JUSTC

Article

http://justc.ustc.edu.cn

Received: October 11, 2023; Accepted: January 30, 2024

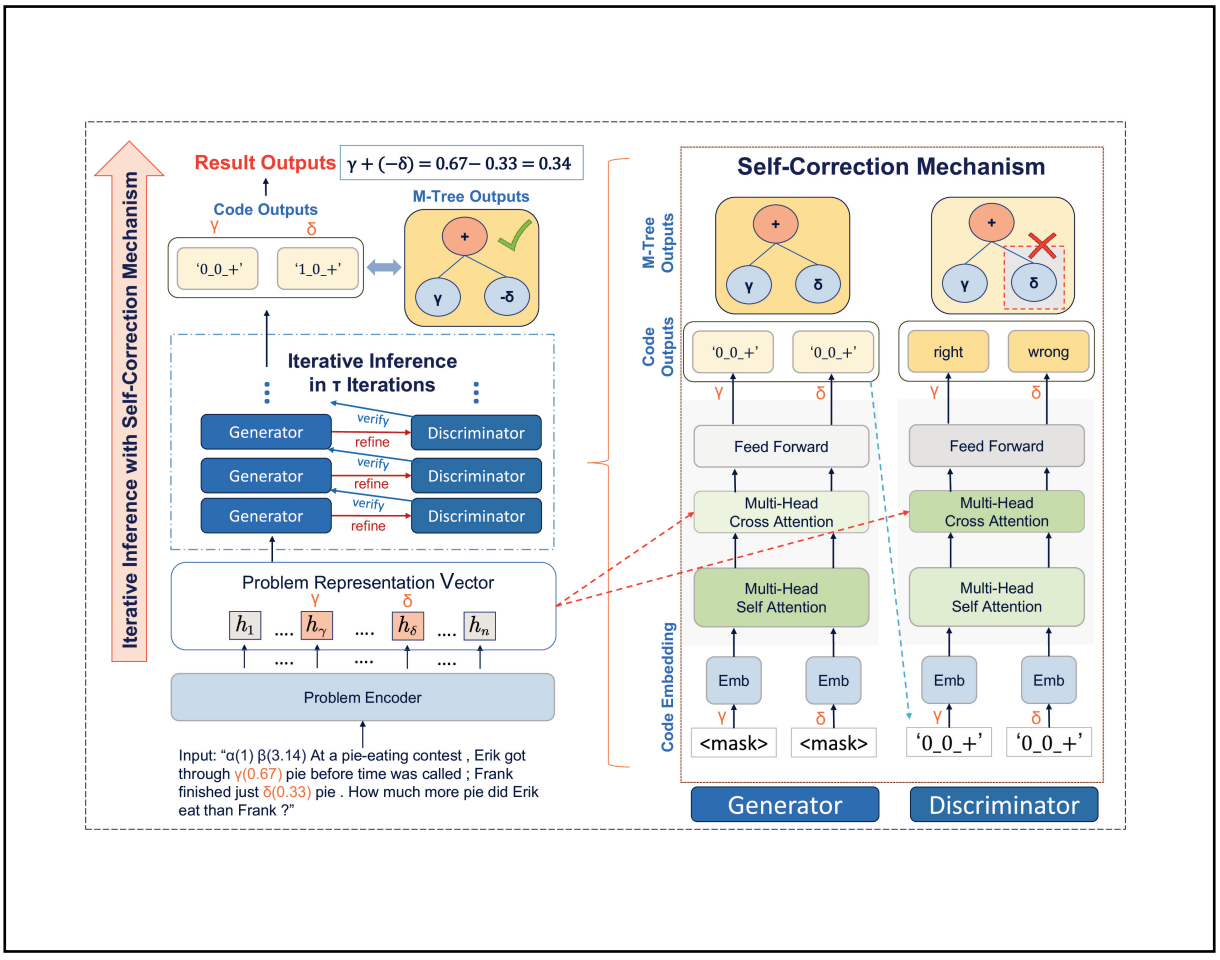
A unified M-tree self-correction solver for math word problems

Zhiyuan Ma[1,2], Jiayu Liu[1,2], and Zhenya Huang[1,2] ✉

[1]*School of Computer Science and Technology, School of Artificial Intelligence and Data Science, School of Data Science, University of Science of Technology of China, Hefei 230027, China;*
[2]*State Key Laboratory of Cognitive Intelligence, Hefei 230088, China*

✉Correspondence: Zhenya Huang, E-mail: huangzhy@ustc.edu.cn

Graphical abstract



The overall pipeline of our proposed UTSC-Solver.

Public summary

- We propose a novel UTSC-Solver for solving math word problems, which uses an iterative inference to generate mathematical expressions in a non-autoregressive framework.
- The key contribution is iterative inference with self-correction mechanism, which refines mathematical expressions by alternating between a generator and a discriminator, improving the accuracy and interpretability of mathematical reasoning.

Citation: Ma Z Y, Liu J Y, Huang Z Y. A unified M-tree self-correction solver for math word problems. *JUSTC*, **2025**, 55(7): 0703. CSTR: 32290.14.JUSTC-2023-0145. DOI: 10.52396/JUSTC-2023-0145

JUSTC

Article

http://justc.ustc.edu.cn

Received: October 11, 2023; Accepted: January 30, 2024

A unified M-tree self-correction solver for math word problems

Zhiyuan Ma[1,2], Jiayu Liu[1,2], and Zhenya Huang[1,2] ✉

[1]*School of Computer Science and Technology, School of Artificial Intelligence and Data Science, School of Data Science, University of Science of Technology of China, Hefei 230027, China;*
[2]*State Key Laboratory of Cognitive Intelligence, Hefei 230088, China*

✉Correspondence: Zhenya Huang, E-mail: huangzhy@ustc.edu.cn

Cite This: *JUSTC*, **2025**, 55(7): 0703 (10pp)

Read Online

Abstract: Automatically answer math word problems is a challenging task in artificial intelligence. Previous solvers constructed mathematical expressions in sequence or binary tree. However, these approaches may suffer from the following issues: Models relying on such structures exhibit fixed-order reasoning (e.g., left-to-right), limiting flexibility and increasing error susceptibility; prior models rely on autoregressive reasoning in a single pass, accumulating minor errors (e.g., incorrect math symbols) during generation, resulting in reduced accuracy. To address the above issues, we emulate the human "check and modify" process in reasoning and propose a unified M-tree self-correction solver (UTSC-Solver) by iterative inference with self-correction mechanism. First, we use an iterative, non-autoregressive process for generating mathematical expressions, free from fixed generation orders to handle complex and diverse problems. Additionally, we design a self-correction mechanism based on alternating execution between a generator and a discriminator. This module iteratively detects and rectifies errors in generated expressions, leveraging previous iteration information for subsequent generation guidance. Experimental results show that our UTSC-Solver outperforms traditional models in accuracy on two popular datasets, while it improves the interpretability of mathematical reasoning.

Keywords: mathematical reasoning; non-autoregressive generation; math word problems

CLC number: TP391 **Document code:** A

1 Introduction

With the impressive advancement of artificial intelligence (AI), it becomes increasingly crucial for AI to solve fundamental tasks[1,2], including mathematical reasoning. Mathematical reasoning plays a pivotal role in AI's capabilities and encompasses tasks like solving math word problems (MWPs), which continue to present a challenge. MWPs are deceptively simple yet intricate elementary-level applied problems[3], which requires a system to understand the mathematical meaning of the problem, construct appropriate expressions, and perform accurate calculations. As depicted in Fig. 1, a standard MWP[3] provides a brief narrative describing mathematical relationships between quantities, such as "4 roses", and poses a query about an unknown quantity, like "total earned money". MWP solvers are expected to automatically construct a mathematical expression for calculating the final answer, which is considered an important stride toward general artificial intelligence. In addition, MWP solvers can be applied in various domains such as intelligent education[4], decision-making[5], and engineering applications[6]. Therefore, MWP solvers are meaningful for both general AI and practical applications, and they have attracted extensive research interest[7–9].

In recent years, various effective MWP solvers have been proposed[10–12]. Based on different forms for organizing the expressions, they can be roughly categorized into the following four types: (i) Sequence-based method like DNS[13] and GSF[14] are based on an encoder-decoder architecture[15] and construct mathematical expression sequences sequentially from left to right (e.g., "$30+7\times(9-4)$" in Fig. 1). (ii) Binary tree-based method like GTS[16] and HMS[17] use a goal-driven decoder to construct the binary tree of mathematical expression from top to bottom, which improves the reasoning accuracy by structure information. (iii) DAG-based method like Seq2DAG[18] extend the tree structure to encompass directed acyclic graphs, allowing for the representation of multiple equations to solve multi-equation math problems. (iv) M-tree based method like SUMC-Solver[19] considers that there exist various equivalent expressions for the same problem, thereby adopting a unified M-tree structure to unify them in a singular, standardized structure. Specifically, as illustrated in Fig. 1, the M-tree structure, through its multi-branch (M-ary) structure and composite operators, unifies the above three different binary tree structures. This significantly reduces the uncertainty in mathematical reasoning (details will be presented in Section 3.1).

Despite the advancements in reasoning capabilities of MWP solvers, they still struggle with notable challenges in achieving complex mathematical reasoning[20]. A primary limitation originates from the widespread use of the autoregressive framework like DNS[13] and GTS[16]. This framework dictates a rigid generation sequence, either from left to right or by constructing binary trees from top to bottom. Such inflexibility hampers the model's adaptability, preventing it from

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

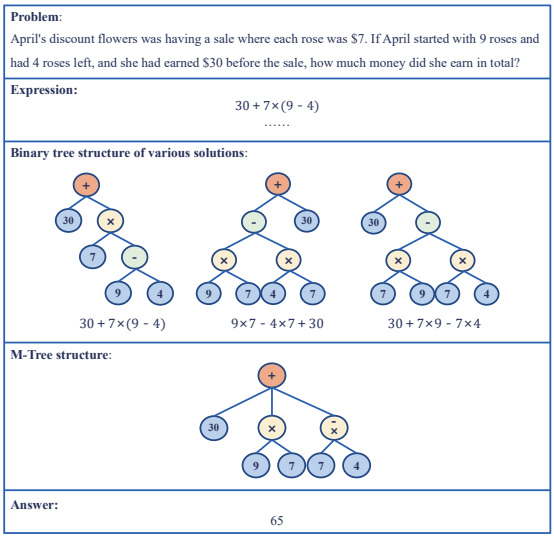


Fig. 1. A typical MWP with a problem, various solutions and final answer.

adjusting the generation order based on context reasoning. Also, the sequence and binary tree structure lead to multiple mathematical expression variants for a single problem, making it difficult to grasp the mapping from input to output.

Furthermore, current solvers primarily reason as one-pass systems, constructing mathematical expressions in a single sweep[21], and do not change. This design inherently lacks the capability to fix errors during the reasoning phase, leading to an accumulation of inaccuracies. The precision required in mathematical expression generation surpasses that in typical text generation. A minor error in a symbol or number can invalidate the entire mathematical expression, unlike text generation, where similar errors may not substantially affect the overall meaning. This contrast emphasizes the need for specialized techniques to address cumulative errors.

To tackle the challenges mentioned above, we take inspiration from the human reasoning process, where individuals continually write, check, and modify their work[22]. Emulating this approach, we introduce a unified M-tree self-correction solver (UTSC-Solver) for MWPs. This solver is designed for multiple iterative self-correction processes in a non-autoregressive manner. Unlike the fixed generation orders, our method can construct mathematical expressions in various directions through iterations, which enhances flexibility and addresses the cumulative error issue inherent in autoregressive solvers. Furthermore, to achieve more precise mathematical reasoning[21], we devise an iterative inference process featuring a self-correction mechanism, which refines mathematical expressions by alternating between the generator and the discriminator. In each iteration, we detect errors in the generated expressions using the discriminator, and then employ the generator to re-create them, facilitating explicit error detection and correction. By incorporating insights from previous iterations, it directs subsequent generation steps, guaranteeing the creation of progressively precise mathematical expressions. Through convergence analysis, we've discovered that this self-correction mechanism substantially enhances accuracy with only a few iterations, yielding commendable accuracy.

Our experimental results on widely recognized datasets, MAWPS[23] and Math23K[13], reveal that UTSC-Solver surpasses state-of-the-art traditional models in terms of answer accuracy. These results also underscore the efficacy of the self-correction mechanism in rectifying errors.

2 Related work

2.1 Math word problem

The journey of MWP solvers has evolved significantly over the decades, mirroring the broader shifts and advancements[24,25] in the field of AI and logical reasoning[26,27]. Tracing back to the 1960s, the initial stages of MWP solvers were dominated by rule-based methods[28,29]. These methods, while pioneering, were heavily reliant on manually formulated rules and heuristic approaches. The subsequent era saw the emergence of statistical machine learning methods[30] and semantic parsing techniques[31]. Despite their advancements, these non-neural network methods had their limitations, particularly in scalability and adaptability to new problem types.

With the emergence of deep learning, the scenario of MWP solvers witnessed a significant shift. Wang et al.[13], in 2017, pioneered this transformation with the introduction of DNS, a deep learning-based solver that utilized the power of RNNs. Following this, the field saw innovations like GTS by Xie et al.[16], which leveraged goal-driven binary tree structures, and Graph2Tree by Zhang et al.[32], which tapped into graph structures for enhanced problem comprehension. The integration of external knowledge, as seen in the KAS2T model by Wu et al.[33], further enriched the capabilities of MWP solvers. The recent trend of integrating large-scale pretrained models, such as BERT[34], has also been explored, with models like MWP-BERT by Liang et al.[35], aiming to refine numerical representation and reasoning.

The inherent variability in mathematical expressions, influenced by arithmetic laws such as commutativity, associativity, and distributivity[36], has long posed challenges in the domain. As shown in Fig. 1, we can calculate the answer following the rules of "$price \times count_1 - price \times count_2$" or "$count_1 \times price - count_2 \times price$", leading to different expressions and binary tree structures. Such variability can lead to multiple valid representations for the same mathematical concept, introducing ambiguities and uncertainties in the model's output space. This not only complicates the learning process but also makes error correction more challenging, as there isn't a single "correct" representation to target. To address this, the M-tree structure was introduced by Wang et al.[19] in their SUMC-Solver. The primary advantage of the M-tree is its ability to unify various expression variants into a singular, standardized structure. By doing so, it provides a unique framework and direction for corrections, significantly reducing uncertainties associated with multiple possible representations. In our work, we adopted the M-tree structure as the foundation of UTSC-Solver, reducing uncertainty and unifying various binary tree structure variants. Additionally, it provides a singular direction for the multiple potential correction strategies associated with our designed self-correction mechanism. This enables the generator and discriminator to operate with greater precision.

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

2.2 Text correction

Text correction has long been a pivotal area of research in natural language processing[37], aiming to rectify erroneous text sequences. Historically, the field was dominated by rule-based and statistical models[38,39], exemplified by Lee and Seneff's syntax tree-based statistical model[40]. However, the advent of sequence-to-sequence models ushered in a new era of advancements[41–43], with Chollampatt et al.[44] pioneering an end-to-end text correction approach. Despite their potential, these models are often criticized for their sluggish training and inference speeds.

Considering the challenges posed by autoregressive frameworks in mathematical reasoning, as discussed earlier, recent research has shifted towards non-autoregressive models for their efficiency in text generation tasks. Ge et al.[45] were pioneers in employing non-autoregressive models for text correction, introducing the MaskPredict network. This innovative approach masks potentially erroneous words in the input sequence and predicts accurate replacements, eliminating the need to generate the entire output sequence.

Mathematical reasoning is inherently complex and being highly sensitive to textual accuracy. Even minor text errors can disrupt the entire inference process. By utilizing non-autoregressive models, we can streamline the correction process and enhance the resilience of mathematical reasoning, ensuring that generated expressions are both efficient and precise.

3 Methodology

As depicted in Fig. 2, the UTSC-Solver has been designed to emulate the cognitive reasoning process of humans in iterative self-correction to solve MWPs[22]. This section will provide a detailed description of the methodology behind UTSC-Solver, including problem definition, model architecture, inference process, training methods, convergence analysis.

3.1 Preliminary

In this section, we will formally introduce the definition of the MWP task, as illustrated in Fig. 3. For each MWP, we define the input problem $P=\{p_1,p_2,\cdots,p_n\}$, like "Input: α β At a...?". It is the raw problem in datasets that have undergone preprocessing, which comprises a set of numbers $V_P=\{v_1,v_2,\cdots,v_m\}$, like "$\gamma$: 0.67, δ: 0.33". The normal output should be a mathematical expression. However, to avoid multiple different expression variants for a single problem, we follow SUMC-Solver[19] and output unified M-tree codes. So the ultimate goal is to obtain target code outputs through a target vector set $C=\{code_1,code_2,\cdots,code_m\}$, where $code_i$ is an l-dimensional vector corresponding to the number v_i, such as δ: ['1_0_+']. The vector set C can construct a target M-tree to calculate the final answer. Next, we will elaborate on the structure of the M-tree, and thereby explain why we need to output such a vector set.

As hinted at in Section 2.1, we leverage the M-tree structure to unify various mathematical expressions and determine a unique correction direction to reduce uncertainty. The M-tree comprises both leaf and internal nodes, with internal nodes representing operators, including $\{+,\times,\times-,+/\}$. Traditional operators include $\{+,-,\times,/\}$, where "–" and "/" are non-commutative, leading to variations. In the M-tree, commutative composite operations $\{\times-,+/\}$ are employed as replacements, and "–" and "/" are moved to leaf nodes. The composite operation "+/" represents the division of the summation of multiple values. For instance, "+/" with operands

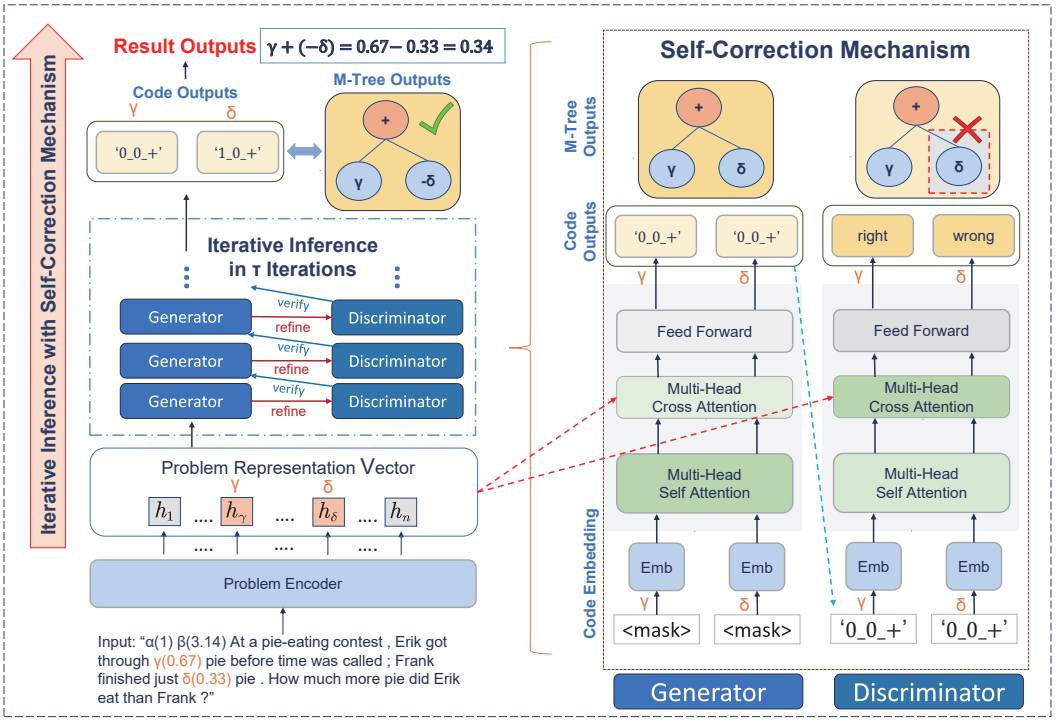


Fig. 2. The overall pipeline of our proposed UTSC-Solver.

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

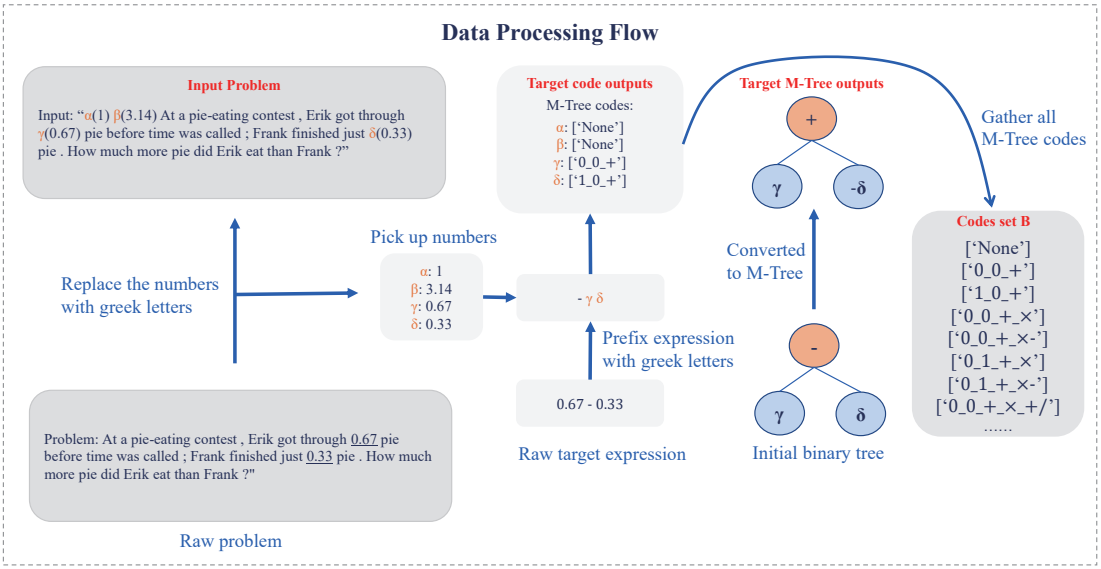


Fig. 3. Illustration of the data preprocessing workflow for a MWP.

$\{1,2,3\}$ is equivalent to"/{1,2,3}" . Leaf nodes in the MWP represent the values associated with the operators. In MWPs, inherent values v can manifest in four variations within leaf nodes: $\{v, -v, \frac{1}{v}, -\frac{1}{v}\}$, signifying the implicit $\{-, /\}$ operations. This design choice ensures the consistency and commutativity of the M-tree, addressing potential mathematical inconsistencies.

To enhance the flexibility in constructing the M-tree, we transform it into M-tree codes following the approach of the SUMC-solver[19]. Fig. 3 illustrates target M-tree and target codes outputs for a given problem. Each numerical value in the problem corresponds to one or multiple M-tree codes. These codes consist of two parts: a two-bit binary number representing the four forms of the value in the leaf nodes and a string that records the path from the root node to the leaf node, maintaining the M-tree's structure. For example, since δ appears in the form of $-v$ in a leaf node, its corresponding code is ["1_0_+"]. Conversely, since α does not appear in the M-tree, its corresponding code is ["None"]. So, our desired output is one or multiple codes for each numerical value in the problem.

We merge all codes from datasets into a code set B to facilitate the code generation process. Consequently, B is represented as $\{b_1, b_2, \cdots, b_l\}$, and all the codes we want to generate are included in the code set B. Therefore, we only need to simplify the code generation process by outputting the indices of B, specifically a target vector set $C = \{code_1, code_2, \cdots, code_m\}$. For example, $code_i$ may be represented as $[1,0,0,\cdots,0]$, with "1" indicating that $code_i$ corresponds to ["None"] (the first code in set B). It is essential to note that $code_i$ might contain multiple "1"s. However, the initially generated codes may not always be accurate. To address this issue, we employ an iterative self-correction mechanism to optimize the generated codes, ensuring more precise mathematical reasoning.

3.2 Iterative inference with self-correction mechanism

Inspired by human cognitive reasoning patterns, we have conceptualized iterative reasoning with a self-correction mechanism. It meticulously fine-tunes mathematical expressions by alternating between a generator and a discriminator. The overall process of UTSC-Solver is depicted in Fig. 2. On the left side, it involves a multi-step pipeline that includes an encoder, an iterative self-correction process with a generator, and a discriminator. Ultimately, this process yields the codes outputs and result. On the right side, we can see the specific details of the generator and discriminator. The inputs for the generator are the embeddings of every number in the problem from the output of encoder and partially masked codes, then the generator predicts the codes for masked number. The "Emb" represents the combination of numerical embeddings and code embeddings. The inputs for the discriminator are the number embeddings and the codes from the generator, then the discriminator predicts whether the codes are right or wrong.

The inference process is depicted in Algorithm 1. The function `IterativeInference`(P) encompasses the entire process from the problem encoder to the iterative inference, ultimately resulting in the output codes. The UTSC-Solver begins with the problem encoder, which interprets the input mathematical problem ("At a pie-eating contest...?"). The generator, informed by the discriminator's feedback, inputs mathematical expressions with masked judgments. Using transformer blocks, it regenerates accurate codes for these masked portions. Notably, the generator constructs mathematical expressions in a non-autoregressive manner, generating codes for all values in a single step (e.g., γ: ["0_0_+"]). The discriminator, a binary classifier, evaluates the rationality of the previously generated code outputs. It classifies each code as either *right* (1) or *wrong* (0).

During each iteration, the discriminator pinpoints inaccuracies in the generated expressions. Accurate codes receive a *right* label, while erroneous ones are tagged as *wrong*. These erroneous codes are then replaced with `<mask>`, prompting the generator to regenerate them, sequentially forming iterative self-correction. As illustrated in Fig. 2, the

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

generator's initial input is entirely <mask>. The iterative inference persists until either all codes are approved by the discriminator or the iteration count reaches the predefined maximum τ. Finally, the generated codes are utilized to construct the M-tree structure, which is then transformed into an equivalent mathematical expression, leading to the final result.

For example, the first iteration is depicted on the right side of Fig. 2. Initially, the generator receives a completely masked input, and it produces two code outputs: γ: ["0_0_+"] and δ: ["0_0_+"]. Subsequently, we use these code outputs as input for the discriminator, which results in the statement: Code ["0_0_+"] is wrong for number δ. After the iteration, it is corrected to the accurate code ["1_0_+"], leading to the final correct answer. Next, we will provide a detailed description of the structure of each component of UTSC-Solver.

3.3 Problem encoder

First in the problem encoder, we input the preprocessed problem $P = \{p_1, p_2, \cdots, p_n\}$ to obtain representations $\{h_1, h_2, \cdots, h_n\}$. We extract the representation h_i as e_i for each numerical value v_i in V_P. Second, we perform average pooling on problem representations to obtain the semantic representation of the entire problem, denoted as $P_{\text{goal}} = \frac{1}{n} \cdot \sum h_i$. Finally, we merge numerical value representation and problem representation to obtain the numerical embedding $n_i = [e_i, P_{\text{goal}}]$. This encoding scheme allows the model to understand semantic relationships between each numerical value in the problem and obtain the overall semantic representation of the problem through average pooling. This representation serves as the holistic objective to drive the model for mathematical reasoning.

3.4 Generator and discriminator

Now, we will introduce the architecture of the generator and discriminator. The generator is a decoder based on the transformer architecture[46], utilizing a non-autoregressive framework. In total, the input consists of the representation vector n_i from the encoder and the M-tree codes from previous iteration with some portions partially masked. It is required to predict the M-tree codes for each number. We add the representation vector n_i and the code vector $code_i$ corresponding to the partially masked unified M-tree codes, resulting in $d_i = n_i + c_i$ as the input to the generator. The masked c_i corresponds to the special character <mask>. For d_i, the self-attention mechanism aims to better understand the M-tree structure composed of the current code sequence. It is applied as shown in Eqs. (1) and (2):

$$Q = \boldsymbol{W}_Q \cdot D, \quad K = \boldsymbol{W}_K \cdot D, \quad V = \boldsymbol{W}_V \cdot D, \tag{1}$$

$$MulitiHead(Q, K, V) = Concat(head_1, head_2, \cdots, head_h)\boldsymbol{W}^O, \tag{2}$$

where D is the input sequence of the decoder $\{d_1, d_2, \cdots, d_m\}$, and $head_i$ is defined as:

$$head_i(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}}\right) \cdot V, \tag{3}$$

where $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V$, and $\boldsymbol{W}^O$ are learnable parameter matrices. Next, we use the cross-attention mechanism to incorporate the semantic information from the original problem, which helps drive the construction of the complete M-tree. The computation process for the cross-attention mechanism is similar, with the only difference being $Q = W_Q \cdot N$, where $N = n_1, n_2, \cdots, n_m$, representing the output of the encoder as the query vector. Finally, the vector $\boldsymbol{d}_i'$ obtained from the two attention modules is passed through a three-layer feed-forward neural network (FFNN) to generate M-tree codes for the decoder. The specific formula is shown in Eq. (4):

$$code_i = (\sigma(\sigma(\boldsymbol{d}_i'^{\mathrm{T}} \cdot W_1 + B_1) \cdot W_2 + B_2))^{\mathrm{T}} \cdot W_3 + B_3, \tag{4}$$

where σ is the activation function, and W, B are learnable parameters in the FFNN. It should be noted that the one-hot vectors $code_1$, $code_2$, $\cdots$, $code_m$ are generated independently and in parallel.

The discriminator, a binary classifier, encodes all $code_i$ from the current output to obtain $c_i = FFNN(code_i)$. We then add the corresponding numerical representation as the input $d_i = c_i + n_i$ for the discriminator. Subsequently, it undergoes the same multi-head self-attention and multi-head cross-attention modules as the generator. It then classifies (0 or 1) through a three-layer FFNN, where a value of 0 indicates a wrong code that requires masking.

In essence, our model strives to enhance code generation accuracy by iteratively refining and rectifying generated

Algorithm 1: UTSC-Solver iterative inference with self-correction mechanism.

Input: Problem P from datasets D.

Output: M-tree codes $Codes$ for each number to compute the result.

Function `IterativeInference`(P):
  $H \leftarrow$ `Encoder`(P);
  Extract numerical embeddings $N = \{[e_i, P_{\text{goal}}]\}$ in H for each number v_i in V_P;
  Initialize $Codes$ with all $code_i$ as <mask>;
  for *iteration* = 1 **to** τ **do**
    $Codes \leftarrow$ `Generator`($Codes, N$);
    $Judgments \leftarrow$ `Discriminator`($Codes, N$);
    if *Judgments are all right* **then**
      break;
    end
    for $i \leftarrow 1$ **to** *len*($Judgments$) **do**
      $j \leftarrow Judgments[i]$;
      $code_i \leftarrow Codes[i]$;
      if j *is wrong* **then**
        $code_i \leftarrow$ <mask>;
      end
    end
  end
  return $Codes$ and convert to mathematical expressions to compute the result.

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

codes, leveraging both the generator's capabilities and feedback from the discriminator.

3.5 Training process of UTSC-Solver

Given a training dataset $D=\{(P_i,C_i)|i=1,\cdots,N\}$, where P_i denotes the ith problem and C_i is the corresponding M-tree codes for the mathematical expression. Our training process involves iterative refinement through a joint training mechanism of the generator and discriminator.

3.5.1 Generator loss

The generator's loss is computed using the cross-entropy between the predicted codes and the target C_i. However, in the iterative self-correction mechanism, only the masked positions contribute to the loss, ensuring the model's focus on rectifying erroneous predictions:

$$loss_{\text{generator}} = \sum_{(P_i,C_i)\in D}\sum_{c_i\in C_i}\frac{1}{l}\sum_{j=1}^{l}-[C_{ij}\cdot\log(c_{ij})+(1-C_{ij})\cdot\log(1-c_{ij})], \tag{5}$$

where c_i is the code at the masked position and l denotes the dimensionality of the code vector, equivalent to the size of the code set B.

3.5.2 Discriminator loss

The discriminator evaluates the validity of all code outputs from the generator. Its loss is derived from the cross-entropy between the predicted probability p_i of $code_i$ being valid and the actual label y_i:

$$loss_{\text{discriminator}}=\frac{1}{m}\sum_{i}[-[y_i\cdot\log(p_i)+(1-y_i)\cdot\log(1-p_i)]]. \tag{6}$$

3.5.3 Joint loss formulation

The joint training process optimizes a combined loss, which is a weighted sum of the generator and discriminator losses, where λ is a hyperparameter determining the balance between the generator and discriminator losses:

$$loss=\lambda\cdot loss_{\text{generator}}+(1-\lambda)\cdot loss_{\text{discriminator}}. \tag{7}$$

3.6 Convergence analysis

The UTSC-Solver conducts mathematical reasoning through an iterative self-correction mechanism. Given the time complexity of the reasoning process, we set the maximum number of iterations to τ. Now, we attempt to prove that our algorithm can converge to a desired accuracy threshold within a fixed number of iterations. Formally, We aim to demonstrate that there exists a maximum iteration number, denoted as τ, such that the accuracy can converge to a desired threshold P_E (e.g., 95%) within τ iterations.

We start by defining some important notations as follows: The code discriminator has a minimum accuracy threshold α. The code generator has a minimum generation accuracy threshold β. And the accuracy in the kth iteration is P_k.

Then, we divide the iterative process of UTSC-Solver into three kinds of situations:

(i) The discriminator correctly identifies valid codes.

(ii) The discriminator misidentifies valid codes, which are then rectified by the generator.

(iii) The discriminator correctly identifies invalid codes, which are rectified by the generator.

From these scenarios, the lower bound on the accuracy for the next iteration is:

$$P_{k+1}=C_1\cdot P_k+C_2, \tag{8}$$

where $C_1=\alpha+\beta-2\alpha\beta$ and $C_2=\alpha\beta$.

Using mathematical induction, we can derive that:

$$P_n=C_1^n\cdot P_0+\frac{C_2\cdot(1-C_1^n)}{1-C_1}. \tag{9}$$

Given the constraints on α and β, we deduce $C_1<1$. Thus, as n increases, P_n approaches a constant value. To achieve P_E, the required iterations are:

$$n_E=\lceil\log_{C_1}\frac{P_E-\frac{C_2}{1-C_1}}{P_0-\frac{C_2}{1-C_1}}\rceil. \tag{10}$$

By setting $\tau=n_E$, we ensure convergence to the desired accuracy within a finite number of iterations, optimizing the efficiency of the inference process.

4 Experiments

4.1 Datasets

To validate the effectiveness of our UTSC-Solver, we evaluated our model on two widely used MWP datasets: MAWPS[23] which consists of 2373 English problems, and Math23K[13] which consists of 23162 Chinese problems. For Math23K, we used the publicly available test set containing 1000 problems. For MAWPS, we performed five-fold cross-validation. We used answer accuracy as the evaluation metric, which is determined by checking if the mathematical expression obtained by converting the unified M-tree codes generated by UTSC-Solver is equivalent to the provided answer.

4.2 Experimental settings

All experiments were conducted on a Linux server equipped with four 2.30 GHz Intel Xeon Gold 5218 CPUs and one Tesla V100 GPU. The initial global learning rate for both datasets was set to 2×10^{-5} and was dynamically adjusted using the warm-up learning rate algorithm[47]. We employed the AdamW[48] optimizer for training. For the FFNN in the decoder, the dimensions were set to $(2048,1024,l)$, where l represents the size of the code set B. For Math23K, the size of l was 153, while for MAWPS, it was 28. During inference, the maximum number of iterations τ was set to 10.

4.3 Comparisons with baselines

To evaluate and analyze the effectiveness of our proposed UTSC-Solver, we compared its performance with state-of-the-art baselines. First, we compare our proposed UTSC-Solver with baseline models based on the autoregressive (AR) generation framework:

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

● T-RNN[49]: This model applies a sequence-to-sequence approach to predict a tree structure template that includes inferred numbers and unknown operators.

● Seq2Seq[13]: This model uses a basic sequence-to-sequence model with attention mechanism to directly translate MWPs into equation templates.

● GTS[16]: GTS employs a heuristic, target-driven inference approach to construct expression binary trees in a top-down manner.

● Graph2Tree[32]: This model is based on the graph2tree framework and utilizes an external graph-based encoder to capture dependencies and positional information between numerical values, enriching the representation of quantities in the problem.

● HMS[17]: HMS utilizes a hierarchical word-clause-question relationship to better extract the semantic meaning of math problems.

● NS-Solver[50]: NS-Solver incorporates four auxiliary tasks into the training process to better handle mathematical symbol constraints.

● BERT-GTS: This model replaces the RNN-based encoder in the original GTS model with a pretrained language model BERT.

● GPT-3[51]: An advanced autoregressive large language model capable of generating human-like text. GPT-3 utilizes a transformer-based model with 175 billion parameters, making it capable of understanding and generating human-like text based on the input. We input the math word problem by the API and extract the numeric answer from its text response.

Next, we compare our proposed UTSC-Solver with solvers based on the non-autoregressive (NAR) generation framework:

● SUMC-Solver[19]: This is the first solver to perform MWP mathematical reasoning in a non-autoregressive manner. It introduces the concept of M-tree, a unified binary tree representation that accounts for diversity.

4.3.1 Comparative analysis of experimental results

The experimental results, as tabulated in Table 1, emphasize the prowess of UTSC-Solver, particularly its self-correction mechanism and non-autoregressive generator. First, UTSC-Solver consistently outperforms both autoregressive solvers and the non-autoregressive SUMC-Solver across Math23K and MAWPS datasets, demonstrating the effectiveness of our overall architecture and the iterative reasoning process. Second, UTSC-Solver surpasses the SUMC-Solver[19] by 1%+ on both datasets, emphasizing the significance of the iterative self-correction mechanism. Third, when compared to sequence-based solvers like T-RNN and Seq2Seq, UTSC-Solver exhibits an improvement of 10% or more. Even when compared to advanced binary tree-based solvers like GTS, HMS, Graph2Tree, and NS-Solver, UTSC-Solver surpasses them, highlighting its superior performance. These results underscore the necessity of the M-tree structure, which outperforms sequences and binary trees in mathematical reasoning. Especially, UTSC-Solver surpasses BERT-GTS by approximately 3%. This directly underscores the efficacy of the attention-based generator and discriminator, since they harness the same problem encoder (i.e., BERT). In essence, the iterative and self-correcting nature of UTSC-Solver, as detailed in Fig. 2, not only mirrors human-like problem-solving but also ensures superior mathematical reasoning. Finally, we observe that GPT-3[51] has an accuracy of 64.3% on the MAWPS dataset, indicating that GPT-3 possesses basic mathematical reasoning abilities. Its accuracy on Math23K is not good, standing at 18.1%. This suggests that GPT-3 may lack training data for Chinese math word problems and has room for improvement in multi-step reasoning and accurate calculation. Additionally, this conclusion potentially proves the necessity of enabling models to self-correct even for large language models (LLMs). Our UTSC-Solver can achieve this through a non-autoregressive mechanism, indicating a direction towards empowering LLMs with the ability to self-correct.

Table 1. Answer accuracy comparison of different models on Math23K and MAWPS* datasets.

Type	Model	Math23K	MAWPS*
AR	T-RNN[49]	0.669	0.668
AR	Seq2Seq[13]	0.640	0.797
AR	GTS[16]	0.756	0.752
AR	Graph2Tree[32]	0.766	0.781
AR	HMS[17]	0.761	0.803
AR	NS-Solver[50]	0.757	–
AR	BERT-GTS	0.795	0.798
AR	GPT-3[51]	0.181	0.643
NAR	SUMC-Solver[19]	0.825	0.820
NAR	UTSC-Solver (ours)	**0.829**	**0.834**

4.4 Ablation study

In order to study the influence of each module in the UTSC-Solver model, namely the self-correction mechanism, attention mechanism, and problem semantic encoding P_{goal}, we conducted ablation experiments. The purpose and results of these experiments are presented in Table 2. The UTSC-Solver represents the complete model, while "w/o Self-Correction mechanism" refers to the model with the self-correction mechanism removed, performing only one generation. "w/o Attention" indicates the model without self-attention and cross-attention mechanisms in the decoder. Finally, "w/o P_{emb}" represents the model without problem semantic representation in the encoder, using only numerical representations as the encoder's output.

From the experimental results, it can be observed that the accuracy of our model decreases when any of the components is removed. This indicates that all the components have an effect and complement each other in terms of problem semantic understanding, mathematical expression generation, and optimization. Furthermore, the results highlight the significant role of the self-correction mechanism in improving answer accuracy and enhancing the mathematical reasoning ability of the UTSC-Solver model. Furthermore, the impact of removing P_{goal} (referred in Section 3.3) is minimal. The marginal performance decrease could be attributed to the

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

Table 2. Ablation experiment results on Math23K and MAWPS* datasets.

Model	Math23K	MAWPS*
UTSC-Solver	0.829	0.834
w/o Self-correction mechanism	0.820	0.819
w/o Attention	0.824	0.820
w/o P_{goal}	0.826	0.831

rich information inherent in numerical value representations. Even without P_{goal}, the encoder's self-attention and the decoder's cross-attention mechanisms, along with the self-correction mechanism, compensate for its absence.

4.4.1 Effectiveness of self-correction mechanism

To offer a detailed perspective on iteration effects, we introduce code accuracy in addition to answer accuracy. Code accuracy signifies the proportion of correctly generated code segments relative to the total number of code segments in the problem. To visualize the impact of the self-correction mechanism, we depict the answer accuracy and code accuracy on the MAWPS dataset as the number of iterations increases in Fig. 4. Both code accuracy and answer accuracy exhibit gradual improvement with deeper iterations, illustrating the self-correction mechanism's role in iteratively rectifying errors in mathematical expressions.

During each iteration, the generator leverages intermediate results from the previous round and feedback from the discriminator. By integrating information from previously generated code segments through the self-attention mechanism, the generator gains awareness of other sub-expressions, resulting in the construction of a more coherent and comprehensive M-tree structure. Furthermore, we observe that beyond $t > 2$, the model's accuracy stabilizes. When we plug in the accuracy of the generator and discriminator into Eq. (10), we obtain a maximum number of iterations between 2 and 3. Therefore, when $t > 2$, the UTSC-Solver achieves a saturated maximum accuracy.

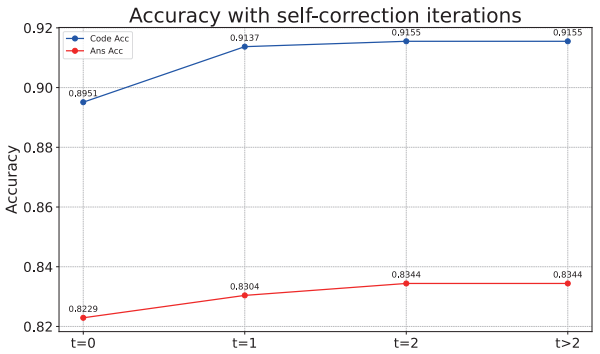


Fig. 4. Enhancement of answer accuracy and code accuracy by iterative inference.

4.5 Case study

To provide a clearer understanding of the iterative and self-correcting characteristics of UTSC-Solver, we present a comparative case study between UTSC-Solver and SUMC-Solver[19] using the MAWPS and Math23K datasets, as visualized in Fig. 5. In both cases, we first present the problem text

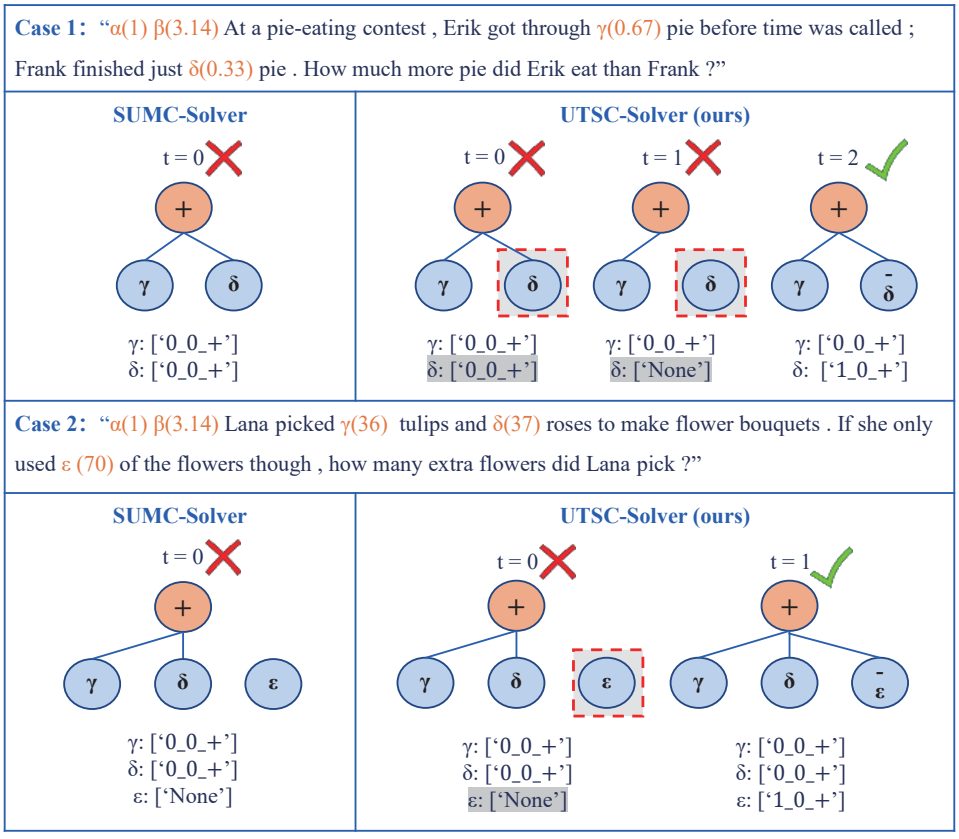


Fig. 5. Case study of UTSC-Solver's iterative self-correction mechanism.

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

and the solution obtained by SUMC-Solver. Next, we illustrate the iterative reasoning process of UTSC-Solver, with different "t" representing different iterations.

In case 1, the task is to calculate how much more pie Erik ate compared to Frank. However, the model generates an incorrect addition operation at first. Through two iterations, the addition operation is successfully corrected to subtraction. Specifically, for $t = 0$, the code corresponding to the number δ is ["0_0_+"], with a probability value of 0.9610975. Additionally, the code ["1_0_+"] has a probability value of 0.08277654. For $t = 1$ and $t = 2$, the self-correction mechanism identifies the code ["0_0_+"] as incorrect and replaces it with the code ["1_0_+"] as the new output. As a result, the self-correction mechanism, through three iterations, corrected the incorrect operator from "$\gamma + \delta$" to "$\gamma - \delta$". On the other hand, SUMC-Solver[19] produces the same error and ultimately gives an incorrect answer. Similarly, in case 2, the UTSC-Solver model corrects the missing number ϵ with the ["1_0_+"] form, resulting in the correct M-tree structure.

These case studies underscore UTSC-Solver's prowess in emulating human-like iterative refinement, a demonstration of its innovative design and the efficacy of its self-correction mechanism.

5 Conclusions

In this work, we introduced the unified M-tree self-correction solver (UTSC-Solver) to address challenges in traditional MWP solvers. UTSC-Solver employed a non-autoregressive framework for flexible mathematical reasoning, aligning with human cognitive patterns. A key feature was the self-correction mechanism, which iteratively refined mathematical expressions. By alternating between a generator and a discriminator, it identified and corrected minor errors, improving accuracy and offering interpretable reasoning. Extensive experiments on MAWPS and Math23K datasets demonstrated UTSC-Solver's superiority over traditional MWP models, highlighting the self-correction mechanism's effectiveness. UTSC-Solver advanced MWP solvers by emulating human-like iterative refinement, promising more accurate and interpretable automated mathematical reasoning. Future work could apply this mechanism to large language models (LLMs) to address hallucination issues[52].

Acknowledgements

This work was supported by the National Natural Science Foundation of China (62106244), the Fundamental Research Funds for the Central Universities (WK2150110021), and the University Synergy Innovation Program of Anhui Province (GXXT-2022-042).

Conflict of interest

The authors declare that they have no conflict of interest.

Biographies

Zhiyuan Ma is a current master's student at the University of Science and Technology of China. His research mainly focuses on knowledge learning and knowledge reasoning.

Zhenya Huang is currently an Associate Professor at the University of Science and Technology of China (USTC). He received his Ph.D. degree from USTC in 2020. His research mainly focuses on artificial intelligence, knowledge reasoning, and intelligent education.

References

[1] He X, Jiang Y, Hu Y. Application of a newly developed naive Bayes algorithm in fire alarm. *JUSTC*, **2022**, *52*(6): 5.
[2] Liu J, Wu J, Liu A, et al. Preoperative diagnosis of hepatocellular carcinoma patients with bile duct tumor thrombus using deep learning method. *JUSTC*, **2022**, *52* (12): 6.
[3] Verschaffel L, Schukajlow S, Star J, et al. Word problems in mathematics education: A survey. *ZDM*, **2020**, *52*: 1–16.
[4] Lu Y, Pian Y, Chen P, et al. RadarMath: An intelligent tutoring system for math education. *Proceedings of the AAAI Conference on Artificial Intelligence*, **2021**, *35*: 16087–16090.
[5] Li S, Puig X, Paxton C, et al. Pre-trained language models for interactive decision-making. In: Koyejo S, Mohamed S, Agarwal A, et al., editors. Advances in Neural Information Processing Systems. New York: Curran Associates, Inc. , **2022**, *35*: 31199–31212.
[6] Kenney R, An T, Kim S H, et al. Linear programming models: Identifying common errors in engineering students' work with complex word problems. *International Journal of Science and Mathematics Education*, **2020**, *18*: 635–655.
[7] Li J, Wang L, Zhang J, et al. Modeling intra-relation in math word problems with different functional multi-head attentions. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA, USA: ACL, **2019**: 6162–6167.
[8] Huang Z, Lin X, Wang H, et al. DisenQNet: Disentangled representation learning for educational questions. In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. New York: ACM, **2021**: 696–704.
[9] Zhao C, Li X, He M, et al. Sequential recommendation via an adaptive cross-domain knowledge decomposition. In: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management. New York: ACM, **2023**: 3453–3463.
[10] Huang Z, Liu Q, Gao W, et al. Neural mathematical solver with enhanced formula structure. In: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM, **2020**: 1729–1732.
[11] Liu J, Huang Z, Lin X, et al. A cognitive solver with autonomously knowledge learning for reasoning mathematical answers. In: 2022 IEEE International Conference on Data Mining (ICDM). Orlando, FL, USA: IEEE, **2022**: 269–278.
[12] Lin X, Huang Z, Zhao H, et al. Learning relation-enhanced hierarchical solver for math word problems. *IEEE Transactions on Neural Networks and Learning Systems*, **2024**, *35*: 13830–13844.
[13] Wang Y, Liu X, Shi S. Deep neural solver for math word problems. In: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing. Copenhagen, Denmark: ACL, **2017**: 845–854.
[14] He D, Xiao J. Goal selection and feedback for solving math word problems. *Applied Intelligence*, **2023**, *53* (12): 14744–14758.
[15] Sutskever I, Vinyals O, Le Q V. Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems 27 (NIPS 2014). New York, USA: Curran Associates, Inc., **2014**, *4*: 3104–3112.
[16] Xie Z, Sun S. A goal-driven tree-structured neural model for math word problems. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. Macao, China: IJCAI, **2019**: 5299–5305.
[17] Lin X, Huang Z, Zhao H, et al. HMS: A hierarchical solver with dependency-enhanced understanding for math word problem. *Proceedings of the AAAI Conference on Artificial Intelligence*, **2021**,

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703

35: 4232–4240.
[18] Cao Y, Hong F, Li H, et al. A bottom-up DAG structure extraction model for math word problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, **2021**, *35*: 39–46.
[19] Wang B, Ju J, Fan Y, et al. Structure unified M-tree coding solver for math word problem. In: Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing. Abu Dhabi, United Arab Emirates: ACL, **2022**: 8122–8132.
[20] Patel A, Bhattamishra S, Goyal N. Are NLP models really able to solve simple math word problems? In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Online: ACL, **2021**: 2080–2094.
[21] Welleck S, Lu X, West P, et al. Generating sequences by learning to self-correct. arXiv: 2211.00053, **2022**.
[22] Madaan A, Tandon N, Gupta P, et al. Self-refine: iterative refinement with self-feedback. arXiv: 2303.17651, **2023**.
[23] Koncel-Kedziorski R, Roy S, Amini A, et al. MAWPS: A math word problem repository. In: Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Stroudsburg, PA, USA: ACL, **2016**: 1152–1157.
[24] Wu J, Wu Y. Machine learning in data envelopment analysis: A smart mechanism for indicator selection. *JUSTC*, **2022**, *52* (12): 5.
[25] Zhuang W, Chen C, Qiu G. A new deep reinforcement learning model for dynamic portfolio optimization. *JUSTC*, **2022**, *52* (11): 3.
[26] Liu J, Huang Z, Ma Z, et al. Guiding mathematical reasoning via mastering commonsense formula knowledge. In: Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. New York: ACM, **2023**: 1477–1488.
[27] Liu J, Huang Z, Zhai C, et al. Learning by applying: A general framework for mathematical reasoning via enhancing explicit knowledge learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, **2023**, *37*: 4497–4506.
[28] Fletcher C R. Understanding and solving arithmetic word problems: A computer simulation. *Behavior Research Methods, Instrument, & Computers*, **1985**, *17*: 565–571.
[29] Ma Y, Zhou Y, Cui G, et al. Frame-based calculus of solving arithmetic multi-step addition and subtraction word problems. In: 2010 Second International Workshop on Education Technology and Computer Science. Wuhan, China: IEEE, **2010**: 476–479.
[30] Roy S, Roth D. Mapping to declarative knowledge for word problem solving. *Transactions of the Association for Computational Linguistics*, **2018**, *6*: 159–172.
[31] Shi S, Wang Y, Lin C Y, et al. Automatically solving number word problems by semantic parsing and reasoning. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Stroudsburg, PA, USA: ACL, **2015**: 1132–1142.
[32] Zhang J, Wang L, Lee R K, et al. Graph-to-tree learning for solving math word problems. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA, USA: ACL, **2020**: 3928–3937.
[33] Wu Q, Zhang Q, Fu J, et al. A knowledge-aware sequence-to-tree network for math word problem solving. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). Stroudsburg, PA, USA: ACL, **2020**: 7137–7146.
[34] Devlin J, Chang M-W, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of NAACL-HLT 2019. Minneapolis, Minnesota, USA: ACL, **2019**: 4171–4186.
[35] Liang Z, Zhang J, Wang L, et al. MWP-BERT: numeracy-augmented pre-training for math word problem solving. In: Findings of the Association for Computational Linguistics: NAACL 2022. Seattle, United States: ACL, **2022**: 997–1009.
[36] Meyer R. The consistency of arithmetic. *The Australasian Journal of Logic*, **2021**, *18*: 289–379.
[37] Wang Y, Wang Y, Dang K, et al. A comprehensive survey of grammatical error correction. *ACM Transactions on Intelligent Systems and Technology*, **2021**, *12*: 1–51.
[38] Anbukkarasi S, Varadhaganapathy S. Neural network-based error handler in natural language processing. *Neural Computing and Applications*, **2022**, *34*: 20629–20638.
[39] Liang Y, Li L. Heterogeneous models ensemble for Chinese grammatical error correction. In: International Conference on Artificial Intelligence, Virtual Reality, and Visualization (AIVRV 2022). Chongqing, China: SPIE, **2023**: 125880F.
[40] Lee J, Seneff S. Correcting misuse of verb forms. In: Proceedings of ACL-08: HLT. Columbus, Ohio, USA: ACL, **2008**: 174–182.
[41] Stahlberg F, Kumar S. Seq2Edits: sequence transduction using span-level edit operations. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). Online: ACL, **2020**: 5147–5159.
[42] Lai S, Zhou Q, Zeng J, et al. Type-driven multi-turn corrections for grammatical error correction. In: Findings of the Association for Computational Linguistics: ACL 2022. Dublin, Ireland: ACL, **2022**: 3225–3236.
[43] Malmi E, Krause S, Rothe S, et al. Encode, tag, realize: high-precision text editing. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. Hong Kong, China: ACL, **2019**: 5054–5065.
[44] Ren H, Yang L, Xun E. A sequence to sequence learning for Chinese grammatical error correction. In: Natural language processing and Chinese computing. Cham: Springer, **2018**: 401–410.
[45] Ghazvininejad M, Levy O, Liu Y, et al. Mask-predict: parallel decoding of conditional masked language models. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. Hong Kong, China: ACL, **2019**: 6112–6121.
[46] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Syste. New York: ACM, **2017**: 6000–6010.
[47] Xu Z, Dai A M, Kemp J, et al. Learning an adaptive learning rate schedule. arXiv: 1909.09712, **2019**.
[48] Loshchilov I, Hutter F. Decoupled weight decay regularization. In: International Conference on Learning Representations. New Orleans, Louisiana, United States: ICLR, **2019**.
[49] Wang L, Zhang D, Zhang J, et al. Template-based math word problem solvers with recursive neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, **2019**, *33*: 7144–7151.
[50] Qin J, Liang X, Hong Y, et al. Neural-symbolic solver for math word problems with auxiliary tasks. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). Online: ACL, **2021:** 5870–5881.
[51] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners. In: Proceedings of the 34th International Conference on Neural Information Processing System. New York: ACM, **2020**: 1877–1901.
[52] Mündler N, He J, Jenko S, et al. Self-contradictory hallucinations of large language models: evaluation, detection and mitigation. arXiv: 2305.15852, **2023**.

CSTR: 32290.14.JUSTC-2023-0145
DOI: 10.52396/JUSTC-2023-0145
JUSTC, **2025**, 55(7): 0703