# EDG-Based Question Decomposition for Complex Question Answering over Knowledge Bases

Xixin Hu, Yiheng Shu, Xiang Huang, and Yuzhong Qu(✉)

State Key Laboratory
for Novel Software Technology, Nanjing University, Nanjing, China
{xixinhu,yhshu,xianghuang}@smail.nju.edu.cn, yzqu@nju.edu.cn

**Abstract.** Knowledge base question answering (KBQA) aims at automatically answering factoid questions over knowledge bases (KBs). For complex questions that require multiple KB relations or constraints, KBQA faces many challenges including question understanding, component linking (e.g., entity, relation, and type linking), and query composition. In this paper, we propose a novel graph structure called Entity Description Graph (EDG) to represent the structure of complex questions, which can help alleviate the above issues. By leveraging the EDG structure of given questions, we implement a QA system over DBpedia, called `EDGQA`. Extensive experiments demonstrate that `EDGQA` outperforms state-of-the-art results on both LC-QuAD and QALD-9, and that EDG-based decomposition is a feasible way for complex question answering over KBs.

**Keywords:** Entity description graph · Question answering · Question decomposition

## 1 Introduction

Knowledge base question answering (KBQA) aims at answering factoid questions over knowledge bases (KBs) such as DBpedia, Freebase, and Wikidata. One of the widely used approaches of KBQA is semantic parsing, where natural language questions (NLQs) are transformed into formal queries on a KB [4]. Existing semantic-parsing-based approaches [2,26] have achieved promising results on simple questions that can be answered by matching a single relation (or path) in a KB [24].

However, answering complex questions that require retrieving multiple KB facts is still very challenging. For instance, the golden formal query shown in Fig. 1 (a) contains various components including a grounded entity, an intermediate variable, a variable for the final answer, and several KB predicates and type constraints. It requires accurate component linking and sub-query composition to build such a complicated formal query, which is not trivial. Several approaches enumerate possible relation paths and rank them all [1,4,13]. In this way, the search space grows exponentially with the length of relation paths, and the enumeration makes it difficult to find the precise query path.

A. Hotho et al. (Eds.): ISWC 2021, LNCS 12922, pp. 128–145, 2021.
https://doi.org/10.1007/978-3-030-88361-4_8

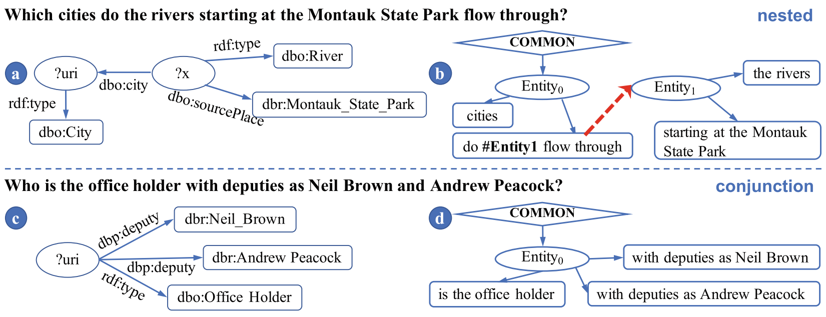

**Fig. 1.** SPARQL (a. and c.) and EDG (b. and d.) on two exemplar questions. The dashed line connects a description to an intermediate entity.

Question decomposition is a practical way of solving complex questions and has been used in recent researches. Existing approaches cast the decomposition problem as an instance of span prediction [15] or split point prediction [20] problem to generate sub-questions. However, multiple relations or constraints may still exist in sub-questions, making it difficult to be answered. Besides, they only define limited patterns of sub-question composition, which are inadequate for various scales of complex questions. In addition to sub-question generation, Abstract Meaning Representation (AMR) is also used to represent NLQs [11]. AMR helps to identify and compose the relations in NLQs but also brings granularity mismatch with KBs. To bridge the mismatch, complicated rules are designed. In general, the existing approaches have not yet reached a satisfactory level.

In this paper, we propose a novel graph structure called **Entity Description Graph** (EDG) to represent the structure of NLQs. Entities are at the center of how people represent and aggregate knowledge, which is the basic driving force of EDG. Examples of EDG are shown in Fig. 1 (b, d). The root, a diamond node, indicates the question type, e.g., "`COMMON`" means the question intends to find an entity or a set of entities. The entities in the EDG, oval nodes, are described by some verb phrases or noun phrases, indicated by rounded rectangles. In some cases, complex descriptions refer to intermediate entities, e.g., "*do #Entity1 flow through*" in Fig. 1 (b). By doing so, EDG can represent the graph structure of complex questions. Intuitively, EDG is an entity-centric graph, where phrase-level descriptions provide relational assertion or constraints about entities.

To generate EDG for a given question, we design some rules based on the constituency tree. We leverage EDG to generate sub-queries and integrate them based on the EDG structure. Furthermore, we implement a KBQA system based on EDG, called `EDGQA`[1]. Our experiments show that it outperforms state-of-the-art results on both LC-QuAD [22] and QALD-9 [23].

The remainder of the paper is organized as follows. Section 2 proposes the Entity Description Graph. Section 3 presents an overview of `EDGQA`. Section 4 details the EDG decomposition process, and Sect. 5 details the query generation.

[1] Source code is available at https://github.com/HXX97/EDGQA.

Section 6 presents our experiments. Section 7 discusses related work. Finally, the conclusions are given in Sect. 8.

**Table 1.** The types of nodes and edges in EDG

| Type of nodes | Comments |
|---|---|
| Root node (diamond) | Indicates the question type, e.g., COUNT: count the number of entities that meet the requirements JUDGE: determine whether a fact is established or not LIST: give a full list of entities that meet the requirements COMMON: find the answer that completes the question into a declarative sentence |
| Entity node (oval) | A placeholder for an entity to be found in the KB |
| Description node (rounded rectangle) | A phrase that describes its corresponding entity node |
| **Type of edges** | **Comments** |
| Quest edge | Connects a root node to the target entity |
| Constraint edge | Connects an entity node to its corresponding description nodes |
| Reference edge | Connects a description node to an intermediate entity nested in the description |

## 2 Entity Description Graph

In this paper, we use an RDF graph as a KB for question answering. An RDF graph $\mathcal{K}$ is a collection of triples in the form of $(s, r, o)$, e.g., (`dbr:Current_River`, `dbo:sourcePlace`, `dbr:Montauk_State_Park`)[2], where $s$, $r$, $o$ represent the subject, relation, and object, respectively. For a factoid question, the answer $\mathcal{A}$ is a subset of the entities in $\mathcal{K}$, or the result of a calculation such as aggregation (`COUNT`) and assertion (`ASK`). Hence, answering a question is to find the target *entities* in a KB that match the *descriptions* in the question, where the calculation is needed in some situations. For example, as indicated in Fig. 1 (d), "*is the office holder*", "*with deputies as Neil Brown*" and "*with deputies as Andrew Peacock*" are descriptions about the target entities of the question, say $Entity_0$.

Conjunction and nesting are common forms of complex questions [3,10]. In the case of *nested* questions, a description of an entity may contain intermediate entities described by some sub-descriptions. For example, given a phrase "*do the rivers starting at the Montauk State Park flow through*", we can extract an intermediate entity $Entity_1$ described by "*the rivers*" and "*starting at the Montauk State Park*". The intermediate entity is crucial to finding the final answer to the question.

In the case of *conjunction* questions, the descriptions about the target entity usually come from different aspects. For question "*Who is the office holder with deputies as Neil Brown and Andrew Peacock*", we can extract three different descriptions. "*is the office holder*" describes the occupation of the target entity, "*with deputies as Neil Brown*" and "*with deputies as Andrew Peacock*" describe the people associated with the target entity.

[2] dbr: http://dbpedia.org/resource, dbo: http://dbpedia.org/ontology.

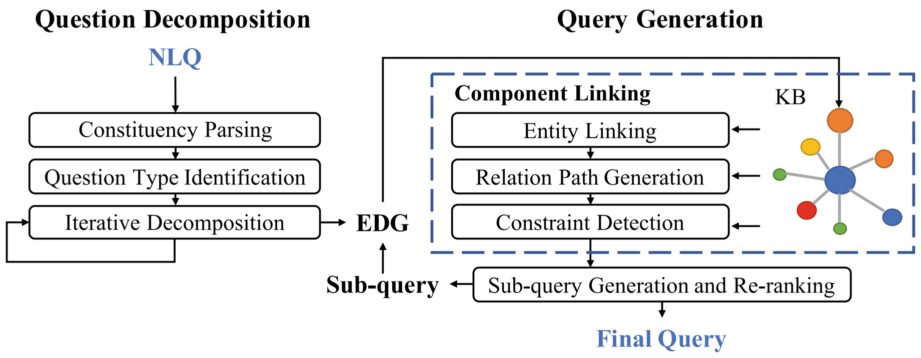

**Fig. 2.** `EDGQA` system overview: question decomposition phase and query generation phase.

By decomposing the question into the form of entities and their descriptions, we can have a straightforward view of the question structure, which indicates how the sub-queries should be integrated. Descriptions are at the phrase level, including verb phrases or noun phrases, etc.

Based on the above idea, we introduce the **Entity Description Graph** (EDG) to describe NLQs. An EDG is a rooted directed acyclic graph, consisting of three types of nodes and three types of edges, as shown in Table 1. The target entity is connected to the root node of an EDG. Several descriptions can be connected to an entity node. Intermediate entities can be referred to from some complicated descriptions. An entity and its descriptions form an **EDG block**, e.g., Block$_0$ and Block$_1$ in Fig. 3 (c). These two blocks are connected by a reference edge.

Representing NLQs by EDG brings the following benefits. 1) Candidate phrases are provided for component linking. 2) A structural basis for the sub-query composition of complex questions is provided. We will discuss these issues in Sect. 6.4.

## 3 Overview

Our proposed KBQA system `EDGQA` consists of two phases: question decomposition and query generation. The question decomposition phase generates an EDG for the NLQ, and the query generation phase generates and combines sub-queries based on the EDG. A system overview is illustrated in Fig. 2.

For an NLQ $q$, we first obtain a constituency parsing tree [5] of $q$, and then identify the question type according to the result of constituency parsing (Sect. 4.1). The generation process of EDG $\mathcal{G}$ is iterative (Sect. 4.2). Initially, the whole $q$ is viewed as a single description of the target entity. Then the description is iteratively parsed until no additional entity nodes or description nodes are generated. Finally, the EDG $\mathcal{G} = \langle root, \{b_0, b_1, ..., b_L\}\rangle$ is generated to represent the input NLQ $q$, where *root* denotes the root node and $b_i$ denotes the EDG block of $entity_i$.

After question decomposition, we generate the formal query $s$ based on EDG $\mathcal{G}$ (Sect. 5). For an EDG block $b_i$, we perform entity linking, relation path generation, and constraint detection for each description in $b_i$. All the detected components are ranked and combined to generate candidate sub-queries for block $b_i$, and top-$k$ sub-queries are retained as the query for $b_i$ after re-ranking (Sect. 5.1). If a description contains another entity nested within it, the intermediate entity is handled recursively. Then the sub-query for the intermediate entity is integrated into the query for block $b_i$ (Sect. 5.2). Finally, the formal query $s$ for $b_0$ is generated and executed against the KB $\mathcal{K}$ to get the final result of $q$.

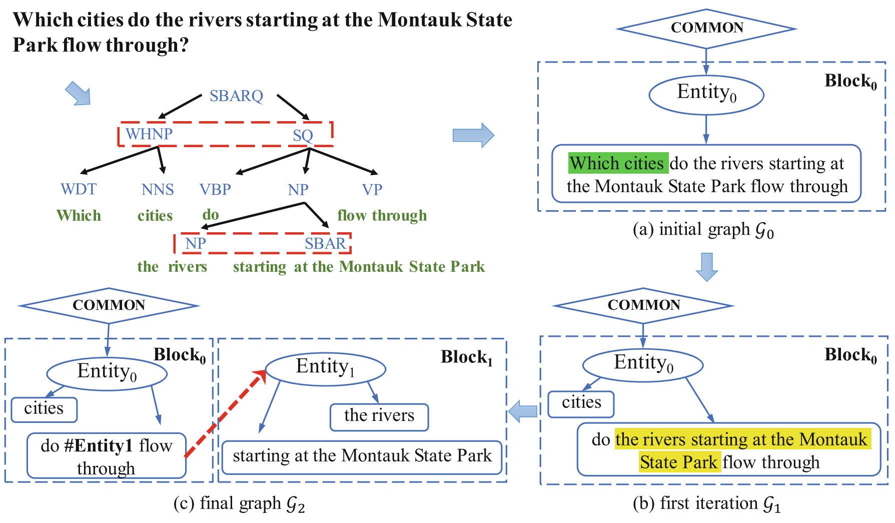
Which cities do the rivers starting at the Montauk State Park flow through?
SBARQ
WHNP
SQ
WDT
NNS
VBP
NP
VP
Which
cities
do
flow through
NP
SBAR
the rivers
starting at the Montauk State Park
COMMON
Entity0
Block0
Which cities do the rivers starting at the Montauk State Park flow through
(a) initial graph $\mathcal{G}_0$
COMMON
Entity0
Block0
cities
do the rivers starting at the Montauk State Park flow through
(b) first iteration $\mathcal{G}_1$
COMMON
Entity0
Block0
cities
do #Entity1 flow through
Entity1
Block1
the rivers
starting at the Montauk State Park
(c) final graph $\mathcal{G}_2$

**Fig. 3.** Running example of question decomposition, spans identified by W-rule and A-rule are marked.

## 4 Question Decomposition

In this phase, we decompose a question into an EDG. Our method for question decomposition consists of 3 steps: 1) constituency parsing, 2) question type identification and 3) iterative decomposition. The decomposition process is illustrated in Algorithm 1, which is detailed as follows.

### 4.1 Constituency Parsing and Question Type Identification

Constituency parsing aims to extract a constituency-based parse tree from a sentence that represents its syntactic structure according to a phrase structure

**Algorithm 1.** QuestionDecomposition

```
Input: A natural language question $q$
Output: An EDG $\mathcal{G}$ for $q$
 1: $\mathcal{T} \leftarrow$ ConstituencyParsing($q$)
 2: $\mathcal{G}_0, b_0 \leftarrow$ QuestionTypeIdentification($\mathcal{T}$)
 3: $i \leftarrow 0$, $j \leftarrow 0$ // the number of iteration and the number of current block
 4: repeat
 5:    $\mathcal{G} \leftarrow \mathcal{G}_i$
 6:    for all description $d$ in $b_j$ do
 7:       if A-Rule($d, \mathcal{T}$) or N-Rule($d, \mathcal{T}$) is matched then
 8:          $e_{j+1}, \bigcup_{k=1}^{h} d_{j+1,k} \leftarrow$ ExtractNewEntity($d$) // extract a new entity from $d$
 9:          $b_{j+1} \leftarrow \bigcup_{k=1}^{h} d_{j+1,k} \cup \{e_{j+1}\}$ // $h$ is the number of description nodes for
            $e_{j+1}$.
10:          $\mathcal{G}_{i+1} \leftarrow \mathcal{G}_i \cup b_{j+1}$ // add the newly generated block
11:          $i \leftarrow i + 1$
12:          $j \leftarrow j + 1$
13:       else if W-Rule($d, \mathcal{T}$) or C-Rule($d, \mathcal{T}$) is matched then
14:          $d_1^{'}, d_2^{'} \leftarrow$ SplitDescription($d$) // split $d$ into sub-descriptions
15:          $b_j \leftarrow b_j - \{d\}$
16:          $b_j \leftarrow b_j \cup \{d_1^{'}, d_2^{'}\}$ // replace $d$ with sub-descriptions
17:          $i \leftarrow i + 1$
18:       end if
19:    end for
20: until $\mathcal{G}$ equals $\mathcal{G}_i$
21: return $\mathcal{G}$
```

grammar. Since EDG is a phrase-level structural representation of the question, it is natural to generate EDG from the constituency parsing. We generate the constituency tree $\mathcal{T}$ based on Stanford CoreNLP Parser[3], which is shown in the upper left part of Fig. 3. Our decomposition is based on the phrase structure grammar of constituency parsing.

Typically, the root label of the constituency tree indicates the question type. For example, `SBARQ` denotes a direct question introduced by a *wh*-word and `SQ` denotes an inverted boolean question. We create a root node to represent the question type according to the root label. We have defined four question types: `COUNT`, `JUDGE`, `LIST`, and `COMMON`, detailed in Table 1. To identify the question type more precisely, we collect a set of trigger words based on co-occurrence frequency, e.g., "*count the number of*" usually indicates `COUNT` questions.

After question type identification, we create an entity node representing the target entity of the question, denoted by $entity_0$, and connect it to the root node with a quest edge. The whole question is viewed as a single description connected to the target entity before the following decomposition. The initial graph is denoted as $\mathcal{G}_0$, consisting of a root node and a block $b_0$, which is shown in Fig. 3(a).

[3] https://stanfordnlp.github.io/CoreNLP/.

## 4.2 Iterative Decomposition

After initial graph generation, we decompose the descriptions according to the constituency tree iteratively. The decomposing module is a rule-based system that detects the predefined phrase-structure patterns and then decomposes the descriptions into several sub-descriptions. In general, this module consists of the following rules:

**W-Rule** identifies wh-phrase labels, such as `WHNP`, `WHPP`, `WHADJP`. A wh-phrase (e.g. "*Which cities*") consists of a wh-word and a phrase that describes an aspect of the target entity. Most type constraints are derived from the wh-phrases and thus it is important to identify them at first. We split the original description into two sub descriptions: the phrase introduced by the wh-word and the others, shown in Fig. 3 (a).

**C-Rule** identifies coordinating conjunction tags, such as `CC` (e.g., "*and*") or `RB` (e.g., "*also*"). Conjunction is a regular form of complex questions, which joins multiple constraints or relations. The clauses split by a conjunction word usually introduce different constraints. Note that there may exist coordination ambiguity in such sentences. For example, given a description "*with deputies as Neil Brown and Andrew Peacock*", we need to know that "*Andrew Peacock*" is conjoined with "*Neil Brown*" instead of "*with deputies as Neil Brown*". In this case, we rewrite the description into two sub-descriptions: "*with deputies as Neil Brown*" and "*with deputies as Andrew Peacock*".

**A-Rule** identifies attributive clauses or adverbial clauses labeled by `SBAR`, `SBARQ` or `SQ`. An attributive clause (e.g., "*the rivers starting at the Montauk State Park*") serves as an attribute to a noun or pronoun in the main clause. It usually indicates the existence of an intermediate entity when such an attributive clause is identified. We extract the clause (e.g., "*starting at the Montauk State Park*") along with the antecedent (e.g., "*the rivers*") from the original description and create an entity node representing this intermediate entity, as is shown in Fig. 3 (b).

**N-Rule** identifies compound noun phrases which contain prepositional phrases (e.g., "*the daughter of Obama*") labeled by `PP` or possessive phrases (e.g., "*Obama's daughter*") labeled by `POS`. Similar to attributive clauses, compound noun phrases tend to include intermediate entities. We extract such compound noun phrases and create an entity node referred from the original description.

In summary, W-Rule and C-Rule are designed to split a description into several sub-descriptions (**SplitDescription** in Algorithm 1). A-Rule and N-Rule are designed to extract intermediate entities from the original descriptions (**ExtractNewEntity** in Algorithm 1).

At each iteration, we apply the above rules to each description in the block, and then split the descriptions or generate new entity nodes according to the matched rules. This process is repeated until no description is split or extracted. As shown in Fig. 3, the graph after the first iteration is denoted as $\mathcal{G}_1$ and the final graph is denoted as $\mathcal{G}_2$, which means the process is iterated twice. Finally, $\mathcal{G}_2$ is returned as the EDG generated for the original question.

## 5 Query Generation

An EDG gives a hierarchical decomposition of an NLQ and provides a clear structure for sub-query generation and composition without the need to enumerate a large number of possible relation paths or sub-query combinations. The query generation starts from the EDG structure.

We denote the EDG as $\mathcal{G} = \langle root, \{b_0, b_1, ..., b_L\}\rangle$, where $root$ is the root node, $b_0$ is the block of the target entity asked by the question and $b_i$ is the block of intermediate $entity_i$. Intuitively, a block is a structured representation for an entity, and thus we can map a block to a formal sub-query whose execution results are the answers for the entity. Note that a reference edge connects a block $b_i$ to another block referred from $b_i$. In this way, the EDG blocks form a tree. We follow the trace of the block tree to generate sub-queries for each block recursively. If a block $b_i$ refers to one or more blocks, we generate the sub-queries of its referred blocks and then merge them into the query for $b_i$. Figure 4 shows the process of query generation for the example in Fig. 3, which will be detailed below. The process of generating queries for each block is given in Algorithm 2. It is a recursive algorithm and the query generated for $b_0$ is viewed as the query for the full NLQ.

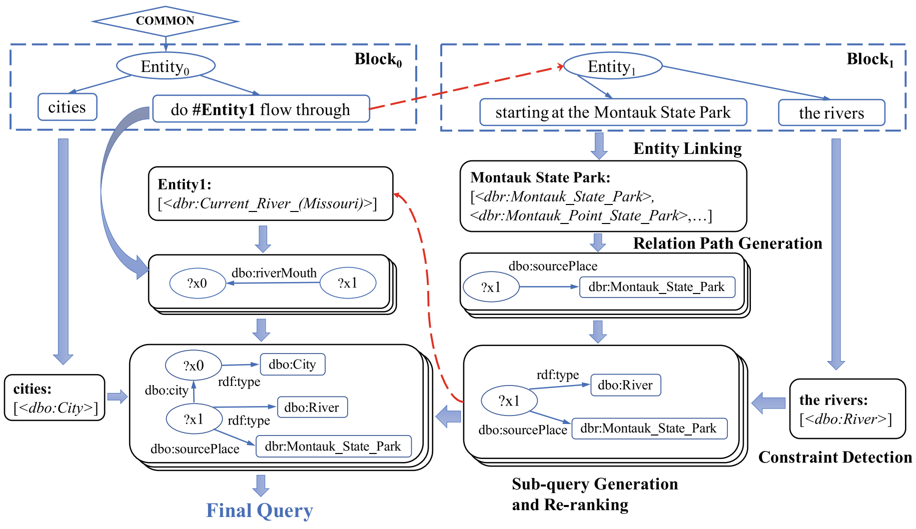

**Fig. 4.** Running example of sub-query generation and composition.

### 5.1 Block Query Generation

We generate the query for a block through a pipeline consisting of 4 modules: 1) entity linking, 2) relation path generation, 3) constraint detection, and 4) sub-query generation and re-ranking. The starting three modules form a phrase-level component linking module (line 4 in Algorithm 2) and the last module integrates the components to generate candidate sub-queries for this block (line 6 in Algorithm 2).

**Entity Linking.** We ensemble Dexter [21], EARL [8] and Falcon [18] as one entity retriever. We assume that a description contains at most one entity in the KB. For a description $d_i$ in block $b$, the entity retriever gives a list of candidate entities denoted as $E_i$, e.g., `dbr:Montauk_State_Park` and `dbr:Montauk_Point_State_Park` in Fig. 4. Then we average distance-based string similarity metrics (Jaro distance, Dice coefficient, and Levenshtein distance) to measure the confidence score $p_{ent}(e)$ of each $e \in E_i$. To reduce the ambiguity brought by the retriever, we assume that the distance of candidate entities in a block should be relatively close in the KB [8]. If a candidate entity in a description is not reachable in 2 hops from any candidate in other descriptions in the KB, the entity is abandoned. In this way, EDG can help the entity disambiguation.

**Relation Path Generation.** We follow the work of [16] to generate candidate relations of $E_i$ for each description $d_i$. The candidate relation list is denoted as $R_i$.

A major challenge is to properly rank these candidate relations according to their relevance to the question. Since EDG has already decomposed the question into phrase-level descriptions, there is no need to detect relation mentions. The description naturally provides mentions for relation linking. To select the accurate relation mapping to description $d_i$, we measure the similarity from both literal and semantic aspects.

For semantic similarity, we use a BERT-based [6] semantic matching model. Specifically, the model takes the text of $d_i$ with entity-masked mention (e.g.,

**Algorithm 2.** BlockQueryGeneration

```
Input: An EDG block $b = \{d_i\}_{i=1}^h$
Output: A set of candidate queries $\mathcal{Q}$ for $b$
1: $\mathcal{Q} \leftarrow \emptyset$
2: if $b$ contains no reference edge then
3:    for all description $d_i$ in $b$ do
4:       $E_i \leftarrow$ EntityLinking($d_i$), $R_i \leftarrow$ RelationLinking($d_i$), $C_i \leftarrow$
         ConstraintDetection($d_i$)
5:    end for
6:    $\mathcal{Q} \leftarrow$ Sub-query generation and re-ranking for $b$ given $(\langle E_i, R_i, C_i \rangle)_{i=1}^h$
7: else
8:    for all description $d_i$ in $b$ do
9:       if exists $b_j$ referred from $d_i$ then
10:         $Q_j \leftarrow$ BlockQueryGeneration($b_j$) // generate sub-query for $b_j$
11:         $d_i \leftarrow$ SubQueryIntegration($d_i$, $Q_j$) // take the sub-query results
12:      end if
13:   end for
14:   $\mathcal{Q} \leftarrow$ BlockQueryGeneration($b$) // no reference edge in block $b$ after the above
      loop
15: end if
16: return $\mathcal{Q}$
```

"*starting at*") and the candidate relations in $R_i$ (e.g., `dbo:sourcePlace`) as input. The output is the probability of the sentence-pair binary classification (related or not related), regarded as the semantic similarity $sim_{sem}$ between the relation and the description. Due to the lack of training data, we collect distant supervision data from the training sets of QA datasets. More precisely, for a question and query pair in QA datasets, we mask the entity mentions in the original question. Then we extract all the relations from the golden query as positive cases, which means the relations can be inferred from the masked question. To collect negative cases, we randomly sample some relations associated with the grounded entities in the question.

For literal similarity, we use string similarity $sim_{lit}$ in the same way as entity linking. We also use a paraphrase dictionary [25] to get the paraphrases of the relation and calculate the similarity $sim_{dict}$ between the paraphrases and the descriptions. Finally, the relation confidence $p_{rel}$ is calculated as Eq. 1.

$$p_{rel}(r) = \mu \ sim_{sem}(r,m) + (1-\mu) \ \max(sim_{lit}(r,m), sim_{dict}(r,m))) \tag{1}$$

where $r$ is the label of the relation and $m$ stands for the mention (description with entity masked). The hyper-parameter $\mu$ is used to balance the importance of semantic score and literal score.

**Constraint Detection.** We mainly focus on type constraints and adopt DBpedia Ontology[4] as the type bank. For noun phrase descriptions (e.g., "*the rivers*" in Fig. 4) where no entities are spotted, we measure its string similarity to all the DBpedia types and reserve types with high similarity score (e.g., "`dbo:River`") as candidates. The list of all candidate types for description $d_i$ is denoted as $C_i$.

**Sub-query Generation and Re-ranking.** When all the components are linked by previous modules, we generate the sub-query $s$ for block $b$ by compositing the candidate entities, relations, and constraints. More specifically, as shown in Fig. 4, we combine the triple patterns generated from descriptions into a query $s$ for a block. The query $s$ can be modeled as a set of triples $s = \{\langle ?x_b, r_i, e_i \rangle_{i=1}^{h}\}$, where $r_i \in R_i \cup \{\texttt{rdf:type}\}$, $e_i \in E_i \cup C_i$ and $?x_b$ denotes the answer of block $b$. To measure the quality of this query, we multiply the confidence scores of all the components in $s$ to get the component-level score $p_{comp}$.

$$p_{comp}(s) = \prod_{i=1}^{h} p_{rel}(r_i) \cdot p_{ent}(e_i) \tag{2}$$

where $h$ is the number of triples in $s$.

Besides the above component-level score $p_{comp}(s)$, we also take advantage of the context of this block by a block-level semantic matching model to estimate the quality of sub-query candidates. Specifically, we convert the block $b$ and the query $s$ to two sequences with additional special tokens, and then utilize a BERT-based semantic matching model to measure the correlation between the

[4] https://wiki.dbpedia.org/services-resources/ontology.

two sequences. Take the block $b_1$ in Fig. 4 as an example, we represent block $b_1$ as:

`[BLK]` `[DES]` the rivers `[DES]` starting at the Montauk State Park

where `[BLK]` tags the start of a block and `[DES]` tags the start of a description. Similarly, we represent the candidate query as:

`[TRP]` x1 type river `[TRP]` x1 source place Montauk State Park

where `[TRP]` denotes the start of a triple in the query. The semantic matching task is also modeled as a sentence-pair classification task here, and the score given by the model is denoted as $p_{blk}$. The final score of a sub-query is shown in Eq. 3 by combining the above two scores.

$$p_{query}(s) = \lambda \; p_{comp}(s) + (1 - \lambda) \; p_{blk}(s) \tag{3}$$

where $\lambda$ is a hyper-parameter. Finally, a ranked list of candidate sub-queries for block $b$ is generated. We select top-$k$ sub-queries as the output of the block query generation.

### 5.2 Sub-query Integration

If an EDG block refers to one or more blocks, we recursively solve referred blocks beforehand and get a collection of candidate sub-queries. We execute the candidate sub-queries to get candidate intermediate entities, e.g., $\#Entity_1$ of block $b_0$ in Fig. 4. The candidate intermediate entities are viewed as the entity linking results of the descriptions referring to them, e.g., "do $\#Entity_1$ flow through" of block $b_0$ in Fig. 4. In this way, we can generate the query for this block. Finally, the formal query $s_0$ for $b_0$ is generated and executed against the KB to retrieve the answer to the question.

Instead of generating sub-queries separately [3,19,20], we consider them together, because the results of the sub-queries bring extra evidence for follow-up query generation. Take the block $b_0$ in Fig. 4 as an example, it is unlikely to select the correct relation `dbo:city` if we do not get the intermediate entity `dbo:Current_River_(Missouri)` at first. Besides, we avoid enumerating possible combinations of sub-queries as [19], since the composition of sub-queries is guided by the EDG structure.

## 6 Experiments

### 6.1 Experimental Setup

**Datasets.** We use two KBQA datasets to evaluate `EDGQA`.

- **LC-QuAD** [22] is the **L**arge-scale **C**omplex **Qu**estion **A**nswering **D**ataset over DBpedia (2016-04), with 4,000 training and 1,000 test questions. The average length of questions is 11.46 words.
- **QALD-9** [23] is an open-domain QA campaign over DBpedia (2016-10) with 408 training and 150 test questions. The average length of questions is 7.49 words.

All experiment results are reported on the LC-QuAD and QALD-9 test set. The training set is used for distant supervision for relation linking and sub-query re-ranking.

**Metrics.** For all the experiments, the metrics are precision (P), recall (R), and macro F1 (F) for both datasets. Macro-F1-QALD [23] (F-Q) is also included for QALD-9.

**Comparative Approaches.** We compare EDGQA with several methods including non-decomposition and decomposition approaches, as well as the graph-driven approach. All comparative approaches include the entity linking module.

1) **WDAqua** [7] is a KB agnostic QA system based on keyword extraction and query templates.
2) **QAmp** [24] uses unsupervised message passing, which propagates confidence scores obtained by parsing the NLQ and matching KB terms to possible answers.
3) **gAnswer** [9] generates the semantic query graph to model the query intention in the NLQ in a structural way.
4) **NSQA** [11] uses AMR to parse questions and performs logical queries via neural networks.

The results of WDAqua and QAmp for LC-QuAD 1.0 are from the official leaderboard[5], and the results of WDAqua and gAnswer for QALD-9 are from the QALD-9 official report [23].

**Implementation Details.** For the semantic matching model in relation path ranking and sub-query re-ranking, we fine-tune the BERT model[6] for the sentence-pair classification task with default hyper-parameters. For hyper-parameters of the weighted average in Eq. 1 and 3, we set $\mu$ to 0.45 and $\lambda$ to 0.55. We set $k$ to 2 for intermediate entities, which means we execute the top-2 ranked sub-queries and merge the results into a candidate set. For target entities of the question, we only reserve the top-1 ranked query.

## 6.2 Results and Discussion

As is shown in Table 2, EDGQA outperforms state-of-the-art results on both LC-QuAD and QALD-9. Both precision and recall of EDGQA are at a high level compared with competitors. It makes a significant improvement on LC-QuAD compared with the best competitor (39.5% improvement on F1). QALD-9 questions are more difficult to answer, but EDGQA still achieves competitive results. gAnswer with semantic query graph and NSQA with AMR parsing perform relatively well, and the latter one had the highest precision on QALD-9.

To compare EDGQA with other question decomposition methods, we generate sub-questions with the model from [20] and follow its original composition strategy to implement a comparative approach with existing linking tools. This model

[5] http://lc-quad.sda.tech/lcquad1.0.html.
[6] https://github.com/google-research/bert.

**Table 2.** QA performance on LC-QuAD and QALD-9

| Approaches | LC-QuAD 1.0 | | | QALD-9 | | | |
|---|---|---|---|---|---|---|---|
| | P | R | F | P | R | F | F-Q |
| WDAqua [7] | 0.22 | 0.38 | 0.28 | 0.261 | 0.267 | 0.250 | 0.289 |
| QAmp [24] | 0.25 | 0.50 | 0.33 | – | – | – | – |
| gAnswer [9] | – | – | – | 0.293 | 0.327 | 0.298 | 0.430 |
| NSQA[11] | 0.38 | 0.40 | 0.38 | **0.314** | 0.322 | 0.309 | 0.453 |
| Pointer network* [20] | 0.220 | 0.241 | 0.230 | 0.096 | 0.109 | 0.102 | 0.190 |
| EDGQA | **0.505** | **0.560** | **0.531** | 0.313 | **0.403** | **0.320** | **0.461** |

FQ: Macro-F1-QALD measure, *: our implementation

named **pointer network** [20] identifies split points in the NLQ and decomposes the NLQ into spans. In our implementation, if it outputs the empty set of answers for a sub-question, we solve it directly without decomposition. It is denoted as pointer network* in Table 2.

The decomposition of the pointer network does not present a significant advantage to the QA task. Since its decomposition is only a simple combination of spans and not entity-centric, there may still be multiple relations or constraints in a sub-question, and the entity to which they belong is not clear. In contrast, EDGQA composes individual sub-queries in an entity-centric manner, and their composition has a structure to follow.

## 6.3 Quality of Question Decomposition

Currently, it is hard to make a comprehensive criterion for evaluating the quality of question decomposition. We attempt to evaluate the quality of EDG generation in terms of question type identification and the number of EDG blocks.

Correct question type identification is the prerequisite for the QA task. The type of most questions are identified correctly on both datasets (type accuracy in Table 3).

In the ideal decomposition, an EDG block corresponds to an entity to be queried on the KB (Sect. 2), which is represented by a variable in a SPARQL query. We count the number of questions where the number of EDG blocks (#b) is equal to the number of SPARQL variables (#v) and the number of those questions where #b is not equal to #v, separately. The F1 measure of answering these two types of questions is shown in Table 3, respectively. The significant difference between them shows that the relationship between #b and #v for each question is a distinctive feature that influences the QA performance. Questions that meet this criterion (#b = #v) are answered significantly better than other questions. Thus, it can be regarded as one criterion to judge the quality of EDG generation since it is an objective criterion and highly related to QA.

We observe that most of the questions meet this criterion on both LC-QuAD and QALD-9 (79.4% and 73.3%). The EDG block representing an entity can

**Table 3.** Detailed comparison of EDGs with golden SPARQL queries

| Datasets | Criterion | #question | Type accuracy | F |
|---|---|---|---|---|
| LC-QuAD | $\#b = \#v$ | 794 | 0.997 | 0.650 |
| | $\#b \neq \#v$ | 206 | 1.000 | 0.107 |
| QALD-9 | $\#b = \#v$ | 110 | 0.945 | 0.416 |
| | $\#b \neq \#v$ | 40 | 0.900 | 0.014 |

**Table 4.** Ablation study on `EDGQA`, EL: entity linking, RL: relation linking

| Methods | LC-QuAD 1.0 | | | QALD-9 | | | |
|---|---|---|---|---|---|---|---|
| | P | R | F | P | R | F | F-Q |
| `EDGQA` | **0.505** | **0.560** | **0.531** | **0.313** | **0.403** | **0.320** | **0.461** |
| w/o decomposition | 0.368 | 0.455 | 0.407 | 0.219 | 0.272 | 0.226 | 0.368 |
| w/o phrase level RL | 0.301 | 0.367 | 0.331 | 0.138 | 0.230 | 0.137 | 0.345 |
| w/o phrase level EL & RL | 0.284 | 0.350 | 0.314 | 0.138 | 0.230 | 0.137 | 0.345 |
| w/o re-ranking | 0.485 | 0.540 | 0.511 | 0.294 | 0.390 | 0.301 | 0.430 |

correspond well to the SPARQL variable. As an entity-centric decomposition, EDG gives an appropriate granularity.

## 6.4 Ablation Study

To illustrate the impact of EDG on component linking and query generation, we perform several ablation experiments as shown in Table 4.

**W/o Decomposition.** To verify the effect of decomposition on complex query generation, we solve the whole question as a single description directly. The F1 measure decreases by 23.35% and 29.37% on two datasets, respectively. It shows that only the linking and re-ranking (line 2–6 Algorithm 2) are not sufficient for complex questions. The split of descriptions and the recursive query generation (line 7–15 in Algorithm 2) brought by EDG are also necessary.

**W/o Phrase Level Entity and Relation Linking.** To verify the usefulness of the EDG decomposition for linking, we put the whole question into linking tools instead of these phrases, and still apply the recursive query generation. The F1 measure decreases by 40.87% and 57.19% on two datasets, respectively. It shows that complex questions remain a challenge for existing linking tools. The granularity of the EDG decomposition helps the correspondence of NLQ to entities or relations in the KB.

**W/o Re-ranking.** To verify the effect of the re-ranking method (Sect. 5.1) with EDG block information, we run QA without the re-ranking step. The F1 measure decreases by 3.77% and 5.94% on two datasets, respectively. While phrase-level component linking has worked well, the re-ranking method still aids the QA task.

It also shows that our query generation does not highly depend on a ranking model compared to the methods enumerating relation paths [4,12].

### 6.5 Error Analysis

We analyze all the questions answered incorrectly by `EDGQA` (F1 $<$ 1.0). The major causes of errors are summarized as follows.

**Relation Linking Error (38.63%).** We observe that relation linking is a bottleneck of `EDGQA`. Although we leverage several semantic matching models, it is still difficult to select the correct relation from all the candidates. There are two main reasons for error relation linking: (1) indistinguishable relations, e.g., the golden query contains "`dbp:founders`"[7] while our system selects "`dbp:founder`"; (2) implicit relations with no explicit evidence, e.g., given a question "*Give me all animals that are extinct.*", the golden query contains a relation path `?uri dbo:conservationStatus ''EX''` to represent "*are extinct*" while `EDGQA` fails to identify this relation.

**Entity Linking Error (37.03%).** Some questions require complex reasoning and disambiguation for entity linking, which is challenging for existing entity linking tools. For example, `dbr:Lee_Robinson_(American_football)` is a desired entity URI in the question "*What city has the football team in which Lee Robinson debuted?*". However, `EDGQA` links it to "`dbr:Lee_Robinson_(footballer)`".

**Question Decomposition Error (11.29%).** About 11.29% of questions are decomposed incorrectly due to constituency parsing errors and limited coverage of our decomposition rules. Since we generate the query based on EDG, an incorrect EDG usually results in an inaccurate query. For example, given a question "*Which monarchs of the United Kingdom were married to a German?*", `EDGQA` decomposes the question into two descriptions: "*monarchs of the United Kingdom*" and "*were married to a German*", where `EDGQA` fails to identify the intermediate entity represented by "*a German*".

## 7 Related Work

**Complex Question Answering Over KBs.** Traditional KBQA researches [2,17] mainly focus on learning semantic parsers that translate questions into logical forms for simple questions. To address the challenges brought by complex questions, recent researches propose an alternative way by reducing semantic parsing to query graph generation [1,13,14,26]. They first identify the main relation path and then adds constraints to it. To select the correct query, they enumerate possible candidate query graphs and rank them by neural semantic matching models. As the number of candidates grows, it is more difficult to find accurate query graphs. To reduce the search space of candidate queries, existing works introduce constraints from various perspectives. Chen et al. [4] leverage

[7] http://dbpedia.org/property.

the structural to restrict candidates. Lan et al. [12] present a beam-search based method to generate the query graph iteratively. The main difference between our approach and previous ones is that we leverage EDG to generate sub-queries, and avoid exhaustive enumeration of query candidates.

**Question Decomposition.** There are mainly two directions for question decomposition: split-based and template-based methods. Talmor and Berant [20] use a pointer network to find split points for an NLQ. Zhang et al. [27] further utilize this decomposition for semantic parsing. Min et al. [15] decompose questions in a span prediction manner for machine reading comprehension task. These split-based decomposition methods take spans as sub-questions while a sub-question may still contain multiple relations and constraints. Another direction is template-based methods, which decompose the NLQ $q$ by pre-collected templates. The basic idea is to calculate the similarity between sub-sentences of $q$ and natural language patterns [28]. Shin et al. [19] build a question-query graph library to match sub-questions to sub-query graphs and then combine them into a complete query. Instead of collecting templates, EDG provides guidance on sub-query generation and integration, which helps to generate accurate complete queries.

## 8 Conclusions

In this paper, we studied the problem of decomposing complex questions for KBQA. The main contributions of this paper are summarized as follows.

- We propose a novel graph structure called Entity Description Graph (EDG) to represent complex questions, which is an entity-centric decomposition.
- We present a rule-based method to decompose a question into an EDG iteratively.
- We implement an EDG-based KBQA system called `EDGQA`. By leveraging the EDG structure, `EDGQA` recursively generates and integrates sub-queries effectively.
- Experiments show that `EDGQA` outperforms state-of-the-art results on both LC-QuAD and QALD-9. In particular, it makes significant improvements on LC-QuAD.

In future work, we consider the use of neural networks to generate EDGs. Besides, we are going to apply EDG to answering complex questions on other KBs such as Freebase. Furthermore, it is also interesting to incorporate EDG with a retrieve-based method or template-based method for answering complex questions.

**Acknowledgements.** This work was supported by the National Natural Science Foundation of China under Grant 61772264. We would like to thank Xiaoyin Pan for her efforts in the early stages of this work.

## References

1. Bao, J., Duan, N., Yan, Z., Zhou, M., Zhao, T.: Constraint-based question answering with knowledge graph. In: COLING, pp. 2503–2514 (2016)
2. Berant, J., Chou, A., Frostig, R., Liang, P.: Semantic parsing on Freebase from question-answer pairs. In: EMNLP, pp. 1533–1544 (2013)
3. Bhutani, N., Zheng, X., Jagadish, H.V.: Learning to answer complex questions over knowledge bases with query composition. In: CIKM, pp. 739–748 (2019)
4. Chen, Y., Li, H., Hua, Y., Qi, G.: Formal query building with query structure prediction for complex question answering over knowledge base. In: IJCAI, pp. 3751–3758 (2020)
5. De Marneffe, M.C., MacCartney, B., Manning, C.D., et al.: Generating typed dependency parses from phrase structure parses. In: LREC, vol. 6, pp. 449–454 (2006)
6. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: NAACL-HLT, pp. 4171–4186 (2019)
7. Diefenbach, D., Singh, K., Maret, P.: WDAqua-core0: a question answering component for the research community. In: Dragoni, M., Solanki, M., Blomqvist, E. (eds.) SemWebEval 2017. CCIS, vol. 769, pp. 84–89. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69146-6_8
8. Dubey, M., Banerjee, D., Chaudhuri, D., Lehmann, J.: EARL: joint entity and relation linking for question answering over knowledge graphs. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11136, pp. 108–126. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00671-6_7
9. Hu, S., Zou, L., Yu, J.X., Wang, H., Zhao, D.: Answering natural language questions by subgraph matching over knowledge graphs. TKDE **30**(5), 824–837 (2018)
10. Kalyanpur, A., Patwardhan, S., Boguraev, B., Lally, A., Chu-Carroll, J.: Fact-based question decomposition in deepQA. IBM J. Res. Dev. **56**(3.4), 13:1–13:11 (2012)
11. Kapanipathi, P., et al.: Question answering over knowledge bases by leveraging semantic parsing and neuro-symbolic reasoning. arXiv preprint arXiv:2012.01707 (2020)
12. Lan, Y., Jiang, J.: Query graph generation for answering multi-hop complex questions from knowledge bases. In: ACL, pp. 969–974 (2020)
13. Luo, K., Lin, F., Luo, X., Zhu, K.: Knowledge base question answering via encoding of complex query graphs. In: EMNLP, pp. 2185–2194 (2018)
14. Maheshwari, G., Trivedi, P., Lukovnikov, D., Chakraborty, N., Fischer, A., Lehmann, J.: Learning to rank query graphs for complex question answering over knowledge graphs. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11778, pp. 487–504. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30793-6_28
15. Min, S., Zhong, V., Zettlemoyer, L., Hajishirzi, H.: Multi-hop reading comprehension through question decomposition and rescoring. In: ACL, pp. 6097–6109 (2019)
16. Pan, J.Z., Zhang, M., Singh, K., Harmelen, F., Gu, J., Zhang, Z.: Entity enabled relation linking. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11778, pp. 523–538. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30793-6_30
17. Reddy, S., Lapata, M., Steedman, M.: Large-scale semantic parsing without question-answer pairs. TACL **2**, 377–392 (2014)

18. Sakor, A., et al.: Old is gold: linguistic driven approach for entity and relation linking of short text. In: NAACL, pp. 2336–2346 (2019)
19. Shin, S., Lee, K.H.: Processing knowledge graph-based complex questions through question decomposition and recomposition. Inf. Sci. **523**, 234–244 (2020)
20. Talmor, A., Berant, J.: The web as a knowledge-base for answering complex questions. In: NAACL HLT, vol. 1, pp. 641–651 (2018)
21. Trani, S., Ceccarelli, D., Lucchese, C., Orlando, S., Perego, R.: Dexter 2.0: an open source tool for semantically enriching data. In: ISWC, pp. 417–420 (2014)
22. Trivedi, P., Maheshwari, G., Dubey, M., Lehmann, J.: LC-QuAD: a corpus for complex question answering over knowledge graphs. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 210–218. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_22
23. Usbeck, R., Gusmita, R.H., Ngomo, A.C.N., Saleem, M.: 9th challenge on question answering over linked data (QALD-9) (invited paper). In: Semdeep/NLIWoD@ISWC (2018)
24. Vakulenko, S., Garcia, J.D.F., Polleres, A., de Rijke, M., Cochez, M.: Message passing for complex question answering over knowledge graphs. In: CIKM, pp. 1431–1440 (2019)
25. Xue, B., Hu, S., Zou, L., Cheng, J.: The value of paraphrase for knowledge base predicates. In: AAAI, pp. 9346–9353 (2020)
26. Yih, W.t., Chang, M.W., He, X., Gao, J.: Semantic parsing via staged query graph generation: question answering with knowledge base. In: ACL-IJCNLP, pp. 1321–1331 (2015)
27. Zhang, H., Cai, J., Xu, J., Wang, J.: Complex question decomposition for semantic parsing. In: ACL, pp. 4477–4486 (2019)
28. Zheng, W., Yu, J.X., Zou, L., Cheng, H.: Question answering over knowledge graphs: question understanding via template decomposition. In: Proceedings of the VLDB Endowment (2018)