Towards Automatic Tutoring of Custom Student-Stated Math Word Problems

Pablo Arnau-González[1][0000−0001−9048−4659], Ana Serrano-Mamolar[2][0000−0002−0027−7128], Stamos Katsigiannis[3][0000−0001−9190−0941], and Miguel Arevalillo-Herráez[1][0000−0002−0350−2079]

[1] Departament d'Informàtica, Universitat de València, Valencia, Spain
{pablo.arnau, miguel.arevalillo}@uv.es
[2] Departamento de Ingenieria Informatica, Universidad de Burgos, Burgos, Spain
asmamolar@ubu.es
[3] Department of Computer Science, Durham University, Durham, UK
stamos.katsigiannis@durham.ac.uk

Abstract. Math Word Problem (MWP) solving for teaching math with Intelligent Tutoring Systems (ITSs) faces a major limitation: ITSs only supervise pre-registered problems, requiring substantial manual effort to add new ones. ITSs cannot assist with student-generated problems. To address this, we propose an automated approach to translate MWPs to an ITS's internal representation using pre-trained language models to convert MWP to Python code, which can then be imported easily. Experimental evaluation using various code models demonstrates our approach's accuracy and potential for improvement.

Keywords: Math Word Problems · Algebra Tutoring · Intelligent Tutoring Systems · Automatic Code Generation.

1 Introduction

Math Word Problem (MWP) solving is the task of providing a numerical solution to a mathematical problem expressed in natural language [2]. The computation of the numerical answer of the MWP requires the correct identification of the quantities expressed in the problem statement, together with the relationships between these quantities [4]. MWPs are widely used in Intelligent Tutoring Systems (ITSs) to teach math and arithmetic by emulating human tutor tasks such as interpreting problem statements, validating processes, and adapting problems to individual learners [1]. However, ITSs are limited to supervising registered problems and cannot assist with self-introduced problems, as they have to be registered in the system's own knowledge representation schema.

In this work, we present a two-stage process to automate the encoding of problem solutions from natural language statements. A large code model, i.e. a Large Language Model (LLM) trained on source code, generates an intermediate representation of the problem solution as Python code, which can then be

used to create a bipartite graph solution specification using compiler technology. This automation has benefits for learners and experts, enabling problem-specific supervision and easy addition of new exercises to the ITS. In our evaluation on a dataset of 1,000 MWP, our method correctly solved 39% of the problems, consistent with other state-of-the-art MWP solving methods, while at the same time facilitating the automated encoding of MWPs for ITS.

2 Proposed Methodology

Given a natural language problem description S that consists of n words w_i, $S = \{w_0, w_1, ..., w_{n-1}\}$, the proposed approach will generate Python source code that first defines a list Q_S of m quantities that appear in S, $Q_S = \{q_0, q_1, ..., q_{m-1}\}$, and then computes and returns the numerical answer y_S to the problem S. The main benefit of proposing a solution expressed in source code is that it allows to construct a graph using the mathematical relations between the quantities, and simultaneously allows for the automatic naming of the identified quantities so that the system is capable of semantically matching the user input to the correct quantity. The generated Python source code is essentially an intermediate representation of the MWP that can be subsequently converted to the internal representation of an ITS in an automated manner. To generate Python code for a MWP, we propose initialising a LLM's prompt with an example problem statement and the expected output code, followed by the target problem statement. The example problem is introduced as a code comment followed by a function definition (`sol()`), with known quantities defined in a Map-like structure using Python's dictionary data structure, as shown in Fig. 1a. One operation is defined per line until the solution is defined and returned. The model output is the source code of the `sol()` function for the target problem, as shown in Fig. 1b. The code can then be compiled into the required representation for any ITS.

The pre-trained LLMs examined in this work are the 350M, 2B, 6B, and 16B parameter variations of Saleforce's CodeGen [9] model ("mono" version), a transformer-based model trained on a general text corpus and fine-tuned first on a corpus with source code from multiple languages and then on a Python corpus, and the 1B and 6B parameters variations of Facebook's InCoder [3] Casual Language model that has been trained on a corpus that contained source code from Github and StackOverflow.

3 Results

Our proposed approach is evaluated against a common benchmark dataset for MWP solving, SVAMP [10]. SVAMP is a collection of 1,000 MWP, expressed in natural language (English), along with the numerical answer and an algebraic expression to solve the problem. The performance of the examined models on the SVAMP dataset was evaluated according to the accuracy metric, defined in this case as the percentage of problems that were solved with the first solution returned by the model.

```
""" A book has 3 chapters. The first chapter is 91 pages long
the second chapter is 23 pages long and the third chapter is
25 pages long. How many more pages does the first chapter have
than the second chapter? """
def sol():
    context = dict()
    context['number of chapters'] = 3
    context['number of pages first chapter'] = 91
    context['number of pages sencond chapter'] = 23
    context['number of pages third chapter'] = 25
    context['pages more first chapter'] = (
         context['number of pages first chapter']
         - context['number of pages second chapter']
    )
    return context['pages more first chapter']
""" Each pack of dvds costs 76 dollars.
If there is a discount of 25 dollars on each pack.
How much do you have to pay to buy each pack? """
def sol():
```

(a) Prompt

```
context = dict()
context['price of dvds'] = 76
context['discount'] = 25
context['price of dvds with discount'] = (
    context['price of dvds'] - context['discount']
)
return context['price of dvds with discount']
```

(b) Output

Fig. 1: Example input problem statement with one example provided to the model and expected solution for a MWP.

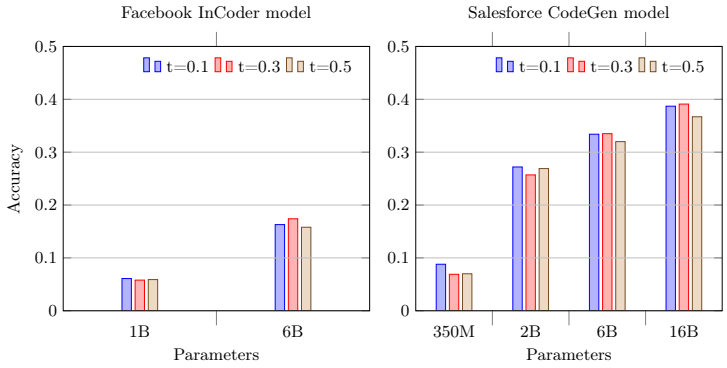
Table 1: Accuracy for the best-performing temperature (t) for each of the examined models

Model	# Params	Temperature	Accuracy
InCoder [3]	1B	0.1	0.061
	6B	0.3	0.174
CodeGen [9]	350M	0.1	0.088
	2B	0.1	0.272
	6B	0.3	0.335
	16B	0.3	**0.391**

Note: M: Million, B: Billion. Best performance in bold.

For the performance evaluation procedure, a random problem from SVAMP was selected and manually solved by implementing the respective `sol()` function, according to the specifications described in Section 2. This problem was then used as the initialisation of all the language models' prompts, in order to ensure a fair evaluation across the different examined models. Then, each model was queried with the randomly chosen example problem and the statement for the problem to be solved, as shown in Fig. 1a. Each model was queried to generate $n = 10$ solutions for each of the 1,000 problems of the SVAMP dataset, which were then tested by running the generated Python source code and evaluating whether the result was equal to the expected solution or not.

Both examined models (CodeGen, InCoder) were evaluated for all the publicly available parameter number variations. CodeGen was tested for all the aforementioned model variations, while InCoder was tested for the 1B and 6B parameter versions. The inference was carried out by tuning Softmax's temperature (t) parameter to values 0.1, 0.3, and 0.5. The temperature is used to control the randomness of a model's predictions, with higher values of temperature resulting in a model becoming more random and less certain of its predictions, whereas lower values result in more certain predictions. To this end, each parameter number variation of the CodeGen and InCoder models was evaluated three times using the three aforementioned temperature values, respectively.



(a) Facebook InCoder model (b) Salesforce CodeGen model

Fig. 2: Accuracy obtained for different temperatures (t) and number of parameters for each model. (a) Facebook InCoder; (b) Salesforce CodeGen.

Results in terms of accuracy, are reported in Table 1 for the temperature that provided the best accuracy for each examined model variation. From this table, it is evident that the 16B parameter CodeGen model achieved the best performance on the SVAMP dataset in terms of the examined metric, reaching an accuracy of 39.1%. In addition, one of the main takeaways from Table 1 is that there is a clear relationship between the number of parameters and the performance, with variants of a model with more parameters achieving higher accuracy (Pearson's $\rho = 0.77$), as shown in Fig. 2a and 2b, for the InCoder and CodeGen models respectively. Fig. 2 depicts the accuracy for each examined model variant and temperature value. The figure shows clearly the relationship between the number of model parameters and the accuracy. However, it also makes evident that this relationship is not purely linear, but rather it appears to follow a logarithmic trend. In addition, it is evident that the temperature parameter has minimal effects on the achieved accuracy.

The presented results compete directly with the latest state-of-the-art MWP solving methods, with CodeGen-16B's maximum accuracy of 39.1% outperforming 6 out of the 9 current top performing methods, as shown in Table 2, without being explicitly designed to solve MWPs. In addition and in contrast to other available MWP solving methods, the proposed approach enables the automation of the task of translating MWP from natural language to the internal representation of ITSs, thus addressing one of their major limitations. Consequently, other state-of-the-art MWP solving methods cannot replace the proposed approach for the task at hand, as they cannot provide the required source code representation of the MWP. It is expected that as large language models get even bigger in terms of the number of parameters, the proposed approach will be capable of surpassing the state-of-the-art MWP solving methods without requiring specific domain knowledge.

Table 2: Current top performing MWP solving methods on the SVAMP dataset.

Model	Year	Problem encoding	Accuracy
DeductReasoner [5]	2022	No	0.473
Roberta-Graph2Tree [10]	2021	No	0.438
Roberta-GTS [10]	2021	No	0.410
CodeGen-16B (Ours)	2023	**Yes**	0.391
Graph2Tree [12]	2020	No	0.365
BERT-Tree [8]	2021	No	0.324
GTS [11]	2019	No	0.308
Roberta-Roberta [6]	2022	No	0.303
BERT-BERT [6]	2022	No	0.248
GroupAttn [7]	2022	No	0.215

4 Conclusions

In this work, we presented a method for solving MWPs, expressed in natural language, by using Large Language Models to produce Python source code that can solve the problem and can automatically convert it to the internal representation of Intelligent Tutoring Systems. Apart from the automated solving of MWPs, the proposed approach is also capable of naming the quantities in the MWP. Together, they allow the translation of problem statements into the ITS' internal knowledge representation schema, thus allowing learners to add new MWPs in an ITS and tutors to add new MWPs at scale. Our experimental evaluation showed that Saleforce's CodeGen model with 16B parameters achieved the highest accuracy (39.1%) for the proposed approach on the SVAMP MWP dataset. In addition, despite some state-of-the-art MWP solving methods achieving higher accuracy, they are not able to encode the MWP in a form that would enable automated import to an ITS, thus they cannot replace the proposed method.

The success of the proposed approach relies heavily on the performance of code generation models, as shown by the significant performance differences between the examined CodeGen and InCoder models. However, the observed correlation between the size of the model and performance suggests that the quality of the generated solutions will improve with the future development of more advanced models with a larger number of parameters.

Nevertheless, the proposed solution has some limitations. Since it uses plain Python, it is not capable of proposing solutions to problems that cannot be arithmetically solved. In addition, it only generates one solution graph per problem. Although this is in general the most obvious solution, it is not necessarily the only one. Future work will seek to find strategies to deal with these weaknesses. Additionally, it would be interesting to study the different generated source code snippets in order to synthesise all the possible resolution paths to a given problem.

Acknowledgements This research has received support from project TED2021-129485B-C42/C43, funded by the Ministry of Science and Innovation (Strate-

gic Projects Focused on the Green and Digital Transition); project PGC2018-096463-B-I00, funded by MCIN/AEI/10.13039/501100011033 and "ERDF A way of making Europe"; project AICO/2021/019 and Grupos de Investigación Emergentes, funded by Generalitat Valenciana; MS'22-24 Grant, awarded by University of Valencia funded through NextGenerationEU funds; and project "AGENCY", funded by the Engineering and Physical Sciences Research Council [EP/W032481/1], United Kingdom.

References

1. Arnau, D., Arevalillo-Herráez, M., González-Calero, J.A.: Emulating human supervision in an intelligent tutoring system for arithmetical problem solving. IEEE Transactions on Learning Technologies **7**(2), 155–164 (2014). https://doi.org/10.1109/TLT.2014.2307306
2. Bobrow, D.G.: Natural language input for a computer problem solving system. Tech. Rep. AIM-066, Massachusetts Institute of Technology (1964)
3. Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.t., Zettlemoyer, L., Lewis, M.: Incoder: A generative model for code infilling and synthesis. arXiv:2204.05999 (2022)
4. Jie, Z., Li, J., Lu, W.: Learning to reason deductively: Math word problem solving as complex relation extraction. In: Annual Meeting of the Association for Computational Linguistics (2022)
5. Jie, Z., Li, J., Lu, W.: Learning to Reason Deductively: Math Word Problem Solving as Complex Relation Extraction. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics. pp. 5944–5955 (2022)
6. Lan, Y., Wang, L., Zhang, Q., Lan, Y., Dai, B.T., Wang, Y., Zhang, D., Lim, E.P.: Mwptoolkit: An open-source framework for deep learning-based math word problem solvers. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 13188–13190 (Jun 2022). https://doi.org/10.1609/aaai.v36i11.21723
7. Li, J., Wang, L., Zhang, J., Wang, Y., Dai, B.T., Zhang, D.: 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. pp. 6162–6167. Florence, Italy (Jul 2019). https://doi.org/10.18653/v1/P19-1619
8. Li, Z., Zhang, W., Yan, C., Zhou, Q., Li, C., Liu, H., Cao, Y.: Seeking patterns, not just memorizing procedures: Contrastive learning for solving math word problems. arXiv preprint arXiv:2110.08464 (2021)
9. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: Codegen: An open large language model for code with multi-turn program synthesis. ArXiv preprint, abs/2203.13474 (2022)
10. 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. pp. 2080–2094 (2021)
11. Xie, Z., Sun, S.: A goal-driven tree-structured neural model for math word problems. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence. pp. 5299–5305 (2019)
12. Zhang, J., Wang, L., Lee, R.K.W., Bin, Y., Wang, Y., Shao, J., Lim, E.P.: Graph-to-tree learning for solving math word problems. In: Proc. of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 3928–3937 (2021)