MathGraph: A Knowledge Graph for Automatically Solving Mathematical Exercises

Tianyu Zhao[1(✉)], Yan Huang[2], Songfan Yang[2], Yuyu Luo[1], Jianhua Feng[1], Yong Wang[1], Haitao Yuan[1], Kang Pan[1], Kaiyu Li[1], Haoda Li[1], and Fu Zhu[1]

[1] Tsinghua University, Beijing, China
{zhaoty17,wangy18,yht16,pk16,ky-li18,lhd16,zhuf18}@mails.tsinghua.edu.cn
{luoyuyu,fengjh}@tsinghua.edu.cn
[2] TAL Education Group, Beijing, China
{galehuang,yangsongfan}@100tal.com

Abstract. Knowledge graphs are widely applied in many applications. Automatically solving mathematical exercises is also an interesting task which can be enhanced by knowledge reasoning. In this paper, we design MathGraph, a knowledge graph aiming to solve high school mathematical exercises. Since it requires fine-grained mathematical derivation and calculation of different mathematical objects, the design of MathGraph has major differences from existing knowledge graphs. MathGraph supports massive kinds of mathematical objects, operations, and constraints which may be involved in exercises. Furthermore, we propose an algorithm to align a semantically parsed exercise to MathGraph and figure out the answer automatically. Extensive experiments on real-world datasets verify the effectiveness of MathGraph.

Keywords: Knowledge graph · Mathematical exercise · Knowledge reasoning

1 Introduction

Currently, large scale knowledge graphs are widely used in many real-world applications, such as semantic web search, and question-answer systems, natural language processing, and data analytic. For example, if we ask "What is the highest mountain?" on a web search engine, it may directly show the answer "Everest" with the help of a knowledge graph.

Recently intelligent education is more and more popular and automatically resolving mathematical exercises can help students improve the comprehensive ability. However, it is rather challenging to automatically resolve mathematical exercises without knowledge graphs, because it requires to use complex semantics and extra calculations. In this paper, we propose **MathGraph**, a knowledge graph aiming to solve high school mathematical exercises. MathGraph must be specially designed and differentiated from other knowledge graphs. The reasons are listed as follows:

G. Li et al. (Eds.): DASFAA 2019, LNCS 11446, pp. 760–776, 2019.
https://doi.org/10.1007/978-3-030-18576-3_45

1. ***Knowledge in MathGraph belongs to a very specific domain.*** Building MathGraph requires specific mathematical knowledge. Traditional knowledge graphs are built based on extensive semantic data, e.g., Wikipedia. However, it is very hard to get the semantic data for mathematical problems.
2. ***Knowledge in MathGraph is stored in class-level rather than instance-level.*** Most of the traditional knowledge graphs focus on extracting instances, categories, and relations among instances. For example, a 3-tuple (Beijing, *isCaptialOf*, China) shows a relation between two instances. However, in MathGraph, there is no instance in the origin graph, but only many class-level mathematical objects (such as `Complex Number`, `Ellipse`, etc.). Only if an exercise is given, instances will be created accordingly.
3. ***MathGraph supports mathematical derivation and calculation.*** The reasoning process of mathematical problems is different from other problems, because besides logical relation, mathematical derivation must be included in the knowledge graph to solve mathematical exercises.

Thus, in this paper, we focus on building a knowledge graph MathGraph for resolving mathematical problems. We propose an effective algorithms to align a mathematical problem to MathGraph, and use the aligned sub-graph to resolve a mathematical exercise. The contributions of this paper are as follows.

- We specially design the structure of MathGraph to support mathematical derivation and calculation. We model different mathematical objects, operations and constraints in MathGraph. To the best of our knowledge, this is the first attempt to build a knowledge graph for resolving mathematical problems.
- We propose an algorithm to align a mathematical problem to MathGraph.
- We design a method to resolve mathematical exercises by the help of a semantic parser.
- Experimental study shows great performance of MathGraph and our proposed method.

Figure 1 gives an overview of the exercise-solving process with MathGraph. We detail the structure of MathGraph and the exercise-solving algorithm later.

2 Related Work

2.1 Reasoning with Knowledge Graph

Since knowledge graphs can provide well-structured information and relations of the entities, it is known to be useful to do reasoning in many tasks, such as query answering and relation inference (i.e., to infer missing relations in the knowledge graph [4,11,12]). Guu et al. [7] proposed a technique to answer queries on knowledge graph by "compositionalizing" a broad class of vector space models, which preforms well on query answering and knowledge graph completion. Toutanova et al. [17] proposed a dynamic programming algorithm to incorporate all paths in knowledge graph within a bounded length, and modelled entities

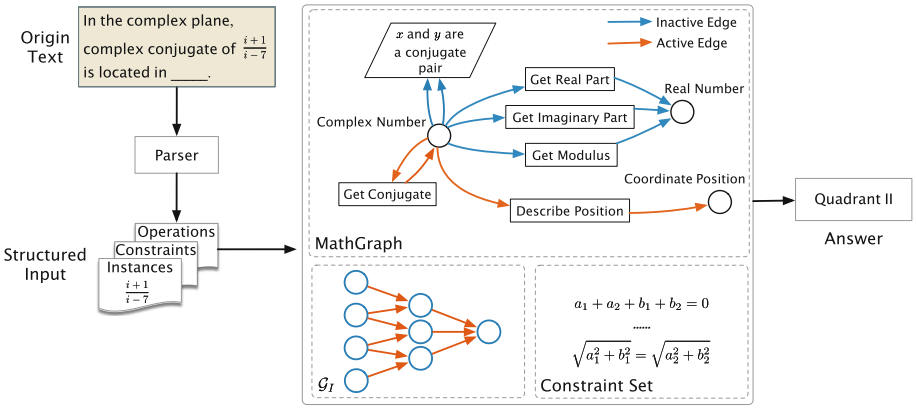


Fig. 1. Overview of using MathGraph to solve a mathematical exercise

and relations in the compositional path representations. Zhang et al. [18] proposed a deep learning architecture and a variational learning algorithm, which can handle noise in the question and do multi-hop reasoning in knowledge graph simultaneously. Zheng et al. [19] used a large number of binary templates rather than semantic parsers to query knowledge graph with natural language. A low-cost technique that can generate a large number of templates automatically is also proposed. Our work is different from above works. Firstly, there are some differences between the structure of MathGraph and existing knowledge graphs (e.g. Freebase and NELL [2]). Secondly, to solve a math exercise usually requires multi-step mathematical derivation, and the derivation procedures need to be output as the problem-solving process. Thirdly, derivation and calculation should be performed simultaneously when solving an exercise to retrieve the answer.

2.2 Automated Solving Mathematical Problems

Automated solving mathematical problems has been studied over years. But they only focused on easy problems, e.g., mathematical problems in primary schools.

Kojiri et al. [10] constructed a mechanism called solution network to automatically generate the answers for mathematical exercises. The solution network is represented as a tree to describe inclusive relations of exercises.

Tomas et al. [16] proposed a framework of Constraint Logic Programming to automatically generate and solve mathematical exercises. This paper proposed to concentrate on the solving procedures rather than many simple exercise templates so that the generation and explanation of these exercises is easy.

Ganesalingam et al. [6] proposed a method that solves elementary mathematical problems using logical derivation and shows solutions which are made difficult to distinguish from human's writing.

However, these works all have their own limits. For example, some can solve problems only involving elementary math (e.g. set theory, basic algebraic oper-

ation) and have no deeper theorems; some only support very limited logical derivation. Thus, in this paper, we decide to use a knowledge graph to represent as many mathematical entities and logical relationships as possible.

3 Preliminaries

In this section, we describe the entities that may appear in MathGraph, including mathematical objects and instances, operations, and constraints.

Mathematical Object and Instance. A mathematical object is an abstract object which has a definition, some properties, and can be taken as the target of some operations or derivation. Note that a mathematical object can be defined in terms of other objects. A concrete object that satisfies the definition of the mathematical object is called an instance.

For example, `Complex Number` can be considered as a mathematical object:

- *Definition:* A complex number is a number that can be in the form $a + bi$, where a and b are both real numbers and i is the imaginary unit which satisfies $i^2 = -1$.
- *Property example:* Imaginary part is a property of a complex number. The imaginary part of a complex number $a + bi$ is b.
- *Operation example:* $(a_1 + b_1 i) \cdot (a_2 + b_2 i) = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i$
- *Derivation example:* If $(a_1 + b_1 i)$ and $(a_2 + b_2 i)$ are conjugated to each other, then $a_1 = a_2$ and $b_1 + b_2 = 0$.

And $2 + 3i$ and $(i + 1)(i - 3)$ are instances of `Complex Number`.

Different mathematical objects should be described as different structures in MathGraph. Thus, in MathGraph, a mathematical object is represented with a tuple of **key properties** $(p_1, p_2, \cdots, p_n)$. The key properties of a mathematical object are those properties that together can form and describe all the information of an instance of the object. Table 1 shows examples of key properties of some mathematical objects. Two instances of a mathematical object is equivalent if and only if all the key properties are equivalent.

In a mathematical exercise, instances can be categorized into certain instances and uncertain instances depending on whether it contains some uncertain values as its key properties. An instance is a *certain instance* if all key properties are certain; *uncertain instance* otherwise. For example, a real number 2.3 and a function $f(x) = x + sin(x)$ are certain; a complex number $3 + ai$ (where $a \in \mathbb{R}$) and a random triangle $\triangle ABC$ are uncertain.

Operation. Generally, an operation is an action or procedure which, given one or more mathematical objects as input (known as operands), produces a new object. Simple examples include addition, subtraction, multiplication, division, and exponentiation. In addition, other procedures such as calculating the real part of a complex number, the derivative of a function, and the area of a triangle can also be considered as operations.

Table 1. Examples of key properties of different mathematical objects

Mathematical object	Example instance	Key properties
Complex number	$ai + b$	(a, b)
Elementary function	$f(x) = \langle$an algebraic expression about $x\rangle$	$\langle$the algebraic expression$\rangle$
Triangle	$\triangle ABC$	$(a, b, c, \angle A, \angle B, \angle C)$
Line	$Ax + By + C = 0$	(A, B, C)
Ellipse	$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$	(a, b)

Constraint. A constraint is a description or condition about one or more instances, at least one of which is an uncertain instance. There are four types of constraints: *descriptive constraints* (e.g. complex numbers x and y are conjugated), *equality constraints* (e.g. $a + 2 = b$), *inequality constraints* (e.g. $a^2 \leq 5$), and *set constraints* (e.g. $a \in \mathbb{N}$).

Most descriptive constraints cannot be applied directly to solve the exercise, but can be converted into other three types of constraints using some definitions or theorems. For example, if an exercise says "$a + 3i$ and $7 - bi$ are a conjugate pair", by the definition of conjugate complex, we can know that $a = 7$ and $3 + (-b) = 0$ by derivation.

4 The Structure of MathGraph

MathGraph is a directed graph $\mathcal{G} = \langle V, E\rangle$, in which each node $v \in V$ denotes a mathematical object, an operation or a constraint, and each edge $e \in E$ is the relation of two nodes.

4.1 Nodes

In general, nodes are categorized into three different types: **object nodes**, **operation nodes** and **constraint nodes**.

Object Nodes. An object node $v_o = (t, P, C)$ represents a mathematical object, where t denotes an instance template of this mathematical object; $P = (P_1, P_2, \cdots, P_n)$ is a tuple indicating key properties of the mathematical object; and C is a set of constraints that, according to the definition or some theorems, must be satisfied by this mathematical object. Table 2 shows an example of "triangle" as an object node. We can see that properties and theorems of triangles are included in the constraint set.

Operation Nodes. An operation node $v_p = (X_1, X_2, \cdots, X_k, Y, f)$ represents a k-ary operation, where $X_i (i = 1, 2, \cdots, k)$ and Y are object nodes representing the domain of the i^{th} operand x_i and the result of the operation y respectively, and f is a function that implements the operation and can be finished by a series

Table 2. An example of object node: triangle

Mathematical object	Triangle
Instance template	$\triangle ABC$
Key properties	(a, b, c, A, B, C)
Constraint set	$\{a, b, c > 0,$ $0 < A, B, C < \pi,$ $A + B + C = \pi,$ $a + b > c, a + c > b, b + c > a,$ $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C},$ $a^2 = b^2 + c^2 - 2bc \sin A,$ $b^2 = a^2 + c^2 - 2ac \sin B,$ $c^2 = a^2 + b^2 - 2ab \sin C\}$

of symbolic execution [1,3,9] process using a symbolic execution library (e.g. SymPy [13], Mathematica [8]) even if some operands are uncertain instances.

For example, *getting the modulus of a complex number* is an unary operation where $X_1 = \langle \texttt{Complex Number} \rangle$, $Y = \langle \texttt{Real Number} \rangle$, and f can be implemented by the following symbolic execution process: (1) Get the real part of x_1; (2) Get the imaginary part of x_1; (3) Return the squared root of the sum of (1) squared and (2) squared.

Constraint Nodes. A constraint node $v_c = (d, X_1, X_2, \cdots, X_k, f)$ represents a descriptive constraints of k instances, where d is the description of the constraint, $X_i (i = 1, 2, \cdots, k)$ are object nodes representing the domain of each involving instance, and f is a function which maps this descriptive constraint into several equality constraints, inequality constraints and set constraints.

For example, a constraint node represents that x_1 and x_2 are a conjugate pair, where $X_1 = X_2 = \langle \texttt{Complex Number} \rangle$, and f can be implemented by the following process: (1) Get the real part of x_1 as a_1; (2) Get the real part of x_2 as a_2; (3) Get the imaginary part of x_1 as b_1; (4) Get the imaginary part of x_2 as b_2; (5) Return two equality constraints: $a_1 = a_2$ and $b_1 + b_2 = 0$.

4.2 Edges

There are two types of edges in MathGraph: the DERIVE edges and the FLOW edges.

The DERIVE Edge. For two object nodes X and Y, there may be a DERIVE edge $e_{\text{DERIVE}} = (X, Y, f)$ to indicate a general-special relationship between them, such as *Triangle* and *Isosceles Triangle*. If $X \xrightarrow{\text{DERIVE}} Y$, an instance of X can be reassigned as an instance of Y if certain conditions are met. These conditions are encapsulated into a function $f : X \to \{\text{False}, \text{True}\}$: if these conditions are

met, the function f will return True and reassign the instance from X to Y; otherwise it will simply return False.

For example, there is a DERIVE edge from object node `Triangle` to `Isosceles Triangle`, where the function f can be implemented as: (1) if the values of key properties or a constraint shows that two angles or lengths of two edges of the origin instance are equal, return an instance of *Isosceles Triangle* with the same key properties; (2) return False otherwise.

When solving an exercise, reassigning an instance to a more specific object node will bring more constraints of this object and help find the answer. For example, for a rhombus $ABCD$, if we know that $\angle A = 90°$, we can infer, by the DERIVE edge from object node `Rhombus` to `Square`, that $ABCD$ is a square and has constraints that $\angle A = \angle B = \angle C = \angle D = 90°$.

The FLOW Edge. A FLOW edge $e_{\text{FLOW}} = (X, Y)$ indicates the flow direction of instances during the exercise solving process, which may only exist from an object node to an operation node, from an operation node to an object node, or from an object node to a constraint node.

The FLOW edges between object nodes and operation nodes represent the process of passing instances as parameters before the operation and the process of returning a new instance after it. For example, in Fig. 2, the two FLOW edges pointing to the operation node "addition" indicate that this operation takes two instances of complex number as its input values, and the edge leading from this operation node indicates that it returns a new instance of complex numbers.

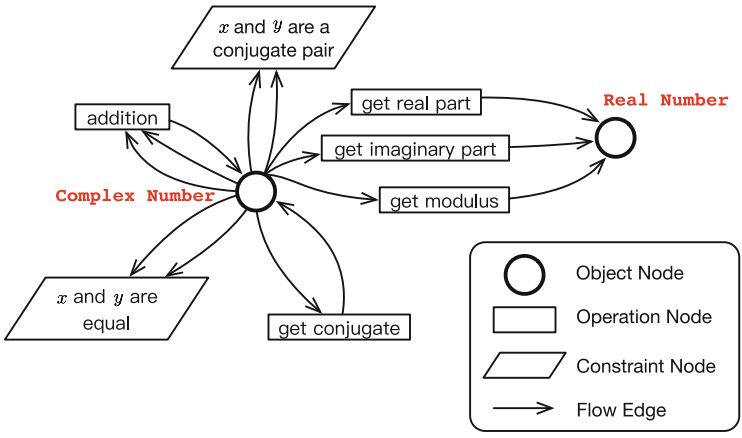


Fig. 2. Example of the FLOW edges

The FLOW edges from object nodes to constraint nodes also represent the process of passing parameters of the constraints. For example, in Fig. 2, the two FLOW edges pointing to the constraint node "x and y are a conjugate pair" indicates that this constraint takes two complex number as its input. Note that

constraints nodes only convert descriptive constraints into other types of constraints and generate no instances, so there are no FLOW edges from a constraint node to an object node.

In summary, MathGraph is a well-structured graph supporting different mathematical objects, operations and constraints. Next, we will discuss how to solve mathematical exercises using it.

5 Solving Mathematical Exercises with MathGraph

In this section, we propose a framework to solve a mathematical exercise using MathGraph. First, we use a semantic parser mapping exercise text to the instances, operations and constraints respectively. Then, we solve the constraints and update uncertain instances. Finally, we return the answer of this exercise.

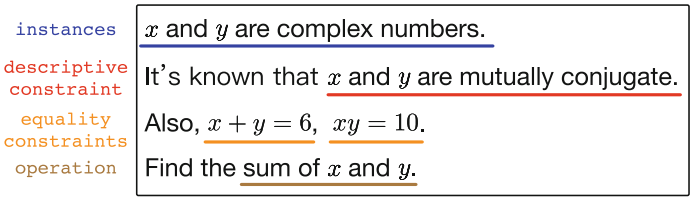


Fig. 3. Example of parsing the text into nodes in MathGraph

5.1 Mapping Text in MathGraph

Considering the limited information and expression in the mathematical exercises, we can easily use a rule-based semantic parser to parse the exercise text and then map them to corresponding nodes in MathGraph.

The rule-based semantic parser uses a set of rules to parse every sentence of the exercise and recognize the logical relationship in the text. For example, "Let x and y be complex numbers" will be parsed as declaration of two uncertain instances; "Find the coordinates of the conjugate complex of $(i+1)(i-1)$" will be parsed as a declaration of a certain instance and two operations.

Mapping Instances. With the semantic parser, every instance generated from the exercise should have already mapped into the corresponding object node. That is, a set of instances $\mathcal{I} = \{(x_1, X_1), \cdots, (x_k, X_k)\}$ is generated by parsing the text of the exercise, where x_i denotes the instance and X_i denotes the corresponding object node.

Instances are classified as certain instances or uncertain instances depending on if the exercise provide certain values or expressions of them. For uncertain instances generated from text, key properties with unknown value should be

generated as instances, since they may be used in the operations and constraints of this exercise. For example, for the exercise shown in Fig. 3, x and y are both uncertain instances of object node `Complex Number`. Therefore, we need to generate a_x, b_x, a_y and b_y as four uncertain instances of object node `Real Number`, where a_x and b_x stand for the two key properties of x, and a_y and b_y stand for the key properties of y.

Mapping Operations. The semantic parser can also parse out the a set of operations from the text. Every operation $(o, (x_1, X_1), \cdots, (x_n, X_n))$ in it will be aligned to the corresponding operation node in MathGraph o with its operands, trigger the function in the operation node, and then finally generate a new instance as the output of the operation.

Mapping Constraints. Similar to mapping operations, for every descriptive constraint $(c, (x_1, X_1), \cdots, (x_n, X_n))$ in the exercise, the semantic parser can map it to the corresponding constraint node c with some involving instances, trigger the function in the node, and convert it to several equality/inequality/set constraints.

Also, note that when an uncertain instance is generated, some constraints may also be generated according to the constraint set of the corresponding object node. After that, we gather all the constraints in the exercise as a set for further using.

Algorithm 1 shows the process of mapping text of the exercise.

5.2 Solving Uncertain Instances and Constraints

After parsing all the instances and operations in the exercise, the answer of the exercise should already be generated as an instance (from the text or by an operation). If this instance is a certain instance, we can directly return the value of this instance as the answer; otherwise, we must deal with these uncertain instances and solve the constraints in the exercise to update their values and finally retrieve the answer of the exercise.

Reassign Uncertain Instances. First, we need to check every uncertain instance if it can be reassigned to a more specific object node in MathGraph by a DERIVE edge. For an uncertain instance i that is assigned to an object node v_o, we check every outgoing DERIVE edge of v_o, and if the function of an edge e returns true, then we reassign i to the object node that e points to and add all the constraints in this node to the constraint set. Algorithm 2 shows the pseudo code of this process.

For example, if we have an uncertain instance $\triangle ABC$, and there is a constraint $\angle B = \angle C$ in the constraint set, then the DERIVE edge from `Triangle` to `Isosceles Triangle` should return true. So the instance should be reassigned to `Isosceles Triangle`, and a new constraint $AB = AC$ should be added to the constraint set.

Algorithm 1. MAPPINGTEXT$(t, \mathcal{G})$

Input: t : text of the exercise;
$\mathcal{G}$: MathGraph

Output: $\mathcal{I}_{\text{certain}}$: a set of certain instances;
$\mathcal{I}_{\text{uncertain}}$: a set of uncertain instances;
$\mathcal{C}$: a set of constraints;
$\mathcal{S}_{\text{dependency}}$: a set denoting dependencies of uncertain instances;

begin
  Initialize P as a semantic parser;
  $\mathcal{I}_{\text{certain}}, \mathcal{I}_{\text{uncertain}} \leftarrow P$.MAPPINGINSTANCES$(t, \mathcal{G})$;
  $\mathcal{O} \leftarrow P$.MAPPINGOPERATIONS$(t, \mathcal{G})$;
  $\mathcal{C} \leftarrow P$.MAPPINGCONSTRAINTS(t, G);
  Let $\mathcal{S}_{\text{dependency}}$ be an empty set;
  for each $(x, X) \in \mathcal{I}_{uncertain}$ **do**
    for each key property$(p, X_p) \in x.keyProperties$ **do**
      if p is an uncertain instance **then**
        $\mathcal{I}_{\text{uncertain}} \leftarrow \mathcal{I}_{\text{uncertain}} \cup \{(p, X_p)\}$;
        $\mathcal{S}_{\text{dependency}} \leftarrow \mathcal{S}_{\text{dependency}} \cup \{(p, x)\}$;
  for each $(o, (x_1, X_1), \cdots, (x_k, X_k)) \in \mathcal{O}$ **do**
    $(y, Y) = o.f(x_1, \cdots, x_k)$;
    if y is a certain instance **then**
      $\mathcal{I}_{\text{certain}} \leftarrow \mathcal{I}_{\text{uncertain}} \cup \{(y, Y)\}$;
    else
      $\mathcal{I}_{\text{uncertain}} \leftarrow \mathcal{I}_{\text{uncertain}} \cup \{(y, Y)\}$;
      $C \leftarrow S \cup y.ConstraintSet$;
      for $i = 1$ **to** k **do**
        if x_i is an uncertain instance **then**
          $\mathcal{S}_{\text{dependency}} \leftarrow \mathcal{S}_{\text{dependency}} \cup \{(x_i, y)\}$;
  for each $(c, (x_1, X_1), \cdots, (x_k, X_k)) \in \mathcal{C}$ **do**
    if c is a descriptive constraint **then**
      $c \leftarrow c.f(x_1, \cdots, x_k)$;
  return $\mathcal{I}_{certain}, \mathcal{I}_{uncertain}, \mathcal{C}, \mathcal{S}_{dependency}$

Algorithm 2. ReassignUncertainInstances($\mathcal{G}, \mathcal{I}_{\text{uncertain}}, \mathcal{C}$)

Input: $\mathcal{G}$: MathGraph;
$\mathcal{I}_{\text{uncertain}}$: the set of uncertain instances;
$\mathcal{C}$: the constraint set;

```
1 begin
2     for each instance (x, X) ∈ I_uncertain do
3         for each DERIVE edge (X_e, Y_e, f_e) ∈ G do
4             if X_e == X and f_e(x) == True then
5                 C ← C ∪ Y_e.ConstraintSet;
6                 update (x, X) as (x, Y);
```

Algorithm 3. OrganizeUncertainInstances($\mathcal{I}_{\text{uncertain}}, \mathcal{S}_{\text{dependency}}$)

Input: $\mathcal{I}_{\text{uncertain}}$: a set of uncertain instances;
$\mathcal{S}_{\text{dependency}}$: the set denoting dependencies of uncertain instances;
Output: $\mathcal{G}_I$: the graph to organize the uncertain instances;
$\mathcal{S}_I$: a set denoting all instances in $\mathcal{G}_I$ without incoming edges;

```
1 begin
2     Let G_I⟨V_I, E_I⟩ be an empty graph;
3     for (x, y) ∈ S_dependency do
4         V_I ← V_I ∪ {x, y};
5         E_I ← E_I ∪ {(x, y)};
6     S_I ← {v | v ∈ V_I ∧ ∀u ∈ V_I, (u, v) ∉ E_I};
7     return G_I, S_I
```

Organizing Uncertain Instances. Note that for two uncertain instances α and β, there may be a dependency relationship between them, which is caused due to either α is one of the input of an operation node and β is the output or α is one of the key properties of β.

Thus, we use a graph $\mathcal{G}_I = \langle V_I, E_I \rangle$ to describe dependency of all the uncertain instances, where $v \in V_I$ is a node representing an uncertain instance and $e \in E_I$ is a directed edge representing a dependency relationship of two nodes. Note that $\mathcal{G}_I$ is always a DAG, since there will be no dependency loop in it.

Let $\mathcal{S}_I = \{v | v \in V_I \wedge \forall u \in V_I, (u, v) \notin E_I\}$ denote the set containing all node without any incoming edges in $\mathcal{G}_I$. It is obvious that if all nodes in $\mathcal{S}_I$ can turn into certain instances, the instance corresponding to the answer can be derived to a certain instance. Algorithm 3 demonstrates this process.

For example, Fig. 4 shows $\mathcal{G}_I$ of the exercise in Fig. 3, where x and y depend on their respective key properties, and $z = x + y$ depends on its two operands. In this context, $\mathcal{S}_I = \{a_x, b_x, a_y, b_y\}$ and the instance corresponding to the answer is z.

Algorithm 4. ProcessConstraints($\mathcal{C}, \mathcal{G}_I, \mathcal{S}_I$)

Input: $\mathcal{C}$: the constraint set;
$\mathcal{G}_I$: the graph for dependency of uncertain instances;
$\mathcal{S}_I$: the set denoting all instances in $\mathcal{G}_I$ without incoming edges;

begin
  for each $(c, (x_1, X_1), \cdots, (x_k, X_k)) \in \mathcal{C}$ **do**
    for $i = 1$ **to** k **do**
      if $x_i \notin \mathcal{S}_I$ **then**
        Replace (x_i, X_i) with its key properties $(p_1, P_1), \cdots, (p_n, P_n)$;
  SolveConstraints($\mathcal{S}_{\text{constraint}}, \mathcal{S}_I$);

Organizing and Solving Constraints. After the last step, we now have a set of constraints. First, we need to make sure every variable in every constraint is in $\mathcal{S}_I$. If not, this constraint needs to be rewritten by using its key properties as the variable. For example, for the exercise in Fig. 3, the set of the constraint is $\{x + y = 6, xy = 10, a_x = a_y, b_x + b_y = 0\}$. Since $x, y \notin \mathcal{S}_I$, the first two constraints will be rewritten as $a_x{+}b_x i{+}a_y{+}b_y i = 6$ and $(a_x{+}b_x i)(a_y{+}b_y i) = 10$.

Now the constraint set includes and formalizes all the constraints in the exercise. So we can apply methods of a symbolic execution library [8,13] or some approximation algorithms [5,15] to solve these equations and/or inequalities. Finally, we will get the value (or range of value) of every instance in $\mathcal{S}_I$. Algorithm 4 shows this process.

Updating Uncertain Instances and Retrieving the Answer. After solving all the constraints in the exercise, we need to update the value of the rest instances in $\mathcal{G}_I$. Since we now know the value of instances in $\mathcal{S}_i$, we can traverse every instance in $\mathcal{G}_I$ in the topological sorting order and update their values in turn. Finally, we return the value of the instance corresponding to the answer. Algorithm 5 shows the complete process of using MathGraph to solve exercise.

Algorithm 5. SolvingExercise($t, \mathcal{G}$)

Input: t : text of the exercise;
$\mathcal{G}$: MathGraph
Output: answer of the exercise

begin
  $\mathcal{I}_{\text{certain}}, \mathcal{I}_{\text{uncertain}}, \mathcal{C}, \mathcal{S}_{\text{dependency}} \leftarrow$ MappingText($t, \mathcal{G}$);
  Mark the instance corresponding to the answer as x_{ans};
  ReassignUncertainInstances($\mathcal{G}, \mathcal{I}_{\text{uncertain}}, \mathcal{C}$);
  $\mathcal{G}_I, \mathcal{S}_I \leftarrow$ OrganizeUncertainInstances($\mathcal{I}_{\text{uncertain}}, \mathcal{S}_{\text{dependency}}$);
  ProcessConstraints($C, \mathcal{S}_I$);
  Update the value of every node in $\mathcal{G}_I$ in the topological sorting order;
  return value of x_{ans};

6 Experiments

In this section, we conduct extensive experiments on real mathematical datasets to evaluate the performance of our method.

6.1 Datasets and Experiment Setting

We collect four real-world datasets of mathematical exercises of Chinese high schools, namely `Complex`, `Triangle`, `Conic` and `Solid`. The exercises are stored in plain text, and the mathematical expressions are stored in the LaTeX format.

- `Complex`: This dataset contains 1526 mathematical exercises related to calculation and derivation of complex numbers, including basic algebraic operation, the modulus and the conjugate of a complex number, Argand plane, polar representation, etc.
- `Triangle`: This dataset contains 782 mathematical exercises related to solving triangles (using Law of Sines and Law of Cosines), which includes finding missing sides and angles, perimeter, area, radius of the circumscribed circle, etc.
- `Conic`: This dataset contains 1196 exercises related to Conic sections, including calculation and derivation on ellipse, hyperbola and parabola.
- `Solid`: This dataset contains 653 exercises related to solid geometry, which involves a variety of geometries in three-dimension Euclidean space, including pyramids, prisms, etc.

Exercises in the four datasets are categorized into three levels (i.e. easy, medium, and hard) based on the difficulty (which is classified according to the accuracy of many high school students). Table 3 shows the number of exercises with different difficulty levels in the datasets.

Table 3. Summary of exercises in the datasets

	Easy	Medium	Hard	Total
`Complex`	685	634	207	1526
`Triangle`	179	470	133	782
`Conic`	486	602	108	1196
`Solid`	217	336	100	653

In the experiments, we use Neo4j [14] as the graph database platform to build and index MathGraph. For the datasets, we build the knowledge graph manually involving only the instances, operations and constraints that may exist in these exercises. The experiments are implemented in Python 3.7. Sympy is used to do the work of symbolic execution. All the experiments are conducted in a machine with 2.40 GHz Intel Xeon CPU E52630, 48 GB RAM, running Ubuntu 14.04.

6.2 Evaluation and Discussion

We implement a rule-based baseline method as the following procedures:

1. We still use a rule-based semantic parser to parse the text and extract the information.
2. A large quantity of rules are written in advance to match different situations of exercises. Every rule represents an exercise type and has a built-in solving process only for this exercise type.
3. If the exercise matches a rule, then we apply the solving process of the rule and return the answer.

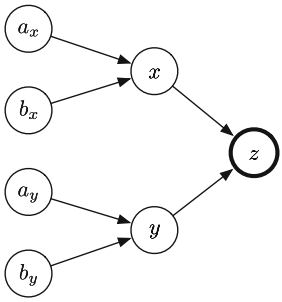


Fig. 4. $\mathcal{G}_I$: a DAG to organize the uncertain instances

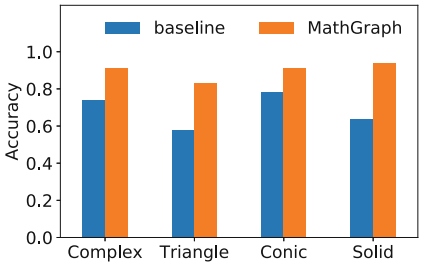
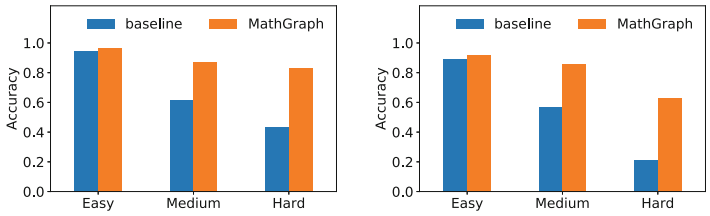


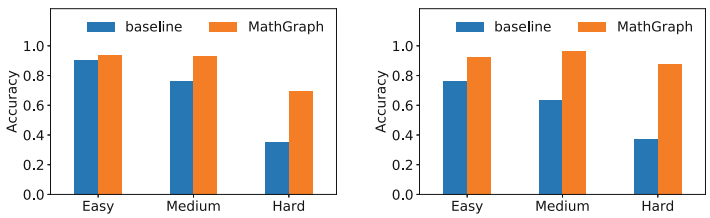
Fig. 5. Overall accuracy on four datasets

Figure 5 shows the exercise-solving accuracy on four datasets. We can see that in every dataset, our method achieves higher accuracy than baseline, e.g., 20% higher accuracy. This result shows the effectiveness of solving problems using MathGraph.

Figure 6 demonstrates the exercise-solving accuracy on different difficulty level. From the experiment result, we have the following observations. Firstly, as



(a) Dataset `Complex` (b) Dataset `Triangle`



(c) Dataset `Conic` (d) Dataset `Solid`

Fig. 6. Accuracy on different difficulty levels

the difficulty of the exercises increases, the accuracy of both methods decreases. Secondly, for easy exercises, the baseline and our method have similar performance; but for medium and hard exercises, MathGraph significantly outperforms the baseline, because our method can use the knowledge graph to do mathematical derivation.

The rule-based baseline considers the exercise as a whole and solving it according to the logic specified by a rule. This means that this method relies on a large amount of rules, and the more complex the exercise is, the more rules and the higher difficult it needs to write. Therefore, this method has a poor performance in hard exercises. However, our method extracts the mathematical objects, calculations, and constraints from these rules and models them into a graph, so it can be used for multi-step calculation and derivation.

7 Conclusion

In this paper, we proposed MathGraph, a knowledge graph for automatically solving mathematical exercises. MathGraph is specially designed to represent different mathematical objects, operations and constraints. Given an exercise, we can use the proposed method to solve it with the help of MathGraph and a prebuilt semantic parser. Experimental study on four real-world datasets demonstrates the accuracy of our method.

Acknowledgements. This work was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61521002, 61661166012), and TAL education.

References

1. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50 (2018)
2. Carlson, A., Betteridge, J., Kisiel, B., Settles, B., Hruschka Jr., E.R., Mitchell, T.M.: Toward an architecture for never-ending language learning. In: Proceedings of the Twenty-Fourth Conference on Artificial Intelligence (AAAI 2010), vol. 5, p. 3. Atlanta (2010)
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM (1977)
4. Dongo, I., Cardinale, Y., Chbeir, R.: RDF-F: RDF datatype inferring framework. Data Sci. Eng. **3**(2), 115–135 (2018)
5. Fletcher, R., Leyffer, S.: Filter-type algorithms for solving systems of algebraic equations and inequalities. In: Di Pillo, G., Murli, A. (eds.) High Performance Algorithms and Software for Nonlinear Optimization, pp. 265–284. Springer, Heidelberg (2003). https://doi.org/10.1007/978-1-4613-0241-4_12
6. Ganesalingam, M., Gowers, W.T.: A fully automatic theorem prover with human-style output. J. Autom. Reason. **58**(2), 253–291 (2017)
7. Guu, K., Miller, J., Liang, P.: Traversing knowledge graphs in vector space. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, 17–21 September 2015, pp. 318–327 (2015)
8. Mathematica, Version 11.3. Wolfram Research, Inc., Champaign (2018)
9. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
10. Kojiri, T., Hosono, S., Watanabe, T.: Automatic generation of answers using solution network for mathematical exercises. In: Khosla, R., Howlett, R.J., Jain, L.C. (eds.) KES 2005. LNCS (LNAI), vol. 3683, pp. 1303–1309. Springer, Heidelberg (2005). https://doi.org/10.1007/11553939_181
11. Li, K., Li, G.: Approximate query processing: what is new and where to go? Data Sci. Eng. **3**(4), 379–397 (2018)
12. Lin, P., Song, Q., Wu, Y.: Fact checking in knowledge graphs with ontological subgraph patterns. Data Sci. Eng. **3**(4), 341–358 (2018)
13. Meurer, A., et al.: SymPy: symbolic computing in Python. PeerJ Comput. Sci. **3**, e103 (2017)
14. Neo4j, Inc.: Neo4j, Version 1.1.12. https://neo4j.com/
15. Polyak, B.T.: Gradient methods for solving equations and inequalities. USSR Comput. Math. Math. Phys. **4**(6), 17–32 (1964)
16. Tomás, A.P., Leal, J.P.: A CLP-based tool for computer aided generation and solving of maths exercises. In: Dahl, V., Wadler, P. (eds.) PADL 2003. LNCS, vol. 2562, pp. 223–240. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36388-2_16
17. Toutanova, K., Lin, V., Yih, W.t., Poon, H., Quirk, C.: Compositional learning of embeddings for relation paths in knowledge base and text. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, pp. 1434–1444 (2016)

18. Zhang, Y., Dai, H., Kozareva, Z., Smola, A.J., Song, L.: Variational reasoning for question answering with knowledge graph. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018), pp. 6069–6076 (2018)
19. Zheng, W., Yu, J.X., Zou, L., Cheng, H.: Question answering over knowledge graphs: question understanding via template decomposition. Proc. VLDB Endow. **11**(11), 1373–1386 (2018)