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 G_{Ont} to summarize and index a data graph G iteratively, to form a hierarchical index structure $\mathbb{G}$. 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 $\mathbb{Q}$ according to G_{Ont} in runtime. The transformed query is searched on the summary graphs in $\mathbb{G}$. 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 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,

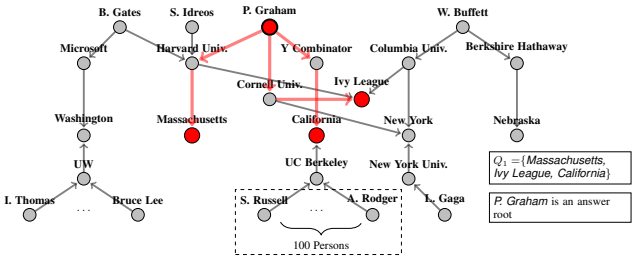


Fig. 1: An example data graph G, a keyword search Q_1 and its answer (bold subtree)

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 G_{Ont}, we index a data graph G into a hierarchy of summary graphs $\mathbb{G}$ as follows. Some node labels of a graph G are generalized with respect to G_{Ont} 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.

- *Jiaxin Jiang, Byron Choi and Jianliang Xu are with the Department of Computer Science, Hong Kong Baptist University, Hong Kong. E-mail: {jxjian, bchoi, xujl}@comp.hkbu.edu.hk*
- *Sourav.S. Bhowmick is with School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: assourav@ntu.edu.sg*

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.

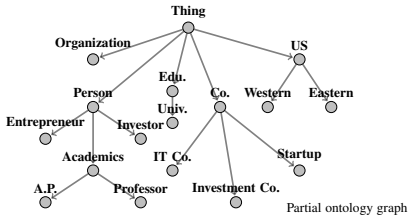


Fig. 2: Ontology graph G_{Ont} of Fig. 1

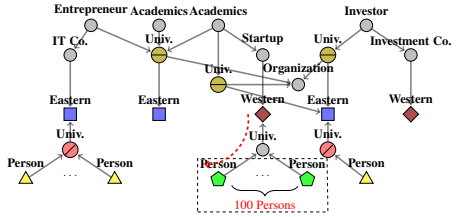


Fig. 3: Generalized graph G_{C_1} of G in Fig. 1

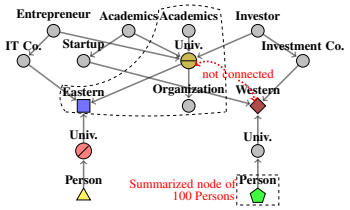


Fig. 4: Summary graph G' of Fig. 3

Example 1.1: We present a simplified knowledge graph G in Fig. 1 and its (partial) ontology graph G_{Ont} in Fig. 2.[3] We generalize some labels of G, according to G_{Ont}, to obtain a generalized graph G_{c_1}, as shown in Fig. 3. Note that there are more similar subgraphs after generalization (*e.g.*, the *Univ.* subgraphs). We then summarize G_{c_1} (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_{max}. Suppose the query keywords Q_1 are {*Massachusetts, Ivy League, California*} and the threshold $d_{max} = 3$. A possible answer is the (red bold) subtree whose root r 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 *Person* vertices, which have the same structure, as shown in G_{c_1}. Hence, they can be summarized, as shown in G'. As a result, we can verify whether or not the summarized *Person* can form an answer *once*, rather than doing this 100 *times* as in Fig. 4 and Fig. 3.

Consider another query Q_2 ={*California*, *P. Graham*} if the same d_{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, *California* (resp., *P. Graham*) is generalized as *Western* (resp., *Academics*). When searching forward from *P.Graham* in G, *Harvard Univ.* and *Cornel Univ.* are pruned one by one since they cannot reach *California* within d_{max} hops. In contrast, the subgraph rooted at *Univ.* does not connect to the keyword *Western* within d_{max} hops. Hence, it is pruned from G' and both *Harvard Univ.* and *Cornel 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 = {*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 *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 naïve 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.

3. Due to space limitation, we omit the specific keywords, such as person's and state names , but show the types in Fig. 2

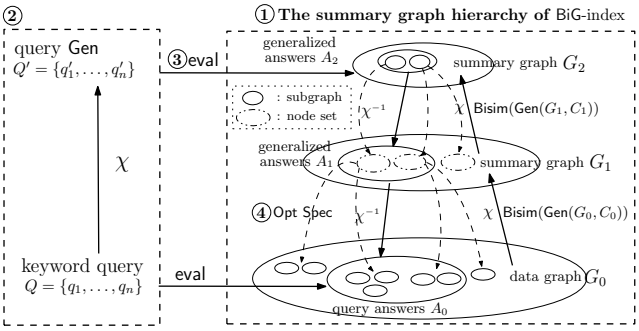


Fig. 5: Schematic of the evaluation eval_{Ont} of Q for the answer A_0 in BiG-index of the data graph G_0

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 ① 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 ② a cost-based generalization of query keywords. We then propose ③ a query evaluation algorithm on BiG-index. We propose ④ 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.

TABLE 1: Frequently used notations

Notations	Meaning
a / A	An answer graph / an answer graph set
G_{Ont}=(V_{Ont},E_{Ont})	An ontology graph, where $\ell \in V_{Ont}$ is a type and $(\ell', \ell) \in E_{Ont}$ denotes ℓ' is a supertype of ℓ.
Gen/Spec	Generalization/Specialization function
Bisim / Bisim^{-1}	The summarization function and its reverse. We adopt the bisimulation formalism
equiv(v) / $[v]_{\mathsf{equiv}}$	An equivalent relation of v
χ	The indexing function for generalizing and summarizing a graph G with a configuration C: $\chi(G, C) = \mathsf{Bisim}(\mathsf{Gen}(G, C))$
χ^{-1}	The function for recovering the graph G: $\chi^{-1}(G, C) = \mathsf{Spec}(\mathsf{Bisim}^{-1}(G), C))$

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 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) Σ 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 Σ. 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 $(u_i, u_j) \in B$, the following hold:

- $L(u_i) = L(u_j)$;
- for each $(u_i, v_i) \in E$, 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 [18], *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: $\mathsf{equiv}(v) = \{u \in V | (u, v) \in B\}$ or simply $[v]_{\mathsf{equiv}}$.

Bisimulation can also be intuitively considered as a graph $\mathsf{Bisim}(G)$ in which the nodes of an equivalence class of G are represented by a supernode. We use $\mathsf{Bisim}(v)$ to denote the supernode of v, where $v \in V$. A graph G has a $\mathsf{Bisim}(G)$ 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 $\mathsf{Bisim}(G)$ is used as an index, an efficient "reverse" method (denoted as Bisim^{-1}) is needed for answer generation. In this paper, Bisim^{-1} is implemented by hash tables.

Graph summarization $\mathsf{Bisim}(G)$**.**[4] Given a graph $G = (V, E, L, \Sigma)$. Based on the equivalent relation B of G, we define the summary graph $\mathsf{Bisim}(G) = (V', E', L', \Sigma')$ which can be regarded as yet another graph, where

1) $V' = \{[v]_{\mathsf{equiv}} \mid v \in V\}$;
2) $E' = \{([u]_{\mathsf{equiv}}, [v]_{\mathsf{equiv}}) \mid (u, v) \in E\}$;
3) $L'([v]_{\mathsf{equiv}}) = L(v)$; and $\Sigma' = \Sigma$.

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

Definition 2.1: A summarization method S of a graph G is *path-preserving* if a path $(u_1, u_2, \ldots, u_i)$ exists in G (dentoed as $S(G)$) implies $(S(u_1), S(u_2), \ldots, S(u_i))$ is a path in $S(G)$, where $S(u_j)$ are nodes of $S(G)$, $j \in$ [1,. . .,i].

It is straightforward to verify that bisimulation Bisim has the path-preserving property. This is a key reason why 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 $G_{Ont} = (V_{Ont}, E_{Ont})$, where (a) V_{Ont} is a set of vertices, where ℓ is a label (which models types, etc), $\ell \in V_{Ont}$; and (b) each edge $(\ell', \ell) \in E_{Ont}$ is labeled with $\mathsf{SubClassOf}$ or $\mathsf{SubTypeOf}$ s.t. $\ell', \ell \in V_{Ont}$. ℓ' is also referred to as a direct supertype of ℓ. 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_{Ont}, a generalization configuration (or simply, *configuration*) C is a set of mappings, and each mapping is denoted as $(\ell \rightarrow \ell')$, where $\ell, \ell' \in \Sigma$ and ℓ' is either (i) a supertype of ℓ s.t. $\ell, \ell' \in V_{Ont}$ and $(\ell', \ell) \in E_{Ont}$, or (ii) $\ell = \ell'$ when ℓ has no supertype.

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

Definition 2.2: A generalized graph $G_C = \mathsf{Gen}(G, C)$ is *label-preserving* under C only if for each $v \in V_{G_C}$ either (i) $(\ell \rightarrow \ell') \in C$, where ℓ and ℓ' are the labels of v in G and G_C, respectively, or (ii) the label of v in G and G_C are the same.

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

4. Since the summary graph is smaller than the original graph, we often use the terms *compression* and *summarization* interchangeably.

Rodger are generalized to *Person* (in the dotted rectangle of Fig. 3). *UC Berkeley* (resp. *California*) is generalized to *Univ.* (resp. *Western*). The 100 *Person*s 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 *Person*s can be summarized with one supernode (as shown in the dotted box in Fig. 4), due to bisimulation. It can be observed that G' 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_{max}) which contains a set of keywords $Q = \{q_1, \ldots, q_n\}$ and a distance bound d_{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) $\mathsf{dist}(r, p_i) \leq d_{max}$.

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

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

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

Intuitively, this paper proposes *graph generalization and summarization* for an index function χ (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 χ^{-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_{Ont}, a keyword query Q, and a keyword search algorithm f, this paper studies the index function χ and answer graph generation function χ^{-1} s.t.

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

where eval and eval_{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 fs. The technical discussions of fs only assume χ 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.[5]

5. For illustration, we present the keyword search algorithms that have high citation counts. According to Google Scholar, in Apr. 2019, the numbers of citations of [12] and [15] were 570 and 118, respectively.

3 Bisimulation of Generalized Graph Index BiG-index

In this section, we propose the *Bisimulation 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 bisimulation as a summarization function* since it is path-preserving (see Sec. 2). The index function χ is defined as follows.

Definition 3.1: (BiG-index) The BiG-index of a graph G and its ontology graph G_{Ont} are defined to be a binary tuple $(\mathbb{G}, \mathcal{C})$, where $\mathbb{G}$ 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

$$G^i = \begin{cases} G, & \text{if } i = 0, \\ \chi(G^{i-1}, C^i), & \text{otherwise,} \end{cases} \quad (2)$$

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 Gen and graph summarization Bisim. The index is denoted as $G^1 = \mathsf{Bisim}(\mathsf{Gen}(G^0, C^1))$.

(i) Graph generalization (Gen). As described in Sec. 2, a generalization $\mathsf{Gen}(G^0, C^1)$ simultaneously replaces the labels of the vertices with generalized ones, as specified in $C^1 = \{(\ell_1 \to \ell'_1), \ldots, (\ell_m \to \ell'_m)\}$. C^1 denotes that $\ell_i \in \Sigma$ is generalized to ℓ'_i, where ℓ'_i is one of the supertypes of ℓ_i in G_{Ont} for $1 \leq i \leq m$.

(ii) Graph summarization (Bisim). Bisimulation of a graph G can be regarded as yet another graph. An equivalent relation of vertices U, where $U \subseteq V_G$ (*e.g.*, $U = \{u_1, u_2, \ldots, u_n\}$), can be represented by a supernode s (*e.g.*, $s = [u_1]_{\mathsf{equiv}} = \ldots = [u_n]_{\mathsf{equiv}}$). Then, we denote the summary graph of a graph G as $\mathsf{Bisim}(G)$ by applying the maximal bisimulation 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 bisimulation 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 χ of a given graph G and a configuration C as follows.

$$\mathsf{cost}(G, C) = \alpha \times \mathsf{compress}(G, C) + (1 - \alpha) \times \mathsf{distort}(G, C), \quad (3)$$

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

(i) Compression ratio (compress). Given a data graph, a summarization function 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 (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 specialized summary graphs. Consider a configuration C=$\{\ell_1 \to \ell'_1, \ell_2 \to \ell'_2, \ldots, \ell_n \to \ell'_n\}$. Denote the domain of C as X=$\{\ell_i \mid (\ell_i \to \ell'_i) \in C\}$ and the image as Y=$\{\ell'_i \mid (\ell_i \to \ell'_i) \in C\}$. The *term distortion of* ℓ_i due to C, where $(\ell_i \to \ell'_i) \in C$, is defined as $\mathsf{distort}(\ell_i) = 1 - \frac{1}{|X_{\ell_i}|}$, where $|X_{\ell_i}|$ is the number of the labels that are also generalized to the supertype ℓ'_i (*i.e.*, X_{ℓ_i}=$|\{\ell \mid (\ell \to \ell'_i) \in C\}|$). It quantifies the cost to distinguish ℓ_i from the other labels that are generalized to the same supertype ℓ'_i. Then, the basic distortion of a configuration C is the normalized semantic distortion of labels, as follows:

$$\mathsf{distort}(G, C) = \big(\sum_{\ell_i \in X} \mathsf{distort}(\ell_i)\big)/|X|.$$

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, $\mathsf{sup}(\ell_i) = |V_{\ell_i}|/|V|$, where $V_{\ell_i} = \{v \mid v \in V$ and $L(v) = \ell_i\}$. The proposed $\mathsf{distort}(G, C)$ function is as follows:

$$\mathsf{distort}(G, C) = \big(\sum_{\ell_i \in X} \mathsf{distort}(\ell_i) \times \mathsf{sup}(\ell_i)\big) / \big(|X| \times \sum_{\ell_i \in X} \mathsf{sup}(\ell_i)\big).$$

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

Estimating the cost of χ 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 $O(2^{|\Sigma|})$. 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_v. The sample graph is the *node-induced subgraph* of V_v. 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.

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

Input: A data graph G, an ontology graph G_{Ont}, a cost threshold θ and max. number of generalizations Π
Output: A configuration C

```
1  initialize the configuration C = ∅
2  initialize a priority query C_grdy in the ascending order of the
     estimated cost
3  for c_i = (ℓ_i → ℓ'_i) ∈ E_ont and ℓ ∈ Σ do
4  |  C_grdy.insert(<c_i, cost(G, {c_i})>)
5  while C_grdy.isNotEmpty or |C| < Π do
6  |  <c_i, cost(G, {c_i})> = C_grdy.removeTop()
7  |  if cost(G, C ∪ {c_i}) ≤ θ then
8  |  |  C.insert(c_i)
9  |  else
10 |  |  return C
11 return C
```

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 $c_i = (\ell_i \to \ell'_i) \in E_{ont}$, w.r.t the data graph G and its ontology G_{Ont}, we estimate the cost of c_i as $\mathsf{cost}(G, \{c_i\})$ (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 θ, 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 Π. We then use C returned by Algo. 1 to generalize and summarize G to obtain G'. Users may increase the value of θ and apply Algo. 1 on G' and G_{Ont} again to obtain another layer of the index.

Maintenance of BiG-index**.** When the data graph is updated, the summary graph hierarchy $\mathbb{G}$ must be maintained. We adopt the efficient maintenance of the Bisim of $\mathbb{G}$ 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

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 $\mathcal{C}$=[C^1,C^2,...,C^h], we use the symbol $\chi^m(G,\mathcal{C}^m)$ to denote the index at the m-th layer by applying χ on G with $\mathcal{C}^m$, *i.e.*, χ^m=$\chi(\ldots\chi(\chi(G,C^1),C^2),\ldots,C^m)$, where $\mathcal{C}^m$=[C^1,C^2,...,C^m] is a subsequence of $\mathcal{C}$. Again, we omit $\mathcal{C}$ when $\mathcal{C}$ is not relevant to the discussion. We use $\chi^m(u)$ to denote the summarized vertex of u in $\chi^m(G)$. Given a set of query keywords, Q=$\{q_1,q_2,\ldots,q_n\}$, and a subsequence of configurations, $\mathcal{C}^m$=[C^1,...,C^m], BiG-index generalizes Q to the m-th layer by applying χ on Q with $\mathcal{C}^m$, denoted as $\mathsf{Gen}^m(Q,\mathcal{C}^m)$=$\mathsf{Gen}(\ldots\mathsf{Gen}(\mathsf{Gen}(Q,C^1),C^2),\ldots,C^m)$.[6] Similarly, we use $\mathsf{Gen}^m(Q)$ when $\mathcal{C}^m$ is irrelevant.

Then, we formulate the *cost model* of query evaluation at the m-th layer: $\mathsf{cost_q}(m,\chi(G,\mathcal{C}^m))$ =

$$\beta(1-\frac{|\chi^m(G,\mathcal{C}^m)|}{|G|})+(1-\beta)\frac{\sum_{i=1..n}\mathsf{sup}(\mathsf{Gen}^m(q_i,\mathcal{C}^m),G^m)}{\sum_{i=1..n}\mathsf{sup}(q_i,G)}, \quad (4)$$

where $\mathsf{sup}(q_i,G)$ 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 $\mathsf{Gen}^m(Q)$. This is slightly different from the $\chi(Q)$ symbol of the problem statement.

Algorithm 2: Hierarchical query processing (eval_{Ont})

Input: Query Q, BiG-index: $(\mathbb{G},\mathcal{C})$, $1<m\leq|\mathcal{C}|$
Output: Query answer A

1 $A^m=\mathsf{eval}(G^m,Q^m,f)$, where $Q^m=\mathsf{Gen}^m(Q,\mathcal{C}^m)$, the generalization of Q to the layer m using $\mathcal{C}^m$ (Def. 4.1)
2 $A^{m-1}=\mathsf{Spec}(A^m)$
3 $A^{m-1}=\mathsf{filter}(A^{m-1},\mathsf{Gen}^{m-1}(Q,\mathcal{C}))$
4 while A^1 is not obtained (*i.e.*, $m\neq 1$), do repeat Step 2 with m=$m-1$
5 **return** $\bigcup_{a\in A^1}\mathsf{ans\_graph\_gen}(a^m,a)$ where $a^m=\chi(a)$

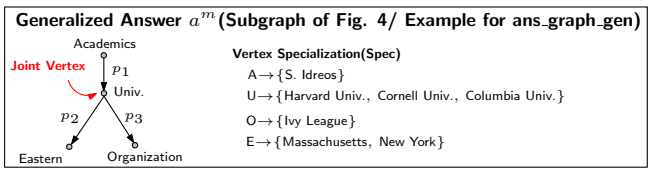


Fig. 6: A generalized answer graph and vertex specialization

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

1) $|\mathsf{Gen}^m(Q,\mathcal{C}^m)|=|Q|$; and
2) for all k, k is a layer of BiG-index and $m\neq k$, $\mathsf{cost_q}(m)\leq\mathsf{cost_q}(k)$ (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 $\mathsf{eval}_{Ont}(G,Q,f)$ are summarized in Algo. 2. The user-defined keyword search algorithm is f (as discussed in Sec 2), and the BiG-index is denoted as $(\mathbb{G},\mathcal{C})$.

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 $A^m=\{a_0^m,\ldots,a_n^m\}$, *e.g.*, the solid ovals in G_2 of Fig. 5.

Step 2. We specialize each generalized answer a_i^m of A^m to the $(m$-1)-th layer to form A^{m-1}, denoted as $\mathsf{Spec}(a_i^m)$ (as discussed in Sec. 2). For each vertex u_j^m in a_i^m, BiG-index specializes u_j^m to the $(m$-1)-th layer by $\mathsf{Spec}(u_j^m)$. Then, $A^{m-1}=\mathsf{Spec}(A^m)=\{\mathsf{Spec}(a_i^m)\mid a_i^m\in A^m\}$ (Line 2 of Algo. 2).

Step 3. A vertex of a_i^m 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_j^m of a generalized answer a_i^m that matches a query keyword q_k, at the m-th layer G^m. Denote that the set of vertices U_j^{m-1} is obtained by specializing u_j^m. $v\in U_j^{m-1}$ is pruned if $L^{m-1}(v)\neq\mathsf{Gen}^{m-1}(q_k,\mathcal{C}^{m-1})$. □

For concise presentation, we skip the verbose details of filter that implements the pruning. The set of vertices after filter forms a generalized answer $a_i^{m-1}=(V_a^{m-1},E_a^{m-1})$ at the $(m$-1)-th

Algorithm 3: Answer graph generation (Step 5 of Algorithm 2: ans_graph_gen)

Input: A generalized answer graph $a^m = (V_a^m, E_a^m)$, its specialized node set $a = (V_a^1, E_a^1)$ where $E_a^1 = \emptyset$
Output: A set of specialized answer graphs A^0

```
1  initialize intermediate partial ans. graph G_par = ∅
2  initialize A^0 = {G_par}
3  build a specialization order O for a with a^m
4  for a_i ∈ O do
       // Enlarge the partial answers
5      A^0 = enlarge(a_i, a, A^0)
6  return A^0
7  Function enlarge(a_i, a, A^0)
8      initialize A^0_next = ∅
       // v is a specailized node of a_i
9      for v ∈ Spec(a_i) do
10         for G_par ∈ A^0 do
11             if v is qualified to enlarge G_par by Def. 4.2 then
12                 A^0_next = A^0_next ∪ {G_par.add(v)}
13     return A^0_next
```

layer. It is worth noting that $E_a^{m-1} = \emptyset$ 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 $a \in \mathsf{eval}(G, Q, f)$ and v is a vertex of a implies $\exists a_i^1$ s.t. $\chi^1(v) \in a_i^1$, a_i^1 is a vertex of A^1 and A^1 is obtained from Steps 1–4 of eval_{Ont}.* □

Proof: (Sketch) The lemma is established by a proof of contradiction. Suppose there is an answer node v that is returned by $\mathsf{eval}(G, Q, f)$ but $\chi^1(v)$ is not returned by Steps 1-4 of eval_{Ont}, (i) since v is an answer node, $\chi^m(v)$ must be in $\mathsf{eval}(G^m, Q^m, f)$, 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 $a^m = (V_a^m, E_a^m)$, a partial answer G_{par} (which is a subgraph of $G^0 = (V^0, E^0)$), a vertex $v \in V^0$, and a query keyword q, v is *qualified to enlarge* G_{par} w.r.t q *iff*

- $\chi(v) \in V_a^m$, $L(\chi(v)) = \mathsf{Gen}(q)$, $L(v) = q$; and
- $\forall u \in V_{G_{par}}$, $(\chi(u), \chi(v)) \in E_a^m \Rightarrow (u, v) \in E^0$.

Example 4.1: Consider a query Q={*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=[*Univ.*, *Eastern*, *Academics*, *Organization*]. Spec(*Univ.*)={*Harvard Univ.*,*Cornel Univ.*,*Columbia Univ.*}, where each vertex in Spec(*Univ.*) forms a partial answer G_{par}. Without loss of generality, we take G_{par} = {*Harvard Univ.*} as an example. Suppose *Eastern* is specialized to Spec(*Eastern*)={*Massachusetts*, *New York*}.*Massachusetts* is qualified to enlarge G_{par} since (*Univ.*, *Eastern*) $\in E_a^m$ and (*Harvard Univ.*, *Massachusetts*) $\in E^0$. However, *New York* is not qualified to enlarge G_{par} since there is no edge between *Massachusetts* and *Columbia Univ.* in E^0. Hence BiG-index enlarges G_{par} as {*Harvard Univ.*, *Massachusetts*} as shown in Fig. 7

Theorem 4.2. $\mathsf{eval}_{Ont}(G, Q, f) = \mathsf{eval}(G, Q, f)$. □

Proof: (Sketch) By Lemma 4.1, Algo. 2 correctly computes A^0. The remaining argument is to establish that $\mathsf{eval}(G, Q, f) = \bigcup_{a \in A^1} \mathsf{ans\_graph\_gen}(a^m, a)$. ($\Rightarrow$) Suppose there is an answer graph a in $\mathsf{eval}(G, Q, f)$ but not returned by Algo. 2. This implies: i) $\exists v \in V_a$, $\chi(v) \notin A^1$; or ii) $\exists (u, v) \in E_a$, $(\chi(u), \chi(v)) \notin E_{A^m}$. The former contradicts Lemma 4.1, whereas the latter contradicts the path-preserving property and Prop. 4.2. ($\Leftarrow$) The answer graphs generated by Step 5 from A^1 are in $\mathsf{eval}(G, Q, f)$. 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 and 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 *iff* it is matched to a generalized query keyword. isKey helps to prune candidate nodes in two ways.

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

4.3.2 Specialization order

Since ans_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 $a = \{a_1, a_2, \ldots, a_s\}$ 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 χ^{-1}. Specifically, the specialization order of a is a permutation of a, denoted as $O = (a_1', a_2', \ldots, a_s')$, 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* $a_i' \preceq a_j'$.

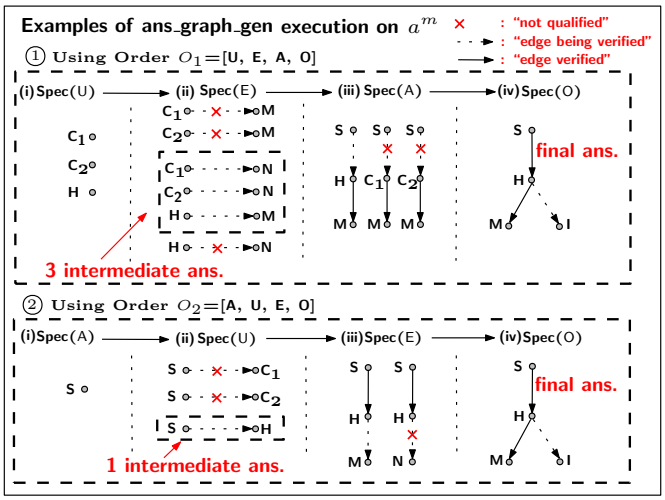
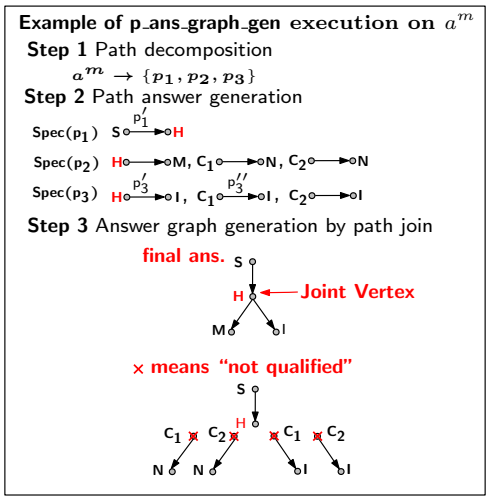


Fig. 7: Answer graph generation



Abbreviation of keywords in examples

Keyword	Abbr.
Academics	A
Univ.	U
Eastern	E
Organization	O
S. Idreos	S
Ivy League	I
Harvard Univ.	H
Cornell Univ.	C_1
Columbia Univ.	C_2
Massachusetts	M
New York	N

Fig. 8: Path-based answer 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 $(s, \ldots, t)$, 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 $p_i = (s_i, \ldots, t_i) \in P$, we specialize p_i into a path set A_{p_i}, such that $\forall p'_i = (s'_i, \ldots, t'_i) \in A_{p_i}$, $s'_i \in \chi^{-1}(s_i)$,... and $t'_i \in \chi^{-1}(t_i)$.

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 $p_j \in A_{p_i}$, 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 $p = (s, \ldots, t)$ in G^0, p is *qualified to enlarge* G_{par} *iff*

- $\forall p' = (s', \ldots, t')$, 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 $p_1 = (\textit{Academics}, \textit{Univ.})$, $p_2 = (\textit{Univ.}, \textit{Eastern})$ and $p_3 = (\textit{Univ.}, \textit{Organization})$. 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 $p'_1 = (\textit{S. Idreos}, \textit{Harvard Univ.})$, and p_3 is specialized into two paths, $p'_3 = (\textit{Harvard Univ.}, \textit{Ivy League})$ and $p''_3 = (\textit{Cornel Univ.}, \textit{Ivy League})$. We assume the current partial answer is $G_{par} = p'_1$. 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 $A^m = \{a^m_1, a^m_2, \ldots, a^m_s\}$ in the m-th layer. BiG-index specializes one a^m_i at a time, by Spec(a^m_i) from 1 to i 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 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

Algorithm 4: Path-based answer graph generation (Step 5 of Algorithm 2: p_ans_graph_gen)

Input: A generalized answer graph $a^m = (V_a^m, E_a^m)$, its specialized node set $A^1 = (V^1, E^1)$ where $V^1 = \emptyset$
Output: A set of specialized answer graphs A^0

initialize $V_{joint} = \emptyset$, $A^0 = \{G_{par}\}$
// decompose the generalized answer graph
$P = \mathsf{answer\_decomposition}(a^m)$
for $p_i \in P$ **do**
  // specialized the generalized path
  $A_{p_i} = \mathsf{ans\_graph\_gen}(p_i, A^1)$
  $A^0 = \mathsf{enlarge}(A_{p_i}, A^0)$ // enlarge the partial answers
return A^0
Function answer_decomposition(a^m)
  initialize $P = \emptyset$
  for $v \in V_a^m$ **do**
    if v.isJoint **then**
      V_{joint}.add(v)
  decompose a at the joint vertex set to generate P
  return P
Function enlarge(A_{p_i}, A^0)
  initialize $A^0_{next} = \emptyset$
  for $p \in A_{p_i}$ // p is a specialized path of p_i **do**
    for $G_{par} \in A^0$ **do**
      if p is qualified to enlarge G_{par} by Def. *4.3* **then**
        $A^0_{next} = A^0_{next} \cup \{G_{par}.\mathsf{add}(p)\}$
  return A^0_{next}

keyword search, and shortest path as examples in Secs. 5.1 to 5.3, respectively. Each algorithm is plugged into the evaluation framework (eval) of BiG-index, where specialization, pruning, and answer generation are adopted. Our framework can be also applied to optimize the algorithms that contain these operations with minor modifications, *e.g.*, [12], [15], [1], [14], [32].

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_{q_i}*. We denote *the set of vertices that could reach one of the vertices in V_{q_i} as V_i*.

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_{q_i} 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_{q_i}, for all q_i in Q.

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 $\mathsf{eval}(\chi^m(G), \mathsf{Gen}^m(Q), \mathsf{bkws})$. The answer are 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 summarized 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 $\mathsf{reach}(u, v\, G) \Rightarrow \mathsf{reach}(\chi^m(u), \chi^m(v), \chi^m(G))$, where u and v are the vertices of G, and $\mathsf{reach}(u, v, G)$ denotes the reachability from u to v in G.

(2) Specialization and pruning (Spec). For each subtree $a^m = (V_a^m, E_a^m) \in A^m$, we denote the keyword node set $K^m \subseteq V_a^m$ ($K_i^m = \{u_i\} \subseteq K^m$, where u_i has the label $\mathsf{Gen}^m(q_i)$) of bkws. BiG-index specializes K^m and $V_a^m \backslash K^m$ layer by layer. For each keyword node u_j^m ($u_j^m \in K_i^m$), BiG-index specializes u_j^m into a set of vertices in the $(m-1)$-th layer by $\mathsf{Spec}(u_j^m)$, denoted as $U_{i,j}^{m-1}$. The node $v \in U_{i,j}^{m-1}$ is pruned if $L^{m-1}(v) \neq \mathsf{Gen}^{m-1}(q_i)$. Then, BiG-index specializes K^m by $K_i^{m-1} = \bigcup_{u_j^m \in K_i^m} U_{i,j}^{m-1}$ and $K^{m-1} = \bigcup_{q_i \in Q} K_i^{m-1}$. Then, BiG-index simply specializes $V_a^m \backslash K^m$, 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_j^0 is initialized as *a candidate partial answer graph*, denoted as G_{par}. To reduce the number of candidate answers, BiG-index specializes the set U_j^0, among $U_1^0, \ldots, U_n^0$, that has the fewest keyword nodes. According to this specialization order, BiG-index specializes each vertex $a_i \in V_{a^m}$. If the vertex $v \in \mathsf{Spec}(a_i)$ is qualified to enlarge G_{par}, G_{par} is enlarged with v. BiG-index continues to enlarge G_{par}s. Otherwise, if a G_{par} 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.

5.2 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_is are matched to a set of keyword nodes, denoted as V_{q_i}s. The *search space* of the query answers is denoted as SP=$(V_{q_1}, \ldots, V_{q_n})$. 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 $u_i \in V_{q_i}$ and V_{q_j}s, where $i \neq j$ and $j \in [1, n]$. In particular, it computes $a'=\{u'_1, \ldots u'_i, \ldots, u'_n\}$ as a candidate best answer, where $u'_j = \arg \min_{\forall u_j \in V_{q_j}} \mathsf{dist}(u_i, u_j)$. 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: $SP_i=(V_{q_1}, \ldots, V_{q_i} \backslash \{u_i\}, \ldots, V_{q_n})$, $i \in [1, n]$. 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 $\mathsf{eval}(\chi^m(G), \mathsf{Gen}^m(Q), \mathsf{dkws})$. 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, $\mathsf{dist}(\chi^m(u), \chi^m(v), G^m) \leq \mathsf{dist}(u, v, G)$, where u and $v \in V$, and $\mathsf{dist}(u, v, G)$ denotes the shortest distance from u to v in G.

Proof: We denote the shortest path between u and v as $p = (u, v_0, v_1, \ldots, v)$ in G. For any edge $e = (s, t) \in p$, there is an edge $e' = (s', t')$ in G^i 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 $p' = (\chi^m(u), \chi^m(v_0), \chi^m(v_1) \ldots, \chi^m(v))$ in G^m. The shortest path between $\chi^m(u)$ and $\chi^m(v)$, $\mathsf{dist}(\chi^m(u), \chi^m(v), G^m) \leq |p'| = \mathsf{dist}(u, v, G)$. □

(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 ℓ, ordered by their shortest distances to ℓ. Each pair of a vertex v and a keyword ℓ is associated with a *node-keyword map* that stores the shortest distance from v to the vertex containing ℓ.

Expanding backward and forward. Given a keyword query $\{q_1, \ldots, q_n\}$, we initialize the set of vertices V_{q_i} 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 $\mathsf{rank}(a, Q, G, \mathsf{scr})$, where $a = (V_a, E_a)$ is the matching answer, Q is the keyword query, G is the data graph, and scr is a score function. We illustrate scr with the distance-based score function of [12]: $\mathsf{scr}(a) = \sum_{u,v \in V_a} \mathsf{dist}(u, v)$. Given two matching answers a and b, $\mathsf{scr}(a) \leq \mathsf{scr}(b)$ *iff* $\mathsf{rank}(a,Q,G,\mathsf{scr}) \leq \mathsf{rank}(b,Q,G,\mathsf{scr})$.

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-bsed score function $\mathsf{scr}(a) = \sum_{u,v \in V_a} \mathsf{dist}(u, v)$, if $\mathsf{rank}(a, Q, G^m, \mathsf{scr}) \leq \mathsf{rank}(b, Q, G^m, \mathsf{scr})$, then $\mathsf{rank}(a^0, Q, G^0, \mathsf{scr}) \leq \mathsf{rank}(b^0, Q, G^0, \mathsf{scr})$, 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 $u, v \in V_{a^0}$ $\mathsf{dist}(\chi^m(u), \chi^m(v)) \leq \mathsf{dist}(u, v)$ (by Prop. 5.2). (2) We next prove that $\mathsf{dist}(\chi^m(u), \chi^m(v)) \geq \mathsf{dist}(u, v)$. The proof is similar to Prop 5.2. Denote the path between $\chi^m(u)$ and $\chi^m(v)$ as $p = (\chi^m(u), \ldots, \chi^m(v))$. 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, $\mathsf{dist}(\chi^m(u), \chi^m(v)) \geq \mathsf{dist}(u,v)$.

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

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

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 $\mathsf{eval}(\chi^m(G), \mathsf{Gen}^m(Q), \mathsf{rkws})$. 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 $a^m = (V_a, E_a) \in A^m$, the specialization and pruning techniques are the same as those in boost-bkws.

(3) Answer generation and verification. The answer generation and verification function are similar to those of boost-bkws. As in Prop. 5.3, given two generalized answer graphs a and b in the query layer, and the ranking function 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	$\|V\|$	$\|E\|$	$\|V_{ont}\|$	$\|E_{ont}\|$
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	1,673,076	6,074,782	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

TABLE 3: Index size of the 1 layer of BiG-index

Datasets	Layer 1 size ($\|V\|$+$\|E\|$)	Size ratio
YAGO3	0.39M + 1.80M	0.2785
Dbpedia	1.74M + 11.3M	0.6052
IMDB	0.46M + 2.38M	0.3666
synt-1M	0.53M + 2.98M	0.8775
synt-2M	1.02M + 5.93M	0.8687
synt-4M	1.61M + 7.67M	0.7730
synt-8M	3.18M + 15.01M	0.7579

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 the 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.[7] 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,[8] assigning entities to more than 350K classes. YAGO3 separates the ontology information as *TAXONOMY*. We generated an ontology graph by combining the tables *yagoTaxonomy* and *yagoTypes*.

Dbpedia.[9] 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 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.[11] We conducted our experiment on a benchmark of keyword search [5] on IMDB graph. IMDB is a movie dataset. We downloaded the dataset and the queries from the 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 θ and Π 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.

7. http://www.mpi-inf.mpg.de/yago
8. https://en.wikipedia.org/wiki/Category:Systems
9. http://dbpedia.org
10. We remark that for the general graphs, similar to Dbpeida, 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
12. https://joel-coffman.github.io/resources.html

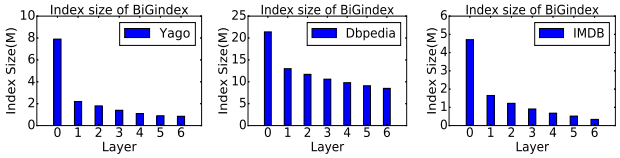


Fig. 9: Summary graph sizes ($|V| + |E|$) at different layers

TABLE 4: Benchmarked queries

ID	Queries	Counts in the data graph
Q1	(Actresses, Films)	(3364, 20119)
Q2	(Administrative, Player)	(111629, 20318)
Q3	(Club, Player, England)	(8336, 20318, 37269)
Q4	(Feature, Managers, Films)	(17635, 1571, 20119)
Q5	(Season, Mountain, Actors)	(19689, 5030, 13809)
Q6	(Computer, People, Food, Films)	(1687, 32638, 1241, 20119)
Q7	(Administrative, Player, Season, Mountain, Actors)	(111629, 20318, 19689, 5030, 13809)
Q8	(Season, Mountain, Actors, Feature, Managers, Films)	(19689, 5030, 13809, 17635, 1571, 20119)

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={*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 = {*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 τ_{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

13. http://nlp.stanford.edu:8080/corenlp/process

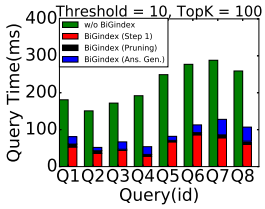


Fig. 10: Query times of Blinks on YAGO3

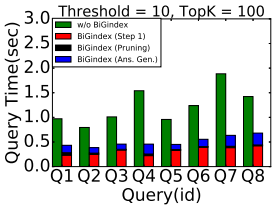


Fig. 11: Query times of Blinks on Dbpedia

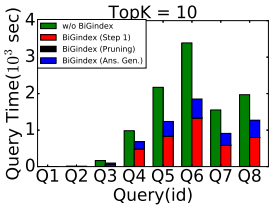


Fig. 13: Query times of r-clique on YAGO3

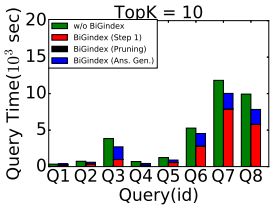


Fig. 14: Query times of r-clique on Dbpedia

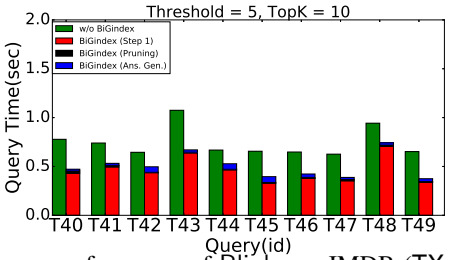
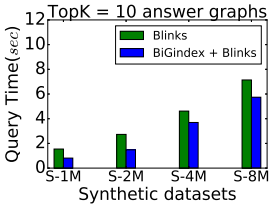


Fig. 12: Query performance of Blinks on IMDB (TX is the X-th topic in [5])



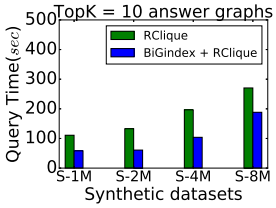


Fig. 15: Query times of Blinks (RHS) and r-clique (LFS)

shows that r-clique can not 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.

Optimal query layer m. We present the performance of the queries on each layer in Fig. 19. We varied the β of the query generalization model from 0.1 to 0.9. We observed that β can be set to a range of 0.3 to 0.7 for efficient BiG-index. We set β to 0.5. The model accurately predicts the optimal query layers of 6 queries, except for Q3 and Q6 (*i.e.*, 75% accuracy).

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_ans_graph_gen. We evaluated the query sets with and without the p_ans_graph_gen on YAGO3. The results are reported in Fig. 18. The p_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 Bisim once. We first generalized the keywords to their types once, so that Bisim 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.

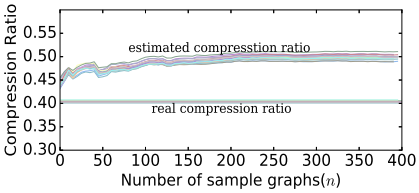


Fig. 16: The estimated compress vs sample sizes

Fig. 17: Specialization order optimization

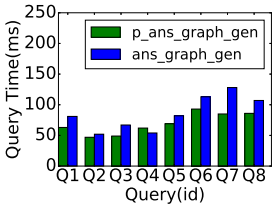


Fig. 18: Path-based answer generation optimization

7 Related Work

Keyword search semantics. Recently, keyword search has attracted a lot of interests from both industry and research communities (*e.g.*, [12], [15], [1], [14]). Bhalotia et al. [1] proposed for keyword search on relational databases. He et al. [12] proposed an index and search strategies to reduce the keyword search time. Kargar et al. [15] proposed distance restrictions on the keyword nodes, (*i.e.*, the shortest distance between each pair of keywords nodes is smaller than r). Shi et al. [25] proposed a top-k relevant semantic place retrieval (kSP) which finds the tightest semantics places ranked by the distance to the query location and the semantics looseness to the query keywords. In contrast, we propose a generic framework for keyword search. These keyword search semantics could be built on top of our framework without rewriting the entire algorithms. Due to space restriction, we omit the implementations of some of these semantics in the paper.

Ye et al. [32] proposed a search strategy based on a compressed signature to avoid the flooding search strategy which may lead to massive communication cost for the keyword search on distributed graphs. Given a set of query keywords, Yang et al. [30] proposed path-based indexes to find the tree pattern over knowledge graph with an approximate algorithm based on a sampling approach. These studies optimize a specific keyword search semantic. In contrast, our work optimizes the efficiency of different existing keyword search semantics. Since our indexes are yet another set of smaller graphs, their algorithms can be also implemented on top of our framework to reduce the searching cost.

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.

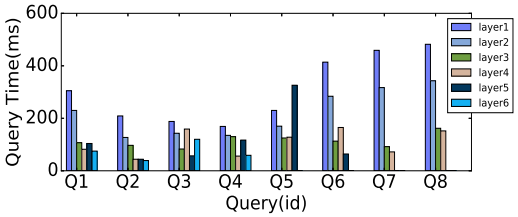


Fig. 19: Query performance by varying m (layer)

Query-preserving query. Fan et al. [10] proposes a query-preserving 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-preserving 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 keyword query 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

[1] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings of the 18th International Conference on Data Engineering*, pages 431–440, 2002.

[2] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. D. Viglas. Vectorizing and querying large xml repositories. In *Proceedings of the 21st International Conference on Data Engineering*, pages 261–272. IEEE, 2005.

[3] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed xml. In *Proceedings of 29th International Conference on Very Large Data Bases*, volume 29, pages 141–152, 2003.

[4] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 134–144, 2003.

[5] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 729–738. ACM, 2010.

[6] O. Corby, R. Dieng-Kuntz, F. Gandon, and C. Faron-Zucker. Searching the semantic web: Approximate query processing based on ontologies. *IEEE Intelligent Systems*, 21(1):20–27, 2006.

[7] J. Deng, B. Choi, J. Xu, H. Hu, and S. S. Bhowmick. Incremental maintenance of the minimum bisimulation of cyclic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 25(11):2536–2550, 2013.

[8] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 836–845, 2007.

[9] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 155–169, 2017.

[10] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2012.

[11] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective community search for large attributed graphs. *Proceedings of the Very Large Data Bases Endowment*, 9(12):1233–1244, 2016.

[12] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 305–316, 2007.

[13] J. Jiang, B. Choi, J. Xu, and S. S. Bhowmick. A generic ontology framework for indexing keyword search on massive graphs. *https://www.comp.hkbu.edu.hk/∼jxjian/tr2018.pdf*, 2018.

[14] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[15] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *Proceedings of the Very Large Data Bases Endowment*, 4(10):681–692, 2011.

[16] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 129–140, 2002.

[17] F. Mahdisoltani, J. Biega, and F. Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *Seventh Biennial Conference on Innovative Data Systems Research*, 2014.

[18] R. Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[19] T. Milo and D. Suciu. Index structures for path expressions. In *Database Theory - ICDT '99, 7th International Conference*, pages 277–295, 1999.

[20] N. Nakashole, T. Tylenda, and G. Weikum. Fine-grained semantic typing of emerging entities. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1488–1497, 2013.

[21] N. Nakashole, G. Weikum, and F. Suchanek. Patty: a taxonomy of relational patterns with semantic types. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1135–1145. Association for Computational Linguistics, 2012.

[22] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–432, 2008.

[23] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 827–838, 2014.

[24] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the Very Large Data Bases Endowment*, 8(5):617–628, 2015.

[25] J. Shi, D. Wu, and N. Mamoulis. Top-k relevant semantic place retrieval on spatial RDF data. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1977–1990, 2016.

[26] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 567–580, 2008.

[27] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, pages 405–416, 2009.

[28] Y. Wu, S. Yang, M. Srivatsa, A. Iyengar, and X. Yan. Summarizing answer graphs induced by keyword queries. *Proceedings of the Very Large Data Bases Endowment*, 6(14):1774–1785, 2013.

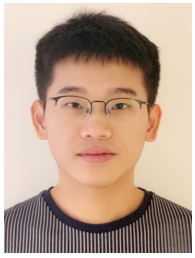
[29] Y. Wu, S. Yang, and X. Yan. Ontology-based subgraph querying. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 697–708. IEEE, 2013.

[30] M. Yang, B. Ding, S. Chaudhuri, and K. Chakrabarti. Finding patterns in a knowledge base using keywords to compose table answers. *Proceedings of the Very Large Data Bases Endowment*, 7(14):1809–1820, 2014.

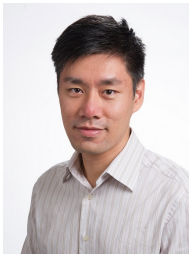
[31] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu. Autog: A visual query autocompletion framework for graph databases. *Proceedings of the Very Large Data Bases Endowment*, 9(13):1505–1508, 2016.

[32] Y. Yuan, X. Lian, L. Chen, J. X. Yu, G. Wang, and Y. Sun. Keyword search over distributed graphs with compressed signature. *IEEE Trans. Knowl. Data Eng.*, 29(6):1212–1225, 2017.

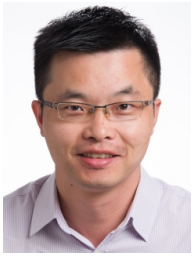
[33] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao. Semantic sparql similarity search over rdf knowledge graphs. *Proceedings of the Very Large Data Bases Endowment*, 9(11):840–851, 2016.



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.



Jianliang Xu is a Professor in the Department of Computer Science, Hong Kong Baptist University (HKBU). He held visiting positions at Pennsylvania State University and Fudan University. 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.

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 many papers in major venues in these areas such as SIGMOD, VLDB, ICDE, SIGKDD, MM, TKDE, VLDB Journal, and Bioinformatics.

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*

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

where α ($0 \leq \alpha \leq 1$). □

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.*, $c_k = (x_i \vee \bar{x_j})$, 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.*, $c_1 \wedge c_2 \wedge \ldots \wedge c_n$, 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_{Ont}. The overall G and G_{Ont} are presented in Fig. 20.
- We set $\alpha \approx 1$. 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_{Ont}. Each x_i can be generalized to either $x_{i\to\mathsf{T}}$ or $x_{i\to\mathsf{F}}$ 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_k, 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'_k do not have the same label and hence, are not bisimilar.
- An important property is that when both v_i and v'_k are generalized to $x_{i\to\mathsf{T}}$, then v_i and v'_k 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_{Ont} for each clause c_k as follows.
  - Add $(v'_k, x_{i\to\mathsf{T}})$ to G_{Ont} if x_i is in c_k; and
  - Add $(v'_k, x_{j\to\mathsf{F}})$ to G_{Ont} if $\bar{x_j}$ is in c_k.
- Another important property is that v'_k can be generalized to a label of either one of 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 $(v'_k \to x_{i\to\mathsf{T}}) \in C$, then we assign T to x_i. If $(v'_k \to x_{j\to\mathsf{F}}) \in C$, then we assign F to x_j. We assign T to other variables that do not have a truth value (if any). The construction of the truth assignment is obviously in PTIME.

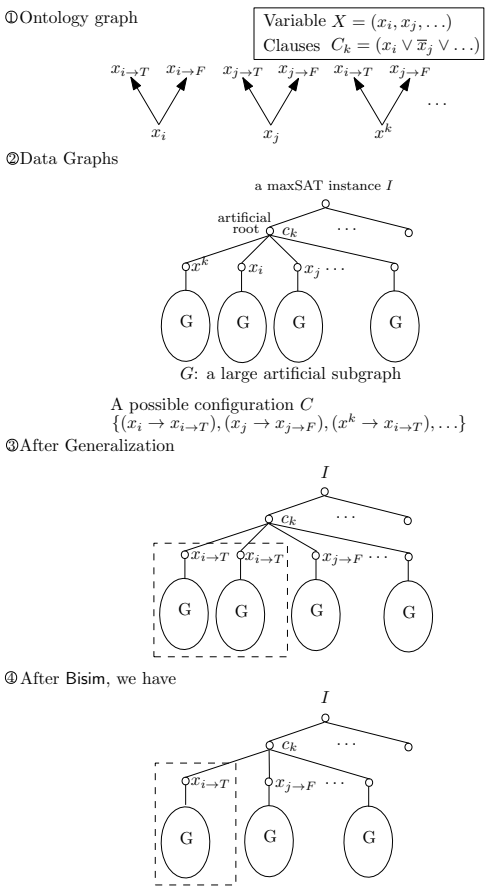


Fig. 20: An illustration of the OptGen instance constructed from a maxSAT instance

Suppose $(v'_k \to x_{i\to\mathsf{T}}) \in C$. It means that x_i is in c_k. The assignment x_i=T makes c_k true. Similar argument applies to $(v'_k \to x_{j\to\mathsf{F}})$. 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 = T, we add a mapping rule $(x_i, x_{i\to\mathsf{T}})$ to C'. Otherwise, we add $(x_i, x_{i\to\mathsf{F}})$ to C'. For each c_k where A' makes true, there is either a variable x_i or $\bar{x_j}$ that makes c_k true. Hence, we add $(v'_k \to x_{i\to\mathsf{T}})$ or $(v'_k \to x_{i\to\mathsf{F}})$ 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^Y_{Ont}. Next, we present the major steps of using G^Y_{Ont} to type a general graph G below.

Step 1. We look up each keyword $\ell \in \Sigma$ of G from G^Y_{Ont}. If ℓ is found, G^Y_{Ont} contains the ontology information of ℓ for the BiG-index framework.[14]

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

Step 3. Otherwise, we type ℓ' using the topmost type of G^Y_{Ont}.

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^Y_{Ont}.