26] Z. Zeng, A. K. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 25–36, 2009.
[27] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri, "Querying knowledge graphs by example entity tuples," *IEEE TKDE*, 2015.
[28] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas, "Exemplar queries: a new way of searching," *PVLDB*, vol. 25, pp. 741–765, 2016.
[29] M. H. Namaki, Q. Song, Y. Wu, and S. Yang, "Answering why-questions by exemplars in attributed graphs," in *SIGMOD*, 2019, pp. 1481–1498.
[30] Q. Song, M. H. Namaki, and Y. Wu, "Answering why-questions for subgraph queries in multi-attributed graphs," in *ICDE*, 2019, pp. 40–51.
[31] M. H. Namaki, Y. Wu, and X. Zhang, "Gexp: Cost-aware graph exploration with keywords," in *SIGMOD*, 2018, pp. 1729–1732.
[32] P. Peng, L. Zou, and R. Guan, "Accelerating partial evaluation in distributed sparql query evaluation," in *ICDE*, 2019, pp. 112–123.
[33] X. Zhang and L. Zou, "IMPROVE-QA: an interactive mechanism for RDF question/answering systems," in *SIGMOD*, 2018, pp. 1753–1756.
[34] S. Hu, L. Zou, and X. Zhang, "A state-transition framework to answer questions over knowledge base," in *EMNLP*, 2018, pp. 2098–2108.
[35] W. Zheng, J. X. Yu, L. Zou, and H. Cheng, "Question answering over knowledge graphs: question understanding via template decomposition," *PVLDB*, vol. 11, no. 11, pp. 1373–1386, 2018.
[36] H. Ma, M. Alipourlangouri, Y. Wu, F. Chiang, and J. Pi, "Ontology-based entity matching in attributed graphs," *PVLDB*, 2019.
[37] N. Nakashole, T. Tylenda, and G. Weikum, "Fine-grained semantic typing of emerging entities," in *ACL*, 2013, pp. 1488–1497.
[38] R. Navigli and S. P. Ponzetto, "Babelnet: Building a very large multilingual semantic network," in *ACL*, 2010, pp. 216–225.
[39] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
[40] "Freebase links," http://downloads.dbpedia.org/2016-10/core-i18n/en/.
[41] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic parsing on freebase from question-answer pairs," in *EMNLP*, 2013, pp. 1533–1544.
[42] T. Neumann and G. Weikum, "Rdf-3x: A risc-style engine for rdf," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.
[43] A. Bordes, N. Usunier, A. García-Durán, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *NIPS*, 2013, pp. 2787–2795.