A Generic Ontology Framework for Indexing Keyword Search on Massive Graphs

Jiaxin Jiang, Byron Choi, Jianliang Xu and Sourav S Bhowmick

Abstract—Due to the unstructuredness and the lack of schema information of knowledge graphs, social networks and RDF graphs, keyword search has been proposed for querying such graphs/networks. Recently, various keyword search semantics have been designed. In this paper, we propose a generic ontology-based indexing framework for keyword search, called Bisimulation of Generalized Graph Index (BiG-index), to enhance the search performance. The novelties of BiG-index reside in using an ontology graph \( \mathcal{G}_{\text{Out}} \) to summarize and index a data graph \( \mathcal{G} \) iteratively, to form a hierarchical index structure \( \mathcal{C} \). BiG-index is generic since it only requires keyword search algorithms to generate query answers from summary graphs having two simple properties. Regarding query evaluation, we transform a keyword search \( q \) into \( \mathcal{Q} \) according to \( \mathcal{G}_{\text{Out}} \) in runtime. The transformed query is searched on the summary graphs in \( \mathcal{C} \). The efficiency is due to the small sizes of the summary graphs and the early pruning of semantically irrelevant subgraphs. To illustrate BiG-index’s applicability, we show popular indexing techniques for keyword search (e.g., Blinks and \( r \)-clique) can be easily implemented on top of BiG-index. Our extensive experiments show that BiG-index reduced the runtimes of popular keyword search work Blinks by 50.5% and \( r \)-clique by 29.5%.

1 INTRODUCTION

Knowledge graphs, social networks and RDF graphs have a wide variety of emerging applications, including semantic query processing [33], information summarization [28], community search [11], collaboration and activities organization [26] and user-friendly query facilities [31]. Such networks often lack useful schema information for users to formulate their queries. Keyword search has been proposed to make it easier for users to query data from such networks/graphs. In a nutshell, a user specifies a set of keywords \( Q \) on a data graph \( \mathcal{G} \) as his/her query. Depending on the search semantics, the answer of \( Q \) is the subgraphs that either contain the keywords and/or are ranked as top-\( k \) subgraphs. For instance, Google’s knowledge graph search API\(^1\) helps users to find the answer in their knowledge database. The query answers are returned in the form of subgraphs. The subgraph answers (a) make it easy for users to explore some relevant keywords and (b) are connected and present the relationships among the query keywords.

Recently, there has been a stream of works proposing various interesting keyword search semantics on massive networks/graphs, as well as various relevant techniques for optimizing their query performance (e.g., [12], [15]). Furthermore, the knowledge graphs are large in size. For instance, the latest version of one semantic knowledge base, YAGO, contains 4.5 million entities and 24 million facts. Some previous approaches face scalability challenges. This warrants a revisit to the research on keyword search.

Ontology information, such as information of properties, classes, and their super classes, is typically encoded in an ontology graph and often accompanied with knowledge graphs. Some existing techniques exploit it to aid users to formulate their queries. In comparison, this paper proposes a generic framework,

\[ \mathcal{G}_{\text{Out}} \]

that the Bisimulation of Generalized Graph Index (BiG-index), that exploits ontology information to index the graphs themselves. Moreover, this paper illustrates how existing research on keyword search can be implemented on top of BiG-index with minor modifications.

More specifically, given an ontology graph \( \mathcal{G}_{\text{Out}} \), we index a data graph \( \mathcal{G} \) into a hierarchy of summary graphs \( \mathcal{C} \) as follows. Some node labels of a graph \( \mathcal{G} \) are generalized with respect to \( \mathcal{G}_{\text{Out}} \) to yield a generalized graph that has a high potential for compression/summarization. It is then summarized by using classical graph summarization methods\(^2\) to obtain a summary graph. The benefits of deriving summary graphs are threefold. First, a summary graph can often be iteratively generalized and summarized further until it cannot be further summarized efficiently. Second, queries are evaluated over the summarized graphs so as to avoid duplicated computation and save I/O costs. The query results at the generalized layers are decompressed only when they are qualified to form query answers. Third, the summary graphs are yet another set of graphs. Existing indexing techniques for keyword search can be readily applied to them.

2. As a proof of concept, we adopt bisimulation for graph summarization. In a nutshell, bisimulation is a path-preserving formalism. Since some representative keyword search semantics are defined with paths, the support of their query algorithms requires little modifications.

---

\( \mathcal{G}_{\text{Out}} \)

1. https://developers.google.com/knowledge-graph/

2. As a proof of concept, we adopt bisimulation for graph summarization. In a nutshell, bisimulation is a path-preserving formalism. Since some representative keyword search semantics are defined with paths, the support of their query algorithms requires little modifications.
Example 1.1: We present a simplified knowledge graph \( G \) in Fig. 1 and its (partial) ontology graph \( G_{\text{Ont}} \) in Fig. 2. We generalize some labels of \( G \), according to \( G_{\text{Ont}} \), to obtain a generalized graph \( G_{\text{gen}} \), as shown in Fig. 3. Note that there are more similar subgraphs after generalization (e.g., the \( \text{Univ.} \) subgraphs). We then summarize \( G_{\text{gen}} \) (detailed in Sec. 2) and yield a small summary graph \( G' \), as shown in Fig. 4.

Consider the keyword query semantics [1] for example. It finds a subtree that contains the query keyword and whose paths are smaller than a user-specified threshold \( d_{\text{max}} \). Suppose the query keywords \( Q_1 \) are \{Massachusetts, Ivy League, California\} and the threshold \( d_{\text{max}} = 3 \). A possible answer is the (red bold) subtree whose root is \( P. \ Graham \) and whose leaf vertices contain \( Q_1 \).

The example query above reveals a few efficiency issues of keyword search. First, some processing would not eventually yield answers. The vertices \{S. Russell, …, A. Rodger\} are candidate answers and are pruned after some verification. Then, suppose these vertices are generalized to \( \text{Person} \) vertices, which have the same structure, as shown in \( G_{\text{gen}} \). Hence, they can be summarized, as shown in \( G' \). As a result, we can verify whether or not the summarized \( \text{Person} \) can form an answer once, rather than doing this \textit{100 times} as in Fig. 4 and Fig. 3.

Consider another query \( Q_2 = \{ \text{California, P. Graham} \} \) if the same \( d_{\text{max}} \) of \( Q_1 \). One possible evaluation is to traverse the graph \( G \) in a breadth first manner bidirectionally, to check distances between these nodes. Such traversals can be more efficient in the summary graph \( G' \). In the example, \textit{California} (resp., \textit{P. Graham}) is generalized as \textit{Western} (resp., \textit{Academics}). When searching forward from \textit{P. Graham} in \( G \), \textit{Harvard Univ.} and \textit{Cornell Univ.} are pruned by one since they cannot reach \textit{California} within \( d_{\text{max}} \) hops. In contrast, the subgraph rooted at \textit{Univ.} does not connect to the keyword \textit{Western} within \( d_{\text{max}} \) hops. Hence, it is pruned from \( G' \) and both \textit{Harvard Univ.} and \textit{Cornell Univ.} are pruned together.

We also remark that ontology information naturally helps users, who often lack detailed knowledge of the data graph, to formulate their queries. Consider a query \( Q_3 = \{ \text{Person, Univ., Startup} \} \). The keywords are the generalized ones. By the original search semantics, the answer of \( Q_3 \) is an empty set. However, in this example, the subtree rooted at \textit{P. Graham} is a possible answer generated from the summary graph.

To the best of our knowledge, ontology information has not been used to index graphs before. A naive method is to generalize all the labels of nodes a layer at a time until the generalization is not possible anymore and the queries are evaluated from the topmost layer to the data graph layer. However, this method can be inefficient for two reasons. The index hierarchy can be as deep as the ontology graph. When keyword queries are evaluated deep in the hierarchy, the answers at the data graph layer require many steps to recover. Second, as the summary graphs are iteratively generalized, the “compression” potentials diminish. The saving in I/O from evaluating queries with summary graphs reduces.

Contributions. This paper proposes a generic hierarchical index of summary graphs by exploiting an ontology graph, called BiG-index. The overall framework of BiG-index is visualized in Fig. 5. Specifically, this paper makes these contributions:

- We formalize the hierarchical graph summaries BiG-index and the core operations of BiG-index. We propose a cost model to determine a space- and query-efficient BiG-index.
- We introduce a cost-based generalization of query keywords. We then propose a query evaluation algorithm on BiG-index. We propose various optimizations for query answer generation such that the subgraphs that are irrelevant to the keyword search and intermediate query results are minimized.
- We illustrate how the existing algorithms for keyword search can be implemented on top of the BiG-index framework by taking Blinks [12] and r-clique [15] as examples which contain the fundamental operations including backward traversal, ranked keyword search and shortest path evaluation. Therefore, the proposed framework is fundamental and aims to be orthogonal to specific query semantics.
- We conduct extensive experiments on BiG-index and the results show that BiG-index can reduce the runtimes of some popular keyword search work such as Blinks on average by 50.5% and r-clique on average by 29.5%.

Organization. The rest of the paper is organized as follows: Sec. 2 presents the background, and problem statement. Sec. 3 presents BiG-index and its cost model for efficient index construction. Sec. 4 presents how to implement keyword search on the top of BiG-index. Sec. 6 presents the experimental evaluation. Sec. 7 discusses the related work. Sec. 8 concludes the paper.
### 2 Background and Problem Statement

This section presents the background for the technical discussions. Some notations needed are summarized in Tab 1. It then presents the problem statement of the paper.

**Knowledge base.** We discuss our technical development with knowledge graphs as they are often associated with an ontology graph. The support of general graphs (by associating types to nodes using existing ontology graphs or tools) is discussed in Appendix A.2. A knowledge base consists of a set of entities \( V_E \) and a set of attributes \( V_A \). Each entity \( e \in V_E \) has some attributes \( A \subset V_A \). There might be some relationships among the entities. Hence, it is natural to construct the knowledge base as a graph.

**Knowledge graphs.** This paper considers a directed and labeled graph \( G = (V, E, L, \Sigma) \), where (a) \( V \) is a set of vertices, such that \( V_E \subseteq V \) and \( V_A \subseteq V \); (b) \( E \subseteq V \times V \) is a set of edges; (c) \( \Sigma \) is a set of labels; and (d) \( L \) is a mapping function defined on \( V \) such that for each vertex \( v \in V \), \( L(v) \) maps \( v \) to a label in \( \Sigma \). We use labels to model the values of entities or attributes and may use types or keywords to refer to them when they are more intuitive to the discussions.

This paper uses the number of nodes and edges as the graph size, denoted as \(|G| = |V| + |E|\).

**Graph bisimulation.** This paper adopts a classical graph formalism [18], namely backward bisimulation (or simply bisimulation), to summarize graph structures. A binary relation \( B \subseteq V \times V \) is a bisimulation relation of a graph \( G \). For each \( (u_i, v_j) \in B \), the following hold:

- \( L(u_i) = L(v_j) \);
- for each \( (u_i, v_j) \in E \), there is an edge \( (u_j, v_j) \in E \) and \( (v_i, v_j) \in B \); and
- for each \( (u_j, v_j) \in E \), there is an edge \( (u_i, v_i) \in E \) and \( (v_i, v_j) \in B \).

We remark that due to different applications, there have been various bisimulation definitions, such as backward and/or forward bisimulations. This paper adopts backward bisimulation since it seamlessly aligns with the graph traversals of popular keyword search algorithms (e.g., [12], [15]).

Given a graph \( G \), there is a unique maximal bisimulation relation \( B \). \( B \) is an equivalent relation of nodes \( V \), i.e., \( B \) is reflexive, symmetric, and transitive. Hence, a vertex is bisimilar to itself. We denote the equivalent class containing vertex \( v \) as follows: \( \equiv(v) = \{ u \in V \mid (u, v) \in B \} \) or simply \( [v]_{\equiv} \).

Bisimulation can also be intuitively considered as a graph \( \text{Bisim}(G) \) in which the nodes of an equivalence class of \( G \) are represented by a supernode. We use \( \text{Bisim}(v) \) to denote the supernode of \( v \), where \( v \in V \). A graph \( G \) has a \( \text{Bisim}(G) \) of the smallest size by applying the maximal bisimulation relation.

Many index schemes, e.g., [3], [4], [16], [19], have been derived from such a bisimulation graph. When \( \text{Bisim}(G) \) is used as an index, an efficient “reverse” method (denoted as \( \text{Bisim}^{-1} \)) is needed for answer generation. In this paper, \( \text{Bisim}^{-1} \) is implemented by hash tables.

**Graph summarization.** Given a graph \( G = (V, E, L, \Sigma) \), the support of \( G \), we define the summary graph \( G_{\text{Spec}}(G) = (V', E', L', \Sigma') \) which can be regarded as yet another graph, where

- \( V' = \{ [v]_{\equiv} \mid v \in V \} \);
- \( E' = \{ ([u]_{\equiv}, [v]_{\equiv}) \mid (u, v) \in E \} \);
- \( L'(\[v\]_{\equiv}) = L(v) \); and \( \Sigma' = \Sigma \).

Intuitively, each vertex \([v]_{\equiv} \in V'\) stands for a set of equivalent vertices \( U \), where \( U \subseteq V \) (e.g., \( U = \{ u_1, u_2, \ldots, u_n \} \) and \( (u_1, u_2) \in B \). Specifically, for each \( v \in V \), there is a vertex \([v]_{\equiv} \in V'\), denoted as \( \text{Bisim}(v) = [v]_{\equiv} \). Similarly, for each edge \((u, v) \in E\), there is an edge \(([u]_{\equiv}, [v]_{\equiv}) \in E'\).

**Definition 2.1:** A summarization \( S \) of a graph \( G \) is path-preserving if a path \((u_1, u_2, \ldots, u_i) \in G \) (denoted as \( S(u_i) \)) simultaneously replaces the supertypes with their subtypes when \( S(u_i) = S(u_i) \).

It is straightforward to verify that bisimulation \( \text{Bisim} \) has the path-preserving property. This is a key reason why \( \text{Bisim} \) is adopted since popular keyword search algorithms involve numerous traversals of graphs, e.g., [12], [15].

**Ontology for label generalization.** Next, ontology information is modeled by a directed acyclic graph called an ontology graph, denoted as \( G_{\text{Ont}} = (V_{\text{Ont}}, E_{\text{Ont}}) \), where (a) \( V_{\text{Ont}} \) is a set of vertices, where \( \ell \) is a label (which models types, etc.), \( \ell \in V_{\text{Ont}} \); and (b) each edge \((\ell', \ell) \in E_{\text{Ont}} \) is labeled with \( \text{SubClassOf} \) or \( \text{SubTypeOf} \) of \( s.t. \ell', \ell \in V_{\text{Ont}} \). \( \ell' \) is also referred to as a direct supertype of \( \ell \). Next, we present some definitions for using ontology to generalize graphs.

**Generalization configuration \((C)\).** Given a knowledge graph \( G = (V, E, L, \Sigma) \) and its ontology graph \( G_{\text{Ont}} \), a generalization configuration \((C)\) is a set of mappings, and each mapping is denoted as \((\ell \rightarrow \ell')\), where \( \ell, \ell' \in \Sigma \) and \( \ell' \) is either (i) a supertype of \( \ell \) s.t. \( \ell, \ell' \in V_{\text{Ont}} \) and \((\ell', \ell) \in E_{\text{Ont}} \) or (ii) \( \ell \leftrightarrow \ell' \) when \( \ell \) has no supertype.

**Graph generalization (Gen) and specialization (Spec).** A generalization \( \text{Gen} \) of a graph \( G \) w.r.t. an ontology graph \( G_{\text{Ont}} \) and a configuration \( C \), denoted as \( \text{Gen}(G, C) \), is to simultaneously apply all mappings of \( C \) to the labels of the vertices of \( G \) to obtain a generalized graph \( GC \). Specialization is the reverse process of generalization to recover the original graph \( G \) from \( GC \). Given a generalized graph \( GC \) and a configuration \( C \), \( \text{Spec}(GC, C) \) simultaneously replaces the subtypes with their supertypes according to \( C \). For simple exposition, we may skip \( C \) when it is clear from the context.

**Definition 2.2:** A generalized graph \( GC = \text{Gen}(G, C) \) is label-preserving under \( C \) only if for each \( v \in V_{\text{GC}} \), either (i) \( \ell \leftrightarrow \ell' \) or (ii) the label of \( v \) in \( G \) and \( GC \) are the same.

**Example 2.1:** Consider the data graph \( G \) in Fig. 1 and the ontology graph \( G_{\text{Ont}} \) in Fig. 2. Suppose \( S. \text{Russell} \), ..., A.

---

**TABLE 1: Frequently used notations**

<table>
<thead>
<tr>
<th>Notations</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a / A )</td>
<td>An answer graph / an answer graph set</td>
</tr>
<tr>
<td>( G_{\text{Ont}} = (V_{\text{Ont}}, E_{\text{Ont}}) )</td>
<td>An ontology graph, where ( \ell \in V_{\text{Ont}} ) is a type and ((\ell', \ell) \in E_{\text{Ont}} ) denotes ( \ell' ) is a supertype of ( \ell ).</td>
</tr>
<tr>
<td>Gen/Spec</td>
<td>Generalization/Specialization function</td>
</tr>
<tr>
<td>Bisim / Bisim(^{-1})</td>
<td>The summarization function and its reverse. We adopt the bisimulation formalism</td>
</tr>
<tr>
<td>( \equiv(v) / [v]_{\equiv} )</td>
<td>An equivalent relation of ( v )</td>
</tr>
<tr>
<td>( \chi )</td>
<td>The indexing function for generalizing and summarizing a graph ( G ) with a configuration ( C ): ( \chi(G, C) = \text{Bisim}(\text{Gen}(G, C)) )</td>
</tr>
<tr>
<td>( \chi^{-1} )</td>
<td>The function for recovering the graph ( G ): ( \chi^{-1}(G, C) = \text{Spec}(\text{Bisim}^{-1}(G, C)) )</td>
</tr>
</tbody>
</table>
Rodger are generalized to Person (in the dotted rectangle of Fig. 3). UC Berkeley (resp. California) is generalized to Univ. (resp. Western). The 100 Persons are bisimilar because 1) they have the same label Person, and 2) their child node (Univ) is bisimilar since the bisimulation is a reflexive relation, i.e., the Univ node is bisimilar to itself. The 100 Persons can be summarized with one supernode (as shown in the dotted box in Fig. 4), due to bisimulation. It can be observed that C” in Fig. 4 is path-preserving, e.g., the path (S. Russell, UC Berkeley, California) is summarized into (Person, Univ., Western), and label-preserving.

Exact keyword search. Several keyword query semantics have been proposed. For instance, He et al. [12] proposes that a keyword query is a 2-ary tuple \( (Q, d_{\text{max}}) \) which contains a set of keywords \( Q = \{q_1, \ldots, q_n\} \) and a distance bound \( d_{\text{max}} \). Given a graph \( G = (V, E, L, \Sigma) \), a match of \( Q \) in \( G \) is a subtree of \( G \), denoted as \( T = \{r, p_1, \ldots, p_n\} \), such that

1) \( T \) is a tree rooted at \( r \);
2) \( p_i \) is a leaf vertex of \( T \) and \( L(p_i) = q_i \); and
3) \( \text{dist}(r, p_i) \leq d_{\text{max}} \).

To find the top-\( k \) answers, He et al. [12] modified the above query semantic that computes the tree which has the smallest \( \sum_{i \in [1,n]} \text{dist}(r, p_i) \), i.e., any tree \( T' = \{r', p'_1, \ldots, p'_n\} \) that satisfies Conditions (1)-(3) implies the following:

\[
\sum_{i \in [1,n]} \text{dist}(r', p'_i) \geq \sum_{i \in [1,n]} \text{dist}(r, p_i)
\]  

(1)

Some existing works propose top-k keyword search semantics [9], [12], [23] that rank the answer graphs on the basis of \( \sum_{i \in [1,n]} \text{dist}(r, p_i) \) and its variations [8].

Intuitively, this paper proposes graph generalization and summarization for an index function \( \chi \) (Sec. 3), which takes a data graph and an ontology graph as input and produces an index structure as output (Def. 3.1), for many keyword search semantics. Moreover, it proposes its reverse function \( \chi^{-1} \) (Sec. 4.2) to support efficient answer generation. Next, we present the problem statement. Condition 1 of Def. 2.3 is the correctness and approach; and Condition 2 is its efficiency.

Definition 2.3: (Problem statement) Given a graph \( G \), its ontology graph \( G_{\text{Ont}} \), a keyword query \( Q \), and a keyword search algorithm \( f \), this paper studies the index function \( \chi \) and answer graph generation function \( \chi^{-1} \) s.t.

1) \( \text{eval}(G, Q, f) = \text{eval}_{\text{Ont}}(G_{\text{Ont}}, Q, f) = \text{eval}(\chi^{-1}(\chi(G), \chi(Q), f), Q) \); and
2) the query time of \( \text{eval}_{\text{Ont}} \) is minimized,

where \( \text{eval} \) and \( \text{eval}_{\text{Ont}} \) are query evaluations with \( f \) on graphs and BiG-index, respectively.

As the first effort to address the problem, we temper the claim of supporting arbitrary \( f \).s. The technical discussions of \( f \)s only assume \( \chi \) possesses label- and path-preserving properties. The two properties are simple. Popular keyword search algorithms [12], [15] can be readily implemented when this assumption holds.  

3. BISIMULATION OF GENERALIZED GRAPH INDEX BiG-index

In this section, we propose the Bismulation of Generalized Graph Index (BiG-index) and its cost model for index construction. The main ideas of BiG-index are that (i) the labels of a graph are generalized on the basis of the ontology, (ii) the generalized graph is summarized, and (iii) these two steps are repeated alternately to form a hierarchical index structure.

3.1 Index Definition

In this paper, we use bismulation as a summarization function since it is path-preserving (see Sec. 2). The index function \( \chi \) is defined as follows.

Definition 3.1: (BiG-index) The BiG-index of a graph \( G \) and its ontology graph \( G_{\text{Ont}} \) are defined to be a binary tuple \((\mathcal{G}, \mathcal{C})\), where \( \mathcal{G} \) is a set of graphs \( \{G^0, \ldots, G^h\} \), \( \mathcal{C} \) is a sequence of label-preserving generalization configurations \( \{C^1, \ldots, C^h\} \), and

\[
G^i =\begin{cases} 
G, & \text{if } i = 0, \\
\chi(G^{i-1}, C^i), & \text{otherwise}
\end{cases}
\]  

(2)

where \( C^i \) is the configuration at \((i - 1)\)-th layer.

To illustrate the details of Def. 3.1, we discuss the simple case of \( h = 1 \). Given a data graph \( G^0 \), we construct the index by graph generalization \( \text{Gen} \) and graph summarization \( \text{Bism} \). The index is denoted as \( G^1 = \text{Bism}(\text{Gen}(G^0, C^1)) \).

(i) Graph generalization (\( \text{Gen} \)). As described in Sec. 2, a generalization \( \text{Gen}(G^0, C^1) \) simultaneously replaces the labels of the vertices with generalized ones, as specified in \( C^1 = \{(\ell_1 \rightarrow \ell_1'), \ldots, (\ell_m \rightarrow \ell_m')\} \). \( C^1 \) denotes that \( \ell_i \in \Sigma \) is generalized to \( \ell_i' \), where \( \ell_i' \) is one of the supertypes of \( \ell_i \) in \( G_{\text{Ont}} \) for \( 1 \leq i \leq m \).

(ii) Graph summarization (\( \text{Bism} \)). Bismulation of a graph \( G \) can be regarded as yet another graph. An equivalent relation of vertices \( U, \text{ where } U \subseteq V_G \) (e.g., \( U = \{u_1, u_2, \ldots, u_n\} \) ), can be represented by a supernode \( s \) (e.g., \( s = [u_1\text{equiv} = \ldots = [u_n\text{equiv}] \)). Then, we denote the summary graph of a graph \( G \) as \( \text{Bism}(G) \) by applying the maximal bismulation relation \( B \) on \( G \) as elaborated in Sec 2.

Since the generalized summary graph \( G^1 \) is yet another graph, we can apply the abovementioned process recursively to construct a hierarchy of graphs. We denote the original graph \( G \) as \( G^0 \). In a recursive call, we have \( G^{i+1} = \chi(G^i, C^i) \), where \( i \geq 0 \). We omit \( C^i \), but use \( G^{i+1} = \chi(G^i) \), when \( C^i \) is not relevant to the discussion. We shall present the termination condition of the construction procedure after the elaborations of the cost model for BiG-index in Sec. 3.2.

Discussions. The main difference between BiG-index and previous studies that exploit bismilation for query processing (e.g., [3], [10]) is that previous work exploits only the graph topologies, whereas BiG-index indexes graphs by exploiting both the graph topologies and ontology information.

3.2 Cost Model for BiG-index Construction

BiG-index affects query performance in non-trivial ways. The intuitions are that, firstly, since BiG-index is an index, it should be small in size for efficient query processing. Secondly, generating answer graphs from the data graph requires specializing generalized candidate answers. As generalizations in indexing cause semantic distortion, querying them leads to false answers, in the candidate answers. Hence, the distortion should be small to reduce...
checking of false answers. These two objectives however are competing. We propose a cost model for modeling the performance of $\chi$ of a given graph $G$ and a configuration $C$ as follows.

$$\text{cost}(G, C) = \alpha \times \text{compress}(G, C) + (1 - \alpha) \times \text{distort}(G, C),$$  \hspace{1cm} (3)$$

where $\alpha (0 \leq \alpha \leq 1)$ is the weight of compress and distort. Next, we elaborate the two components of cost.

(i) Compression ratio ($\text{compress}$). Given a data graph, a summarization function $\text{Bisim}$, and the corresponding summary graph, we define the compression ratio by the ratio of the size of the summary graph to the data graph. The details of estimating compress are given at the end of this section.

(ii) Semantic distortion ($\text{distort}$). Generalizations introduce semantic distortion because they replace specific labels with generalized ones. Intuitively, the less the distortion, the less the chance a query requires filtering out irrelevant generalized summaries. Consider a configuration $C = \{l_1 \rightarrow \ell_1', l_2 \rightarrow \ell_2', \ldots, l_n \rightarrow \ell_n'\}$. Denote the domain of $C$ as $X = \{l_i \mid (l_i \rightarrow \ell_i') \in C\}$ and the image as $Y = \{\ell_i' \mid (l_i \rightarrow \ell_i') \in C\}$. The term distortion of $l_i$ due to $C$, where $(l_i \rightarrow \ell_i') \in C$, is defined as $\text{distort}(l_i) = 1 - \frac{|X|}{|Y|}$, where $|X|$ is the number of the labels that are also generalized to the supertype $\ell_i'$ (i.e., $X = \{l_i \mid (l_i \rightarrow \ell_i') \in C\}$). It quantifies the cost to distinguish $\ell_i$ from the other labels that are generalized to the same supertype $\ell_i'$. Then, the basic distortion of a configuration $C$ is the normalized semantic distortion of labels, as follows:

$$\text{distort}(G, C) = \frac{1}{|X|} \sum_{l_i \in X} \text{distort}(l_i).$$

Note that small distortions of the frequent labels of the data graph could lead to high distortion. Hence, we take the support of the label $l_i$, $\text{sup}(l_i) = |V_{l_i}|/|V|$, where $V_{l_i} = \{v \mid v \in V \text{ and } L(v) = l_i\}$. The proposed distortion $\text{distort}(G, C)$ function is as follows:

$$\text{distort}(G, C) = \frac{1}{|X|} \sum_{l_i \in X} \text{distort}(l_i) \times \text{sup}(l_i).$$

Example 3.1: Consider the data graph $G$ in Fig. 1 and the ontology graph $G_{\text{Ont}}$ in Fig. 2 again. Suppose $C = \{\text{P. Graham} \rightarrow \text{Investor} \}, \{\text{W. Buffett} \rightarrow \text{Investor} \}$. Hence, $\text{distort(}P. \text{Graham}) = \text{distort(}W. \text{Buffett}) = 1/2$. Further, suppose there are $n_1$ labels, such as $l. \text{ Thomas}, \ldots, \text{ Bruce Lee}$, that are generalized to $\text{Person}$. Then, $\text{distort(l. Thomas)} = \ldots = \text{distort(Bruce Lee)} = 1 - 1/n_1$.

It is not surprising that the general problem of optimal index construction is NP-hard. The proof is a reduction from the maxSAT problem. The details are presented in [13].

Theorem 3.1. [OptGen] Given a graph, computing a configuration s.t. cost (Formula 3) is minimum is NP-hard.

Estimated the cost of $\chi$ of $G$ using $C$. The construction of BiG-index involves the search of a configuration that yields a small cost. The number of the possible generalizations is $O(2^{2^{|G|}})$. In addition, it is computationally costly to compute compress of cost, because it involves the data graph.

Graph sampling. A simple fact is that we do not need to explore the whole graph because most keyword search semantics are bounded by $r$ hops. We sample some subgraphs from the data graph, such that their radii are $r$. The average compress values of sample subgraphs due to a configuration gives an indication of compress. We verified by experiments that the estimates accurately indicate the relative compress values of the generalizations.

To sample a subgraph, we randomly select a vertex $v$ from the graph. We determine the vertices which are reachable from $v$ within $r$ hops to form a vertex set, denoted as $V_{r}$. The sample graph is the node-induced subgraph of $V_{r}$. The sampled graphs are obtained by sampling such node-induced subgraphs $n$ times.

Assume that the differences in the compress values of the sampled graphs follow a Gaussian distribution and have a zero mean. The sample size can be determined by estimation of proportion. Given a specific error bounded $E$, we determine the sample size $n$ by the formula $n = 0.5 \times 0.5(\frac{z}{E})^2$, where $z$ is the standard normal value corresponding to a desired level of confidence. For example, when the maximum allowable error $E = 5\%$ and $z = 1.96$, the sample size $n = 400$.

As either the number of hops $r$ or the number of sample graphs increases, the possibilities of compressing the nodes are closer to the accurate compress value. However, the estimated cost increases as well. Hence, we determine the parameters by experiments to efficiently determine the compress cost.

Index construction. Due to the hardness of computing the optimum configuration, we propose a heuristic algorithm, shown in Algo. 1. to construct the index layer by layer. Initially, the configuration $C$ is empty. Given a generalization $c_i = (l_i \rightarrow \ell_i' \in E_{\text{ont}}$, w.r.t the data graph $G$ and its ontology $G_{\text{Ont}}$, we estimate the cost of $c_i$ as $\text{cost}(G, \{c_i\})$ (Formula 3). compress and distort of Formula 3 are computed as discussed in this subsection. We then maintain a priority queue, in the ascending order of the estimated costs. We then check the generalization $c_i$ from the front of the queue iteratively. If the cost of $C \cup \{c_i\}$ is smaller than a user-defined threshold $\theta$, we set $C$ to $C \cup \{c_i\}$. Otherwise, we simply return $C$. This procedure terminates when the queue is empty or the size of $C$ is larger than a user-defined budget $\Pi$. We then use $C$ returned by Algo. 1 to generalize and summarize $G$ to obtain $G'$. Users may increase the value of $\theta$ and apply Algo. 1 on $G'$ and $G_{\text{Ont}}$ again to obtain another layer of the index.

Maintenance of BiG-index. When the data graph is updated, the summary graph hierarchy $G$ must be maintained. We adopt the efficient maintenance of the BiSim of $G$ as the practical incremental maintenance algorithm of BiSim, (e.g., [7]).

While ontologies are often static, in the case of their updates, BiG-index is maintained as follows: (i) new ontologies do not...

---

**Algorithm 1:** One-step heuristic for determining a maximal configuration

**Input:** A data graph $G$, an ontology graph $G_{\text{Ont}}$, a cost threshold $\theta$ and max. number of generalizations $\Pi$

**Output:** A configuration $C$

1. initialize the configuration $C = \emptyset$
2. initialize a priority query $C_{\text{grady}}$ in the ascending order of the estimated cost
3. for $c_i = (l_i \rightarrow \ell_i') \in E_{\text{ont}}$ and $l_i \in \Sigma$ do
4. \hspace{0.5cm} $C_{\text{grady}}$.insert($<c_i, \text{cost}(G, \{c_i\})>$)
5. \hspace{0.5cm} while $C_{\text{grady}}$.isNotEmpty or $|C| < \Pi$ do
6. \hspace{1cm} $<c_i, \text{cost}(G, \{c_i\})>$ = $C_{\text{grady}}$.removeTop()
7. \hspace{1cm} if $\text{cost}(G, C \cup \{c_i\}) \leq \theta$ then
8. \hspace{1cm} \hspace{0.5cm} $C$.insert($c_i$)
9. \hspace{1cm} else
10. \hspace{1cm} \hspace{0.5cm} return $C$
11. return $C$
make a BiG-index incorrect, and BiG-index can be reconstructed periodically to maintain its efficiency; (ii) when a subtype-supertype relationship is removed, BiG-index determines the configurations that are affected by the removal and specializes the summary graphs so that the affected relationships are not involved in any configurations in the updated BiG-index. To minimize the index size, BiG-index can be recomputed occasionally.

4 Query processing with BiG-index

This section presents the major query processing steps of BiG-index (Fig. 5). In a nutshell, we exploit the ontology information to generalize the keyword queries (Sec. 4.1), process the queries on the summary graphs and generates the candidate and then final answers (Sec. 4.2). Further, we propose a few optimizations for improving efficiency (Sec. 4.3).

4.1 Query Generalization

In the first step, we generalize the query keywords to the m-th layer of BiG-index. The query answer at the m-th layer is determined; its node sets are specialized to the 1st layer; and the final answer graphs are generated using the data graph (the 0th layer). The query cost is affected by two major factors. First, query evaluation in the higher layer reduces the query time since the summary graphs are small and reduce redundant computation. However, generating the final answer graphs from a generalized layer requires time to specialize the summary graphs and to prune irrelevant graphs.

Notations for query processing. We introduce some notations for modeling the query cost. Given a graph G and a sequence of configurations C=[C1,C2,...,Cn], we use the symbol \( \chi(G,C^m) \) to denote the index at the m-th layer by applying \( \chi \) on G with \( C^m \), i.e., \( \chi = \chi(\chi(\chi(G,C^1),C^2),...,C^m) \), where \( C^m = [C^1,C^2,...,C^m] \) is a subsequence of C. Again, we omit \( \chi \) when \( C \) is not relevant to the discussion. We use \( \chi(u) \) to denote the summarized vertex of u in \( \chi(G) \). Given a set of query keywords, \( Q = \{q_1,q_2,...,q_n\} \), and a subsequence of configurations, \( C^m = [C^1,...,C^m] \), BiG-index generalizes Q to the m-th layer by applying \( \chi \) on Q with \( C^m \), denoted as Gen\(^m\)(Q,\( C^m \)) = Gen(...Gen(Gen(Q,C^1),C^2),...,C^m). Similarly, we use Gen\(^m\)(Q) when \( C^m \) is irrelevant.

Then, we formulate the cost model of query evaluation at the m-th layer: 

\[
\text{cost}_q(m, \chi(G,C^m)) = 
\beta \left( 1 - \frac{\chi(G,C^m)}{|G|} \right) + (1 - \beta) \frac{\sum_{i=1}^n \sup(q_i, G)}{\sum_{i=1}^n \sup(q_i, G)},
\]

where \( \sup(q_i, G) \) is the percentage of the number of occurrences of \( q_i \) in \( V(G) \), \( |\cdot| \) is the graph size and \( \beta \in [0,1] \) is a weight. The first term is the compression ratio of the summary graph at the m-th layer. The smaller the summary graph, the more efficient the query processing. The second term is the support of the query keywords in the summary graph. It increases as \( m \) increases and more computation is needed to specialize labels to those of the answer graphs at layer 0. We then present the query generalization problem in Def. 4.1.

6. Since Q is a set of query keyword nodes and does not have non-trivial paths, it is more intuitive to present the query generalization using Gen\(^m\)(Q).

Fig. 6: A generalized answer graph and vertex specialization

Definition 4.1: Optimal query generalization. Given a set of query keywords Q : \{q_1,q_2,...,q_n\}, and the configurations of C of a BiG-index, the optimal query layer m of Q for the BiG-index satisfies the following:

1) \( |\text{Gen}^m(Q,C^m)| = |Q| \); and
2) for all k, k is a layer of BiG-index and \( m \neq k \), cost\(_q\)(m) \leq cost\(_q\)(k) (Formula 4)

Condition 1 specifies that the generalized query cannot generalize multiple query keywords into one. Otherwise, this requires the modifications to existing keyword search algorithms for resolving query keywords. Condition 2 simply ensures optimal efficiency. Since the number of layers of BiG-index is often not large and Formula 4 can be efficiently computed, the optimal layer is obtained by exhaustive search.

4.2 Query Processing on Summary Graph Hierarchy

The major steps of the hierarchical query processing eval\(_{out}(G, Q, f)\) are summarized in Alg. 2. The user-defined keyword search algorithm is f (as discussed in Sec 2), and the BiG-index is denoted as (G, C).

**Step 1.** Initially, eval evaluates \( Q^m \) at the m-th layer. We obtain the *generalized answer set at the m-th layer*, denoted as \( A^m = \{a^m_1, a^m_2, ..., a^m_n\} \), e.g., the solid ovals in G\(_2\) of Fig. 5.

**Step 2.** We specialize each generalized answer \( a^m_k \) of \( A^m \) to the \( (m-1)\)-th layer to form \( A^{m-1} \), denoted as Spec\(_\{a^m_k\}\) (as discussed in Sec 2). For each vertex \( u^m_j \) in \( a^m_k \), BiG-index specializes \( u^m \) to the \( (m-1)\)-th layer by Spec\(_{u^m_j}\). Then, \( A^{m-1} = \text{Spec}(A^m) = \{\text{Spec}(a^m_k) | a^m_k \in A^m\} \) (Line 2 of Alg. 2).

**Step 3.** A vertex of \( a^m_k \) that is matched to a keyword \( q_k \) can be specialized to a set of generalized answer vertices at the \( (m-1)\)-th layer, e.g., the dotted ovals in Fig 5. A vertex is pruned if its label is not a supertype of \( q_k \), as stated in Prop. 4.1.

**Proposition 4.1:** [Candidate filtering] Consider a vertex \( u^m_j \) of a generalized answer \( a^m_k \) that matches a query keyword \( q_k \) at the \( m\)-th layer \( G^m \). Denote that the set of vertices \( U^m\) is obtained by specializing \( u^m_j \), \( v \in U^m\) is pruned if \( L^m(v) \neq \text{Gen}^m(q_k,C^m) \).

For concise presentation, we skip the verbose details of filter that implements the pruning. The set of vertices after filter forms a generalized answer \( a^{m-1}_k = (V^{m-1}, E^{m-1}) \) at the \( (m-1)\)-th
Algorithm 3: Answer graph generation (Step 5 of Algorithm 2: ans_graph_gen)

Input: A generalized answer graph \( a^m = (V^m_a, E^m_a) \), its specialized node set \( a = (V_a, E_a) \) where \( E_a = \emptyset \)

Output: A set of specialized answer graphs \( A^0 \)

1. initialize intermediate partial ans. graph \( G_{par} = \emptyset \)
2. initialize \( A^0 = \{G_{par}\} \)
3. build a specialization order \( O \) for \( a \) with \( a^m \)
4. for \( a_i \in O \) do
   // Enlarge the partial answers
5. \( A^0 = \text{enlarge}(a_i, a, A^0) \)
6. return \( A^0 \)

Function \( \text{enlarge}(a_i, a, A^0) \)

8. initialize \( A^0_{\text{next}} = \emptyset \)
// \( v \) is a specialized node of \( a_i \)
9. for \( v \in \text{Spec}(a_i) \) do
10. \( \text{for } G_{par} \in A^0 \) do
11. \( \text{if } v \text{ is qualified to enlarge } G_{par} \) by Def. 4.2 then
12. \( A^0_{\text{next}} = A^0_{\text{next}} \cup \{G_{par} \cdot \text{add}(v)\} \)
13. return \( A^0_{\text{next}} \)

layer. It is worth noting that \( E^m_{a^{-1}} = \emptyset \) when \( m \neq 1 \), to avoid costly intermediate answer graph generation. Therefore, \( a^{m-1} \) when \( m \neq 1 \) is described as a node set.

Step 4. If \( m \) does not equal 1, the algorithm repeats Step 2. Otherwise, Algo. 2 generates the answer graphs \( A^0 \) from \( A^1 \). Lemma 4.1 states that Steps 1–4 of Algo. 2 determine a set of candidate answers of the query.

Lemma 4.1. Given \( a \in \text{eval}(G, Q, f) \) and \( v \) is a vertex of a implies \( \exists a_i \) s.t. \( \chi^1(v) \) is a vertex of \( A^1 \) and \( A^1 \) is obtained from Steps 1–4 of \( \text{eval}_{\text{Out}} \).

Proof: (Sketch) The lemma is established by a proof of contradiction. Suppose there is an answer node \( v \) that is returned by \( \text{eval}(G, Q, f) \) but \( \chi^1(v) \) is not returned by Steps 1–4 of \( \text{eval}_{\text{Out}} \), (i) since \( v \) is an answer node, \( \chi^m(v) \) must be in \( \text{eval}(G^m, Q^m, f) \), by definition; (ii) the specializations of \( \chi^m(v) \) do not satisfy the candidate filtering condition and hence are not pruned. Hence, a contradiction is established.

Step 5. We generate the answer graphs \( A^0 \) from \( A^1 \) (the solid ovals in \( G_0 \) of Fig. 5). Note that Lemma 4.1 implies that the node set returned by Steps 1–4 is not necessarily an answer node set. We present a basic algorithm (Algo. 3) for generating answer graphs.

Algo. 3 constructs \( A^0 \) by enlarging a set of partial answer graphs \( G_{par} \). Initially, \( G_{par} \) is empty. Given a generalized answer graph \( a \), we fix an order to traverse its vertices \( V_a \) (Line 2, to be detailed in Sec. 4.3.2). We obtain the specialized vertices of \( a_i \) (Line 9) and check if they are qualified (see Def. 4.2 for details) to form larger (partial) answer graphs in (Lines 10–12).

Definition 4.2: (Vertex qualification) Given a generalized answer graph \( a^m = (V^m_a, E^m_a) \), a partial answer \( G_{par} \) (which is a subgraph of \( G^m = (V^m, E^m) \)), a vertex \( v \in V^0_a \), and a query keyword \( q \), \( v \) is qualified to enlarge \( G_{par} \) w.r.t. \( q \) iff
- \( (\chi(v) \in V^m_a, L(\chi(v)) = \text{Gen}(q), L(v) = q) \) and
- \( \forall u \in V_{par}, (\chi(u), \chi(v)) \in E^m_a \Rightarrow (u, v) \in E^0 \).

Example 4.1: Consider a query \( Q = \{\text{Eastern, Organization}\} \). The subgraph in the dashed region of Fig. 4 is one possible generalized answer \( a^m \). Suppose the specialization order is \( O_1 = \{\text{Univ., Eastern, Academics, Organization}\} \). \( \text{Spec}(\text{Univ.}) = \{\text{Harvard Univ., Cornell Univ., Columbia Univ.}\} \), where each vertex in \( \text{Spec}(\text{Univ.}) \) forms a partial answer \( G_{par} \). Without loss of generality, we take \( G_{par} = \{\text{Harvard Univ.}\} \) as an example. Suppose \( \text{Eastern} \) is specialized to \( \text{Spec}(\text{Eastern}) = \{\text{Massachusetts, New York}\} \). \( \text{Massachusetts} \) is qualified to enlarge \( G_{par} \) since \( \{\text{Univ., Eastern}\} \in E^m_a \) and \( \{\text{Harvard Univ., Massachusetts}\} \in E^0 \). However, \( \text{New York} \) is not qualified to enlarge \( G_{par} \) since there is no edge between \( \text{Massachusetts} \) and \( \text{Columbia Univ.} \) in \( E^0 \). Hence BiG-index enlarges \( G_{par} \) as \( \{\text{Harvard Univ., Massachusetts}\} \) as shown in Fig. 7.

Theorem 4.2. \( \text{eval}_{\text{Out}}(G, Q, f) = \text{eval}(G, Q, f) \).

Proof: (Sketch) By Lemma 4.1, Algo. 2 correctly computes \( A^0 \). The remaining argument is to establish that \( \text{eval}(G, Q, f) = \bigcup_{a \in A^0} \text{ans}_\text{graph_gen}(a^m, a) \). Suppose there is an answer graph \( a \) in \( \text{eval}(G, Q, f) \) but not returned by Algo. 2. This implies: (i) \( \exists v \in V_a \) s.t. \( \chi(v) \not\in A^1 \); or (ii) \( \exists (u, v) \in E_a \) s.t. \( \chi(u), \chi(v) \not\in E_A \). The former contradicts Lemma 4.1, whereas the latter contradicts the path-preserving property and Prop. 4.2. Hence, the answer graphs generated by Step 5 from \( A^1 \) are in \( \text{eval}(G, Q, f) \). This is established by analyzing all execution paths of Algo. 3.

4.3 Optimizations for Hierarchical Query Processing

The performance of BiG-index is significantly affected by the answer graph generation. Hence, Algo. 2 localizes the logic of answer graph generation at the last layer, in Step 5. The answer graphs at the higher layers \( \{A^{m-1}, \ldots, A^1\} \) are node sets. Next, we present a few optimizations to prune useless nodes from candidate node sets to optimize the generation.

4.3.1 Early specialization of keyword nodes

To distinguish the nodes containing the query keywords from the others, we refer the keyword nodes to be those whose labels are either the query keywords or their generalizations.

Some query keywords are selective and lead to small node sets. However, the generalized keywords are often not as selective. Hence, we propose to introduce an attribute isKey to the nodes of generalized answer graphs. The isKey of a node is true if it is matched to a generalized query keyword. isKey helps to prune candidate nodes in two ways.

First, given a generalized answer node \( a^m \) in \( a^m \) (a generalized answer in \( A^m \)) and its isKey is true, we specialize \( a^m \) and keep it only if one of the nodes in \( \text{Spec}(a^m) \) satisfies the keyword of the query. Second, for any keyword node \( a^m \) in the \( m \)-th layer, we specialize \( a^m \) in the \( m \)-th layer (Line 2 of Algo. 2). For any vertex \( a^{m-1} \) from the specialization of \( a^m \), isKey of \( a^{m-1} \) is true if \( L(a^{m-1}) \) is a supertype of the query keyword.

4.3.2 Specialization order

Since \( \text{ans}_\text{graph_gen} \) involves nodes of the data graph \( G^0 \), we propose the specialization order of answer nodes used in Line 2 of Algo. 3 to minimize the number of partial graphs being constructed. Suppose \( a = \{a_1, a_2, \ldots, a_n\} \) is the set of candidate answer nodes. We order the nodes on the basis of the numbers of nodes they are specialized to, as determined from the implementation of \( \chi^{-1} \). Specifically, the specialization order of \( a \) is a permutation of \( a \), denoted as \( O = \{a_1, a_2, \ldots, a_n\} \) s.t., for any \( a_i, a_j \in O, a_i, a_j \in a \) and \( |\chi^{-1}(a_i)| \leq |\chi^{-1}(a_j)| \) iff \( a_i \preceq a_j \).
Examples of ans_graph_gen execution on a^m

(i) Spec(U)

(ii) Spec(A)

(iii) Spec(A)

(iv) Spec(O)

1 intermediate ans.

Using Order O_2 = [A, U, E, O]

3 intermediate ans.

Using Order O_1 = [U, E, A, O]

Fig. 7: Answer graph generation

Example 4.2: We follow the query Q and the generalized answer a^m in Example 4.1. There are 4! = 24 different specialization orders. We show that specialization ordering will affect the number of intermediate partial answers with Figs. 6 and 7. If we specialize a subgraph using the order O_1 = [Univ., Eastern, Academics, Organization], there are three partial answers generated including {Harvard Univ., Massachusetts}, {Cornell Univ., New York}, {Columbia Univ., New York} for generating the answer subgraphs after the first two vertices are specialized. Consider another specialization order O_2 = [Academics, Univ., Eastern, Organization], there is only one intermediate partial answer generated.

4.3.3 Path-based answer generation

It is worth noting that ans_graph_gen specializes one vertex at a time. A generalized vertex may be checked many times for enlarging different partial answers. Therefore, we specialize one path at a time to avoid such duplicated computation. Moreover, the paths that contain keyword nodes, which are often more selective than other nodes and lead to small candidate answer sets. Enlarging the partial answers with them maintain small candidate sets.

We then elaborate some crucial details of the path-based answer generation algorithm p_ans_graph_gen (Algo. 4), which replaces ans_graph_gen of Step 5 of the (Algo 2).

Joint vertices. A joint vertex (of a summary graph) is a vertex of a degree larger than 2. The isJoint is an attribute of a node v such that v.isJoint is true iff v is a joint vertex. Intuitively, we can decompose a graph into a set of paths at joint vertices.

Step 1. Path decomposition of a generalized answer (Lines 2, 8–16). Given a generalized answer graph a^m, we decompose it into a canonical path set P at its joint vertices. We denote a path p \in P as (s_1, \ldots, t), where s (resp. t) is the source vertex (resp. target vertex) of p. s is a joint vertex and t is a joint vertex or a leaf vertex.

Step 2. Path answer generation (Lines 3–6). We specialize one path at a time using Algo. 3: Given a path p_i = (s_1, \ldots, t) \in P, we specialize p_i into a path set A_{p_i} such that \forall p_i' = (s_1', \ldots, t') \in A_{p_i}, s_1' \in \chi^{-1}(s_1), \ldots, t' \in \chi^{-1}(t).

Step 3. Construction of answer graph by path join (Lines 5, 17–26). We construct the answer graph by joining the paths at joint vertices, which has been recorded in Step 1. Specifically, given a partial answer graph G_{par} and a path p_j \in A_{p_j}, if p_j is qualified to G_{par} (Def. 4.3), we enlarge G_{par} with p_j.

Definition 4.3: (Path qualification) Given a set of generalized paths P, a partial answer G_{par} and a path p = (s_1, \ldots, t) \in G^0, p is qualified to enlarge G_{par} iff

- \forall p' = (s_1', \ldots, t'), p' is a path of G_{par}, \chi(s) = \chi(s'), and

\chi(s).isJoint is true \Rightarrow s = s'.

Example 4.3: Consider the generalized answer graph a^m in Example 4.1. Univ. is a joint vertex, since its degree is larger than 2. We decompose a^m into 3 paths p_1 = (Academics, Univ.), p_2 = (Univ., Eastern) and p_3 = (Univ., Organization). Consider p_1 and p_3. The main difference between the path qualification and vertex qualification is that the path qualification verifies on the joint vertex and is more selective. Moreover, Algo 4 often maintains small intermediate partial answers. After answer generation, p_1 is specialized into a path p_1' = (S. Idreos, Harvard Univ.), and p_3 is specialized into two paths, p_3' = (Harvard Univ., Ivy League) and p_3'' = (Cornell Univ., Ivy League). We assume the current partial answer is G_{par} = p_1' p_3', if p_3 is qualified to enlarge G_{par} since p_1' and p_3' have the same joint vertex Harvard Univ. However, p_3'' is not qualified since the specialized joint vertices are different (see Fig. 8).

4.3.4 Early termination after the first k answers

Finally, in applications where users retrieve the first k answers, BiG-index can be readily modified to return them and stops retrieving the rest. The idea is to specialize the candidate answer nodes only when there are not yet k answers. Specifically, assume that the generalized answer set is A^m = \{a_1^m, a_2^m, \ldots, a_n^m\} in the m-th layer. BiG-index indexes one a_i^m at a time, by Spec(a_i^m) from 1 to t to the data graph layer, and construct its answer graphs, instead of specializing the entire A^m set. The algorithm terminates when the number of answer graphs is k.

5 \textbf{Boost keyword search with the BiG-index framework}

In this section, we present how BiG-index speeds up some algorithms for popular keyword search semantics (e.g., Blinks and r-clique). In particular, we take backward traversal, ranked
We next show boost-bkws, which is the bkws version on top of BiG-index.

(1) **Plugin and evaluation (eval).** Given a set of query keywords, we generalize the query keywords to the optimal query layer, \( m \). BiG-index loads the \( m \)-th layer from the disk and takes bkws as \( f \) to compute \( \text{eval}(\chi^m(G), \text{Gen}^m(Q), \text{bkws}) \). The answer is a set of pairs of answer subtrees and their roots, denoted as \( A^m = \{(a^1_m, r_1), (a^2_m, r_2), \ldots, (a^n_m, r_n)\} \). The backward search of bkws requires the connectivity of the answer graph. By Prop. 5.1 (below), \( A^m \) contains all generalized answers.

To present Prop. 5.1, we overload slightly the notation \( \chi^m(u) \) to denote the specialized vertex of \( u \) in \( \chi^m(G) \), where \( u \in G \). Prop. 5.1 is derived from the path-preserving property of Bisim.

**Proposition 5.1:** Given a graph \( G \) and its \( m \)-th layer index \( G^m \), if reach\((u, v, G) \Rightarrow reach(\chi^m(u), \chi^m(v), \chi^m(G))\), where \( u \) and \( v \) are the vertices of \( G \), and reach\((u, v, G)\) denotes the reachability from \( u \) to \( v \) in \( G \).

(2) **Specialization and pruning (Spec).** For each subtree \( a^m = (V^m_a, E^m_a) \in A^m \), we denote the keyword node set \( K_m \subseteq V^m_a \) \((K^m_m = \{u_i\} \subseteq K_m \) where \( u_i \) has the label \( \text{Gen}_i^m(q_i) \)\) of bkws. BiG-index specializes \( K_m \) and \( V^m_a \) by layer. For each keyword node \( u^m_j \) \((u^m_j \in K^m_m) \), BiG-index specializes \( u^m_j \) into a set of vertices in the \((m - 1)\)-th layer by \( \text{Spec}(u^m_j) \), denoted as \( u^m_{i, j} \). The node \( u^m_{i, j} \) is pruned if \( L^m(v) \notin \text{Gen}^{m - 1}(q_i) \). Then, BiG-index specializes \( K_m \) by \( K^m = \bigcup_{q_i \in Q} u^m_{i, j} K^m_m = \bigcup_{q_i \in Q} u^m_{i, j} K^m_m \). BiG-index simply specializes \( V^m_a \setminus K^m_m \), but it does not prune these nodes by their labels, since they are kept just for the connectivity of each \( a^m \). This procedure terminates when \( m = 1 \).

(3) **Answer generation and verification.** BiG-index performs a depth-first traversal to generate the final answers, i.e., at layer 0. Each keyword node \( u \) in \( U^0_a \) is initialized as a candidate partial answer graph, denoted as \( \text{Gpar}_u \). To reduce the number of candidate answers, BiG-index specializes the set \( U^0_a \) among \( U^1_a, \ldots, U^\infty_a \) that has the fewest keyword nodes. According to this specialization order, BiG-index specializes each vertex \( a \in V^m_a \). If the vertex \( v \in \text{Spec}(u) \) is qualified to enlarge \( \text{Gpar}_u \), \( \text{Gpar}_u \) is enlarged with \( v \). BiG-index continues to enlarge \( \text{Gpar}_u \). Otherwise, if \( \text{Gpar}_u \) cannot be enlarged and is not yet an final answer, it is pruned. This procedure terminates when all the vertices in \( a^m \) are specialized or there are no candidate partial answers.

### 5.1 Backward Keyword Search

We start with the **backward search** in a graph, which ensures the answer graph is connected. In the data graph without any connectivity index, a common algorithm to answer the query is to explore the graph starting at the vertices that contain the query keywords. For example, Bhalotia et al. [1] presented the first backward keyword search algorithm. He et al. [12] have proposed a search strategy on the backward expansion.

**Backward keyword search (bkws).** We next summarize the major steps of the backward keyword search [1].

**Initialization.** Consider a query keyword set \( Q = \{q_1, q_2, \ldots, q_n\} \). We denote the set of vertices that contain the keyword \( q_i \) as \( V^i_q \). We denote the set of vertices that could reach one of the vertices in \( V^i_q \) as \( V^i_q \).

**Backward expansion.** In each search step, the vertex set \( V_i \) with the minimal size is processed as follows. The vertex \( v \in V_i \) that has the shortest distance to \( V^i_q \) is chosen for backward expansion. In the expansion, \( u \) is added to \( V_i \), where \((u, v)\) is an incoming edge of \( v \), and \( u \) is checked whether it can be an answer root. If yes, the answer whose root is \( u \) is recorded. Otherwise, the backward expansion continues.

**Answer discovery.** It discovers an answer root \( r \) such that \( r \) can reach a node from each \( V^i_q \), for all \( q_i \) in \( Q \).

5.2 **Distance-based Keyword Search**

Next, we present the shortest distance computation that is crucial in a representative keyword search \( r\)-clique. We recall that \( r\)-clique [15] determines the subgraph that all pairs of the vertices that contain the keywords are reachable to each other within \( r \) hops, where \( r \) is a user-specified parameter. [15] proposes an approximate algorithm to compute the top-\( k \) answers in PTIME. We use our notations to present the major steps of distance-based keyword search (dkws), as follows:

**Initialization.** Initially, the keywords \( q_i \)'s are matched to a set of keyword nodes, denoted as \( V^i_q \). The search space of the query answers is denoted as \( SP(V^i_q, \ldots, V^i_q) \). \( r\)-clique inserts a pair of the search space \( SP \) and an approximate best answer \( a = \{u_1, \ldots, u_n\} \) of \( SP \) into a priority queue \( S \) in ascending order according to the weight of \( a \), which is the total distance.
between keyword nodes. Given an $SP$, to find the best answer $a$, $r$-clique computes the shortest distances between $u_i \in V_i$, and $V_q$, where $i \neq j$ and $j \in [1, n]$. In particular, it computes $a'_i = \{u_{i1}, \ldots , u_{in}\}$ as a candidate best answer, where $u_{ij} = \arg \min_{u_i \in V_i} \text{dist}(u_i, u_j)$. The best answer is the best $a$ among all candidate answers obtained from the abovementioned method.

**Search space decomposition.** $r$-clique decomposes the search space recursively. In each iteration, the pair $(SP, a)$ in the front of $S$ is removed and $a = \{u_1, \ldots , u_n\}$ is added into the answer set. $r$-clique decomposes the search space $SP$ into $n$ subspaces as follows: $SP_i=(V_{q_i}, \ldots , V_q \setminus \{u_{i1}, \ldots , u_{in}\})$, $i \in [1, n]$. $r$-clique inserts the search subspaces $SP_i$ into $S$ together with their respective best answers.

**Termination.** The search procedure terminates when $S$ is empty or the top-$k$ answers are found.

Next, we present boost-dkws, the dkws version on top of BiG-index.

(1) **Plugin and evaluation (eval).** Kargar et al. [15] index the distance information in a neighbor list [15]. In BiG-index, we adopt the neighbor list and build it on the $m$-th layer. Then, we take dkws as $f$ to compute $\text{eval}(\chi^m(G), \text{Gen}^m(Q), \text{dkws})$.

When an answer pattern $a^m$ in a search space $SB$ is generated, we specialize $a^m$ (to be detailed next) and decompose the search space, which is consistent with [15]. The correctness of boost-dkws is established by applying Prop. 5.2.

**Proposition 5.2:** Given a graph $G$ and its $m$-th layer index $G^m$, $\text{dist}(\chi^m(u), \chi^m(v), G^m) \leq \text{dist}(u, v, G)$, where $u$ and $v \in V$, and $\text{dist}(u, v, G)$ denotes the shortest distance from $u$ to $v$ in $G$.

**Proof:** We denote the shortest path between $u$ and $v$ as $p = \langle u, v_0, \ldots , v_l \rangle$ in $G$. For any edge $e = \langle s, t \rangle \in p$, there is an edge $e' = \langle s', t' \rangle \in G^m$ where $s' = \chi^m(s)$ and $t' = \chi^m(t)$ based on the definition of BiG-index (Sec 3.1). Therefore, there is a path $p' = \langle \chi^m(u), \chi^m(v_0), \chi^m(v_1), \ldots , \chi^m(v) \rangle$ in $G^m$. The shortest path between $\chi^m(u)$ and $\chi^m(v)$, $\text{dist}(\chi^m(u), \chi^m(v), G^m) \leq |p'| = \text{dist}(u, v, G)$.

(2) **Specialization and pruning (Spec).** This step is identical to specialization and pruning discussed in Sec. 5.1.

(3) **Answer generation and verification.** This step is identical to answer generation and verification discussed in Sec. 5.1.

5.3 **Ranked Keyword Search**

Another common algorithm is keyword search ranking. We take Blinks [12] as an example to show how BiG-index speeds up the search procedure and preserves the ranking property. Blinks proposes a single-level index and bi-level index for the distinct root keyword search semantic query. As presented in [12], the single-level index is infeasible for large graphs, since the index requires $O(|V|^2)$ space. Therefore, we only demonstrate how to adopt the bi-level index. We present the major steps of ranked keyword search (rkws) as follows:

**Index construction.** Each keyword $\ell \in \Sigma$ is associated with a keyword-node list containing the vertices which can reach $\ell$, ordered by their shortest distances to $\ell$. Each pair of a vertex $v$ and a keyword $\ell$ is associated with a node-keyword map that stores the shortest distance from $v$ to the vertex containing $\ell$.

**Expanding backward and forward.** Given a keyword query $\{q_1, \ldots , q_n\}$, we initialize the set of vertices $V_q$, containing keyword $q_i$. We expand each keyword in a round-robin manner by traversing the vertex $v$ backward in the keyword-node list. Then, we look up $v$ from the node-keyword map and check if $v$ is an answer root. If the root is not yet found, we continue the expansion above. This procedure terminates when the top-$k$ answers are found.

**Ranking function.** A ranking function is defined for the top-$k$ answers. In this work, we offer an API to input the ranking function as $\text{rank}(a, Q, G, \text{scr})$, where $a = \langle V_a, E_a \rangle$ is the matching answer, $Q$ is the keyword query, $G$ is the data graph, and $\text{scr}$ is a score function. We illustrate $\text{scr}$ with the distance-based score function of [12]: $\text{scr}(a) = \sum_{u \in V_a} \text{dist}(u, v)$. Given two matching answers $a$ and $b$, $\text{scr}(a) \geq \text{scr}(b)$ iff $\text{rank}(a, Q, G, \text{scr}) \leq \text{rank}(b, Q, G, \text{scr})$.

**Proposition 5.3:** Given a graph $G$, its summary graph $G^m$ at the $m$-th layer, $a$ and $b$ as two elements of the answer set $A$, and a distance-based score function $\text{scr}(a) = \sum_{u \in V_a} \text{dist}(u, v)$, if $\text{rank}(a, Q, G^m, \text{scr}) \leq \text{rank}(b, Q, G^m, \text{scr})$, then $\text{rank}(a^0, Q, G^0, \text{scr}) \leq \text{rank}(b^0, Q, G^0, \text{scr})$, where $a^0 \in \chi^{-1}(a)$ and $b^0 \in \chi^{-1}(b)$.

**Proof:** (1) Given any two nodes $u$ and $v$ of an answer graph $a^0$ where $u, v \in V_a \cap \chi^m(u, \chi^m(v)) \leq \text{dist}(u, v, G)$ (by Prop. 5.2). (2) We next prove that $\text{dist}(\chi^m(u), \chi^m(v)) \geq \text{dist}(u, v, G)$. The proof is similar to Prop. 5.2. Denote the path between $\chi^m(u)$ and $\chi^m(v)$ as $p = \langle \chi^m(u), \ldots , \chi^m(v) \rangle$. After answer generation, the vertices in $a^0$ are qualified to form $a$. Given any edge $(x_i, x_{i+1})$ in $p$, there is an edge $(\chi^{-1}(x_i), \chi^{-1}(x_{i+1}))$ in $a^0$ (by Def 4.2). Thus, $\text{dist}(\chi^m(u), \chi^m(v)) \geq \text{dist}(u, v, G)$.

Putting (1) and (2) together, we have $\text{dist}(\chi^m(u), \chi^m(v)) = \text{dist}(u, v)$. $\text{scr}(a^0) = \sum \text{dist}(u, v) = \sum \text{dist}(\chi(u, v)) = \text{scr}(a)$.

Since $\text{rank}(a, Q, G^m, \text{scr}) \leq \text{rank}(b, Q, G^m, \text{scr})$, we have $\text{scr}(a) \leq \text{scr}(b)$. And then we have $\text{scr}(a^0) = \text{scr}(a) \leq \text{scr}(b^0)$. Therefore, we deduce that $\text{rank}(a^0, Q, G^0, \text{scr}) \leq \text{rank}(b^0, Q, G^0, \text{scr})$.

Finally, we present boost-rkws, the rkws version on top of BiG-index.

(1) **Plugin and evaluation (eval).** He et al. define a ranking function based on distance. The pattern answer which has the smaller distance gets a higher ranking. As an example, we take the rkws as $f$ to compute $\text{eval}(\chi^m(G), \text{Gen}^m(Q), \text{rkws})$. The correctness of the ranking function is due to a direct application of Prop. 5.3. We specialize the answer graph once it is found in the query layer.

(2) **Specialization and pruning (Spec).** For each answer graph $a^m = \langle V_a, E_a \rangle \in A^m$, the specialization and pruning techniques are the same as those in boost-dkws.

(3) **Answer generation and verification.** The answer generation and verification function are similar to those of boost-dkws. As in Prop. 5.3, given two generalized answer graphs $a$ and $b$ in the query layer, and the ranking function $\text{scr}$ is a distance-based function, if the ranking of $a$ is higher than that of $b$, then the ranking of $a^0$ is higher than that of $b^0$ where $a^0 \in \chi^{-1}(a)$ and $b^0 \in \chi^{-1}(b)$, respectively. The generalized answer graph with higher ranking returned in query layer can be specialized first.

6 **Experimental Evaluation**

This section presents a detailed experimental evaluation to investigate the efficiency of BiG-index and the effectiveness of the optimization techniques.
TABLE 2: Statistics of real-world and synthetic datasets

| Datasets | $|V|$ | $|E|$ | $|ont|$ | $|cnt|$ |
|-----------|------|------|--------|-------|
| YAGO3     | 2,635,317 | 5,260,573 | 5,699,253 | 17,497,481 |
| Dbpedia   | 5,795,123 | 15,752,299 | 5,699,253 | 17,497,481 |
| IMDB      | 167,127,069 | 6,049,262 | 5,699,253 | 17,497,481 |
| synt-1M   | 1,000,000 | 3,000,000 | 5,000  | 10,000  |
| synt-2M   | 2,000,000 | 6,000,000 | 5,000  | 10,000  |
| synt-4M   | 4,000,000 | 8,000,000 | 5,000  | 10,000  |
| synt-8M   | 8,000,000 | 16,000,000 | 5,000 | 10,000 |

6.1 Experimental Setup

6.1.1 Software and hardware

We implemented BiG-index, Blinks and r-clique in Java. Our experiments were run on a machine with a 2.93GHz CPU and 64GB memory running CentOS 7.4. The implementation was made memory-resident.

6.1.2 Datasets and default indexes

We used both real-life and synthetic datasets in our experiments. Tab. 2 summarizes some characteristics of these datasets and their ontology graphs. The sizes of layer 1 of the BiG-indexes are presented in Tab. 3. Those of other layers are smaller than layer 1's. The sizes of all the layers are reported in Fig. 9.

YAGO3. YAGO3 [17] is a large semantic knowledge base, derived from Wikipedia, WordNet and GeoNames. YAGO3 combines the cleaned taxonomy of WordNet with the Wikipedia category system, assigning entities to more than 350K classes. YAGO3 separates the ontology information as TAXONOMY and yagoTypes.

Dbpedia. Dbpedia (v3.9) is a knowledge graph with 5.8M vertices and 15.8M edges. It extracts structured content from the information created in various Wikimedia projects. Since there are 783 entities in the ontology graph and more than 80% of the entities in a data graph cannot match on the ontology graph, we used the ontology graph of YAGO3 for Dbpedia. We noted that 73.2% of the entities can be matched to some types in the ontology graph, whereas the rest can be simply matched to the topmost type.10

IMDB. IMDB is a movie dataset. We downloaded the dataset and the queries from their website.12 However, since IMDB does not accompany with an ontology graph, we simply applied YAGO3’s ontology graph.

Synthetic datasets. We included some synthetic datasets in the experiments. We generated the ontology graphs by setting an average degree of 5 and a height of 7. This is consistent with the heights and average degrees of the real ontology graphs.

Default indexes. By default, we constructed BiG-index by setting a large values of $\theta$ and $\Pi$ of Algo. 1 so that the labels of the graphs were generalized once when a layer was constructed. This setting helps us to reason the performances of BiG-index.

6.1.3 Queries

YAGO3 and Dbpedia. We generated synthetic keyword queries for the experiments. For each query, we selected 2-6 keywords from the ontology graph which had semantic relationships: for example, the query “The player who works in an England club” could be expressed by $Q_3 = \{\text{Club, Player, England}\}$ listed in Tab. 4. The count of each keyword in the data graph was more than 3000. Due to space limits, we report the results of 8 queries. The reported runtimes are the average of 10 runs.

IMDB. The queries of an existing benchmark [5] are often about topics e.g., “Relevant results identify relationships between Harrison Ford and George Lucas”. We generate keyword queries from the topics by an existing natural language processing tool 13, for example, $Q = \{\text{Harrison Ford, George Lucas}\}$. Also, some existing benchmark queries involve text matching and are not adopted because the textual search has not been the focus of this paper.

6.2 Experimental Results

Exp-1: Query performance. To evaluate the efficiency of BiG-index, we have tested the performance of Blinks and r-clique with and without BiG-index.

Blinks. We used the same setting as [12]. We adopted METIS for partitioning. The average blocks size was 1000. We set the pruning threshold $d_{max}$ (a.k.a $\tau_{\text{prune}}$ in [12]) to 5 to ensure keyword nodes were reachable from the root vertex within 5 hops. The results on YAGO3 and Dbpedia are reported in Figs. 10 and 11. The results show that BiG-index consistently reduced the query times by 61.8% (resp. 57.3% and 32.5%) on YAGO3 (resp. Dbpedia and IMDB) on average.

r-clique. For r-clique, we built the neighbors index, as in [15], with $R = 4$. The results are presented in Figs. 13 and 14. Again, BiG-index consistently reduced the query times by 39.4% (resp. 19.6%) on YAGO3 (resp. Dbpedia) on average. Our experiment

10. We remark that for the general graphs, similar to Dbpedia, we can first map the entities to those of an ontology graph of YAGO3. For the entities which are untyped, we can employ some existing techniques to type them, such as PEARL [20] and Patty [21];
11. http://www.imdb.com

Fig. 9: Summary graph sizes ($|V| + |E|$) at different layers

TABLE 4: Benchmarked queries

<table>
<thead>
<tr>
<th>ID</th>
<th>Queries</th>
<th>Counts in the data graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>(Actresses, Films)</td>
<td>(3364, 20119)</td>
</tr>
<tr>
<td>Q2</td>
<td>(Administrative, Player)</td>
<td>(111629, 20318)</td>
</tr>
<tr>
<td>Q3</td>
<td>(Club, Player, England)</td>
<td>(8336, 20318, 37269)</td>
</tr>
<tr>
<td>Q4</td>
<td>(Feature, Managers, Films)</td>
<td>(17635, 1571, 20119)</td>
</tr>
<tr>
<td>Q5</td>
<td>(Season, Mountain, Actors)</td>
<td>(19698, 5030, 13809)</td>
</tr>
<tr>
<td>Q6</td>
<td>(Computer, People, Food, Films)</td>
<td>(1687, 32638, 1241, 20119)</td>
</tr>
<tr>
<td>Q7</td>
<td>(Administrative, Player, Season, Mountain, Actors)</td>
<td>(111629, 20318, 19698, 5030, 13809)</td>
</tr>
<tr>
<td>Q8</td>
<td>(Season, Mountain, Actors, Feature, Managers, Films)</td>
<td>(19698, 5030, 13809, 17635, 1571, 20119)</td>
</tr>
</tbody>
</table>
shows that r-cliq cannot handle the IMDB dataset since it keeps an $O(mn)$ neighbor list where $n = |V|$ and $m$ is the average number of vertices in the neighborhood [15]. On IMDB, $m$ is closed to 105K. Our experiment estimated that the neighbor list could take 16TB.

**Query performance breakdown.** Figs. 10 to 14 show the major steps of query processing. They show that (a) BiG-index takes a large fraction of time to explore the summary graphs in the hierarchy, (b) the pruning time is negligible, and (c) the time for answer generation (at the data graph layer) contributes to a small part of query time. From the figures, we can observe that BiG-index reduced the query times by up to 61.8%.

**Exp-2: Performance on synthetic datasets.** We evaluated the performances of BiG-index over large synthetic graphs. We set $|Q| = 4$. We evaluated the queries on the datasets listed in Tab. 2. The compression ratio and runtime of BiG-index increase linearly with the graph sizes. Fig. 15 shows that BiG-index reduced the query times of existing keyword algorithms by at least 20%.

**Exp-3: Characteristics of BiG-index.** We further studied the size and construction time of the summary graphs in BiG-index. We computed 7 layers of summary graphs.

**Index sizes.** Tab. 2 and Fig. 9 report the sizes of the summary graphs in BiG-index. For real-life knowledge graphs, the first layer of summary graph of YAGO3 (resp. Dbpedia and IMDB) reduced to its 27.9% (resp. 60.5% and 36.7%). The synthetic datasets can be reduced at least 12%. Figs. 9 shows that in YAGO3, Dbpedia and the IMDB, the sizes of the summary graphs at the higher layers are smaller than those at the lower layers. The compression ratio due to summarization decreases as the layer number increases. The BiG-index size is simply the sum of the summary graphs in the index.

**Construction time.** We report the construction times here. BiG-index takes 20 minutes (resp. 6.4 hours and 6.6 hours) to construct the indexes (all layers) for YAGO3 (resp. Dbpedia and IMDB). For the largest synthetic graph, BiG-index constructed the index in 3 hours.

**Exp-4: Effectiveness of the cost model.** This experiment investigates the cost model for determining efficient index.

**Graph sampling.** To study the effectiveness of the configurations, we sampled some subgraphs in each layer. As depicted in the Fig. 16, we found that when the number of samples $n$ was larger than 400, the estimate of the average compression ratio was stable. Meanwhile, we fixed the sample subgraphs and generated 100 different configurations. We adopted Spearman Rank Correlation Coefficient to measure the generalization accuracy between the relative performance of configurations of the sample graphs and the whole graph. We obtained $r_s = 0.541$ which is larger than the critical value 0.326 with $\alpha = 0.001$. Hence, the estimate is an indication of the relative compression ratio of the original graph.

**Exp-5: Effectiveness of optimization.** We performed a set of experiments to investigate the effectiveness of the proposed optimization. First, we turned the specialization order optimization on and off and ran the query sets of YAGO3. The results are reported in Fig. 17. The specialization order optimization offers 14.8% performance improvement on average.

We also investigated the effectiveness of $p_{\text{ans\_graph\_gen}}$. We evaluated the query sets with and without the $p_{\text{ans\_graph\_gen}}$ on YAGO3. The results are reported in Fig. 18. The $p_{\text{ans\_graph\_gen}}$ offers 21.7% performance improvement on average.

We then evaluated the query sets at different layers. The results are presented in Fig. 19. The performance was the best when some queries (Q1,2,6–8) were evaluated at the highest layer; a possible reason is that it is more efficient to evaluate queries on small graph summaries than the data graph itself. The early pruning is useful for answer generation.

**Exp-6: Comparison with [10].** [10] summarizes graphs using Bism once. We first generalized the keywords to their types once, so that Bism could summarize the graphs. The queries of [10] are different from keyword search. Then, we adopted our query evaluation for [10]. Hence, the query performance of [10] is that of the second layer. Fig. 19 shows that evaluating queries at the second layer is always suboptimal.
Graph summarization. Graph summarizations can be roughly divided into three categories. (a) Some graph summarizations (e.g., [22]) aim to save the storage space. These general methods preserve the information of the entire graph. Our work differs from these studies in the following ways: we summarize the graph to enable efficient keyword search such that queries are evaluated on the summary graphs and final query answers are reconstructed. (b) Some other graph summarizations are designed for path expressions by vertices similarity (e.g., [2], [3], [4], [16], [19]). However, they only consider the structural relation. In contrast, BiG-index exploits both the semantic information from ontologies and the structural information of a data graph. (c) There are graph summarizations, e.g., [26], [28], designed for helping users to understand keyword queries. It is worth noting that these studies are driven by keywords. In contrast, our work aims to offer a platform to optimize keyword search performance.

Query-preserved query. Fan et al. [10] proposes a query-preserved framework for a reachability query and a simulation query. Queries are evaluated in the compressed graph without decompression. To the best of our knowledge, query-preserved compression for keyword search has not been studied. The most recent work is that of Ren et al. [24], which exploits vertex relationships for the subgraph isomorphism query. Their work transforms the original data graph into an adapted hypergraph. In contrast, we exploit the semantic relationships over the vertices on the knowledge graphs for indexing. We organize the summary graphs in a hierarchy to speed up query processing.

Queries by exploiting the ontology relations. Wu et al. [29] proposed an ontology-based filtering-and-verification framework for subgraph querying. Zheng et al. [33] used the ontology information to speed up the SPARQL similarity search over RDF knowledge graphs. Tran et al. [27] took advantage of the type to build an augmented summary graph to index the original graph. Corby et al. [6] proposed an approximate method based on ontologies for SPARQL query. Our work differs from these works in the following: (a) BiG-index is generic to speed up the existing keyword search semantics which is label- and path-preserving rather than a specific query. (b) BiG-index is built on the basis of both ontology information and graph structures.

8 Conclusions
In this paper, we have proposed BiG-index. The main idea is to exploit ontology information associated with a knowledge graph, to semantically index the graph for efficient keyword search. The major feature of BiG-index is that it is generic enough to optimize existing algorithms for keyword search. This paper proposes efficient algorithms to construct and query BiG-index. Our experiments have shown that BiG-index can reduce the runtimes of existing keyword search work Blinks on average by 50.5% and r-clique on average by 29.5%.

For future work, we plan to implement BiG-index with more complicated keyword search semantics, e.g., similarity search. We plan to implement other summarization formalisms for BiG-index.

Acknowledgements. This work is partly supported by HKRGC GRF 12201119, 12232716, 12201518, 12200817, and 12201018, and NSFC 61602395.

References


Jiaxin Jiang is a PhD student in the Department of Computer Science, Hong Kong Baptist University. He received his BEng degree in computer science and engineering from Shandong University in 2015. His research interests include graph-structured databases, distributed graph computation. He is a member of the Database Group at Hong Kong Baptist University (http://www.comp.hkbu.edu.hk/~db/).

Byron Choi is an Associate Professor in the Department of Computer Science at the Hong Kong Baptist University. He received the bachelor of engineering degree in computer engineering from the Hong Kong University of Science and Technology (HKUST) in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002 and 2006, respectively.

Sourav S Bhowmick is an Associate Professor in the School of Computer Science and Engineering, Nanyang Technological University. Sourav’s current research interests include data management, data analytics, computational social science, and computational systems biology. He has published more than 150 technical papers in these areas, most of which appeared in leading journals and conferences including SIGMOD, VLDB, ICDE, TODS, TKDE, and VLDBJ.
**APPENDIX A**

**SUPPLEMENTAL MATERIALS**

**A.1 Proofs**

**Theorem III.1.** [OptGen] Given a graph $G$, determining a configuration $C$ such that cost is minimum is NP-hard, where

$$\text{cost}(G, C) = \alpha \times \text{compress}(G, C) + (1 - \alpha) \times \text{distort}(G, C),$$

where $\alpha (0 \leq \alpha \leq 1)$.

**Proof.** The proof can be established by a reduction from maxSAT to a specific instance of the above problem.

Recall the definition of the maxSAT problem. Given a set of variables $X = \{x_1, x_2, \ldots, x_m\}$ and a set of disjunction clauses $C$: $\{c_1, c_2, \ldots, c_n\}$, e.g., $c_k = (x_i \lor x_j)$, the problem is to find a truth assignment to the variables such that it maximizes the number of clauses in the conjunction of $C$, i.e., $c_1 \land c_2 \land \ldots \land c_n$, true.

**Construction of an OptGen instance.** The major steps of the reduction are presented as follows.

- Given an instance of maxSAT, we construct a data graph $G$ and an ontology graph $G_{\text{Ont}}$. The overall $G$ and $G_{\text{Ont}}$ are presented in Fig. 20.
- We set $\alpha = 1$. Hence, the distortion factor (distort) is negligible to the analysis. The corresponding OptGen instance optimizes compress.
- The ontology graph encodes the truth assignment of variables. Each variable $x_i$ is a ground term (leaf variable node) of $G_{\text{Ont}}$. Each $x_i$ can be generalized to either $x_i \rightarrow \text{T}$ or $x_i \rightarrow \text{F}$ which corresponds to a truth assignment of $x_i$ to true (T) or false (F), respectively.
- The data graph encodes the clauses of maxSAT. We create an artificial root node $r$. For each clause $c_k$, we create a node for $v_k$ and an edge $(r, v_k)$. Further, we create $v_k', (v_k, v_k')$ and $v_k''$ is the only node connected to a large artificial subgraph $G_k$ (see Fig. 20 for illustration). Suppose the variable $x_i$ appears on $c_k$. We construct a node $v_i$ and a large artificial subgraph $G_k'$, and an edge $(v_i, v_i')$. Similarly, $v_i$ is the only node connected to $G_k'$, where $G_k'$ and $G_k$ have identical structure. $G_k$ and $G_k'$ are not compressed because $v_i$ and $v_i'$ do not have the same label and hence, are not bisimilar.
- An important property is that when both $v_i$ and $v_i'$ are generalized to $x_i \rightarrow \text{T}$, then $v_i$ and $v_i'$ have the same label and $G_k$ and $G_k'$ are compressed and clearly reduce the compression cost.
- We construct the data graph of other variables of the clause in a similar manner.
- Then, we augment $G_{\text{Ont}}$ for each clause $c_k$ as follows.
  - Add $(v_k', x_i \rightarrow \text{T})$ to $G_{\text{Ont}}$ if $x_i$ is in $c_k$; and
  - Add $(v_k', x_i \rightarrow \text{F})$ to $G_{\text{Ont}}$ if $x_i$ is in $c_k$.
- Another important property is that $v_k'$ can be generalized to a label of either one of the variable nodes. This encodes the disjunction of a truth assignment of variables.

**Construction of a truth assignment $A$ from the solution $C$ of OptGen.** Suppose OptGen returns configuration $C$, which compresses the artificial subgraphs the most. We construct a truth assignment $A$ of variables as follows: If $G_k$ is compressed and $(v_k', x_i \rightarrow \text{T}) \in C$, then we assign $\text{T}$ to $x_i$. If $(v_k', x_i \rightarrow \text{F}) \in C$, then we assign $\text{F}$ to $x_i$. We assign $\text{T}$ to other variables that do not have a truth value (if any). The construction of the truth assignment is obviously in PTIME.

Suppose $(v_k', x_i \rightarrow \text{T}) \in C$. It means that $x_i$ is in $c_k$. The assignment $x_i = \text{T}$ makes $c_k$ true. Similar argument applies to $(v_k', x_i \rightarrow \text{F})$. The number of $G_k$ graphs compressed is the number of clauses that $A$ makes true.

$A$ is a solution of maxSAT. Suppose there is a truth assignment $A'$ that makes more number of clauses of the maxSAT instance true than $A$ does. Then, we construct a configuration $C'$ as follows. For each $x_i = \text{T}$, we add a mapping rule $(x_i, x_i \rightarrow \text{T})$ to $C'$. Otherwise, we add $(x_i, x_i \rightarrow \text{F})$ to $C'$. For each $c_k$ where $A'$ makes true, there is either a variable $x_i$ or $\bar{x}_i$ that makes $c_k$ true. Hence, we add $(v_k', x_i \rightarrow \text{T})$ or $(v_k', x_i \rightarrow \text{F})$ to $C'$ and this will make $G_k$ compressed by Bisim. After this construction, the number of $G_k$ graphs that are compressed by $C'$ is larger than that by $C$. Hence, $C'$ is not a solution for OptGen. A contradiction is established. Hence, $A'$ does not exist and $A$ is the solution of the maxSAT instance.

Therefore, the specific instance of OptGen (where $\alpha \approx 1$) is NP-hard. Consequently, OptGen is NP-hard.

**A.2 BiG-index on General Graphs**

The main body of the paper presents the technical details of BiG-index with knowledge graphs as they are naturally associated with ontology graphs. In this appendix, we present how to support general graphs.

General graphs may not be accompanied with an ontology graph. We can exploit some publicly available comprehensive ontology graphs. The keywords of general graphs can very often be
found in such ontology graphs. Hence, the ontology information can then be used by the BiG-index framework.

In this appendix, we adopt YAGO3, which is a large semantic knowledge base, derived from Wikipedia, WordNet and GeoNames. In particular, YAGO3 combines the cleaned taxonomy of WordNet with the Wikipedia category system and assigns keywords to more than 350K types (a.k.a classes). YAGO3 stores the taxonomy information and types in yagoTaxonomy and yagoTypes tables, respectively. By combining these two tables, we can generate an ontology graph of YAGO3, denoted as $G_{Ont}^Y$. Next, we present the major steps of using $G_{Ont}^Y$ to type a general graph $G$ below.

**Step 1.** We look up each keyword $\ell \in \Sigma$ of $G$ from $G_{Ont}^Y$. If $\ell$ is found, $G_{Ont}^Y$ contains the ontology information of $\ell$ for the BiG-index framework.\(^{14}\)

**Step 2.** Otherwise, we employ some existing tools to type $\ell$, such as PEARL [20] and Patty [21]. Suppose $\ell'$ is the returned type of $\ell$. If $\ell'$ can be found from $G_{Ont}^Y$, $\ell'$ is used for the type of $\ell$.

**Step 3.** Otherwise, we type $\ell'$ using the topmost type of $G_{Ont}^Y$.

We remark that the above typing steps may not be very surprising. There have been innovative works/approaches that exploit ontologies or types for some other purposes. For instance, Wu et al. [29] proposed an ontology-based filtering-and-verification framework for subgraph querying. Zheng et al. [33] used the ontology information to speed up the SPARQL similarity search over RDF knowledge graphs. Tran et al. [27] took advantage of the type to build an augmented summary graph to index the original graph. Corby et al. [6] proposed an approximate method based on ontologies for SPARQL query. This paper differs from these works that BiG-index is a generic framework for efficient keyword search and BiG-index exploits both ontology information and graph structures.

\(^{14}\) Our experiments showed that in Step 1, we can type 73.2% of the keywords in Dbpedia by using $G_{Ont}^Y$. 