



UNIVERSITY
OF SKÖVDE

Evaluating large language models' capability to generate algorithmic code using prompt engineering

Bachelor Degree Project in Information Technology
Basic level 30 ECTS
Spring 2024

Sebastian Karlsson, Mtanus Farah, Faysal Hassan

Supervisor: Márki, András
Examiner: Bae, Juhee

Summary

The study evaluated the performance of large language models (LLMs) such as Gemini, ChatGPT-4, and GitHub Copilot in generating C++ algorithms for specific tasks using different prompting techniques. The central aim was to assess the effectiveness of these models in creating code solutions that are both functionally correct and complete, using a combination of automated unit tests and human evaluation. Across two main tasks (Social Network and Huffman Encoding), the models showed different levels of success in generating functionally correct code. Github Copilot and ChatGPT-4 generally produced more syntactically accurate and functionally appropriate code than Gemini. There was a notable variation in completeness, whether the code met all the tasks' specified requirements. Some models managed to include all necessary functionalities more consistently than others. Gemini, for instance, excelled in generating complete solutions for the Social Network task but had issues with the Huffman Encoding task, where its output often did not integrate the provided code effectively or correctly.

keywords: LLM, large language model, ChatGPT, Gemini, Github Copilot, Prompt Engineering, Algorithm

Contents

1	Introduction	1
2	Background	2
2.1	Natural language processing	2
2.2	Large Language Models	2
2.2.1	Generative Pre-trained Transformer (GPT)	2
2.2.2	ChatGPT	3
2.2.3	Gemini	4
2.2.4	GitHub Co-pilot	4
2.3	Prompt Engineering	4
2.3.1	Instructive Prompt	5
2.3.2	Question-Answer Prompt	5
2.3.3	Contextual Prompt	7
2.3.4	Mixed Prompt	8
2.4	Evaluation Methods	9
3	Problem	10
3.1	Aim	10
3.2	Motivation	10
3.3	Research Question	11
3.4	Hypotheses	11
3.5	Objectives	11
3.6	Work Distribution	11
4	Related Work	13
5	Method	14
5.1	Experiment	14
5.2	Alternative methods	15
5.2.1	Case study	15
5.2.2	Survey	15
5.2.3	Systematic Literature Review	15
5.3	Approach	15
5.3.1	Prompt techniques	15
5.3.2	Measurement metrics	16
5.3.3	Alternative Approach	18

6	Result	19
6.0.1	Objective 1	19
6.0.2	Objective 2	20
6.1	Objective 3	21
6.2	Correctness and Completeness - Social Network	23
6.2.1	LLMs	23
6.2.2	Prompting Techniques	24
6.3	Correctness and Completeness - Huffman Encoding	25
6.3.1	LLMs	25
6.3.2	Prompting Techniques	26
7	Analysis	27
7.1	Social Network Task	27
7.1.1	Github Co-pilot	27
7.1.2	Gemini	27
7.1.3	Chatgpt-4	28
7.1.4	Prompting Techniques	29
7.2	Huffman Encoding	29
7.2.1	Github Co-pilot	29
7.2.2	Gemini	29
7.2.3	Chatgpt-4	30
7.2.4	Prompting Techniques	30
8	Discussion	32
8.1	Summary	32
8.1.1	Correctness of LLMs	32
8.1.2	Completeness of LLMs	32
8.1.3	Prompting Techniques	32
8.2	Generalization	32
8.3	Ethics & Threats to Validity	34
8.3.1	Ethics	34
8.3.2	Threats to Validity	35
8.4	Future Work	37
9	Conclusion	38
A	Appendix	I
A.1	Appendix A	I
A.2	Appendix B	II
A.3	Appendix C	III
A.4	Appendix D	IV
A.5	Appendix E	VI
A.6	Appendix F	IX
A.7	Appendix G	X
A.8	Appendix H	XI
A.9	Appendix I	XII
A.10	Appendix J	XIII
A.11	Appendix K	XIV
A.12	Appendix L	XV

1 | Introduction

Artificial intelligence (AI) in the realm of software development has marked a transformative era, particularly with the integration of Large Language Models (LLMs) like ChatGPT-4, Gemini Advanced, and GitHub Copilot. These models, developed to understand and generate human-like text, have recently been harnessed to tackle complex algorithmic challenges in coding, revolutionizing the traditional programming paradigms.

This research investigates the efficacy of these LLMs in generating executable C++ code for specific tasks—namely, Social Network analysis and Huffman Encoding—utilizing varied prompt engineering techniques. Each task presents unique challenges that test the models' code syntax capabilities and adhere to functional and logical frameworks critical for successful software execution.

Through a structured experimental setup, this study evaluates the functional correctness and completeness of the code produced by the models. Functional correctness refers to the code's ability to execute as intended without errors, while completeness assesses whether the code fulfills all specified requirements of the tasks. The research employs a dual evaluation methodology combining automated unit tests and human assessments of the LLMs' output.

Additionally, the study explores the impact of different prompting strategies on the quality of generated code. These strategies range from simple instructive prompts to complex contextual prompts, examining how effectively each model translates these prompts into functional program code. The use of prompting techniques not only aids in understanding the strengths and limitations of each LLM but also sheds light on the nuances of prompt engineering—a critical aspect of leveraging AI for programming.

The findings of this research are expected to contribute significantly to the fields of AI and software development, providing insights into the potential of LLMs as tools for coding. Moreover, it aims to refine the approach of using AI in educational settings where programming is taught, enhancing learning outcomes and teaching methodologies.

2 | Background

2.1 Natural language processing

Natural Language Processing (NLP) integrates computer science, mathematics, and linguistics to create a multidisciplinary field with various applications, including vocabulary development, text classification, automatic summarization, and social computing Chen (2023). Central to NLP are two main processes: Natural Language Understanding (NLU) and Natural Language Generation (NLG). NLG focuses on the computational generation of natural language, replicating the human language communication process and enabling computers to generate human-like responses. Conversely, NLU involves understanding and constructing meaningful language based on input data by applying specific semantic and grammatical rules, positioning it as a crucial aspect of computational linguistics and artificial intelligence.

2.2 Large Language Models

A large language model (LLM) is a deep learning model designed to process natural language, generate, translate, and summarize text. The model performs these actions based on knowledge gained from massive amounts of data (Lee 2023). LLMs are characterized by their vast size, usually containing hundreds of millions to billions of parameters (Anh et al. 2023). LLMs are one of the most successful applications of transformer models. This architecture employs self-attention mechanisms to process data in parallel, enabling the model to effectively capture long-range dependencies and contextual relationships. These models are used for teaching AIs human language and writing software code and have several other applications (Lee 2023). Recently, the emergence of AI models for code generation has gained considerable interest. Leinonen et al. (2023) explores the capabilities of ChatGPT-3 in composing code explanations and reports that students rated the explanations provided by ChatGPT-3 higher than those written by humans in terms of accuracy and understandability.

2.2.1 Generative Pre-trained Transformer (GPT)

Generative Pre-trained Transformers, commonly known as GPT, are a series of neural network models that utilize the transformer architecture. They represent a significant breakthrough in artificial intelligence and are instrumental in driving generative AI applications, including ChatGPT (Amazon 2024).

GPT models undergo training through a two-phase approach consisting of pre-training and fine-tuning. This approach is commonly used in large language models to enable understanding of

language and subsequently adapt to particular tasks or domains.

During the pre-training, the models are trained with vast amounts of data collected from different sources such as websites, books, and articles. Using this data, the model learns to predict words based on previous words in a process known as masked language modeling. During this phase, the GPT model acquires a significant amount of knowledge by capturing grammar, syntax, and semantics.

GPT models are then, after pre-training, refined to match specific tasks or domains by fine-tuning them on task-specific datasets. This process involves training the model additional times using smaller datasets to specialize in a desired task while maintaining the general knowledge gained during pre-training.

GPT-2 is the successor to GPT, another transformer-based model. In contrast to GPT, it is trained with ten times more data consisting of outbound links from Reddit, which resulted in a training dataset of 10 billion words (Zheng et al. 2021).

2.2.2 ChatGPT

Chat Generative Pre-Trained Transformer is an artificial intelligence chatbot launched in November 2022 by OpenAI. It is a multifunctional conversational chatbot linked to the GPT-3 language model (Prasad et al. 2023).

A Chatbot is an intelligent agent or a clever computer program that can converse with human users and understand their language using natural language processing (NLP). It combines many technologies, models, and algorithms to understand the text and respond accordingly. Chatbots that use NLP can analyze context, identify nuances of human text, and adapt to different conversation styles, allowing for a more personalized interaction with users.

ChatGPT is one such example of an advanced chatbot that is widely used for different applications and tasks. It creates text that closely resembles that of the human language. This close resemblance is achieved through a two-step process: generative and unsupervised learning, which refines its performance for specific tasks

ChatGPT is a part of the NLP generative models and is capable of generating text, images, music, and much more. An AI generative model generates new data based on its knowledge from large datasets. Examples of such models are GPT-2 and GPT-3. The third generation of GPT can generate words based on a given input prompt. It leverages roughly 175 billion parameters and is trained extensively on unlabeled datasets such as text from Wikipedia (Alahmed et al. 2023).

GPT-4 is the latest version of the GPT models released by OpenAI. It can accept images and text as inputs, generating text outputs. While it is less capable than humans in many scenarios, it produces human-level performances in various professional and academic benchmarks. The difference between GPT-3 and GPT-4 is slight in casual conversation. The more significant differences emerge when the complexity of a task reaches a sufficient threshold, in which case GPT-4 outperforms GPT-3.5 in terms of reliability and creativity and can handle more nuanced instructions (OpenAI 2023b).

2.2.3 Gemini

Gemini (formerly Google Bard) is an LLM chatbot by Google. It is trained on massive amounts of text, including code, Wikipedia, Books, Stack Overflow, Google Search, and many other public datasets. Like ChatGPT, it can generate text, translate languages, and create content. The model is free to use and was first released in the US and UK on the 21st of March, 2023. As of September 2023, it is available in over 40 languages and over 230 countries. Gemini is powered by several technologies, including Natural language processing(NLP), Machine learning(ML), and Deep Learning (DL) (Plevris et al. 2023).

2.2.4 GitHub Co-pilot

Github Copilot is an AI assistant tool created to help developers write faster code by offering coding suggestions and code completions. Github Copilot enables integration with the leading code editors, such as Visual Studio, Visual Studio Code, JetBrains, and Neovim (GitHub 2024).

Copilot has been trained on natural language text and source code from publicly available sources, which includes public Github repositories. It is powered by generative AI models by GitHub, OpenAIs GPT-4 (As of the 30th of November, 2023), and Microsoft (GitHub 2024).

Github Copilot also comes with a chat interface allowing interaction with the tool, with the possibility of asking coding-related questions and receiving answers. The chat can be used to prompt Copilot to do various tasks such as generate unit tests, code explanation, code improvements, and much more (GitHubCopilotChat 2024).

Github Copilot has a couple of limitations compared to ChatGPT-4 and Gemini. Copilot can only assist with coding-related topics. This is because Copilot is not a large language model in itself. Copilot's natural language processing abilities are also limited to English when prompting to get the best results (GitHubCopilotChat 2024).

2.3 Prompt Engineering

Prompt Engineering is a method of structuring prompts to receive information as the prompter wants it. A more well-structured prompt would, in turn, when given to an LLM, give a more precise response (Petroni et al. 2020). There are three basic principles to prompt engineering outlined by Whately (2022):

- Being precise in your prompt
- Break down the prompt into multiple parts
- Make the model check its output

It is also important to avoid falling into the following traps when structuring prompts. As prompts play an important role in guiding models to receive the input the user wants, it is important to understand these traps and to keep them in mind when constructing a prompt (Giray 2023).

- **'Ambiguity'**
Do not be vague in the prompts. A clear end goal is needed
- **'Bias Reinforcement'**
Being too specific while also not encouraging discussion
- **'Overfitting'**
By not being too specific when writing a prompt, the LLM has a chance to generate a more diverse response
- **'Lack of Context'**
A model can not know everything. Give it the necessary context to provide the scope and purpose of the question properly
- **'Ethical Considerations'**
Constructing prompts that encourage illegal or illicit activity is heavily discouraged
- **'Unintended Side Effects'**
A focused and coherent prompt is needed. Otherwise, the LLM may generate nonsensical responses.
- **'Unrealistic dependency on model limitations'**
Creating prompts where you expect the model to know everything in the world will not give a satisfying response, nor will it give a correct response

2.3.1 Instructive Prompt

Instructive prompts, as explained by Giray (2023), is a prompt that guides the LLM towards the expected response. Crows and their colors are used as a base to construct prompts according to each technique.

Prompt: Tell me the color of a crow

Response: The color of a crow is primarily black. This includes its feathers, beak, and legs, giving it a distinctive all-black appearance. Some species of crows may have slight variations in color, such as iridescent feathers that can appear to shimmer with bluish, purplish, or greenish hues under certain lighting conditions.

This technique gives a satisfying answer, considering the prompt was simple and straight to the point. An interesting takeaway was that despite only asking for the color, it also explained that crows have slight variations, such as blue, purple, or green iridescent feathers.

2.3.2 Question-Answer Prompt

By structuring a prompt around a question that needs to be answered, the LLM should be able to give a satisfying answer. This also takes into consideration that the question itself does not fall into any of the traps listed before.

Prompt: What color are crows?

Response: Crows are predominantly black in color, including their feathers, beak, and legs. Their

black plumage can exhibit an iridescent sheen, displaying hues of blue, purple, and green under sunlight due to the light reflecting off the structure of their feathers.

This response is on the same level as the instruction prompt. The LLM goes straight to the point and gives what is requested and a little extra.

2.3.3 Contextual Prompt

Adding context to a prompt is expected to make the response much fuller. The LLM can not know everything that is intended in a prompt. Therefore, it is essential to be clear and concise and ask for everything that needs to be included.

Prompt: Considering the color of a crow, analyze why evolution may have chosen this specific color

Response: The predominantly black coloration of crows, as seen in many species within the Corvidae family, can be attributed to several evolutionary advantages:

- 1. **Camouflage and Stealth:** Black provides a level of camouflage in various habitats, from the dark forests to shadows in urban environments. This coloration helps crows blend into their surroundings, making it easier for them to approach prey and avoid predators.*
- 2. **Thermoregulation:** The black coloration helps in thermoregulation. Black feathers absorb more sunlight, which can be advantageous in colder climates by helping the birds to stay warm. Conversely, in warmer conditions, crows can regulate their body temperature by adjusting their feather position and behavior to avoid overheating.*
- 3. **Signal of Fitness:** The intensity of the black coloration in crows can be a signal of individual fitness. Bright, iridescent feathers can indicate a healthy bird, as producing and maintaining such plumage requires energy and good nutrition. This can play a role in mate selection, where healthier individuals are more likely to reproduce.*
- 4. **UV Protection:** The pigments in black feathers can provide protection against the sun's ultraviolet rays. UV radiation can damage feathers, but the melanin that gives feathers their black color can help mitigate this damage, leading to longer-lasting, more durable feathers.*
- 5. **Social Signaling:** Some studies suggest that the iridescence in black feathers can serve as a form of social signaling among crows. The quality of the iridescence may convey information about the individual's age, sex, or social status, which can be crucial in social interactions within their highly intelligent and hierarchical societies.*
- 6. **Resistance to Bacteria and Parasites:** Melanin, the pigment that makes feathers black, has been found to provide resistance against degradation by bacteria and parasites. This means that black feathers not only last longer but also reduce the burden of feather-degrading bacteria and parasites, which can affect the health and flight efficiency of birds.*

In summary, the black coloration of crows is a result of evolutionary pressures that favor camouflage, thermoregulation, signaling fitness, protection against UV radiation, social signaling, and resistance to bacteria and parasites. These advantages contribute to the survival and reproductive success of crows in their respective environments.

This technique inherently asks for more by adding context on what, specifically, is to be answered, and a better response is generated. The essence of the prompt is still the same, where the color of a crow is still asked for, but also why this color could have been chosen in the evolutionary tree.

2.3.4 Mixed Prompt

A mixed prompt is created by adding and combining any of the previous four techniques. This type of prompt is theoretically the most ideal because it provides all the information needed for the LLM to perform its task and for the prompter to receive the information they want.

Prompt: Consider the color of a crow. Analyze why this color may have been chosen during evolution and analyze how albinism may have an effect on the analysis, positive and negative.

Response: The color of a crow, which is typically black, has evolved through natural selection for various adaptive advantages. This coloration can offer camouflage, aid in thermoregulation, protect against UV radiation, and provide resistance to feather degradation. However, the occurrence of albinism—a genetic condition resulting in the lack of melanin, which is responsible for the black coloration in crows—introduces a unique perspective on these evolutionary advantages.

Positive and Negative Effects of Albinism on Crows

Negative Effects:

1. **Camouflage and Predation:** Albinism significantly reduces the camouflage benefits of black coloration. Albino crows are more visible to predators due to their lack of typical black plumage, making them easier targets in their natural environments.
2. **Social Interaction:** Crows are highly social birds, and the stark difference in appearance of an albino crow may affect its interactions within crow societies. There could be social ostracization due to their unusual appearance, potentially impacting their ability to communicate, mate, and participate in communal activities.
3. **Thermoregulation:** Black feathers absorb sunlight, aiding in thermoregulation. Albino crows, lacking this melanin, may struggle more with maintaining their body temperature, particularly in colder environments.
4. **Feather Durability:** Melanin contributes to the structural integrity of feathers. Without it, albino crows may have weaker feathers, leading to quicker wear and tear, affecting their flight and insulation.
5. **UV Sensitivity:** Melanin provides protection against UV radiation. Albino crows are more susceptible to sunburn and potential eye damage due to increased sensitivity to sunlight.

Positive Effects:

The positive effects of albinism in crows are more challenging to identify, as the condition generally presents more disadvantages in the wild. However, a few considerations include:

1. **Human Interest:** Albino wildlife often attracts human interest, which can sometimes lead to protective behaviors from people. In certain cases, this can increase the chances of survival for albino individuals through conservation efforts or in captive environments where predators are absent.
2. **Research and Education:** Albino animals, including crows, can serve as valuable subjects for educational and research purposes. They can help in studying genetics, animal behavior, and the impact of pigmentation on survival and fitness.

3. **Unique Ecology Roles:** *Although significantly challenging, if an albino crow survives to adulthood in the wild, it could play a unique role in its ecosystem, potentially teaching us about the adaptability and resilience of species.*

In essence, while the black coloration of crows is a result of evolutionary processes providing numerous survival benefits, albinism challenges these advantages due to the absence of melanin. The condition highlights the importance of coloration in animal survival strategies and the potential impacts of genetic variations on individual fitness and species adaptation.

This prompt is a mix between context and instruction. As a contextual prompt has been seen to provide what could be argued to be the best response, it was chosen as a base. More context was added to facilitate a better response and mitigate ambiguity, lack of context, bias reinforcement, and overfitting.

2.4 Evaluation Methods

Functional correctness is understood as the extent to which a product or system delivers accurate outcomes when operated by its intended user in accordance with standards established by the International Organization for Standardization/ International Electrotechnical Commission (ISO/IEC).

Completeness refers to a product's ability to offer a comprehensive suite of functions that fulfill all specified tasks and meet the objectives of its users(IEC 2023).

In line with these definitions, unit tests will be used to verify the output's functional correctness. For completeness, we will conduct human evaluations to manually confirm that all necessary functionalities are incorporated into the implementations. Practically, software testing emphasizes the concurrent assessment of correctness and completeness. Completeness ensures the presence of all intended functionalities, but its value is contingent on their correct operation. However, measuring both metrics independently is still possible and adds value to understanding how the models attempt to implement the solutions.

3 | Problem

3.1 Aim

This study aims to compare the performance of different LLMs in solving algorithmic tasks using various prompt engineering techniques. Subsequently, the results from each LLM will be analyzed to determine if there are significant differences in terms of correctness and completeness.

3.2 Motivation

The success of GPT-3 has spurred a surge in interest towards Large Language Models (LLMs) with code generation capabilities. While earlier research primarily focused on generating simple code snippets Yan et al. (2023), LLMs have significant untapped potential to tackle complex coding problems. Their vast capabilities extend beyond basic code generation; exploring this potential is crucial.

Previous work by Yan et al. (2023) has demonstrated the feasibility of LLM-based simple code generation. This study aims to expand on this by investigating an LLM's ability to solve more intricate coding problems. To effectively unlock these capabilities, proper prompt structuring is essential. We propose to explore a set of more straightforward prompting techniques beyond those investigated by Lundblad et al. (2023). Unlike Lundblad et al. (2023), who focused on Java and Python to explore the code generation abilities of ChatGPT-3.5, this study delves into C++. As a syntactically stricter and less abstract language, C++ offers a more challenging test bed for LLMs. Observing how these models handle code generation in a language closer to machine code will be particularly interesting, requiring explicit memory management and offering greater developer control.

LLMs are evolving swiftly and have become increasingly accessible to the general public. The latest iterations of LLMs possess the ability to generate programs and address intricate coding challenges more effectively than their predecessors, using natural language prompts.

In the academic world, educators and researchers globally are evaluating the impact of large language models on education, discussing which of these LLMs are beneficial in an educational setting and which are not. They also believe that, given their public availability, it is crucial to assess their functionality and performance from various perspectives (Prentzas & Sidiropoulou 2023).

3.3 Research Question

RQ1: How do various LLMs compare in terms of the functional correctness of the code generated when using different prompt techniques?

RQ2: How do various LLMs compare in terms of the completeness of the code generated when using different prompt techniques?

3.4 Hypotheses

RQ1:

H_0 : There are no significant differences in either the prompting techniques or the LLMs in terms of functional correctness.

H_1 : There are significant differences in either the prompting techniques or the LLMs in terms of functional correctness.

RQ2:

H_0 : There is no significant difference in the completeness of the generated code across different LLMs.

H_1 : There is a significant difference in the completeness of the generated code across different LLMs.

3.5 Objectives

1. Find LLMs according to selection criteria
2. Define the tasks to be used in the study
3. Study the chosen LLMs
4. Perform the experiment
 - (a) Prompt the LLMs using the chosen techniques
 - (b) Evaluate the output according to our measurement metrics
5. Compare and analyze the differences in the results

3.6 Work Distribution

All participants conducted this study collaboratively, with specific tasks assigned individually. The individual contributions included researching their assigned LLM and prompting that LLM.

1. The selection of LLMs was a collective effort, after which each LLM was assigned to a specific author. The assigned LLMs are as follows:
 - ChatGPT4 - Mtanus Farah
 - Gemini Advanced - Sebastian Karlsson
 - Github Copilot - Faysal Hassan

2. The choice of course and tasks was made collectively.
3. Each author individually studied their assigned LLM, which involved understanding how to use the model for our purposes and reviewing copyright and terms of use.
4. Each author conducted experiments on their assigned LLM, which included prompting the model, gathering data, and evaluating the results according to the measurement metrics. While assistance was available from team members, the experiments were primarily conducted independently.
5. The comparison and analysis of the LLM results were performed individually for the results section. The presentation and discussion of these results were then done collectively.

4 | Related Work

The previous work by Lundblad et al. (2023) was a thesis project titled "The Impact of Task Specification on Code Generated via ChatGPT". Their thesis project explores how different prompt engineering techniques, specifically task specification methods, affect the code generated by ChatGPT in terms of correctness and variety. The study conducted experiments and compared the effectiveness of various task specification methods against a baseline method, evaluating the functional and syntactical correctness of the generated code. The results showed statistically significant differences in some code correctness criteria between the baseline and other methods with variations in the generated code. The study suggests potential areas for future research, including exploring other large language models, programming tasks, programming languages, and prompt engineering techniques.

Yan et al. (2023) mentions that previous work focused on how code generation from LLMs only did simple coding tasks. While Yan et al. (2023) showed how LLMs generated simple code and argued that it is only reasonable to explore the capabilities that LLMs have for more complex code generation.

Prentzas & Sidiropoulou (2023) explains that teachers and researchers in the academic work are assessing the impact that LLMs have on education, Where the discussions are on which LLMs are useful in an educational setting and which are not. Prentzas & Sidiropoulou (2023) says that due to the fact that a lot of LLMs are publicly available, it is important to research their capabilities.

Logozar et al. (2023) investigates ChatGPT's ability to write C/C++ code for solving entry-level programming tasks in a university course. The study focused on using the ChatGPT-3.5 version, which is free to the public. The results show that the model solved most of the tasks correctly, and the study concludes that ChatGPT not only passes the course but does so with excellent grades.

Becker et al. (2023) investigates how an introductory in a computing course looks, assuming that students will be able to easily auto-generate code solutions for their labs and assignments by simply pasting the problem descriptions in an AI-powered tool. They also look at how this affects the computing curricula and believe it poses a challenge to how programming is taught and learned. They conclude that software development in the future will have an increasing amount of auto-generated code and the use of these tools for those training to get a programming job, such as current computer science students.

5 | Method

5.1 Experiment

Experiments allow the researchers to understand the nature of the products and provide models to help solve problems. With an experiment, the researchers can manipulate and create the products. Not all software is identical because various factors influence the outcomes. Development and experimentation are essential to grasping the nature of certain aspects, such as software engineering, leading to differences in the results. (Wohlin et al. 2012). A controlled experiment will be conducted to answer the research questions. Prompt engineering, with four prompting techniques, was used to solve two different tasks. The solutions generated by the different LLMs will then be tested by both human evaluation and unit tests to answer the research questions. Using an experiment allows us to have a specific control of the different variables in the study.

Independent Variable	Values
Large Language Model	ChatGPT4 Gemini Github Co-Pilot
Prompting Techniques	(I) Instructive prompt (QA) Question-Answer Prompt (C) Contextual Prompt (M) Mixed Prompt
Programming Task	(SN) Social Network (HE) Huffman Encoding

Table 5.1: Independent Variables

Dependent Variable	Values
Measurement Metric	Correctness Completeness

Table 5.2: Dependent Variables

This study has three independent variables: the large language models, prompting techniques, and programming tasks. The dependent variables include the evaluation metrics correctness and completeness, which will be used to evaluate the generated code from the prompting techniques. With ten prompts for each technique, there are 80 prompts for each LLM (40 per task).

5.2 Alternative methods

5.2.1 Case study

Case study research is one of the key methods used when conducting empirical research in software engineering. According to Bass et al. (2018), it can be defined as exploring a phenomenon in its natural context, particularly the lines between the phenomenon and its context blur. Using this approach generates fresh hypotheses and helps evaluate the existing ones. They offer a structured approach to gathering data, analyzing, and sharing the findings. Researchers use case study methods mostly to enhance their understanding of the reasons behind a particular occurrence (Bass et al. 2018). A case study is, however, not a viable option for us due to the variability of prompt engineering, and we would not be able to capture the full effects of different LLMs. Our research aims to assess performance metrics such as correctness and completeness across different models, which are not the strengths of a case study.

5.2.2 Survey

A survey is a method to "collect information from or about people to describe, compare, or explain their knowledge, feelings, values, and behavior" (Fink & Kosecoff 2006). Performing a survey to gain information on how people write their prompts, what LLMs they use, and what they use them for would be invaluable data, but having to wait for the data to arrive would take too much time for the scope of this study. So, it was deemed not to be possible. Ultimately, it was decided not to proceed with a survey as it would limit the scope of the study.

5.2.3 Systematic Literature Review

A literature analysis is regarded as helpful with a significant impact on global research (Taherdoost 2023). This technique also helps recognize, as Taherdoost (2023) said, "areas that require more rigorous bunches of evidence." As prompt engineering is a new study, articles have yet to be published. This has changed as more articles were published around prompt engineering in recent years; this was still deemed nonviable as the scope of focus (Taherdoost 2023) was considered too narrow compared to our objectives.

5.3 Approach

5.3.1 Prompt techniques

The initial prompting techniques selected were Instructive, System, Question-Answer, Contextual, and Mixed, as defined by Giray (2023). These techniques were chosen because they were simple enough to emulate a student's prompt when seeking help from a large language model (LLM) to solve an assignment. It soon became apparent that the System prompt was characteristically brief, with an ellipsis indicating a continuation for the LLM, which was impractical for code generation. Consequently, the study proceeded with the remaining four techniques.

Each of these prompting techniques was applied ten times across various LLMs, resulting in a total of 40 prompts. Although the prompts varied in their phrasing, each included essential details to ensure a fair evaluation of the outputs generated. The prompts for the Instructive and Question-Answer techniques were manually crafted, whereas half of the Contextual and all

Mixed prompts were written with assistance from ChatGPT-4 to have a variation in the language used for the prompts. This was done by inputting an already manually written prompt and prompting GPT-4 to formulate the prompt in different ways to avoid repetitive language in each prompt. The Mixed prompts were generated in a way where the first prompts of all other techniques were the input of ChatGPT with a request to combine them. This went on, so all second prompts for Mixed Prompt 2, etc.

All prompts underwent inspection to confirm they contained accurate details and were aligned with the intended nature of the technique. Adjustments were made to the AI-generated prompts as necessary.

5.3.2 Measurement metrics

Correctness

In our first research question (**RQ1**), the aim is to evaluate the correctness of solutions produced by large language models (LLMs) employing various prompt engineering techniques. To achieve this, we utilized the Google Test Library (Google 2024), a specialized tool for implementing unit tests in C++. These tests were complemented with human evaluations of the outputs. This dual approach enabled us to assess the correctness of the solutions objectively, through unit tests, and subjectively, through human judgment.

The Social Network task test suite consists of five individual tests, each targeting specific functionalities of the algorithm.

- **Empty Graph**
Validates the edge case empty graph
- **Single Node**
Validates the edge case single Node
- **BFS traversal linear graph**
Validates accurate BFS traversal in a linear graph
- **BFS traversal cyclic graph**
Validates accurate BFS traversal in a cyclic graph
- **Friend&Adversary classification**
Evaluates correct classification of Friends and Adversaries

The evaluation criteria for the tests are as follows:

Each generated solution can earn up to 2 points for this task. The first four tests, which assess edge cases and BFS traversal, collectively contribute 1 point if passed, while failing any results in 0 points. Completing the Friends and Adversary classification test also earned 1 point; failing this test resulted in 0 points. Thus, the maximum possible score for this task is two points.

Since each prompting technique generates 1-10 solutions, the average correctness is calculated based on this.

$$\frac{Points}{totalPoints} = averageCorrectness$$

For the Huffman Encoding task, the test suite had four separate tests targeting four different functionalities.

- **Frequency Count**
Verifies accurate character frequency calculation.
- **Priority Queue**
Assesses the correctness of prioritizing nodes in the queue based on character weight (frequency).
- **Building the Tree**
Validates the construction of the Huffman tree
- **Printing Codes**
Evaluates the functionality of printing encoded character representations.

The test results of the Huffman encoding task are evaluated according to the following criteria: Each of these tests gives 1 point for passing and 0 for failing. The maximum possible score for the task is 4 points. As with the previous task, each technique is prompted 1-10 times and generates 10 solutions. The average correctness is calculated according to the following formula

$$\frac{Points}{totalPoints} = averageCorrectness$$

Completeness

RQ2 focuses on the completeness of the generated code. The study employed human evaluation to assess the completeness metric. Code inspection was conducted to verify the implementation of functionalities required by the tasks.

Focusing on the social network task, the requirements specified the implementation of a Breadth-First Search (BFS) algorithm and the classification of relationships for a given node based on a predefined rule. Each solution received one point for the presence of each implemented functionality (BFS and classification) and zero points for missing functionalities. Therefore, the maximum attainable score for completeness in the social network task was two points.

Similar to the Social network task, human evaluation determined completeness for the Huffman encoding task. Here, code inspection focused on implementing a Huffman encoding algorithm and utilizing the provided code base within the solution. Each requirement, implementing the algorithm and using the provided code, received one point if present and zero points if absent. Consequently, the maximum achievable score for completeness in the Huffman encoding task was two points.

5.3.3 Alternative Approach

Several different approaches could be adapted. One of these is doing a comparative analysis of prompts generated by humans and those generated by an LLM to determine which prompt would generate better code. Additionally, different prompting techniques can also be used to see if a different outcome is possible. Another alternative approach is possibly evaluating other LLMs, not just the current three being assessed in this study. Some examples include several ones from Huggingface, such as Starcoder or Wizardcoder. These could be an alternative to Github Copilot as the "specifically trained on code" selection criteria in this study

Another approach to this study is to change the programming language. The tasks chosen had a requirement of C++, but this could easily be swapped to another language. While changing the language from C++ to another would infringe on said requirement, using the same tasks with various programming languages would be an interesting alternative. This could show if there is a difference in both the LLMs' understanding of the tasks or if the LLMs are better at generating code in a specific language. Some examples of programming languages that could be used are Java, Python, and C#.

6 | Result

This chapter presents the experiment results for the correctness and completeness metrics across all prompting techniques and models.

6.0.1 Objective 1

For the choice of LLMs for the study, we established a set of selection criteria consisting of some main ones and sub-criteria.

- Able to generate code through prompts
- Widely used
 - Two popular, easy-to-use LLMs
 - One LLM specifically trained on code

After careful consideration, ChatGPT, Gemini, and GitHub Copilot were selected as the primary tools for our research.

These tools are distinguished by their proficiency in generating code in response to user prompts. The version of ChatGPT chosen for our study is the latest, GPT-4. This version is a premium (pay to use) and is reputed to surpass its predecessor, GPT-3.5, in performance and capabilities.

Gemini Advanced was also selected for its promising potential in code generation. Despite being a premium tool with relatively limited research compared to its counterparts, Gemini Advanced is believed to rival or even exceed GPT-4 in certain aspects, making it an intriguing subject for our investigation.

GitHub Copilot completes our trio of tools. Its extensive use in the developer community and ability to generate code from prompts make it an exciting asset to our research. However, Github Copilot is a pair programmer and technically not an LLM. Still, it is a good match since it can do code generation through prompts and as it is built on OpenAI's GPT-4 model with further training on code from publicly available repositories, including the code from Github (Github 2023).

As our prompts focus on algorithms, it would also be interesting to compare the differences between the LLMs that are both built on the GPT-4 model. The regular GPT4 model is used with ChatGPT-4, and another model trained further specifically on code is Github Copilot.

6.0.2 Objective 2

Previous studies have been done for earlier models of ChatGPT-3.5, where the LLM solves basic programming tasks derived from student courses on a university level and has had success in generating correct code. Seeing as these studies showed that LLMs have the potential to solve basic programming tasks, it is only fitting to explore how advanced programming tasks an LLM can solve while also keeping it within student courses. Due to ease of access, we decided to select tasks from the university where we are currently students. These tasks are more advanced than those used in earlier studies because algorithms generally involve complex problems that require a deeper understanding of computational theory and data structures. It also requires a solid understanding of mathematics and theoretical computer science.

The tasks used in this study were derived from assignments in an Algorithms and Data Structures course at the University of Skövde. Two of four potential tasks were selected based on their complexity and suitability for the study. The remaining two tasks were excluded due to their minimal scope and heavy reliance on analytical rather than coding solutions. The chosen tasks were Social Network and Huffman Encoding.

As presented in the table 6.1, the Social Network task primarily involves implementing a Breadth-First Search (BFS) algorithm with additional specific rules. In contrast, the Huffman Encoding task involves a more standardized solution but includes a mandatory code block for completing the task. These elements were particularly intriguing for our study as they allowed us to evaluate how well the models can utilize provided code segments and handle specific rules added to an algorithm. The added functionality to the BFS algorithm and the base code utilization for the Huffman encoding adds another layer of complexity to the algorithms, which, in other cases, would have a more standard solution. The complete task descriptions are shown in Appendices D and E, respectively.

Tasks	Standard	Additional complexity
Social Network	BFS	Friends and Adversary classification rule
Huffman Encoding	Huffman Encoding	Base code utilization

Table 6.1: Tasks

6.1 Objective 3

ChatGPT-4

ChatGPT-4 exhibits a broad range of capabilities, from reading files and solving mathematical problems to generating code, the latter being the primary focus of this thesis project. This model is capable of producing code in various programming languages. When not explicitly directed, it tends to default to Python. Typically, ChatGPT-4 generates all code within a single file; however, there are instances where it outputs code in segmented "fragments." It is important to note that the code generated by ChatGPT-4 may sometimes contain errors. This is attributable to the fact that the model does not execute or validate the code prior to delivering it to the user. Furthermore, there are occasions when ChatGPT 4 may cease code production midway through a document, subsequently prompting the user to decide whether to continue the generation process. Additionally, alongside the code, ChatGPT-4 provides explanatory text detailing the logic and reasoning behind the generated code. This feature aids users in understanding the methodology applied by the model to address the coding task at hand.

On certain occasions, ChatGPT-4 may not generate code but instead, provide a textual description of how to approach specific programming tasks. This necessitates users to resubmit their prompts to ChatGPT-4 multiple times until it produces the desired code output. This iterative interaction highlights an aspect of the models' operational variability in response generation.

Gemini-Advanced

The generated code from Gemini was sometimes "fragmented" where the whole code was given in snippets. Still, it was always in a correct, sequential order, i.e., the main method was given after everything else. These snippets were copied and pasted straight into a solution in Visual Studio, where each prompt got its own project within the solution. A reason as to why the code was fragmented in the prompt output was that Gemini thoroughly explained every part of the generated code. For a student generating an assignment, this could help them understand the code should a teacher or supervisor want them to explain their code. This is also under the pretense that the student wants to learn and not just want to "complete the assignment as fast as possible".

Gemini also had some tendency to create a header file for the code, which was implemented, but the majority of the code generated was in a single .cpp file.

GitHub-Copilot

In the case of GitHub Copilot, unlike platforms such as ChatGPT and Gemini, which feature web interfaces for code generation, it operates through a distinct mode of integration, specifically via extensions within Integrated Development Environments (IDEs) like Visual Studio and Visual Studio Code. To initiate the code generation process within these environments, it is imperative to install two extensions: the primary GitHub Copilot extension and an additional extension, GitHub Copilot Chat.

Upon successful installation of both extensions, to ensure each code generation request (prompt) remained unaffected by prior ones, a fresh "Solution" was created within Visual Studio for each distinct prompt. This approach hopes to minimize potential biases or influences from previous prompts, thereby maintaining the purity of each generation.

Once a code solution is generated, the system presents an option to either accept or reject the generated code. Acceptance of the solution moves the process forward. When the initial prompt did not generate code, it was re-prompted, specifically instructing it to generate the required solution.

Finally, the generated code was saved in a .cpp file within the respective solution for evaluation and further testing.

6.2 Correctness and Completeness - Social Network

This part of the results shows the correctness and completeness of the Social Network task across all prompt techniques and models.

6.2.1 LLMs

Figure 6.1 shows the correctness and completeness of the LLMs code solutions for the Social Network task. Gemini Advanced had the highest completeness with 100% while ChatGPT-4 and Github Copilot followed with 96% and 89%. Github Copilot had the highest correctness with a result of 47.5% correctness of all the prompts, While ChatGPT-4 and Gemini Advanced followed through with a correctness of 33.75% and 21.50%. Gemini Advanced had the highest completeness but also the lowest correctness. In contrast, Github Copilot had the highest correctness, but the lowest completeness and Chatgpt had the second highest completeness and correctness.

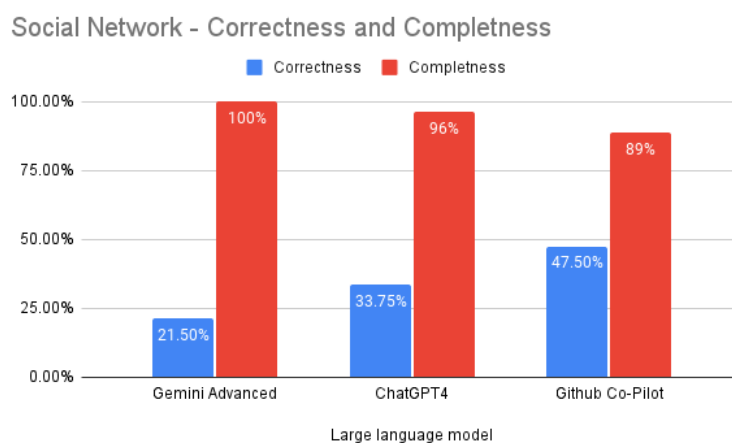


Figure 6.1:

6.2.2 Prompting Techniques

Figure 6.2 illustrates the results for correctness and completeness of the prompting techniques as a combination of all LLMs. Looking at the blue bars, the correctness, we see a downward trend where the correctness is reduced as the prompting technique becomes more advanced.

The orange bars, the completeness metric, are largely the same, with a smaller dip on the Contextual prompt. All techniques managed to generate code that was close to fully complete.

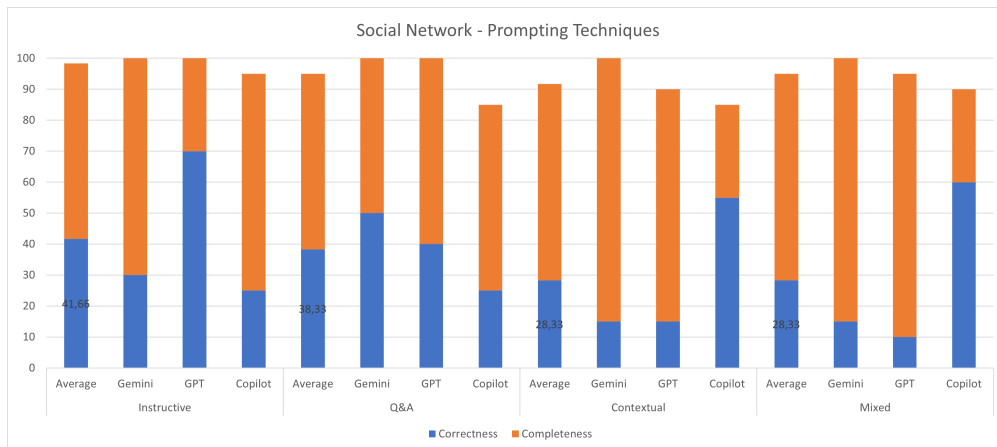


Figure 6.2:

6.3 Correctness and Completeness - Huffman Encoding

This section presents the results of the correctness and completeness metrics across the models and prompt techniques.

6.3.1 LLMs

Figure 6.3 displays the results of correctness and completeness of the LLMs for the Huffman Encoding task. For this task, Github Copilot had the highest correctness rate(51.25%), while Gemini had the lowest, scoring only 11% in correctness. Although the differences between ChatGPT-4 and Github Copilot are insignificant in both metrics, they highly outperformed Gemini.

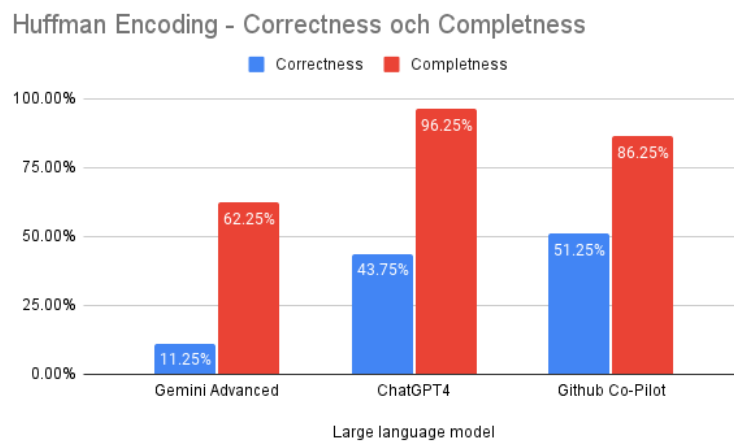


Figure 6.3:

6.3.2 Prompting Techniques

Figure 6.4 shows the correctness and completeness results of the prompting techniques for the Huffman Encoding task. There is a variance in the correctness results of the techniques. The techniques vary in their correctness results, with Instructive at 36.66%, Q&A at 22.50%, Contextual at 45.83%, and Mixed at 25.83%. Contextual got the best results with Instructive following after.

The completeness results are closely similar. The prompting techniques performed mostly similarly, with a slight dip on the Q&A prompt, with it being 6.67% lower than the highest results of the Mixed prompts.

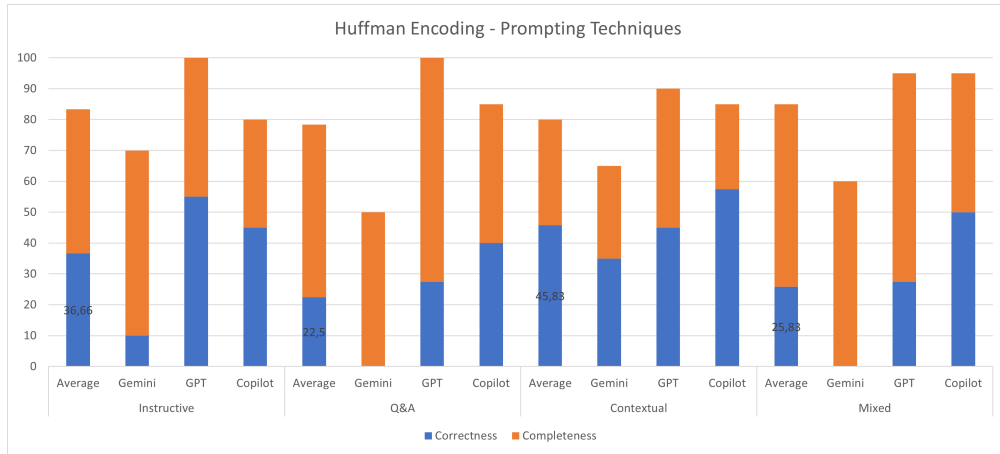


Figure 6.4:

7 | Analysis

The result for the compiled Social Network code for the three LLMs gave an overall higher compiled code result than those for the Huffman encoding assignment. Some things differ between the Social Network and the Huffman Encoding assignment. The main difference is that the Huffman Encoding algorithm has a standardized solution, while the Social Network assignment does not. While the central part of the Social Network task is the BFS of a tree, several additional rules remove a standardized solution. This made it so that the three LLMs needed more information on the prompts to understand what they were supposed to generate. The Three LLMs did give out more compiled code than on the Huffman encoding algorithm assignment. Still, overall, the average correctness on the Social Network assignment for the three LLMs was less than they got for the Huffman encoding algorithm.

7.1 Social Network Task

7.1.1 Github Co-pilot

As illustrated in figure 6.1. Github Copilot showed the best correctness performance for this task compared to the other two LLMs. Copilot had scored 47% in correctness and 89% in completeness. Although it had the lowest completeness, it had the overall best performance for this task.

While GitHub Copilot performed the best in terms of correctness, there is still room for improvement. The model mostly failed the friends and adversary classifications test while correctly implementing the BFS test for most cases. The model also had the least amount of errors in the generated solutions, which could have contributed to why it performed better than the other models on the correctness criteria.

As for the completeness metric, GitHub Copilot scored the lowest out of the three models in the Social Network task. This is primarily due to excluding the friends and adversary classification requirements of the task and instead providing comments on how to implement these requirements in the code.

7.1.2 Gemini

As we look at figure 6.1 it shows the correctness results of all prompting techniques. An abysmal 21.50% correctness rate is the lowest out of all LLMs. This is due to many recurring errors in the code.

The most occurring error type was unidentified identifiers, i.e., Gemini using identifiers that were not declared anywhere or improperly called on. Most of the generated code had this problem, along with other errors, which could deter the use of it as a code generator. This could be due to a student not wanting to fix broken code but instead wanting complete code that is ready to use.

The most common reason for a successfully compiled code to fail the correctness criteria was the incorrect classification. This was tested with unit tests and checked with human evaluation by building the tree according to existing code, drawing the tree with friend/adversary classifications, and checking if the connection between person A and person E was correct. Usually, this failed after leaving the nodes directly connected to the root node. An infrequent occurrence in the code results was it not doing a classification at all but instead a "There is a distance of 3 between person A and person E". There is no hint in the prompt as to why Gemini decided to do this instead.

Gemini also tended to create headers when generating code, not just cram everything into a .cpp file. This was interesting as no sign in the prompts could have given it the hint of creating a header file. Code was also sometimes generated as "fragments," where the code was not given all at once but as small snippets that Gemini explained.

Looking at the same figure and switching to the completeness results, we see that Gemini Advanced performed the best. A 100% completeness rate across all prompt techniques shows that Gemini has the ability to include all parts of a specified code.

After considering both completeness and correctness results, it is a surprise that Gemini performed poorly. All solutions came with a 100% completeness rate, but few met the correctness criteria. Gemini can generate code according to prompts, including all the parts needed, but it fails to construct error-free code for some reason.

7.1.3 Chatgpt-4

ChatGPT-4 is highly sophisticated and accurate when looking at completeness. However, it ranked second overall compared with the different LLMs in both completeness and correctness for the Social Network task, as seen in 6.1.

Most of the errors that occurred from the non-compiled prompts were that ChatGPT-4 sometimes forgot to include and define some variables, which, when later putting the code into Visual Studio code, gave an error because of undefined variables. Another type of error was that the functions had barely any code inside them, but there were comments about what input should be inside them. However, ChatGPT-4 got only half of the prompts as un-compiled, with 20 un-compiled prompts and 20 compiled prompts seen in Appendix C

The points that were not given in correctness were primarily due to ChatGPT-4 not fully understanding the task, and due to that, it did not fully generate code that would operate correctly. This is because the Social Network assignment is not as standardized as the Huffman encoding assignment; this could be shown because when giving a prompt to Chatgpt-4, it mostly tried to understand what the prompt wanted by generating text to say what it understood from it.

7.1.4 Prompting Techniques

As shown in figure 6.2, there is a slight difference between the prompting techniques of the correctness criteria. As said, a downward trend can be seen where as the technique becomes more advanced, the correctness decreases. This is a slight surprise as the more advanced techniques should theoretically give more correct results. With instructive being the best at 41.66% average correctness across all LLMs, it shows that the simpler techniques often are enough depending on your needs.

The completeness results show another result where the downward trend no longer exists. All techniques performed very similarly to each other across all LLMs. Contextual being the worst at 91.66% is also a bit surprising as it was theorized that it should have been the better performing technique instead of performing the worst.

7.2 Huffman Encoding

7.2.1 Github Co-pilot

GitHub Copilot had, once again, the best performance for this task, as shown in figure 6.3. Copilot scored 51.25% in correctness, slightly outperforming ChatGPT-4, which had 43% in correctness. The differences are more significant when comparing Copilot to Gemini, who scored only 11.25% in correctness. In the completeness metric, Copilot scored 86.25%, coming second only to ChatGPT-4.

For this task, GitHub Copilot had a higher average correctness score than the previous task. The most frequent error involved correctly printing the binary codes of the Huffman Tree. Instead of printing the tree in binary codes, Copilot printed the letters of the string to be encoded. This error stemmed from an incorrect implementation of the print function, which was rarely implemented correctly.

Additionally, the most common missing functionality that affected Copilot's completeness score was the exclusion of the function that calculates the frequency of each character in a string. This function is essential for ordering the priority queue, and its absence significantly impacted the completeness and correctness metrics.

7.2.2 Gemini

In figure 6.3, we see the abysmal results from Gemini with an overall correctness rate of 11.25%. The big reason why these metric results are so bad is because Gemini only generated a total of 4 compilable code solutions from a total of 40 (See Appendix L). These are broken down into one Instructive success and three Contextual successes. These four pieces of generated code scored pretty well, with the Instructive result being 100% correct. The contextual prompts had one code piece that failed testing with 2/4 points. This code that failed did not correctly encode each character in the provided string but instead seemed to encode the entire string instead. There were also some common errors that made the generated solutions fail the correctness criteria.

The first common error was the members of the tree class. These members were classified as private, which led to Gemini not correctly accessing them, leading to the errors in the likes of "member Tree:c is inaccessible"

The other common error was in the provided code initialization of the "printTree" function with a "vector& bitString" argument. This vector was intentionally missing an argument list, such as <int>, to see if an LLM could correctly use the provided code when it was incomplete. Gemini had massive problems figuring out that the code needed to be fully implemented, and it had to realize the functions within the provided code.

Switching to completeness shows a better result than correctness but is still bad compared to the other LLMs. Gemini had problems with a couple of prevalent errors. All of these errors stem from the part of the task requiring provided code that has to be used. This tree class and tree wrapper are part of the task completeness. While many generated solutions included the provided code, it failed to utilize even the first part of it, leading to a failure in this completion criteria.

As a result of having to use provided code, Gemini Advanced failed to generate usable code, both because it was not compilable and because it was not salvageable due to the number of errors.

7.2.3 Chatgpt-4

As illustrated in Figure 6.3, ChatGPT-4 achieved the highest percentage of completeness among the various LLMs, with a score of 96.25%. GitHub Copilot followed with 86.25%, and Gemini Advanced scored 62.25%. Regarding correctness, ChatGPT-4 secured the second-highest percentage at 43.75%, with GitHub Copilot leading at 51.25% and Gemini Advanced trailing at 11.25%.

ChatGPT-4 compiled the highest number of prompts, with 27 out of 40 prompts successfully compiled for the Huffman encoding task (see Appendix L). GitHub Copilot compiled 26 prompts, and Gemini Advanced compiled only four prompts. Furthermore, ChatGPT-4 had the lowest rate of uncompiled prompts, with 13 uncompiled prompts, compared to Gemini Advanced's 36 uncompiled prompts and GitHub Copilot's 14 uncompiled prompts.

ChatGPT-4 demonstrated the highest percentage of completeness among the LLMs. This high completeness score did not always correspond to high correctness, as a prompt must successfully compile to be deemed correct. Some prompts for the Huffman encoding task failed to compile, affecting the correctness score. The most common error type for the Huffman encoding task was undefined variables. Also, some errors occurred when the assignment had to include code. ChatGPT-4 occasionally generated code assuming that certain elements were already integrated into the file, resulting in errors for those prompts.

7.2.4 Prompting Techniques

As shown in figure 6.4, there is a mild variance in the correctness results of the prompting techniques. Instead of a downward trend (as seen in 6.2), this trend instead goes down, up, and down again. There are no clear reasons as to why the variance is so large. Contextual having the

best results at 45.83% does match the theory that the more advanced techniques should generate more correct and complete solutions but is later shown otherwise with the 25.83% result of the Mixed technique.

The completeness results are broadly similar, with only a slight variance between them. The LLMs generated roughly the same amount of completed code. Mixed having the best results at 85% also matches the theory of the advanced prompts.

8 | Discussion

8.1 Summary

8.1.1 Correctness of LLMs

In the ANOVA test for comparing the LLMs, the results show a significant difference between the LLMs for the Social Network task(see Appendix H). For this, three samples were used (one per LLM) with a significance level of 0.05. The same ANOVA test is conducted for the Huffman Encoding task (see Appendix I). A significance level of 0.05 shows a significant difference between the LLMs in generating code for the Huffman Encoding task.

As a result of the ANOVA test, which shows a statistically significant difference between the correctness of generated code by the LLMs, H_0 is thrown for **RQ1**.

8.1.2 Completeness of LLMs

A separate Anova test was conducted for both tasks to answer **RQ2** for completeness (See Appendix J and K). The results of the tests for both tasks show a statistically significant difference between the LLMs in terms of completeness of the generated code. The same significance level was used for this test, resulting in a p-value of <0.05 ; thus, H_0 is thrown for RQ2.

8.1.3 Prompting Techniques

We can analyze the data for significant differences with a Kruskal-Wallis one-way ANOVA test shown in Appendix F. Four samples were used for this test, one for each prompting technique with a significance level of 0.05. This test resulted in a p-value of 0.413, thus showing no statistically significant difference between the prompting techniques in the correctness criteria for the Social Network task (See Appendix F). The same test was conducted for the Huffman Encoding task (See Appendix G). The results gave a p-value of 0.220 with a significance level of 0.05, which shows no statistically significant difference between the prompting techniques in the correctness criteria for this task.

8.2 Generalization

Logozar et al. (2023) experimented to see how ChatGPT performs when solving college freshman programming tasks. ChatgPT got five different programming tasks where the results it got are as follows: 80% correct for programming task 1, 60% for programming task 2, 100% for programming task 3, 100% for programming task 4, and 95% for programming task 5. The major

difference between the author and our thesis project is that the author is only using one LLM and one prompt for each assignment, while we are using three different LLMs and four different prompting techniques, where each technique has ten prompts for each assignment. When we compare our results from ChatGPT-4 to their results, their results had a higher rate of correctness. This could be due to the differences in the complexity of tasks used in the studies, resulting in different correctness levels. For example, the tasks used in their paper were mostly simple programming tasks, such as finding the Greatest Common Divisor(GCD) of 2 integers. Additionally, the correctness metric used by (Logozar et al. 2023) differs from our approach. We measured the functional correctness employing unit tests, while (Logozar et al. 2023) assesses correctness through human evaluations.

Yan et al. (2023) uses a different method from our study. The author uses an empirical study to get the results from the code generation of 3 different models, namely GPT-Neo, CodeRL, and GPT-3.5. The authors' methods and test cases differ from those used in this study.

This study compares different LLMs code generation for the same task with varying prompting techniques while the author interviews students to see what the students think are the strengths, weaknesses, and opportunities with ChatGPT (Prentzas & Sidiropoulou 2023). Certain parts are comparable. ChatGPT certainly provided guidance as the generated solutions were explained and given to the user.

While we are comparing different LLMs to each other, the author is comparing Codex to students' assignments. Becker et al. (2023) compares Codex code and students' code to see how Codex performs on programming exercises. We are not using Codex in our project, but we have some similarities with this research article. The main similarity is that this study and the study by Becker et al. (2023) take university assignments to see how well AI can handle them. The study by Becker et al. (2023) shows that Codex performed better when it did not need to adapt existing code. The results in this study show the same: if an LLM had to use the provided code in the Huffman Encoding task, it performed worse than the Social Network task, where it did not have to adapt existing code.

The work made by Lundblad et al. (2023) is somewhat similar to our work. We are both seeing code generation with LLM using different prompting techniques to solve programming assignments from the same school. However, the author uses two programming languages for only one LLM, Chatgpt-3.5. We are using three different LLMs, different programming tasks, and different prompting techniques than the author. While the results by Lundblad et al. (2023) are in-depth, it is hard to compare the results here to theirs. This is due to the different techniques and the broader use of LLMs.

This study's results show that using different LLMs significantly affects the correctness and completeness of the code solution. The prompting techniques showed no significant differences in terms of correctness or completeness. The results indicate that Gemini Advanced should not be used to generate complete code solutions but could be used as an assistant tool.

When not having to include the provided code, the LLMs showed a high degree of completeness, showing that they have the capability to generate algorithmic code that includes all necessary parts for a given task. If the provided code is necessary to complete a given task, the LLMs' ability to generate the algorithmic code diminishes harshly.

8.3 Ethics & Threats to Validity

8.3.1 Ethics

LLMs are trained on such massive amounts of data, so it is nearly impossible to know if copyrighted data was used unless the creators confirm it. OpenAI has confirmed using copyrighted material in their training sets (Zirpoli 2023). At the same time, in terms of use for ChatGPT, they also state that all outputs belong to the user (OpenAI 2023a). Google shoulders this clause by promising legal help to anyone should any output generated by Gemini cause a copyright case against a user. However, this only applies if the user generated copyrighted material unintentionally (Richardson 2023). While this sounds good on paper, it opens up the possibility of LLMs and AI being threatened in the future. Being able to generate copyrighted material, intentionally or not, is dangerous for creators, license holders, and users.

Training LLMs on large data sets is very time-consuming and expensive. The more training data an LLM gets, the more abilities it gets. There is also an essential part regarding the correct training data. Because training an LLM is very time-consuming and expensive, the LLM must get the proper training data. The training data is crucial to how the LLM performs and generates output.

The usage of confidential data differs between the LLMs. Chatgpt-4 and Github copilot both use the GPT-4 model that OpenAI developed. There is no information on whether OpenAI uses confidential data as training data for the GPT-4 model. However, according to the terms of use, the input that the user gives can be used as training data, which means that if a user puts confidential data in a prompt that is later given to OpenAI's LLMs, that input can be used as training data (OpenAI 2023a).

Gemini Advanced does not use user input as training data. According to the Gemini Team & Google (2024), the paid services for Gemini, which is Gemini Advanced, do not use the prompts the users give as training data. This means that should a user give confidential data in their input; it will not be used as training data for the LLM. However, Gemini's non-paid services can use the prompts the user gives as training data. They also confirm that the training data comes from public data. This, however, does not mean that confidential data can not be public, so there is still a chance of confidential data being used as training data.

Another topic on confidential and copyrighted data appears in the code that is included in the training data. Github has stated that they use open-source code as training data (GitHub 2024) but there is no way to know as they could easily be hiding information. While open-source code from GitHub may have been used as training data, it could also easily include code from other sources. These other sources may include both confidential or copyrighted code. While there is nothing a standard user can do to mitigate these problems, it is essential to consider.

Google recently announced the creation of a new TPU chip that offers an alternative to those by NVIDIA (Cherney 2024). Consider the millions or even billions of dollars that may have been put into this research and development. If the R&D for this chip had failed, then all that money would have gone to waste and possibly would have generated tons of e-waste. However, if the outcome of funding the research and development is too successful, then it is vital to consider how strong AI would get. When would an AI get so intelligent that human workers are put out of their jobs? It is essential to consider the implications of both ways, where the development of

AI fails and where the development of AI becomes too successful.

The most considerable impact of this study is societal, where the results help identify the strengths and limitations of the LLMs used in the study. This opens up avenues for future research into how LLMs could be utilized in educational settings. It also demonstrates the impact of prompting techniques has on generated code, aiding in the understanding of how LLMs interpret prompts for coding solutions.

On the positive side, integrating LLMs into various sectors could necessitate updating educational curricula to include LLM-related skills, enhancing learning outcomes, and better preparing students for future job markets.

However, there are also negative implications, such as potential job displacement, which may require addressing through comprehensive retraining programs. Additionally, the long-term trajectory of LLM development remains uncertain, necessitating ongoing research to understand its future impacts.

Economically, the results provide insights into how these LLMs could be used as educational tools. Although the LLMs mostly did not have a high accuracy in generating correct code, they provided detailed explanations of each part of the solution. This feature can be particularly valuable in educational contexts, where understanding code's underlying logic and structure is crucial for learning. The ability to offer detailed code explanations could reduce the demand for additional tutoring resources, thereby lowering educational costs and potentially normalizing access to high-quality coding education.

8.3.2 Threats to Validity

External validity: The study was limited to only two coding tasks, which may not fully capture the wide range of challenges that large language models (LLMs) can encounter. These tasks were derived from a student course assignment that features a limited set of coding problems, potentially limiting the generalizability of the findings to other types of coding tasks. Additionally, the tasks specified using the C++ programming language for problem-solving, this requirement introduces a limitation, as it may restrict the results' applicability to programming environments that use other languages. Moreover, the findings might only apply to some LLMs, mainly because the models evaluated in this study were explicitly trained on code. The outcomes also heavily depend on how the prompts were constructed and the techniques applied during the study. Furthermore, the results may only be valid for newer versions of the LLMs, as these models are continuously being updated and trained, potentially altering their performance capabilities over time. Future work should consider replicating the study across diverse programming languages and a more comprehensive range of coding problems using the updated versions of the models.

Construct validity: Any ambiguity in the measurement metrics of "correctness" and "completeness" could lead to different assessments of the output generated by the LLMs. The correctness and completeness of the study were measured based on the definition of the ISO/IEC. The use of any other definition could lead to different assessments and results.

Conclusion validity: The statistical methods used in this study were chosen to analyze the LLM output adequately, but the analysis could be susceptible to biases, particularly in handling outputs with errors. Assigning zero points to non-compilable code might overly penalize LLMs that generate partially correct but syntactically flawed outputs. An alternative approach might

involve a tiered scoring system that offers partial credit for partially correct code, providing a more nuanced view of each LLM's performance. Furthermore, a detailed error analysis could reveal common patterns or types of errors, offering deeper insights into specific weaknesses in the LLMs' code generation capabilities.

8.4 Future Work

There are many possible ways to build upon this study further. As a bachelor thesis, the scope was limited, and some ideas were scrapped early on in the study. These included ideas such as using open-source LLMs from websites like Huggingface or investigating the time complexity of generated solutions. Another limiting factor was the choice of course from the university. As the course assignments were to be completed in C++, this limited the number of languages that could be investigated.

Therefore, there are multiple ways this study could have branched out and could be investigated further. Future research that would be interesting that failed to do in this study was investigating the time complexity of generated code. The tasks were chosen with the intent to analyze time complexity, but due to the limited scope, this was scrapped. Analyzing the time complexity of generated algorithmic code could be an exciting work to build upon to see if there are differences in time complexity between LLMs or different prompting techniques.

Another way to build upon this study is to further investigate prompts and their many techniques. The techniques used in this study are purposefully less advanced to facilitate a student but remove this limitation, and studies on more advanced techniques are possible. At the same time, it is also possible to increase the number of tested prompts; this study only used 10 prompts for each technique with a total of 80 prompts, whereas more than 10 would give more data. This could also be flipped where further study into the LLMs is viable. This could be done using more techniques, more prompts per technique, or changes of tasks.

ChatGPT-4 and Gemini provided information about their generated solutions, and as mentioned in the article by Prentzas & Sidiropoulou (2023), students use it for guidance and quick responses. Something to investigate in the future would be the accuracy of this explanation for the code given. This could be done using simple tasks for a more straightforward study and testing its accuracy in explaining more advanced code.

Further investigating the capabilities of LLMs in their ability to generate solutions for tasks when they have to adapt and use existing code is an interesting study. Results from this study and Becker et al. (2023) show that it performs worse when using existing code.

Future work could also explore integrating LLMs with other AI technologies, like computer vision, for multidisciplinary applications. Additionally, studying LLMs in domains such as healthcare, finance, and entertainment could reveal novel use cases. Addressing this study's limitations, such as the narrow scope of tasks and languages, with a larger team and more resources could provide deeper insights.

9 | Conclusion

The study aimed to compare LLMs' ability to solve algorithmic coding tasks. Different prompting techniques were used, and their impact on correctness and completeness was evaluated. The results were collected, presented in graphs, and thoroughly discussed.

We conducted ANOVA tests to address our research questions and determine if there are significant differences in correctness and completeness among various LLMs across different prompting techniques.

The first set of tests assessed whether there were significant differences between the LLMs regarding correctness and completeness. The second set evaluated whether significant differences existed between the prompting techniques in these same metrics.

The results indicated no significant differences between ChatGPT-4 and GitHub Copilot regarding correctness and completeness for both tasks. Nevertheless, significant differences were found when comparing Gemini to the other models. In the Social Network task, Gemini performed significantly worse than GitHub Copilot in correctness for both tasks and had a significantly lower score in completeness for the Huffman encoding task. The differences between ChatGPT-4 and Gemini in the Social Network task were insignificant in either metric. Conversely, there was a significant difference in the Huffman encoding task, with Gemini performing significantly worse in both correctness and completeness.

Regarding the prompting techniques, the results showed no significant differences in correctness and completeness across the different techniques.

In conclusion, our results indicate significant differences between the LLMs regarding correctness and completeness, while there are no significant differences between the prompting techniques.

These findings lead to the following conclusions for our hypotheses:

For RQ1, the results reject the null hypothesis (H_0) and support the alternative hypothesis (H_1), indicating significant differences in functional correctness between the LLMs, particularly with Gemini performing worse in certain tasks. However, there are no significant differences based on the prompting techniques.

For RQ2, the results reject the null hypothesis (H_0) and support the alternative hypothesis (H_1), indicating significant differences in the completeness of the code generated by different LLMs, especially with Gemini performing worse in specific tasks. Again, no significant differences were observed based on the prompting techniques.

Based on the results of this study, we recommend using either ChatGPT4 or Github Copilot for your coding needs. While Gemini managed to generate largely complete code, it had too many fails on the correctness criteria. By using the two recommended LLMs instead of Gemini Advanced, one would save more time on debugging the faulty generated code. For students, we recommend not using the LLMs to generate complete solutions. Doing so impedes your ability to learn for the future. Instead of having them complete your work for you, use all LLMs as assistant tools.

Bibliography

- Alahmed, Y., Abadla, R., Badri, A. A. & Ameen, N. (2023), “how does chatgpt work” examining functionality to the creative ai chatgpt on x’s (twitter) platform, *in* ‘2023 Tenth International Conference on Social Networks Analysis, Management and Security (SNAMS)’, pp. 1–7.
- Amazon (2024), ‘What is gpt ai? - generative pre-trained transformers explained - aws’.
URL: <https://aws.amazon.com/what-is/gpt/>
- Anh, D. H., Do, D.-T., Tran, V. & Minh, N. L. (2023), The impact of large language modeling on natural language processing in legal texts: A comprehensive survey, *in* ‘2023 15th International Conference on Knowledge and Systems Engineering (KSE)’, pp. 1–7.
- Bass, J. M., Beecham, S. & Noll, J. (2018), Experience of industry case studies: A comparison of multi-case and embedded case study methods, *in* ‘2018 IEEE/ACM 6th International Workshop on Conducting Empirical Studies in Industry (CESI)’, pp. 13–20.
- Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J. & Santos, E. A. (2023), *in* ‘PROCEEDINGS OF THE 54TH ACM TECHNICAL SYMPOSIUM ON COMPUTER SCIENCE EDUCATION, VOL 1, SIGCSE 2023’, Assoc Comp Machinery; ACM Special Interest Grp Comp Sci Educ, pp. 500–506.
- Chen, Z. (2023), Neural language models in natural language processing, *in* ‘2023 2nd International Conference on Data Analytics, Computing and Artificial Intelligence (ICDACAI)’, pp. 521–524.
- Cherney, M. A. (2024), ‘Google unveils arm-based data center processor, new ai chip | reuters’.
URL: <https://www.reuters.com/technology/google-unveils-arm-based-data-center-processor-new-ai-chip-2024-04-09/>
- Fink, A. & Kosecoff, J. (2006), *How to conduct surveys: A step-by-step guide*, SAGE.
- Giray, L. (2023), ‘Prompt engineering with chatgpt: A guide for academic writers’, *Annals of Biomedical Engineering* **51**, 3.
- Github (2023), ‘Github co-pilot faq’.
URL: <https://github.com/features/copilotfaq>
- GitHub (2024), ‘Github copilot’. Accessed: 2024-04-20.
URL: <https://github.com/features/copilot>
- GitHubCopilotChat (2024), ‘About github copilot chat’. Accessed: 2024-04-20.
URL: <https://docs.github.com/en/copilot/github-copilot-chat/about-github-copilot-chat>

- Google (2024), ‘Google test’.
URL: <https://google.github.io/googletest>
- IEC, I. (2023), ‘Iso/iec 25010:2023(en)’.
URL: <https://www.iso.org/obp/ui/en/iso:std:iso-iec:25010:ed-2:v1:en>
- Lee, A. (2023), ‘What are large language models and why are they important?’.
URL: <https://blogs.nvidia.com/blog/what-are-large-language-models-used-for/>
- Leinonen, J., Denny, P., MacNeil, S., Sarsa, S., Bernstein, S., Kim, J., Tran, A. & Hellas, A. (2023), Comparing code explanations created by students and large language models, *in* ‘Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1’, ITiCSE 2023, Association for Computing Machinery, New York, NY, USA, p. 124–130.
URL: <https://doi-org.libraryproxy.his.se/10.1145/3587102.3588785>
- Logozar, R., Mikac, M. & Hizak, J. (2023), Chatgpt on the freshman test in c/c++ programming, *in* ‘2023 IEEE 21st Jubilee International Symposium on Intelligent Systems and Informatics (SISY)’, pp. 000255–000264.
- Lundblad, J., Thörn, E. & Thörn, L. (2023), The impact of task specification on code generated via ChatGPT, PhD thesis.
- OpenAI (2023a).
URL: <https://openai.com/policies/terms-of-use>
- OpenAI (2023b), ‘Gpt-4 openai’. Accessed: 2024-02-13.
URL: <https://openai.com/research/gpt-4>
- Petroni, F., Lewis, P., Piktus, A., Rocktäschel, T., Wu, Y., Miller, A. H. & Riedel, S. (2020), ‘How context affects language models’ factual predictions’.
- Plevris, V., Papazafeiropoulos, G. & Jiménez Rios, A. (2023), ‘Chatbots put to the test in math and logic problems: A comparison and assessment of chatgpt-3.5, chatgpt-4, and google bard’, *AI* 4(4), 949–969.
URL: <https://www.mdpi.com/2673-2688/4/4/48>
- Prasad, S. G., Sharmila, V. C. & Badrinarayanan, M. (2023), Role of artificial intelligence based chat generative pre-trained transformer (chatgpt) in cyber security, *in* ‘2023 2nd International Conference on Applied Artificial Intelligence and Computing (ICAAIC)’, pp. 107–114.
- Prentzas, J. & Sidiropoulou, M. (2023), Assessing the use of open ai chat-gpt in a university department of education, *in* ‘2023 14th International Conference on Information, Intelligence, Systems Applications (IISA)’, pp. 1–4.
- Richardson, L. (2023), ‘Our responsible approach to building guardrails for generative ai’.
URL: <https://blog.google/technology/ai/our-responsible-approach-to-building-guardrails-for-generative-ai/>
- Taherdoost, H. (2023), ‘Towards nuts and bolts of conducting literature review: A typology of literature review’, *Electronics* 12(4).
URL: <https://www.mdpi.com/2079-9292/12/4/800>
- Team, G. & Google (2024), ‘Gemini: A family of highly capable multimodal models’.
URL: https://storage.googleapis.com/deepmind-media/gemini/gemini_v15r_report.pdf

- Whately, B. (2022), ‘3 principles for prompt engineering with gpt-3’.
URL: <https://www.linkedin.com/pulse/3-principles-prompt-engineering-gpt-3-ben-whately/>
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B. & Wesslén, A. (2012), *Experimentation in Software Engineering*, Springer, Germany.
- Yan, D., Gao, Z. & Liu, Z. (2023), A closer look at different difficulty levels code generation abilities of chatgpt, *in* ‘2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)’, pp. 1887–1898.
- Zheng, X., Zhang, C. & Woodland, P. C. (2021), Adapting gpt, gpt-2 and bert language models for speech recognition, *in* ‘2021 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)’, pp. 162–168.
- Zirpoli, C. T. (2023), ‘Generative artificial intelligence and copyright law’.
URL: <https://crsreports.congress.gov/product/pdf/LSB/LSB10922>

A | Appendix

A.1 Appendix A

Social Network Instructive	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	✓	✓	X	X	✓	X	✓	X	X	X
ChatGPT4	✓	✓	✓	✓	✓	✓	✓	✓	✓	X
Github Co-Pilot	✓	✓	✓	X	✓	✓	✓	✓	✓	X
Social Network QA	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	X	X	✓	✓	X	X	X	X	✓	X
ChatGPT4	X	✓	X	X	✓	X	✓	✓	X	✓
Github Co-Pilot	✓	✓	✓	✓	✓	✓	✓	✓	✓	X
Social Network Contextual	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	X	X	X	✓	X	X	✓	X	✓	X
ChatGPT4	✓	X	X	✓	X	✓	X	X	X	X
Github Co-Pilot	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Social Network Mixed	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	X	✓	X	X	X	X	✓	✓	✓	X
ChatGPT4	X	X	X	X	X	✓	X	X	✓	✓
Github Co-Pilot	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

This table shows the result of each prompt for each technique in the Social Network task for Gemini. A checkmark means that the code from that prompt was successfully compiled, and an X means it was not.

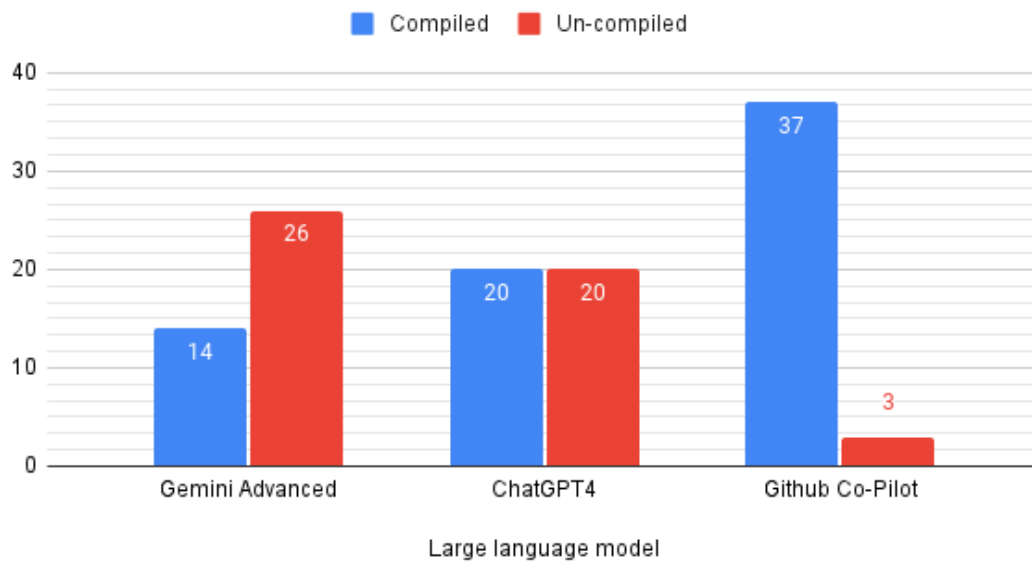
A.2 Appendix B

Huffman Encoding Instructive	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	X	X	✓	X	X	X	X	X	X	X
ChatGPT4	✓	✓	✓	✓	✓	✓	✓	X	✓	✓
Github Co-Pilot	X	✓	✓	X	✓	✓	X	X	✓	✓
Huffman Encoding QA	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	X	X	X	X	X	X	X	X	X	X
ChatGPT4	X	✓	X	✓	X	✓	X	✓	X	✓
Github Co-Pilot	✓	✓	✓	X	✓	✓	X	X	X	X
Huffman Encoding Contextual	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	✓	X	✓	X	X	X	✓	X	X	X
ChatGPT4	✓	✓	✓	X	X	X	✓	✓	✓	✓
Github Co-Pilot	✓	✓	X	✓	✓	✓	✓	✓	✓	✓
Huffman Encoding Mixed	Prompt 1	Prompt 2	Prompt 3	Prompt 4	Prompt 5	Prompt 6	Prompt 7	Prompt 8	Prompt 9	Prompt 10
Gemini Advanced	X	X	X	X	X	X	X	X	X	X
ChatGPT4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Github Co-Pilot	✓	✓	X	X	✓	✓	X	✓	X	X

This table shows the result of each prompt for each technique in the Huffman Encoding task for Gemini. A checkmark means that the code from that prompt was successfully compiled, and an X means it did not.

A.3 Appendix C

Social Network - Compiled och Un-compiled



A.4 Appendix D

Problem 2 – Social Network (Related lectures: F4, F5, F9)

The concept of social networks in which members can make friends in various ways has become increasingly popular. You will learn a new innovative idea to help members find new friends in a social network. The idea is to allow all members of the network to specify their adversaries and model this as a graph, i.e. an *adversary graph*. Then this graph can be used for a member to find new friends according to following description:

- An adversary's adversary is a friend.
- A friend's adversary is an adversary.

All members in the graph are classified according to the above description. Note that, based on the adversary graph, a member can be classified as a friend or an adversary and can choose to classify a person according to “the shortest distance” rule, i.e. the path going through the least number of edges (an edge is a connection between two members). Below is an example of an adversary graph where members are classified as friends or adversaries based on the above recursive description (all directed edges in the graph below models adversary relations):

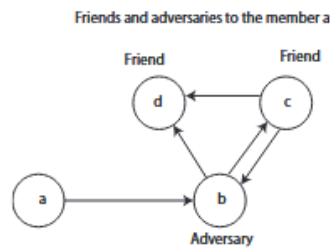


Figure 1: A example to show the friends and adversaries of a from outside the adversary graph.

Note that member d is classified as a 's adversary via the path $a \rightarrow b \rightarrow c \rightarrow d$. However, there is a shortest path $a \rightarrow b \rightarrow d$ where d is classified as a friend.

Task 2.1 (The algorithm must be demonstrated to the lab tutor) – Design an algorithm based on breadth-first search so that a specific individual can find all friends by using the **adversarygraph** . Implement the algorithm in C/C++. It should be easy to test whether the algorithm works for any adversary graph. **The algorithm should use a queue from the STL library.**

The following parts must be included in the report:

- *Intuitive description of the algorithm.*
- *Abstract algorithm in pseudo code.*
- *Time complexity analysis with $O(\cdot)$ -notation*
- *C/C++ code of the algorithm implementation.*

A.5 Appendix E

Problem 3 – Compression by Huffman encoding (Related lectures: F6, F8)

A method of compressing a text string is to use so-called Huffman coding which is to encode the characters that occur most frequently in the string with a short sequence of bits as possible. We can take the text string `aaaabbbc` as a simple example. Each letter in this sequence contains eight bits (according to ASCII table) making a total of $8 * 8$ bits i.e. eight bytes for the entire sequence. However, we can see that this string consists of only three characters in which case we do not need eight bits to represent a single character, we could instead make use of bit encoding `a - 0`, `b - 11`, `c - 10`, which means we could represent the string `aaaabbbc` by the bit sequence `000011111110` with only 12 bits (2 bytes). Compared to the previous example, we saved 6 bytes in this simple example by using this coding approach, which could be used for the purpose of compressing a text string (or file). This coding approach is called Huffman coding. An important feature of Huffman coding is that the characters that occur most frequently in the text string always get shortest bit sequence to obtain the most efficient compression as possible (e.g. `a` is encoded by only one bit since the character appeared most frequently in the string of the above example). The actual Huffman coding can be represented effectively through a tree representation, as illustrated next.

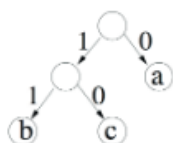


Figure 2: Tree representation of Huffman coding

One can use the above tree for coding the bit-sequence `000011111110` by following the branches from the root of the tree and print each character in

the leaves when it is reached, and then repeat the process for each character. For example, we reach **a** four times, followed by **b** etc., to get the encoded bit-sequence of the compressed text. The algorithm for constructing Huffman trees could use a min-priority queue ADT, as described below:

1. count the occurrence of the various characters that occur in the string and construct a tree T consisting of a node containing the character and a weight w equal to the occurrence of the character in the string
2. Insert each node in a min priority queue where the min relation is determined by weight w
3. Pick the current “top” element T_1 (a tree) from the priority queue (i.e. that is the element with the minimum priority, which will be retrieved and removed)
4. Pick the current “top” element T_2 (a tree) from the priority queue (i.e. it will be retrieved and removed, as well)
5. Construct a new tree $T_{1,2}$ by creating a new node with weight $w_1 + w_2$ (i.e. the sum of the weights of trees T_1 , and T_2). T_1 and T_2 are subtrees of $T_{1,2}$. Place the tree $T_{1,2}$ in the priority queue.
6. Repeat from step 3 as long as the priority queue contains more than one trees.
7. Return the remaining tree in the priority queue (Huffman coding).

The figure below illustrates how the above procedure works on the string **aaaabbbc** (where the numbers indicate the tree weights – occurrence of each character in the tree):

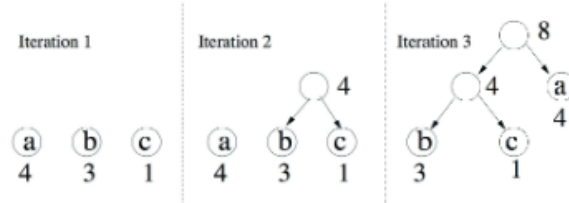


Figure 3: Trees in the priority queue after each iteration.

Note that, in each iteration you should pick the trees that have minimum weights, assemble them into a new tree, and then you put this tree in the priority queue.

Task 3.1 *Implement the above-mentioned Huffman coding algorithm by using a priority queue from STL and a tree class with the following member functions:*

```

class Tree {
public: Tree(int w, char c);
Tree(int w, Tree* t1, Tree* t2); ~Tree();
int getWeight() const;
void printTree(vector& bitString) const;
private:
Tree* left;
Tree* right;
int weight;
char c;
};

struct TreeWrapper {
TreeWrapper() {
tree = NULL;
}
TreeWrapper(Tree* t) {
tree = t;
}
bool operator<(const TreeWrapper &tw) const {
return tree->getWeight() > tw.tree->getWeight();
}
Tree* tree;
};

```

The function printTree(...) to print Huffman coding of the different characters like the following example:

```

0 : a
1 0 : c
1 1 : b

```

where Huffman coding is based on the inserted values in the above example using the priority queue. printTree(...) is used to print out recursively (with input argument vector<char>& bitString)). Here is an example of how to invoke the printTree(...)

```

// t is a tree
vector<char> bitString;
t->printTree(bitString);

```

The following parts should be included in the report:

- *Intuitive description of the algorithm*
- *Abstract algorithms in pseudo code*
- *Time complexity analysis in $O(\cdot)$ -notation*
- *C/C++ code*

A.6 Appendix F

This appendix shows the ANOVA test of correctness between prompting techniques in the Social

Data set statistics

Sample name	Number of samples	Mean	Standard deviation	Standard error of the mean	Median
Instructive	30	0.833	0.834	0.152	1.000
Q&A	30	0.767	0.774	0.141	1.000
Contextual	30	0.667	0.711	0.130	1.000
Mixed	30	0.500	0.630	0.115	0.000

Test summary

Test name	Kruskal-Wallis one-way ANOVA
Total number of samples	N = 120
Groups	<ul style="list-style-type: none">• Instructive: Count =30, Mean rank=65.733• Q&A: Count =30, Mean rank=63.633• Contextual: Count =30, Mean rank=59.867• Mixed: Count =30, Mean rank=52.767
Test statistic	$H(3) = 2.868$
Statistical significance (Two-tailed)	$p = 0.413$
Conclusion	Based on a p -value threshold of 0.05, there is not a statistically significant difference between: {Instructive,Q&A,Contextual,Mixed}.

Network task.

A.7 Appendix G

This appendix shows the ANOVA test of correctness between prompting techniques in the Huff-

Data set statistics

Sample name	Number of samples	Mean	Standard deviation	Standard error of the mean	Median
Instructive	30	1.467	1.525	0.278	1.000
Q&A	30	0.900	1.447	0.264	0.000
Contextual	30	1.533	1.502	0.274	1.500
Mixed	30	1.033	1.402	0.256	0.000

Test summary

Test name	Kruskal-Wallis one-way ANOVA
Total number of samples	N = 120
Groups	<ul style="list-style-type: none">• Instructive: Count =30, Mean rank=65.200• Q&A: Count =30, Mean rank=53.033• Contextual: Count =30, Mean rank=67.450• Mixed: Count =30, Mean rank=56.317
Test statistic	$H(3) = 4.410$
Statistical significance (Two-tailed)	$p = 0.220$
Conclusion	Based on a p -value threshold of 0.05, there is not a statistically significant difference between: {Instructive,Q&A,Contextual,Mixed}.

man Encoding task.

A.8 Appendix H

This appendix shows the ANOVA test of correctness between LLMs in the Social Network task.

Data set statistics

Sample name	Number of samples	Mean	Standard deviation	Standard error of the mean	Median
Gemini	40	0.425	0.675	0.107	0.000
ChatGPT-4	40	0.725	0.905	0.143	0.000
Github Co-pilot	40	0.900	0.496	0.078	1.000

Test summary

Test name	Kruskal-Wallis one-way ANOVA
Total number of samples	N = 120
Groups	<ul style="list-style-type: none">• Gemini: Count =40, Mean rank=48.562• ChatGPT-4: Count =40, Mean rank=59.913• Github Co-pilot: Count =40, Mean rank=73.025
Test statistic	$H(2) = 11.801$
Statistical significance (Two-tailed)	$p = 0.003$
Conclusion	Based on a significance level of 0.05, there is a statistically significant difference between: {Gemini, ChatGPT-4, Github Co-pilot}.

A.9 Appendix I

This appendix shows the ANOVA test of correctness between LLMs in the Huffman Encoding Data set statistics

Sample name	Number of samples	Mean	Standard deviation	Standard error of the mean	Median
Gemini	40	0.225	0.768	0.121	0.000
GitHub Co-pilot	40	1.925	1.639	0.259	3.000
ChatGPT-4	40	1.550	1.319	0.209	2.000

Test summary

Test name	Kruskal-Wallis one-way ANOVA
Total number of samples	N = 120
Groups	<ul style="list-style-type: none">• Gemini: Count =40, Mean rank=39.075• GitHub Co-pilot: Count =40, Mean rank=74.713• ChatGPT-4: Count =40, Mean rank=67.713
Test statistic	$H(2) = 29.189$
Statistical significance (Two-tailed)	$p < 0.001$
Conclusion	Based on a significance level of 0.05, there is a statistically significant difference between: {Gemini, GitHub Co-pilot, ChatGPT-4}.

task.

A.10 Appendix J

This appendix shows an ANOVA test of completeness between LLMs in the Social Network task.

Data set statistics

Sample name	Number of samples	Mean	Standard deviation	Standard error of the mean	Median
Gemini	40	2.000	0.000	0.000	2.000
GitHub Co-pilot	40	1.775	0.423	0.067	2.000
ChatGPT-4	40	1.925	0.350	0.055	2.000

Test summary

Test name	Kruskal-Wallis one-way ANOVA
Total number of samples	N = 120
Groups	<ul style="list-style-type: none">• Gemini: Count =40, Mean rank=66.000• GitHub Co-pilot: Count =40, Mean rank=52.612• ChatGPT-4: Count =40, Mean rank=62.888
Test statistic	$H(2) = 12.980$
Statistical significance (Two-tailed)	$p = 0.002$
Conclusion	Based on a significance level of 0.05, there is a statistically significant difference between: {Gemini, GitHub Co-pilot, ChatGPT-4}.

A.11 Appendix K

This appendix shows the ANOVA test of correctness between LLMs in the Huffman Encoding Data set statistics

Sample name	Number of samples	Mean	Standard deviation	Standard error of the mean	Median
Gemini	40	1.225	0.577	0.091	1.000
GitHub Co-pilot	40	1.725	0.452	0.071	2.000
ChatGPT-4	40	1.775	0.530	0.084	2.000

Test summary

Test name	Kruskal-Wallis one-way ANOVA
Total number of samples	N = 120
Groups	<ul style="list-style-type: none">• Gemini: Count =40, Mean rank=41.525• GitHub Co-pilot: Count =40, Mean rank=67.688• ChatGPT-4: Count =40, Mean rank=72.287
Test statistic	$H(2) = 25.088$
Statistical significance (Two-tailed)	$p < 0.001$
Conclusion	Based on a significance level of 0.05, there is a statistically significant difference between: {Gemini, GitHub Co-pilot, ChatGPT-4}.

task.

A.12 Appendix L

Huffman Encoding - Compiled och Un-compiled

