Evaluating Code Generation of LLMs in Advanced Computer Science Problems

Emir Catir*[1] [0009-0001-6314-5720], Robin Claesson*[1] [0009-0003-5083-582], and Rodothea Myrsini Tsoupidi[2] [0000-0002-8345-2752]

[1] Royal Institute of Technology, Stockholm, Sweden
`{robcla,catir}@kth.se`
[2] Independent Researcher, Stockholm, Sweden
`rtsoupidi@acm.org`

Abstract. Large Language Models (LLMs), such as GitHub Copilot and ChatGPT have become popular among programming students. Students use LLMs to assist them in programming courses, including generating source code.

Previous work has evaluated the ability of LLMs in solving introductory-course programming assignments. The results have shown that LLMs are highly effective in generating code for introductory Computer Science (CS) courses. However, there is a gap in research on evaluating LLMs' ability to generate code that solves advanced programming assignments. In this work, we evaluate the ability of four LLM tools to solve programming assignments from advanced CS courses in three popular programming languages, Java, Python, and C. We manually select 12 problems, three problems from introductory courses as the baseline and nine programming assignments from second- and third-year CS courses. To evaluate the LLM-generated code, we generate a test suite of 1000 test cases per problem and analyze the program output.

Our evaluation shows that although LLMs are highly effective in generating source code for introductory programming courses, solving advanced programming assignments is more challenging. Nonetheless, in many cases, LLMs identify the base problem and provide partial solutions that may be useful to CS students. Furthermore, our results may provide useful guidance for teachers of advanced programming courses on how to design programming assignments.

Keywords: large language models · programming assignments · computer science · advanced courses

1 Introduction

In recent years, advances in machine learning have enabled high-quality analysis of natural language for diverse purposes, including chatbots, image, and code generation. Textual Large Language Models (LLMs) are generative machine-learning models that use large quantities of data during the training process.

* Both authors contributed equally

Given a prompt, namely a text sequence, the trained model generates a text response that is likely to follow the provided prompt.

Many LLMs have the ability to generate code when the data they are trained on includes code examples. Automatic Code Generators (ACGs), such as Github Copilot[3], are specialized LLMs that are trained for source-code generation. Chatbots, such as ChatGPT[4] and Hugging Face chats[5] are general-purpose LLMs that may be trained on code examples. Many students use different LLM tools, including chatbots and ACGs to receive assistance for their programming assignments [2, 5, 9, 10, 12]. A survey by Keuning et al. with 264 responses shows that the use of LLM tools among students has increased in recent years [10]. As a response, many teachers have started including LLMs in their educational plan or adopt policies to ensure that the students receive the required education.

Previous research has shown that LLM tools are highly capable at solving introductory programming tasks [3, 6, 7] and perform well in introductory Computer Science (CS) courses [16]. In particular, given a prompt that describes the program to implement in natural language, LLM tools can generate functional code after one or multiple attempts [3, 6, 7].

Recent work has shown that advanced programming students believe that the use of LLM tools assists them but they also face problems such as inaccurate or incorrect answers [1, 4, 11]. In this work, we investigate how LLM tools handle advanced programming assignments. In particular, we focus on programming assignments from second and third-year university courses to investigate how LLM tools handle them. Typically, these problem descriptions aim to train the problem-solving abilities of the students and describe an everyday-life problem that the students need to solve. The question here is whether LLM tools can decipher a natural-language described programming assignment, recognize the algorithmic problem to solve, and generate functional code that solves the problem.

More specifically, in this work, we evaluate the ability of a set of publicly available LLM-based tools to generate code for a selected set of nine programming assignments from advanced CS courses. We ask the LLMs to generate code in three popular programming languages, Java, Python, and C.

We pose the following research questions to guide our research:

RQ1: How effective are LLM tools at solving advanced programming assignments correctly?
RQ2: What is the capability of LLM tools at identifying the problem to solve?
RQ3: How does the choice of LLM tool and programming language affect source-code generation?

All evaluation data and prompts used for generating code are available at `https://github.com/Emir-Catir-and-Robin-Claesson/publish`.

[3] Copilot: `https://github.com/features/copilot`
[4] ChatGPT `https://chat.openai.com/chat`
[5] Hugging Face: `https://huggingface.co/chat`

2 Related Work

Evaluating LLMs in CS courses. In a broad scope, Šavelka et al. evaluate the performance of GitHub Copilot at solving multiple computer education tests, including multiple-choice questions and programming exercises in Python. Their findings show that while the ACG model fails to pass the course, it achieves high score in most exercises and manages to improve the answers based on the auto-grader feedback [16]. In a different approach, Reeves et al. evaluate the performance of Github Copilot at solving Parsons Problems, namely programs where the program code is given in incorrect order. They show that Copilot can solve 80% of the problems in Python if ignoring indentation errors [14].

Evaluating LLMs in introductory programming assignments. Denny et al. investigate the solving ability of GitHub Copilot on a large set of simple programming exercises in Python. When the LLM assistant fails to generate a correct solution, the authors try to provide more clear instructions in natural language (prompt engineering). In total, Copilot fails to provide correct solution to 20% of the problems [6]. Finnie-Ansley et al. assess the ability of GitHub Copilot to solve introductory programming problems, such as the Rainfall problem [15]. Their work includes evaluating multiple generated programs for a set of programming tests in a programming course. Their findings show that GitHub Copilot received high scores on the programming tests outperforming most of the students [7].

Evaluating LLMs in advanced programming assignments. Michailidis et al. investigate the use of LLMs for the automatic transformation of textual problem descriptions into concrete Constraint Programming (CP) specifications[6]. Their experiments include a set of exercises from a CP course and show that the assisted LLM can solve from 65% for easier problems up to 35% for the more complex assignments. Finnie-Ansley et al. investigate the performance of and ACG, OpenAI Codex, at solving exams questions of a CS2 course, *Data structures and Algorithms*. Their results show that the ACG performs better than students in this course Finnie-Ansley et al.. Instead, in this paper, we evaluate different ACGs on more advanced courses, namely CS4 and CS5.

3 Motivating Example

Consider the *bin-packing* problem, a famous NP-complete problem where the objective is to place a set of weighted items in the minimum number of bins of specified capacity. Figure 1 shows the high-level description of the bin-packing problem that was given during a lecture in a CS5 course, *Algorithms, Data Structures, and Complexity.*

We want to investigate, whether LLM tools are able to 1) identify the problem and 2) generate correct code to solve or approximate the problem. For example, identifying the problem as the *bin-packing* problem is a significant assistance

[6] Constraint programming is a method for solving combinatorial problems.

A person is moving out of their house and need to pack all their belongings into boxes. They have an infinite number of boxes available, but want to use as few as possible. The person has a list of all their items that need to be packed in boxes. All boxes have the same weight capacity.

Fig. 1: Problem description prompt for the `boxes` problem (Table 1).

to a student. Apart from identifying the problem, an LLM tool may be able to generate code that solves the problem correctly, in our case, to find the optimal solution to the bin-packing problem.

When we provided the problem prompt in Figure 1 to Github Copilot, the tool generated the code in Listing 1. The code defines a function `pack_items`, which takes as input the capacity of the boxes, `weight_capacity`, and the list of `items` that need to fit in the boxes. In the generated code snippet, line 1 defines the method, and lines 2-3 declare needed variables. The generated algorithm (lines 5-15) uses a straightforward heuristic: all listed items are sequentially assigned into a box (lines 9 to 10). When adding an item to the current box surpasses the weight limit (line 6), the algorithm assigns the item to a new box (line 7 and 8). This solution is a greedy heuristic and may return the optimal solution in some cases but does not find the optimal solution in the general case.

It is worth mentioning that we attempted to modify the original prompt, with clear indications that the tool should use a *complete* algorithm or use Constraint Programming (CP), a combinatorial solving technique, with no success. In the first case, the generated code was again a heuristic, and in the second case, the generated code was incorrect. Another interesting result is that all the tested LLM tools in all three languages we tested, Java, Python, and C, produce similar heuristics.

To evaluate the solution in Listing 1, we generated 1000 random problem instances and compared the answers of the tool-generated solution with our reference solution. Out of the 1000 problem instances, the tool-generated code produces 754 correct answers. Although the generated solution is not correct, the code is still interesting because it partially solves the problem and may provide a significant assistance to students.

To identify similar cases, where an LLM has solved part of the problem, we introduce the measure of *accuracy* that measures the rate of correctly verified test cases over the total number of test cases. We define this metric in Section 4.4.

4 Methodology

Figure 2 shows the overview of the methodology, which consists of two main parts 1) code generation and 2) code verification part.

The code generation part includes selecting the LLM tools (Section 4.1) and the problems (Section 4.2) to evaluate. For each of the problems, we create a

```python
def pack_items(weight_capacity, items):
  boxes_needed = 0
  current_box_weight = 0

  for item_weight in items:
    if current_box_weight + item_weight > weight_capacity:
      boxes_needed += 1
      current_box_weight = item_weight
    else:
      current_box_weight += item_weight

  if current_box_weight > 0:
    boxes_needed += 1

  return boxes_needed;
...
```

Listing 1: Python code generated by Github Copilot for the `Boxes` problem.

prompt and provide this prompt as input to each LLM tool to generate source code (Section 4.3).

The code verification part includes evaluating the generated solutions against our reference solutions using a test suite that we generate per problem (Section 4.4).

4.1 LLM Selection

The selection of LLM tools to evaluate is based on the purpose of this study to evaluate code generation and the tools that are widely accessible to students. ACGs are specially trained to generate code, thus the evaluation uses two ACGs, Github Copilot and CodePal[7]. GitHub Copilot is a code-generating tool that integrates with popular IDEs such as Intellij and Visual Studio Code. CodePal is based on various OpenAI models, mainly on GPT-3.5 and GPT-4, and is a tool that focuses on code generation. Students use also chatbots to generate code [10]. To investigate general-purpose chatbots, we select two freely available chatbots in Hugging Face, *Llama3-70b*, with 70 billion parameters, and *Mistral-Nemo-Instruct-2407*, with 12 billion parameters.

4.2 Problem Selection

We handpicked programming assignments and problem descriptions from two CS degree programs. Table 1 shows an enumeration of the problems (first column), a short name describing the problem (second column), the CS level course that

[7] CodePal: `https://codepal.ai`

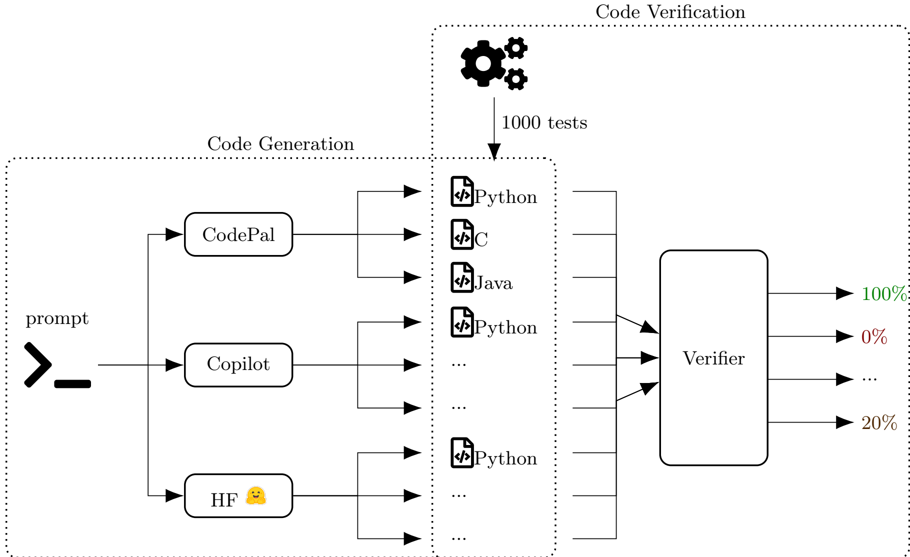

Fig. 2: Code Generation and Verification

Table 1: Selected problems for evaluation. *PN* stands for the problem number, *Level* shows Computer Science course level, and *DA* stands for online description availability. Column *Comments* describes the algorithm of each problem.

PN	Problem	Level	DA	Comments
P1	Temperature	CS1	✓	Rainfall problem [15] variant
P2	Sorting	CS1	✗	Sorting of string and numeric values
P3	Shortest Path	CS1	✓	Minimum path via intermediate stations
P4	Boxes	CS5	✗	Bin-packing problem
P5	TSP	CS5	✗	Traveling salesman problem
P6	Outlets	CS5	✗	Minimum spanning tree
P7	Cow Game	CS5	✓	Shortest path in grid with obstacles
P8	Dice	CS5	✓	Shortest path in graph given allowed moves
P9	Traveling	CS5	✓	All shortest paths with constraints
P10	TV-Zapping	CS5	✓	Minimum increment to match inputs with constraints
P11	Cut Boards	CS4	✓	Dynamic Programming
P12	Train Shunting	CS4	✓	Similar to Tower of Hanoi

the problem corresponds to (third column), and finally, whether the description is available online (forth column). The fifth column provides a brief description of the underlying algorithm.

First, we selected three problems at the introductory CS level as the baseline. Two of the problems, P1 and P3 are lab assignments from a CS1 course, *Programming I* at KTH Royal Institute of Technology in Sweden. P1, `Temperature`, is a variant of the Rainfall problem [15]. P3 To minimize the chance that the

LLM that we use is trained on the actual problem solutions, we introduced an additional problem that was handwritten by two of the authors of this paper and consists of sorting a list of people.

Subsequently, we selected a set of nine problems in advanced programming courses, in particular, CS4 and CS5 courses. The selected problems have clear textual descriptions that allow the selected LLMs to find solutions. Three of the problems, `Boxes`, `TSP`, and `Electrical Outlets` are part of the teaching material, including lecture notes and assignments, in a CS5 course, *Algorithms, Data Structures, and Complexity* taught at KTH Royal Institute of Technology. `Boxes` is an instance of the *bin-packing* problem (see Section 3), `TSP` is an instance of the traveling-salesman problem, and `Outlets` is a minimum-spanning tree problem.

Four of the problems, the `Cow Game`, `Dice`, `Traveling`, and `TV-Zapping` are taught in a CS5 course on *Programming Languages I* at the National Technical University of Athens in Greece. The *TSP* was rewritten as a teacher handing out tests in a classroom rather than a salesman traveling between cities, as it is a classic CS problem with many available implementations. `Cow Game` is a path-finding problem in a grid under time-induced constraints. `Dice` is a shortest-path graph problem with the constraint of each move having to use specific predefined dice values. `Traveling` is a shortest-path problem from a source to all destinations given that the last edge of the shortest path to each destination becomes unusable. `TV-Zapping` is the problem of converging a number of values (channels) to the same value (channel) given the constraint than one channel cannot change two consecutive times. For these problems the assignment descriptions are available online, but we were not able to find any of the solutions in GitHub or other repositories. It is worth noting that the teachers of this course specify that the students should not upload their solutions online.

The last two problems, `Cut Boards` and `Train Shunting`, are problems that are part of a CS4 course, *Programming II* at KTH. `Cut Boards` is the problem of finding the optimal cutting order of a board in smaller parts with goal to minimize the cutting cost. The problem is designed to be solved with dynamic programming, but can also be solved with a greedy algorithm. `Train Shunting` is a problem of performing a set of moves to change the order of wagons in a train. Given three stations, a number of wagons and their initial and desired order, as well as a description of the allowed moves, the task is to rearrange the train by only performing valid moves. The problem is similar to the famous Tower of Hanoi problem, and the constraint lies in rearranging the train wagon order, only using valid moves.

4.3 Code Generation

The next step in the evaluation is creating the prompt text from the selected problems and requesting the LLM tools to generate code.

The prompts for every problem description (LLM prompt) consist of three parts. The first part is the problem description, explaining the problem that the generated code should solve. Figure 1 shows the problem description for the

`Boxes` problem. The second part describes how the input text for a problem instance is formatted. For the `Boxes` problem the input part is the following:

> The input will be given to standard input in this order: The first row contains the weight capacity of the boxes. The second row contains the number of items. The following rows contain the weight of each item.

The last part describes the expected output for a problem instance. Standardized input and output for the generated problems allow for automating the testing of the code.

> The output should be printed to standard output in this order: The number of boxes needed to carry all the items.

Figure 2 shows the code generation part in the left most box. We provide the three-part prompt as input to the selected LLMs, GitHub Copilot (Copilot in Figure 2), CodePal, and two models, Llama3-70b and Mistral-Nemo-Instruct-2407, from Hugging Face (HF in Figure 2). We ask each of the LLMs to generate code in Python, Java, and C.

Note, that we applied some minor changes to the LLMs-generated solutions to run through the verification process. Table 3 in Appendix A lists all the generated solutions that required changes and the change they required. None of these changes alters the algorithm of the generated code.

4.4 Code Verification

Figure 2 shows the code verification part of the evaluation (right-most box). Given the generated programs by the four LLM tools, we generate 1000 random test cases for each of the twelve problems. The test cases for each of the LLM solutions use the same random seed. We run each ACG-generated program separately for each test case, and the answer given by the program is recorded and verified using our reference solution, the Verifier (Figure 2). From the collected answers, we report how many test cases were correct for each generated program and how many cases where wrong. The output of the Verifier is the *accuracy* of the generated code based on Definition 1.

Definition 1 (Accuracy). *Accuracy is the number of correctly solved test cases s, divided by the total number of test cases, t expressed as Accuracy* $= \frac{s}{t}$. *In this experiment* $t = 1000$.

In the `Boxes` problem (Section 3), the accuracy of the GitHub Copilot solution is 75.4%. The structure of the verification algorithm varies depending on the type of problem. Certain problems have a single correct answer. For these problems, the verification consists of calculating the correct answer from the input data and comparing the LLM result to the correct answer. To account for rounding errors in floating point results, the Verifier accepts any answer up to one decimal point far from the correct answer as correct. Other problems have many possible valid answers, such as the Traveling Salesman Problem (TSP) problem. For these problems, our verification step checks each aspect of the given answer, such as verifying all nodes exist in a path and that bounds are not exceeded. The verification algorithm that checks the `Boxes` problem (see Section 3), uses an exhaustive search to check the solutions that LLMs generate.

We define a problem correctly solved, when the accuracy is 100%.

Definition 2 (Correctness). *A problem is **correct** when the accuracy is 100%, namely Accuracy* $= 1$.

5 Evaluation

Table 2: Evaluation results for each of the problems, LLM tools, and programming languages. GC stands for Github Copilot, CP stands for CodePal, HF stands for Hugging Face, LL stands for Llama, and MS stands for Mistral. RE stands for run-time error, IL stands for infinite loop, and CE stands for compilation error. All values are in %.

PN	Python				Java				C			
	GC	CP	HF		GC	CP	HF		GC	CP	HF	
			LL	MS			LL	MS			LL	MS
P1	100	100	100	RE	96.9	RE	100	96.9	100	100	100	96.9
P2	100	100	100	100	100	0	100	0	100	100	100	0
P3	100	100	100	100	100	100	100	100	100	100	100	100
P4	75.4	71.2	71.2	32.3	71.2	71.2	71.2	32.3	75.4	75.4	71.2	CE
P5	100	0	RE	0	100	0	CE	0	0	0	0	CE
P6	0	0	0	RE	0	RE	0	RE	0	0	0	CE
P7	IL	0	0	7.4	1.7	51.9	0	7.4	77.4	0	0	2.1
P8	23.7	23.7	6.3	RE	23.7	23.7	6.3	RE	23.7	RE	RE	CE
P9	0	13.9	0	0	1.1	0	0	RE	13.9	CE	RE	CE
P10	1.3	96.4	1.4	RE	1.4	1.4	1.4	0.3	96.4	3.0	1.4	CE
P11	100	0	0	0	0	0	0	0	0.1	0	0	CE
P12	0	IL	IL	0	0	IL	CE	0	0	0	CE	0

This section presents the results of the evaluation of the selected LLM tools (see Table 1). Table 2 summarizes the results of the evaluation. For each of problems, P1 to P12, Table 2 shows the accuracy (see Definition 1) of the generated source code for each of the targeted programming languages and each of the LLM tools. To evaluate **RQ1**, we compare the results that are correct in Table 2. To evaluate **RQ2**, we examine how good are the LLM tools at recognizing the algorithm or parts of the algorithm for each problem. To evaluate **RQ3**, we compare the results for the different programming languages anad LLM tools.

5.1 RQ1: How effective are LLM tools at solving advanced programming assignments correctly?

The results in Table 2 show the accuracy of each generated problem. The problems that are *correct* (see Definition 2) are colored green in Table 2. The results in Table 2 show that there is a clear difference between the baseline CS1 problems, P1 to P3, and the problems from more advanced courses, P4 to P12. In particular, for P1, we have six out of the twelve solutions correct, and three of them have accuracy 96.9%. The lower accuracy in the last three cases depends on wrong usage of the minimum value for `double` numbers in Java and C, where the value is 0 and not `-inf` as the algorithm requires. For problem P2, which sorts a combination of strings and integers, the failed cases depend on input read mistakes for CodePal and Mistral in Java and wrong sorting order for Mistral in C. Problem P3 is correct for all LLMs.

Among P4 to P12, only P4, `TSP` and P11, `Cut Boards`, have two and one correct results, respectively. In all these cases, Github Copilot is the LLM that generates correct results. For the `Cow Game` problem, Github Copilot's solution for C gives accuracy 77.4%. Listing 3a shows the error of the generated solution and Listing 3b shows the corrected solution. This error depends on wrong intitialization of the input data. In particular, assignment `grid[x-1][y] = t;` that sets the time unit at which the specific grid position is marked should instead be `grid[x-1][y] = min(grid[x-1][y], t);` to consider possible previous markings.

```c
for (i = 0; i < n; i++) {
  ...
  grid[x][y] = t
  if (x > 0)
    grid[x-1][y] = t;
  ...
}
```

(a) Generated Code

```c
for (i = 0; i < n; i++) {
  ...
  grid[x][y] = min(grid[x][y], t);
  if (x > 0)
    grid[x-1][y] = min(grid[x-1][y], t);
  ...
}
```

(b) Corrected Code

Fig. 3: `Cow Game`: Generated code by Github Copilot in C and corrected code.

Summary The ability of LLMs to generate correct solutions in advanced CS problems compared to CS1 problems is low. Among all LLMs Github Copilot is the only tool that is able to generate correct results.

5.2 RQ2: What is the capability of LLM tools at identifying the problem to solve?

This research question investigates the ability of LLM tools to identify the problem that the prompt describes or provide useful information and partial solutions to students. In Table 2, we focus on the results that have partially correct answers or accuracy lower than 100% but higher than 0%.

For problem P4, the `Boxes` problem (see Section 3), all LLM solutions in Table 2 are implementations of heuristics that approximate the result. The LLMs seem to recognize the problem but only provides suboptimal greedy solutions rather than following the prompt instructions.

When examining the solutions for P6, the `Outlets` problem, we can see that some solutions have functions and comments that refer to minimum spanning tree (MST), which is the correct algorithm to solve this problem. Copilot's solutions in Python and C both reference MSTs and all three solutions generated by Llama contain a function named *prim*, a reference to Prim's algortihm, which finds the MST. This indicates that these solutions have correctly identified the underlying problem. However, none of them is able to solve any instance of the problem correctly. We found that this was mostly due to the specific constraints of the tree's intended root as specified in the prompt.

Problem P7, the `Cow Game` problem, has two solutions, one from Github Copilot and the other from CodePal with accuracy above 50%. The Java solution from CodePal finds the shortest path to the destination cell in a grid, without considering the obstacles. In Github Copilot's C solution, the algorithm is correct, but when reading the input code, the implementation overwrites obstacles (see Listing 3a).

Is P8, the `Dice` problem, seven solutions achieve an non-zero accuracy > 0 and five solutions achieve 23.7% accuracy. The latter solutions find the shortest path to the destination node in a graph without considering the constraints implied by the `dice`, namely the allowed steps.

Problem P9, `Traveling` problem has two solution that achieve 13.9% accuracy. Here, both solutions implement the same algorithm that finds the second shortest paths. However, the problem requests for the shortest path after removing one edge of the initial graph that belongs to the original shortest path. Similar to the previous cases, here, the two LLMs recognize the algorithm but ignore some more problem specific constraints and instructions.

Problem P10, `TV-Zapping` problem, includes two solutions that have a very high accuracy, 96.4%. These solutions solve the main problem, but miss a constraint given in the prompt. That is, it is not allowed to increase one of the inputs two times subsequently.

Problem P11, `Cut Board`, was an exercise in dynamic programming. Almost all LLMs try to solve the problem using dynamic programming, but fail to generate correct code to achieve this. The only solution that is correct uses a greedy algorithm.

Finally, in problem P12, or `Train Shunting`, which is similar to `The Towers of Hanoi` problem, none of the LLMs solves the problem and many LLMs generate solutions that result in infinite loops or compilation errors.

Summary The evaluation shows that many LLMs generate solutions that use heuristics instead of full solutions, solve the problem ignoring some constraints, or recognize the algorithm but fail in the implementation (e.g. Dynamic Programming). This is an insight that may be useful for teachers that want to design problems that require understanding and personal effort from the students.

5.3 RQ3: How does the choice of LLM tool and programming language affect source-code generation?

The choice of language affects the result to some extent. In Table 2, LLM tools fail to generate functional C code. In particular, 12 solutions result in either compilation or runtime errors. However, Github Copilot (GC) provides partial solutions in C for P7 to P10. Python and Java have eight solutions each that result in errors, and different LLMs produce non-runnable code for P1. In terms of generating partly correct solutions for advanced problems, P4 - P12, we look at solutions with accuracy $> 5\%$. LLMs produced twelve solutions with Python, ten with Java, and seven with C that partially solved the problems.

To compare the LLM tools, we see in Table 2 that GC performs the best, followed by CodePal (CP). Out of 36 solutions, GC produces only a single non-functional solution, this is for P7, where GC's Python solution results in an infinite loop. Solutions produced by CP are failing to a larger extent, while Llama and Mistral generate a larger number or non-functional solutions. Another interesting insight is that it seems that CP in Python generates the same heuristic as CG in C, for problems P8 - P10.

6 Limitations and Future Work

In this paper, we have not performed extensive prompt engineering but rather used the original or slightly modified exercises as provided in the respective course. It is possible that prompt engineering may improve the results of LLMs in advanced courses. We leave this as a future work.

The development and improvement of LLM tools is constant, which requires continuous effort to evaluate their capabilities and the effect of their use on CS education. We believe that there is a need to adjust CS course assessments for a fair assessment of the students' knowledge.

7 Conclusion

In this paper, we investigate the capabilities of LLM tools to solve advanced programming assignments. Our results indicate that compared to introductory course assignments, LLM tools struggle to generate correct source code for advanced assignments. However, the tools are often able to recognize the algorithm needed to solve the assignment. Our analysis on partially correct results shows that LLMs have difficulty adjusting known algorithms to specific constraints and often tend to implement popular heuristics instead of following the prompt instructions. We believe that teachers may use these insights to assess the knowledge of students in advanced CS courses by incorporating variations or constraints to known algorithms in advanced programming assignments.

Bibliography

[1] Arora, C., Venaik, U., Singh, P., Goyal, S., Tyagi, J., Goel, S., Singhal, U., Kumar, D.: Analyzing llm usage in an advanced computing class in india. arXiv preprint arXiv:2404.04603 (2024)

[2] Becker, B.A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., Santos, E.A.: Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, pp. 500–506 (2023)

[3] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)

[4] Choudhuri, R., Ramakrishnan, A., Chatterjee, A., Trinkenreich, B., Steinmacher, I., Gerosa, M., Sarma, A.: Insights from the frontline: Genai utilization among software engineering students. arXiv preprint arXiv:2412.15624 (2024)

[5] Cipriano, B.P., Alves, P.: "chatgpt is here to help, not to replace anybody"–an evaluation of students' opinions on integrating chatgpt in cs courses. arXiv preprint arXiv:2404.17443 (2024)

[6] Denny, P., Kumar, V., Giacaman, N.: Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, pp. 1136–1142 (2023)

[7] Finnie-Ansley, J., Denny, P., Becker, B.A., Luxton-Reilly, A., Prather, J.: The robots are coming: Exploring the implications of openai codex on introductory programming. In: Proceedings of the 24th Australasian Computing Education Conference, pp. 10–19 (2022)

[8] Finnie-Ansley, J., Denny, P., Luxton-Reilly, A., Santos, E.A., Prather, J., Becker, B.A.: My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises. In: Proceedings of the 25th Australasian Computing Education Conference, pp. 97–104 (2023)

[9] Kasneci, E., Seßler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., Groh, G., Günnemann, S., Hüllermeier, E., et al.: Chatgpt for good? on opportunities and challenges of large language models for education. Learning and individual differences **103**, 102274 (2023)

[10] Keuning, H., Alpizar-Chacon, I., Lykourentzou, I., Beehler, L., Köppe, C., de Jong, I., Sosnovsky, S.: Students' perceptions and use of generative ai tools for programming across different computing courses. In: Proceedings of the 24th Koli Calling International Conference on Computing Education Research, pp. 1–12 (2024)
[11] Korpimies, K., Laaksonen, A., Luukkainen, M.: Unrestricted use of llms in a software project course: Student perceptions on learning and impact on course performance. In: Proceedings of the 24th Koli Calling International Conference on Computing Education Research, pp. 1–7 (2024)
[12] Margulieux, L.E., Prather, J., Reeves, B.N., Becker, B.A., Cetin Uzun, G., Loksa, D., Leinonen, J., Denny, P.: Self-regulation, self-efficacy, and fear of failure interactions with how novices use llms to solve programming problems. In: Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, pp. 276–282 (2024)
[13] Michailidis, K., Tsouros, D., Guns, T.: Constraint Modelling with LLMs Using In-Context Learning. In: Shaw, P. (ed.) 30th International Conference on Principles and Practice of Constraint Programming (CP 2024), Leibniz International Proceedings in Informatics (LIPIcs), vol. 307, pp. 20:1–20:27, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024), ISBN 978-3-95977-336-2, ISSN 1868-8969, https://doi.org/10.4230/LIPIcs.CP.2024.20, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.20
[14] Reeves, B., Sarsa, S., Prather, J., Denny, P., Becker, B.A., Hellas, A., Kimmel, B., Powell, G., Leinonen, J.: Evaluating the performance of code generation models for solving parsons problems with small prompt variations. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, pp. 299–305 (2023)
[15] Soloway, E.: Learning to program= learning to construct mechanisms and explanations. Communications of the ACM **29**(9), 850–858 (1986)
[16] Šavelka, J., Agarwal, A., Bogart, C., Song, Y., Sakr, M.F.: Can generative pre-trained transformers (gpt) pass assessments in higher education programming courses? Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (2023)

A Changed solutions

We made some small alterations to the LLM-generated solutions before the tests. Changes to a solution depend on one of the following three reasons, 1) to make the program running, 2) the generated solution contained a running algorithm but required minor changes to the output to pass the verification algorithm, and 3) to lower the runtime of the programs. No changes were made to the algorithms of the solutions.

Table 3 list all solutions that had some changes made before running the tests. There were four different types of changes made. The most common change was changes to how the solution printed the answer to a problem instance, as the LLMs often include descriptions in their outputs. Another change regarding the solution outputs was from floating point to integer variables. This made the output of the solutions integers instead of .0 vales, aiding parsing in the verification of the output. The change that occurred on multiple solutions was lowered constants to reduce the runtime of the solutions, most common in the P7. These programs did extensive searches on grids larger than needed, causing a long runtime that was cut down by lowering these limits closer to the maximum test values.

Table 3: Changes made for LLM generated solutions before running the evaluation tests. Output format indicate changes to the result printing. Float to int indicate the solution used floating point variables when only integer was needed. Lowered constants indicates values being lowered to reduce search sizes. Added imports indicate that some missing imports needed for compilation was added. Lastly main() call indicates a call to the main funtion was added.

Problem	**LLM**	**Language**	**Change**
P1	Codepal	C	Output format
P2	Llama	C	Float to int
P2	Llama	Java	Float to int
P2	Llama	Python	Float to int
P2	Mistral	Python	Float to int
P3	Copilot	C	Output format
P3	Copilot	Python	Output format
P3	CodePal	C	Output format
P3	CodePal	Java	Output format
P3	CodePal	Python	Output format
P3	Mistral	Python	main() call
P7	Copilot	C	Lowerd constants
P7	Codepal	C	Lowerd constants
P7	Llama	C	Lowerd constants
P7	Llama	Java	Lowerd constants
P7	Llama	Python	Lowerd constants
P9	Copilot	C	Lowerd constants
P9	Mistral	Java	Added import
P9	Llama	Python	Output format
P10	CodePal	C	Output format