KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Technology

First cycle, 15 credits

Problem Solving Using Automatically Generated Code

EMIR CATIR
ROBIN CLAESSON

Stockholm, Sweden, 2023

Problem Solving Using Automatically Generated Code

EMIR CATIR

ROBIN CLAESSON

Degree Programme in Information and Communication Technology
Date: May 25, 2023

Supervisor: Rodothea Myrsini Tsoupidi
Examiner: Fadil Galjic
School of Electrical Engineering and Computer Science
Swedish title: Problemlösning med automatiskt genererad kod

Abstract

Usage of natural language processing tools to generate code is increasing together with the advances in artificial intelligence. These tools could improve the efficiency of software development, if the generated code can be shown to be trustworthy enough to solve a given problem. This thesis examines what problems can be solved using automatically generated code such that the results can be trusted.

A set of six problems were chosen to be used for testing two automatic code generators and the accuracy of their generated code. The problems were chosen to span a range from introductory programming assignments to complex problems with no known efficient algorithm. The problems also varied in how direct their descriptions were, with some describing exactly what should be done, while others described a real-world scenario with a desired result.

The problems were used as prompts to the automatic code generators to generate code in three different programming languages. A testing framework was built that could execute the generated code, feed problem instances to the processes, and then verify the solutions that were outputted from them. The data from these tests were then used to calculate the accuracy of the generated code, based on how many of the problem instances were correctly solved.

The experimental results show that most solutions to the problems either got all outputs correct, or had few or no correct outputs. Problems with direct explanations, or simple and well known algorithms, such as sorting, resulted in code with high accuracy. For problems that were wrapped in a scenario, the accuracy was the lowest. Hence, we believe that identifying the underlying problem before resorting to code generators should possibly increase the accuracy of the code.

Keywords

automatic code generation, code accuracy, natural language processing, GitHub Copilot, CodePal, problem-solving

Sammanfattning

Användningen av verktyg som bygger på språkteknologi för att generera kod har ökat i takt med framstegen inom artificiell intelligens. Dessa verktyg kan användas för att öka effektiviten inom mjukvaruutveckling, om den genererade koden kan visas tillförlitlig nog för att lösa ett givet problem. Denna avhandling utforskar vilka problem som kan lösas med automatiskt genererad kod på en nivå sådan att resultaten kan dömas tillförlitliga.

En mängd på sex olika problem valdes för att testa två olika kodgenererande verktygs noggrannhet. De utvalda problemen valdes för att täcka ett stort span av programmeringsproblem. Från grundläggande programmeringsproblem till komplexa problem utan kända effektiva algoritmer. Problemen hade även olika nivåer av tydlighet i deras beskrivning. Vissa problem var tydligt formulerade med ett efterfrågat tillvägagångssätt, andra var mindre tydliga med sitt respektive förväntade resultat inbakat i problembeskrivningen.

De utvalda kodgenererade verktygen uppmanades lösa problem enligt sex problembeskrivningar på tre olika programmeringsspråk. Ett ramverk byggdes som skapade probleminstanser, exekverade den genererade koden och verifierade den utmatade lösningen. Resultaten användes för att beräkna den genererade kodens noggrannhet, baserat på hur många av de givna instanserna som lösts korrekt.

Resultaten från testerna visar att de flesta av de genererade lösningarna fick antingen alla eller inga instanser korrekt lösta. Problem med tydliga beskrivningar och enkla välkända algoritmer så som sortering, resulterade i kod med hög noggrannhet. För de mindre tydliga problemen, som resulterade i lägst noggrannhet, bör identifiering av det underliggande problemet öka kodens noggrannhet.

Nyckelord

automatisk kodgenerering, kods träffsäkerhet, språkvetenskap, GitHub Copilot, CodePal, problemlösning

Acknowledgments

We would like to thank our advisors at KTH Royal Institute of Technology, Fadil Galjic and Rodothea Myrsini Tsoupidi, for all of their valuable feedback and guidance through each step of this thesis.

Stockholm, May 2023
Emir Catir Robin Claesson

Contents

Chapter 1

Introduction

Natural language processing (NLP) is the computational process of analyzing human languages [1]. In recent years, many new tools using NLP have emerged that are able to assist with or even fully solve questions and problems stated by humans.

There are some tools that use NLP to specifically assist developers by automatically generating code from natural language. One automatic code generator (ACG) is GitHub Copilot, and a study by the team behind it found that the majority of developers using their tool were more productive and felt more satisfied with their work [2]. With an increase in demand for software developers [3], coding more efficiently and using less resources is becoming increasingly important.

While efficiency is an important part of software development, correctness of the code is an indispensable property. The current ACG are capable of generating code in a fraction of the time it takes developers to write it. However, the total time to a correct program might still be longer if time has to be spent debugging the generated code.

1.1 Background

Solving a problem at any level of difficulty requires combining prior knowledge with information and conditions given in a query and deducing at least one possible solution to the query. Within computer science, a common goal is to find algorithms that solve problems with accuracy and do so efficiently.

Computers have been used for a long time to assist humans with calculations and problem-solving. As the computational power over the years has increased, so has the expectations on what kind of problems computers can solve.

Many problems within computer science are known to be so hard that efficient solutions are yet to be found [4]. Hard problems often require complex solutions, and these solutions need to be correct. The correctness of a solution or algorithm is determined by the algorithm's capacity to give a correct output to its respective input.

For certain problems, a proposed solution may solve some instances of a problem, but not all. When encountering complex problems, this might be a better solution than not solving the problem at all. There are situations where human lives are in the hands of software, for example airplane autopilots and self-driving cars. For this type of software, it is critical that the output is correct every time.

1.2 Problem

No code should be blindly trusted, regardless if it is automatically generated or written by a developer; it should be reviewed before use. Reading and understanding code, so it can be properly reviewed, requires skill and experience. A high quality code review requires a high level of understanding of the project code base [5]. Gaining this understanding when the majority of a code base is automatically generated may prove difficult. There is no human author of the code that the reviewer could ask to explain the thought

and reasoning behind the solution. This implies that if automatically generated code is going to be used to a wide extent in a project, we need to trust that the code is correct. However, to claim that a solution gives correct answers every time is a hard task, and it is often enough that the solution is accurate to a certain degree. Hence, this thesis researches the following question:

RQ. What type of problems can be solved accurately with automatically generated code?

1.3 Purpose

The purpose of this thesis is to help students, teachers, and software developers understand the current limitations and trustworthiness of ACGs based on NLP. The thesis aims to contribute to the understanding of the current state-of-the-art in code generation technology, and provide insights to developers on when and how to use automatically generated code.

1.4 Goal

The main goal of this thesis is to present data on how often code generated from natural language gives a correct solution to an instance of the described problem. The secondary goal is to use this data to determine the reliability of these ACGs, and to identify potential sources of errors and limitations in the generation process.

1.5 Methodology

First, a literature study was done to get a fundamental understanding of NLP and its uses, as well as classical problems in computer science that served as the basis for the problems described to the code generators. Different code generators were also examined to select the ones to include in our research.

The sets of problems to test were developed with varying complexity in both the problems themselves and the descriptions of the problems. For each problem, an instance generator and solution verifier were developed to test the generated code.

The experimental research was then carried out, where each code generator was asked to generate code to solve each problem. These programs were then compiled and tested, using our instance generators and the corresponding solution verifier, to evaluate data on how often a correct answer was given.

Lastly, the results of the experimental research were analyzed and evaluated to reach a conclusion for our research question.

1.6 Delimitations

This thesis has limitations that must be recognized. Only a few different NLP based ACG are used to generate code, and there currently exists no established market leader. Hence, there is a limitation on whether the chosen ones are optimal for our purposes.

Moreover, both the sets of programming languages and the sets of problems are limited. This will reduce the area of where the results can be applied and which conclusions that can be drawn from the result. We will also not provide any proof of correctness in the generated solution, but only measure the ratio of correct and incorrect answers.

Another common measurement within code analysis is that of space and time complexity, this was not accounted for since the research question only sought to answer whether generated code was correct and if so to what extent.

1.7 Outline

Chapter 2 gives theoretical background information needed to understand the rest of the thesis. Chapter 3 describes research approach and methodology. Chapter 4 presents the chosen test problems. Chapter 5 presents our results, and chapter 6 discusses the answer to our research question and our research.

Chapter 2

Theoretical Background

This chapter begins with a short introduction to natural language processing. Further, a metric used in computer science to quantify algorithms is explained. This is followed by a brief introduction to different programming languages and automatic code generators. Lastly, some works that have done similar research as this thesis are presented.

2.1 Natural Language Processing

NLP is the computational process of analyzing human languages, both in its written and spoken form. This technology is used in digital assistants, chatbots, translators and many other areas [1, 6].

NLP can be divided into two subsets. The first subset is Natural Language Understanding that breaks down the structure and context of a sentence to understand it's meaning. The second subset is Natural Language Generation, which is used by programs to produce text in human languages [7]. To properly generate code, ACGs needs to be able to handle both of these subsets.

In the beginning of NLP development, the implementations were hard coded, based on syntax and semantics [1, 6]. These early solutions struggled with interpreting words that have different meaning based on context, such as

metaphors and homographs. Modern NLP uses machine learning, together with statistical analysis, to better derive meaning from texts [7, 8].

2.2 Time Complexity of Algorithms

The time needed for an algorithm to finish is expressed as a function based on the size of its inputs. This is known as the algorithm's time complexity. With this function, it is possible to evaluate algorithms in a generic fashion, without specifying a concrete instance of the input. The time complexity function gives an indicator of how well the running time of the algorithm scales as the input size grows [9]. Hence, it is important to grasp as to what factor the running time will grow in proportion to the input.

The running time of an algorithm depends on the size of the input, but can also be affected by indata. For example, the insertion sort algorithm will perform in linear time if given a completely sorted array or in quadratic time if given an array in reversed order. This is known as best and worst-case time complexity.

A mathematical notation is established to communicate the time complexity of an algorithm. Oftentimes, the input size is described using a single parameter variable. The time complexity function establishes the number of elementary operations the algorithm needs, to solve the problem. Other variables are used to further imply whether the running time consider the best, average or worst-case performance of that particular algorithm [10].

It is common to use the worst-case time complexity to describe the efficiency of a solution. An average time complexity will run the risk of shifting the debate from the algorithm's efficiency, to justifying that the input is truly random [11].

2.2.1 P Problems and NP Problems

As mentioned earlier, time complexity is a metric used by computer scientist to categorize different problems into sets of complexity. One of these sets is the set P. According to the definition, P is the set of all decision problems for which there is a polynomial time algorithm [12]. The problems that belong

to P are considered tractable with efficient algorithms. Efficiently means that they can be solved in polynomial time, in contrast to problems not in P [4].

There are problems that are difficult to a degree that a polynomial solution has yet to be found. Yet, there is no proof that these problems are impossible to solve with a polynomial running time algorithm [4], these are NP-problems. Categorization of these problems can still provide relevant insights, even though efficient solutions may not exist. If a problem can be shown to belong to the set NP, searching for an optimal solution will be in vain [13].

2.3 Programming Languages

There exists a myriad of programming languages, some are general purpose, others are more domain specific. This section gives a brief presentation of the languages that were chosen for generated code when using ACGs, and used to develop the required testing tools.

2.3.1 Python

The Python language is an open source, object-oriented [14], general-purpose programming language, maintained by the nonprofit organization Python Software Foundation [15]. It is interpreted at run time[14] allowing it to be cross-platform and run on Windows, Mac and Linux based machines [15].

Python is currently one of the most popular programming languages and used for widely different purposes including web development, machine learning and data analysis [14, 15]. It is one of the programming languages ACGs state to be the most proficient in, and is therefore one of the languages used to test them.

2.3.2 Java

Java is an object-oriented programming language released by Sun Microsystems in 1995 and is now developed by Oracle [16, 17]. Java is not compiled into machine code that can be executed on the CPU, it is instead compiled into bytecode which is interpreted and executed by the Java Virtual Machine (JVM) on the target system. This allows Java to be portable to many different platforms, since any Java program, once compiled in to bytecode, can run on any platform that has support for JVM. [17].

Java was first designed for firmware in home appliances [18], but because of its portability, it was early on adapted for web based programs [17, 18]. While the usage of Java for websites is now no longer the norm, Java is used on machines in practically all areas, and there are tens of billions of active JVMs today [19].

Java, being one of the most used object-oriented programming languages, is one of the languages chosen used for AGC tests in this thesis.

2.3.3 C

The C language is a general-purpose programming language developed by Dennis Ritchie [20]. It is a cross-platform imperative language that can run on any machine that supports C. It is known to be a fast language due to it simplicity and lack of compile-time error checking. It compiles down to a run-time model that is fast, but can miss critical errors in the code. [21].

Common areas of usage are those in need of machine independent, efficient and fast code. Many market leading operating systems are partly written in the language of C [22]. The language is one of the most popular imperative languages and was chosen to test for non object-oriented-language code generation.

2.3.4 C#

C# is a modern, versatile and popular language [23], developed by Microsoft [24]. It is an object-oriented programming language, with built-in support for functional programming techniques such as lambda expressions. The code compiles into an intermediate language that is then converted to machine code in a just-in-time style by the .NET runtime system, much like the JVM [24].

It can among other things be used for developing applications for desktop, web, and smartphones[24], and it is the language used to build the testing framework for this thesis.

2.4 Automatic Code Generators

There are several ACGs available that are quite capable. Most ACGs are trained on code available across the internet. This gives most ACGs the ability to solve common problems with ease. For students taking a first course in programming, an assignment might be to write a sorting algorithm. This would probably be solved by ACGs in seconds. How far the capabilities of ACGs stretch is a goal for this thesis. The following section describe the ACGs that where chosen to be evaluated in more detail.

2.4.1 GitHub Copilot

GitHub Copilot is released by GitHub and marketed as your AI par programmer, and is available as plugin for many popular development environments [25]. It was developed in partnership with OpenAI and powered by OpenAI Codex, a NLP model trained specifically for code generation [25, 26]. To achieve this it was partly using code from over 50 million public repositories on GitHub. It has the capability to write code in most common programming languages, though it is most proficient in Python [26, 27].

The main strength of OpenAI Codex is interpreting prompts in natural

languages describing a program or function, and generating code in response to that prompt. [26, 27] GitHub Copilot allows the developer to enter prompts for functions as comments, which Copilot then gives solutions to. It also passively gives suggestions for how lines of code and functions can be completed [25].

2.4.2 CodePal

CodePal is a platform that offers a range of services, including code generation, language translation and code simplification [28]. CodePal uses a web interface where the user can enter their prompt or code, and get the result on that webpage. Each query made to CodePal is saved in the users history, making it possible to go back and view previous result. This history is however not private, as any query made with any tool is visible to explore for all users, although which account that made query is hidden

2.5 Related Work

Even though computer aided problem-solving is nothing new, the usage of ACGs for writing code has seen a rise in popularity in the recent months [29]. The discussion often revolves around the foundational architecture and different implementations of ACG and their code validity, only rarely is ACGs problem-solving skills evaluated on modern coding problems.

2.5.1 Why are nlp models fumbling at elementary math?

A report by Sundaram *et al.* evaluates several different word problem solvers, with a focus on the architecture behind them, and discusses how different datasets will affect the problem solver [30].

2.5.2 Assessing the Quality of GitHub Copilot's Code Generation

Yetistiren *et al.* present a study that assesses the quality of Python 3.8 code generated by GitHub Copilot, focusing on validity, correctness and efficiency. They used a set of 164 problems, with associated unit tests released by OpenAI called HumanEval [31, 32].

The two areas most relevant to this thesis is the validity and the correctness. A generated program that did not produce any run time errors during testing was deemed valid. A program could be deemed correct, partially correct or incorrect, depending on how many of the associated unit tests were passed by the program.

They found that 91.5% of the programs were valid, but less than a third of the programs were fully correct, meaning they passed all unit tests [31].

2.5.3 The Robots Are Coming

Finnie-Ansley *et al.* use programming tests taken by first year computer science students and compared the results OpenAI Codex produced with the students' answers. They did this to explore how it could threaten the integrity of student assignments.

They found that OpenAI Codex scored 78% and 78.5% on the two tests, placing it over the 75th percentile when compared to the students. Their conclusion was that these new ACG tools will force a change in the way introductory programming is taught [27].

2.5.4 Do Users Write More Insecure Code with AI Assistants?

A paper by Perry *et al.* looks at how ACGs provide a false sense of security. By using ACGs developers may contribute to writing less secure code, while still being under the impression that they are writing code that is secure. This trust in the ACG is amplified if the ACGs can provide code at the same level or above as the developer [33].

Chapter 3

Method

This chapter covers the methods used to evaluate how well ACGs solve problems. Literature study and different methodologies are presented first. Thereafter, the method for developing the sets of problems is outlined. The main focus in the chapter is on the method for code generation and testing. The chapter ends with the short description of the development tools and the documentation.

3.1 Preliminary Research

This section describes the research methodology of this work, including preliminary research and the main research strategies.

3.1.1 Literature study

At an early stage, we performed a literature study to improve the understanding of the basis of the technologies that are relevant in this thesis. The main focus of the study was NLP. In particular, we review previous results on the effect of ACGs in solving programming assignments and investigate how ACGs are currently used to generate code that solves programming problems of different

levels of difficulty. Different methodologies were also researched to find the best approach to achieve the goals of the thesis.

The literature and material included books from the KTH library, which mainly focused on NLP and programming languages. It also included scientific reports from databases accessed through KTH, for example the digital libraries' dl.acm.org. and IEEE Xplorer. Lastly, it also included websites containing information and documentation regarding current ACGs.

3.1.2 Quantitative and Qualitative Methodologies

Quantitative research methods focus on finding the answers to questions such as ”how many?” and ”how often?”. It uses data collection methods based on random sampling and structured data collection instruments, and does calculations on this data to measure levels of occurrences. Because it is a statistical method based on numerical measurements, the results of quantitative research is often easy to present and compare with other results [34].

Qualitative research methods are, on the other hand, mainly focusing on understanding reasons and motivations, and use linguistic or visual data. The methods of collecting the data is often interviews, focus groups and observations. While it can capture areas that quantitative research can not, such as emotions and feelings, it has the downside of using more subjective results and is therefore harder to compare and replicate than quantitative methods [35].

A qualitative approach was used to choose which problems to generate code for using ACGs. Parameters taken into account were the likelihood of generating partly correct code and the problems' relevance from students-taking-programming-courses perspective.

The collection and evaluation of the data for our results uses a quantitative method where random problem instances were given to the generated code, and the ratio of correct and incorrect solutions were calculated.

3.2 Developing the set of problems

We wanted the set of problems to have a wide range of difficulty to get a fair picture of ACGs problem-solving capabilities, starting with introductory programming assignments, and ending with problems known to belong to NP. The easier problems were deemed to be solvable by a first year university student, and NP-problem are as mentioned yet to be solved efficiently but nonetheless solvable in many different ways, thus an ACGs solution would be of interest. We also wanted some variation in how direct the problem descriptions where, to gain insight in the impact that would have on ACGs capability to generate accurate code.

The problem descriptions were written in the way you would explain them, if they were supposed to be solved by hand. This was made with the intention of avoiding programming terminology that unintended would lead the ACGs in certain directions. Problem descriptions were also written with the intention to avoid ACGs recognizing problems that are considered to be famous or well known, and thus may be part of ACGs training data.

The first step was to look at problems given to us in courses we have taken, as we wanted problems from, or equivalent of, early and later stages of computer science educations. While doing this, we intentionally chose not to include problems that focused on implementing classic data structures, such as stacks and specific sorting algorithms, as there is a countless number of implementations for ACGs to just copy.

After that, we researched various internet sources for problems that would be of relevance. Apart from problem taken from courses at KTH, Advent of Code was used for inspiration and as a problem description source.

Next we looked at known NP-problems. Here, we focused on finding ones that we though possible to reformulate into problem descriptions consistent with our other problems, such as graph coloring and the traveling salesman problem.

Lastly, we looked at the selected problems and which areas they covered, to then be able to create a few own problems that filled in those gaps. These problems were more direct in their description than most of the others, mostly

focusing on direct tasks thought to be how ACGs are most used today.

3.3 Data Collection and Evaluation

This section describes the methods to collect and analyze the data used to answer the research question.

3.3.1 Code Generation

Generated code based on NLP is subject to the variability of the written language, including but not limited to sentence structure, formulation, punctuation, and context. For the code generation phase, the problem descriptions were given to the different ACGs, the generated code was saved, and compiled if necessary.

The prompts given to the ACGs to generate code for consisted of three parts. The first part was the problem description in natural language, explaining what problem the generated code should solve. The second part described how the input text for a problem instance was going to be formatted. The last part described how the output of the generated solution for a problem instance should be formatted. The standardized input and output formats were chosen to be able to automate the testing of the code.

3.3.2 Code Testing Methodology

To test the code from the ACGs, a framework was built that automated the testing of the solutions. The framework contained an abstract test class with functions to generate input data, run the generated code and then verify the given solution. Then, each problem had a test that inherited this base test, with the functions for generating the input and verifying the solution overridden to the specifications of that problem. This system allowed us to use the same structure both for problems with only one possible solution, and for problems with multiple possible solutions.

Figure 3.1 shows the flow of how the generated code was tested. The first step of the test is to generate all the input data for the problem instance in the form that is described in the problem prompt. Then, the ACG generated program is started as a system process, and the input data is redirected into its standard input. When the program is done, the solution is read from the programs standard output. This solution, together with the problem instance, is then given to the solution verifier to assess if the solution is correct or not. The result of this test is saved together with the problem instance and the generated programs solution. These steps were repeated 1000 times for each generated program and the results were written to a file to later be analyzed.

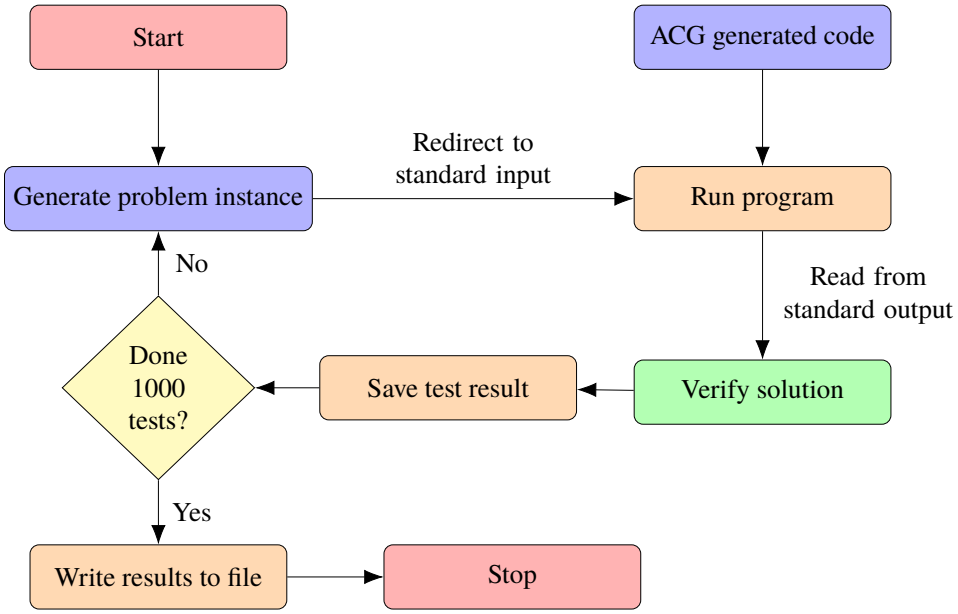

Figure 3.1: Flowchart for testing generated code

3.3.3 Measuring Code Accuracy

After the results from a test was done another developed tool was used to analyze the results and calculate the accuracy of the generated code.

The accuracy measurement is referring to the percentage of correctly solved problem instances. Due to the nature of problems, a given problem instance may vary greatly from another, even though both instances are solved using the

same algorithm. The accuracy measurement is deemed to be the amount of correctly solved problem instances s, divided by the total number of problem instances t prompted to be solved, by the selected ACGs algorithm. Accuracy is then given by the formula $Accuracy = s/t$. A given problem instance is considered to be correctly solved if deemed correct by the verifier.

3.4 Development tools

The testing framework was written in C# in .NET 6 using Visual Studio as the integrated development environment. The C# language has a built-in library for starting external processes and redirecting their standard input and output. This feature made it simple to use the same test for testing different versions of generated code for the same problem, without any modification to the testing framework.

Visual Studio Code is a development tool with available extensions for both writing code in all common programming languages and extensions for the ACGs we tested, which makes the tool ideal for our project.

To compile the generated Java code, we used the Java compiler OpenJDK 17.0.6, and to run the generated Python code we used Python 3.10.10, both in Windows PowerShell. To compile the generated C code we used the GNU gcc 9.4.0 compiler on Ubuntu. The compiled C code was also tested in the same environment, with the testing framework compiled to run on Linux.

3.5 Documentation

All the problems, prompts and generated code, together with the source code for our testing framework and unprocessed raw data from our tests, are available online and accessible through the following link:

https://github.com/Emir-Catir-and-Robin-Claesson/Problem-solving-using-automatically-generated-code

Chapter 4

Selected Problems

For the experiments to provide results of relevance, the problems given to the ACGs were selected based on a number of parameters. The selected problems are aimed to cover a wide range of difficulty, from introductory programming assignments to known NP-hard problems. A few of the problems are relatively known or even famous problems within combinatorial optimization.

For each selected problem, we provide a description of the problems and the reasons why we selected it. Thereafter, the prompt that was given to the ACGs to generate the code solutions is presented. The last information for each problem is the descriptions of what input was generated, and how the ACG-generated solutions were verified.

4.1 Temperature Measurements

This problem was given as the first assignment in the course *ID1018 Programming I* [36], which is an introductory course in programming at KTH. The assignment consists of reading a set temperature measurements for a given amount of weeks, and then calculate the minimum, maximum, and average temperature, both for all the individual weeks and the entire period as a whole. This problem was chosen to see how well the ACGs handled this type of simple problem that mostly consists of reading data and calculating it.

4.1.1 Temperature Measurements Prompt

Temperature measurements are taken in one and the same place over a number of weeks. The measurements are taken a fixed number of times, the same number of measurements in each week. At the end of the measurement period, the collected data must be processed: for each week, the smallest, the largest and the average temperature is determined. The minimum, maximum and average temperature must also be determined for the entire measurement period. All measurements are decimal numbers with 1 point precision.

The input data is given through standard input on the following format. The first row contains the number of weeks and the number of measurements per week separated by spaces. The following rows contain the measurements for each week, one row per week, measurements separated by spaces.

The output data should be printed to standard output given in the following format. One row for each week contains the smallest, largest and average to 1 point precision temperature separated by spaces. One row contains the minimum, maximum and average to 1 point precision temperature for the entire measurement period separated by spaces.

4.1.2 Input Generation and Solution Verification

The program is given 4 to 11 weeks of measurements. Each week has the same amount of measurements, varying between 3 and 7. Each given temperature is a number with one decimal between -10.0 and 31.0.

This is a problem which only has one possible solution, so to verify a given solution we check if it is correct, given the specified input. However, there are differences in how programming languages performs the decimal number rounding to 1 point precision. We therefore accept a difference of 0.1 from the expected value as correct, in any calculated average temperature. Every occurrence of this is noted by the test and shown in the test result files. Any differences between the expected and actual value for any minimum or maximum temperatures is still deemed as incorrect, since no rounding should have been made on them.

4.2 Sorting People

Sorting elements in a list into a specified order is a fundamental skill in programming, which is why we chose this problem to be tested. This problem starts with giving a set of people with the attributes name, height, weight, age and hometown. These people need to be sorted in ascending order with the following priority:

1. Height
2. Weight
3. Age
4. Hometown
5. Name

4.2.1 Sorting People Prompt

We have a list of people with information about them, including name, age, weight, height and hometown. These people must be sorted in ascending order by the order of height, weight, age, hometown, and lastly name. Numbers are integers sorted by their value, words are sorted in alphabetical order.

The input data is given through standard input on the following format. The first row contains the number of people in the list. The following rows contain the data for each person, one row per person, with their name, age, weight, height, and hometown, all separated by spaces.

The output data should be printed to standard output given in the following format. The list of persons sorted according to the instructions, one row per person, with their name, age, weight, height, and hometown, all separated by spaces.

4.2.2 Input Generation and Solution Verification

The program was given a list of between 100 to 300 people. The names and home towns where each randomized from a predefined list of 100 different options. The names of the towns are names of Swedish towns and cities, and the names of the people are also common Swedish names. However, to avoid errors caused by the Swedish letters 'Å', 'Ä' and 'Ö', these letters have been replaced with 'A' and 'O'.

The heights are between 150 to 200 centimeters, the weights are between 50 and 100 kilograms, and the ages are between 20 to 75. As specified in the prompt all number values are integers.

To verify the programs solution we used C#'s collection library LINQ to order the input according to the instructions, and then compare the order of that list with the given solution.

4.3 Box Problem

This problem was given during a lecture in the course *DD2350 Algorithm, Data Structures and Complexity* [37]. It is a rewriting of the known NP-hard problem Subset Sum [38]. The problem was chosen due to its rather simple and straight forward description. The problem can be solved both with a suboptimal and optimal solution, where algorithms that finds the optimal solution run in exponential time. There exists many variations to this problem, but the variation that was given used only positive integers to keep the description simple.

The problem asks to solve the problem of putting items with different weights into boxes with a given weight capacity. There exists an infinite number of boxes, and the real problem is how to utilize the capacity of the boxes such that every box fits as many items as possible. An optimal solution would fill every box as efficiently as possible, and a suboptimal solution would only require that no box exceeds its capacity and all items are put in a box.

Since the known solutions all run in exponential time, rather small instances

were generated so that exhaustive search could find the optimal number of boxes needed.

4.3.1 Box problem prompt

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

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

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

4.3.2 Input Generation and Solution Verification

The input generation consists of generating a few random integers, one for each parameter. The needed parameters are the max load capacity for the boxes, the number of items, as well as a weight for each item. Both the box capacity and the number of items span a range of $3 - 10$. The items weights are limited to less than or equal to the box capacity.

The verification checks for an optimal solution to the generated input by utilizing an exhaustive recursive method that tries all possible solutions. Even though the prompt states that there are an infinite amount of boxes, it also states that the answer should be the least amount of boxes needed. Wrong answers are to be categorized into two different groups, depending on if the number of suggested boxes are less than or greater than the optimal number.

4.4 Advent of Code Problem

Advent of Code (AOC) is an advent calendar with a programming puzzle for each day from December first through 25th, but all problems are accessible anytime after they are published. The difficulty of the problems starts off low for each year, and then increases day by day [39]. The problems in AOC are described as a part of that years story, and a large part of solving them is to figure out what it actually is they want you to solve. This is why we chose to include one of the AOC problems for our testing, as it is a way to see if the ACGs can correctly interpret instructions that are less direct.

The problem we included is from December 17th in 2015 [40], and it is about in how many different ways you can divide 150 liters of eggnog into some specified smaller containers. The descriptive part of the prompt is copied directly from the first part of the problem on the AOC website. The parts describing the input and output of the problem are added by us to keep the format consistent with our other problems.

4.4.1 Advent of Code Problem Prompt

“The elves bought too much eggnog again - 150 liters this time. To fit it all into your refrigerator, you’ll need to move it into smaller containers. You take an inventory of the capacities of the available containers. For example, suppose you have containers of size 20, 15, 10, 5, and 5 liters. If you need to store 25 liters, there are four ways to do it: 15 and 10 20 and 5 (the first 5) 20 and 5 (the second 5) 15, 5, and 5 Filling all containers entirely, how many different combinations of containers can exactly fit all 150 liters of eggnog?”- Advent of code 2015 day 17 [40]

The input data is given through standard input on the following format. The first row contains the number of available containers. Each following line contains the capacity of a container. All capacities are integers and indicate liters.

The output data should be printed to standard output given in the following format. A line with the answer.

4.4.2 Input Generation and Solution Verification

The program is given between 15 to 20 number of containers, with sizes range from 5 to 50 liters. As every problem have exactly one solution, that solution is calculated and compared with the given output in the verification.

4.5 Traveling Salesman Problem

The traveling salesman problem [11] or TSP is a famous problem within computer science. Depending on the formulation, it is either an optimization problem or a decision problem. This problem was chosen due to its properties. The problem is known for being a NP-hard problem, meaning that no algorithms are known to solve it with a time complexity less than exponential. A description given by Garey and Johnson is the following. Given a set of cities $c_1, c_2...c_i$ and distances between each pair, noted as $d_1, d_2...d_j$. How should the salesman travel, such that he visits every city once and travels the shortest possible distance?

For this problem, due to the nature of being a rather known problem, the problem prompt was rewritten as a teacher handing out tests to students in a class, where the students are sitting with different distances between each other and the teacher wants to visit every student once while traveling the shortest distance.

4.5.1 Traveling Salesman Prompt

In a classroom, there is a teacher and a set of students. The teacher has a list of all students, as well as the distance between each pair of students. The teacher wants to hand out tests to all students, passing every student, but no more than once, while taking the shortest path. The students are numbered with a unique student ID, such that student number one has the unique student ID 1 and so on. The teacher must start at the student with unique ID 1 and must also return to this student after visiting all other students. Come up with the shortest path that traverses every student only once and returns to the starting student.

The input will be given to standard input in this order. The first row contains the total number of students. The following rows, one for each pair of students, contain three numbers separated by spaces where the first number is the student ID for a student, the second number is the ID for another student and the third number is the distance between the two students.

The output should be printed to standard output in this order. The first row should contain the total distance traveled. The second row should be the path that the teacher should take, consisting of the student IDs in the order the teacher should walk, where each student ID is separated by a space.

4.5.2 Input Generation and Solution Verification

This problem as well as the Box Problem (problem 4.4) is a known NP problem and thus no efficient solutions are currently known. The input generation is somewhat similar to the other problems and consists of generating a random number within a specified range. The input for the problem consists of a number of students ranging from 4 to 12, and a distance between the students ranging from 1 to 10.

The verification for this problem is, in contrast to the box problem, not an exhaustive search for an optimal solution and instead merely checks that the given answer is valid. For an answer to be considered as valid, the answer needs to choose a path that actually exists, meaning that a path will consist of student numbers from 1 up to the randomized total number of students (4-30). More so, the answer needs to pass all students, but only once, and return to the starting student while correctly calculating the sum of the distance traveled.

4.6 Electrical Outlets Problem

This is a problem which was used in for examining the ability to construct efficient algorithms in the course *DD2350 Algorithms, data structures and complexity* [37] at KTH during the fall of 2022, a mandatory programming course for computer science students in their third year.

The point of the problem is to figure out how to connect power outlets to lamps for an orchestra's sheet stands. There is only one outlet in the wall, so a network of extension cords needs to be constructed to provide power to each lamp. Given a matrix of the distances between the sheet stands, a network that uses the smallest total possible length of extension cord should be found.

There are three reasons why we chose to include this problem to be tested on the ACGs. Firstly, this problem is a good rewrite of the known graph problem minimum spanning tree [41]. Secondly, it was chosen to be able to compare ACGs accuracies of this problem and the temperature problem, since both of them are given as student assignments in differnt stages of the education. Lastly, it was chosen because it contains an element of finding the optimal solution without being a NP-complete problem.

4.6.1 Electrical Outlet Prompt

In a concert hall, the musicians sit in designated places, and each music stand has its own lamp with a lamp cord that just barely reaches the floor. In order for all music stand lamps to be powered, an electrician needs to build a network of branch sockets and cords, where the number of sockets and the length of the cords are adjusted by the electrician according to need. There needs to be an electrical outlet under each music stand, and branch sockets are only allowed to be placed under the music stands. There is only one wall outlet in the concert hall, and it sits 150 centimeters from the nearest music stand. The problem is to design a network of cords with power outlets that use as little cord (power cable) as possible and no unused power outlets.

The input data is given through standard input on the following format. The first row contains N, the number of music stands. The following rows are each a row of the NxN matrix, with the numbers separated by spaces, and place (i, j) indicates how long a cord would be required to be to reach from stand i to stand j. The Nth music stand is the one closest to the wall outlet. The length of the cords are positive integers, representing the length in centimeters.

The output data should be printed to standard output in the following format. A row for each of the branch sockets used to build the network that powers the lamps. Each row should print the length of the cord, the number of sockets

it has, the number of the music stand where the cord is plugged in, and the number of the music stand it lies beneath, all separated by spaces. The wall outlet is considered as music stand 0.

4.6.2 Input Generation and Solution Verification

This problem required a symmetrical matrix generated as input, specifying the distance between the different sheet stands. The number of sheet stands is a randomly generated number in the range of 4 to 10, and the distances between the sheet stands are a random generated number in the range of 100 to 400. The graph that is generated is represented by a symmetrical adjacent matrix.

As there can exist more than one minimum spanning tree with the same root in a graph, there can exist more than one possible solution for each problem instance. The output from the test is verified by first checking that the given solution is a tree that reaches all the sheet stands. It is also verified that the distances between the sheet stands are correct, and that the number of sockets a cord has corresponds to how many other cords say that they are plugged in there.

The verifier then utilizes Prim's algorithm [41], a well known algorithm, to find a minimum spanning tree with root at the power outlet. The sum of the edges in the tree from the output is then compared to the sum of the edges in the tree from Prim's algorithm to verify that the tree from the output is minimum spanning tree.

Chapter 5

Results

This chapter presents the results from running the ACG-solutions for the problems stated in chapter 4. A total of 6 different problems were given to 2 different ACGs, Github Copilot and CodePal. As Github Copilot allows for generating more than 1 solution for each problem, it was used to generate 2 solutions, when only 1 solution was generated for each problem using CodePal, the solutions were evaluated for accuracy by running 1000 randomly generated instances and verifying the ACG-solution output.

5.1 Summary of the Results

The results for different ACGs are summarized below. Figure 5.1 and 5.2 presents the results acquired from running the code generated by CodelPal and GitHub Copilot. In these figures the following abbreviations are used; AoC, is the Advent of code problem, Boxes is the repeated subset sum problem involving packing boxes, Outlets refers to the electrical outlet problem, TSP refers to the rewritten traveling salesman problem, Sorting is the sorting people problem, and Temp refers to the temperature measurement problem.

The results show that CodePal generated accurate solutions for both the temperature and sorting problems. For the advent of code and traveling salesman problems, it got perfect score on two out of three generated solutions.

The boxes problem gave less polarizing results, as all three languages had an accuracy around 75%. As for the electrical outlets problem, no correct solutions were generated.

GitHub Coplilot had more fluctuating results, and did not produce solutions with 100% accuracy to any problem. The temperature, sorting and advent of code had the most accurate solutions, with only a few solutions having low or zero accuracy. The solutions to the boxes problem gave similar results as CodePal, with an accuracy of around 75%, again with one outlier having low scores. The traveling salesman problem had only one solution with accuracy over zero, and the electrical outlets problem had no correct solutions, just as for CodePal.

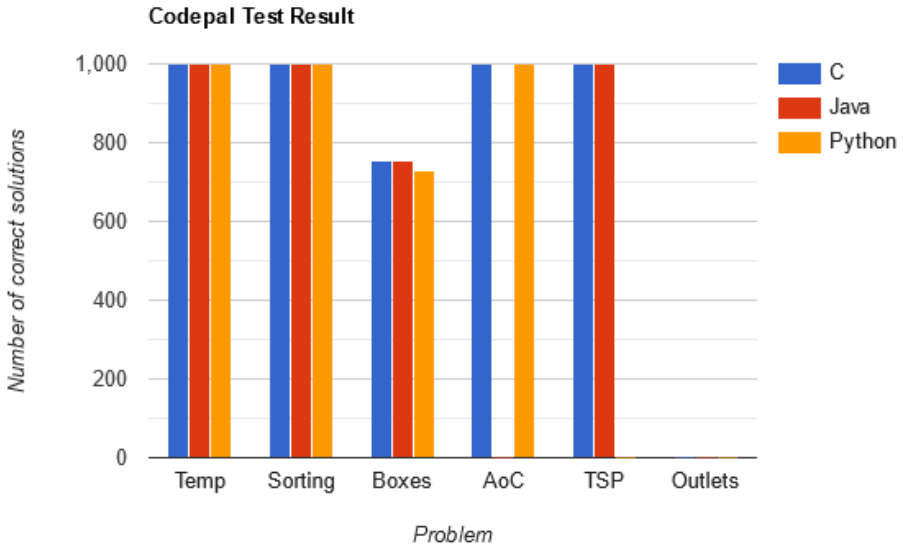

Figure 5.1: CodePal overall test result

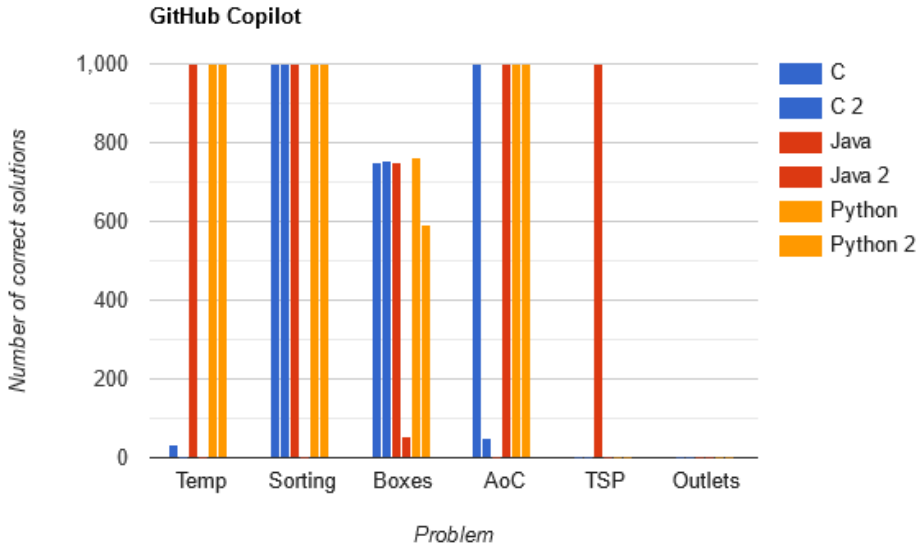

Figure 5.2: Github Copilot overall test result

5.2 Temperature Measurements Results

The results for the temperature measurements problem, given in Table 5.1 show that these two ACGs handles this type of data processing well in most cases. All the code generated by CodePal produced the correct output for every given problem instance, regardless of the programming language.

Table 5.1: Temperature problem test results

ACG	**Python**		**Java**		**C**	
	Correct	Accuracy	Correct	Accuracy	Correct	Accuracy
GitHub Copilot 1	1000	100%	1000	100%	32	3%
GitHub Copilot 2	1000	100%	1	0%	0	0%
CodePal	1000	100%	1000	100%	1000	100%

GitHub Copilot also got perfect accuracy on both of its Python code, and on the first generated Java code as well. GitHub Copilot's second generated Java

code produced only one correct output, and otherwise failed to provide either the correct global min, max or average. The C programs generated by GitHub Copilot performed poorly, with both solutions failing to calculate the correct global average. The first solution managed to get it right 32 times, and was otherwise often within two degrees. The second program however did not produce any correct answers, and was often wrong with around 10 degrees.

5.3 Sorting People Results

The sorting people problem was where the ACG performed the best, generating 100% accurate code almost across the board. As can be seen in Table 5.2, all but one of the generated programs successfully sorted all 1000 of the given lists of people.

Table 5.2: Sorting people problem test results

ACG	**Python**		**Java**		**C**	
	Correct	Accuracy	Correct	Accuracy	Correct	Accuracy
GitHub Copilot 1	1000	100%	1000	100%	1000	100%
GitHub Copilot 2	1000	100%	Runtime error		1000	100%
CodePal	1000	100%	1000	100%	1000	100%

The second solution generated from GitHub Copilot failed every test due to runtime errors caused by it interpreting every field of the persons' information as integers, and therefore tried to parse them as such. This then caused Java to throw an exception when reaching the hometown in the sorting priority.

5.4 Box Problem Results

The box problem, one of the NP-hard problems given to the ACGs, had interesting results that differed from many of the other problem's test results in the way that all ACG solutions got some of the instances correct, as seen

in Table 5.3. The problem prompt was clear, and the output requires only a number that lies in a rather small range of numbers.

Table 5.3: Boxes problem test results

ACG	Python		Java		C	
	Correct	Accuracy	Correct	Accuracy	Correct	Accuracy
GitHub Copilot 1	762	76%	751	75%	748	75%
GitHub Copilot 2	591	59%	56	6%	754	75%
CodePal	728	73%	753	75%	756	76%

Among the failed attempts, the reason for failing differed. All CodePal solutions suggested a number of boxes greater than the optimal amount. For the GitHub Copilot solutions, some suggested a number of boxes greater than the optimal amount or a too low number of boxes, whose capacity sums to less than that of the weight sum of all items.

5.5 Advent of Code Results

The Advent of Code problem results, presented in Table 5.4, follows the general pattern amongst the problems, with most solutions producing either successes for all instances or fails for all instances. All solutions in Python, both solutions generated with GitHub Copilot and CodePal, scored 100% accuracy in the tests. In Java, the CodePal solution, and one of the GitHub Copilot solutions, got perfect accuracy while the other GitHub Copilot solution failed every test.

The generated solutions for C had results where a solution did not get everything right or wrong, as the second solution from GitHub Copilot produced only a few correct solutions, resulting in an accuracy of only 5%.

Table 5.4: Advent of code problem test results

ACG	Python		Java		C	
	Correct	Accuracy	Correct	Accuracy	Correct	Accuracy
GitHub Copilot 1	1000	100%	0	0%	1000	100%
GitHub Copilot 2	1000	100%	1000	100%	51	5%
CodePal	1000	100%	0	0%	1000	100%

5.6 Traveling Salesman Results

The traveling salesman problem, or TSP, was thought to be one of the more difficult problems due to the problem prompt being rewritten to conceal the original problem. Here the Python solutions generated with GitHub Copilot failed to solve a single instance, and CodePals recognized the problem but refused to provide a solution, which is why Table 5.5 says ”no code given” in that field. Instead, the CodePal response to the problem prompt was a short text, explaining how it would require solving the traveling salesman problem, which was too complex for a single function.

Table 5.5: Traveling salesman problem test results

ACG	Python		Java		C	
	Correct	Accuracy	Correct	Accuracy	Correct	Accuracy
GitHub Copilot 1	0	0%	1000	100%	0	0%
GitHub Copilot 2	0	0%	0	0%	0	0%
CodePal	No code given		1000	100%	1000	100%

In Java, one GitHub Copilot and the CodePal generated solution scored 100% accuracy, while only one of the GitHub Copilot generated solutions failed all of its instances. In C, GitHub Copilot struggled to produce code that correctly solved the problem, and did not succeed with either of its attempts. However, CodePal had no problems interpreting the rewritten TSP problem prompt and produced a solution that correctly solved all instances for an accuracy of 100%.

5.7 Electrical Outlets Results

This problem was shown to be difficult for the ACGs to handle. All the tests, presented in Table 5.6, resulted in an accuracy of 0%. In fact, several of the given solutions suggested code consisting of algorithms that solved different problems.

Table 5.6: Electrical outlets problem test results

ACG	**Python**		**Java**		**C**	
	Correct	Accuracy	Correct	Accuracy	Correct	Accuracy
GitHub Copilot 1	0	0%	0	0%	0	0%
GitHub Copilot 2	0	0%	Code not complete		0	0%
CodePal	Runtime Error		0	0%	No code given	

The closest to be correct was the Java code from CodePal, which produced input that at first glance could be correct. However, it missed placing a power cord under the Nth stand, which was the stand closest to the wall outlet. This is the first test performed by the verifier, so there are likely to be other errors as well that did not get detected, as the verifier aborts at any found error.

The output from GitHub Copilot's first generated Java code and second C code, while clearly wrong, also matches the format described in the prompt. The most clear error for both of them is that they give every cord length zero, and gives each power cord only one outlet, which makes it impossible to chain them as would be needed.

The rest of the generated code that was tested either produces runtime errors, prints only zeroes, or only calculates some version of total distances between the music stands. The second attempt, with GitHub Copilot, didn't give a complete program, even when being asked to continue from a previous incomplete solution from itself.

Lastly, when CodePal got asked to generate a solution in C it, just like for the traveling salesman problem, instead gave a response as answer where it stated that the problem was too hard. It identified that it would be a complex problem

that could be solved with Kruskal's or Prim's algorithm. This means that it once again correctly identified the underlying problem in the text, but realized it was too complex for the amount of code it is designed to generate.

5.8 Quality of the Generated Code

This section focuses on the code itself, rather than the results of running the code as in previous sections. It looks at the generated code quality, such as how the code in general is organized, how readable it is and how easy it would be to make corrections. Just as the accuracy of the code varies depending on the problem, the ACG and the programming language, so does the quality.

One thing that consistently keeps a high quality are the names of the variables and functions. A good example of this is in CodePal's Java solution for the temperature problem, presented in listing 5.1. Here, the variable names make it clear that they are intended to store information about temperatures with certain attributes. This clear variable naming helps making the code easier to read, and is persistent through all generated code, with the exception of the loop counters. The naming of these loop counters follows the naming convention of `i, j, k, ...`, which is common in most code standards. While this is often the way these variables are named by programmers, it becomes difficult to follow when there are many nested loops, as for example in GitHub Copilots second solution for the AOC problem in Listing 5.2.

How much of the code that is explained with comments is not regular. Some of the generated code have comments like the ones in Listing 5.1, but not all of them. CodePal has a higher frequency of comments than GitHub Copilot, but there are other factors that play in. Firstly, Python and Java code have more comments than C code. For the C code there are only three generated functions that contain any comments at all. The length of the code also seems to affect whether there are any comments or not. Short programs, such as GitHub Copilot's sorting solution in Listing 5.3 seldom have any generated comments with the code. Lastly, the problems with low accuracy also show a tendency to lack comments. For example, there is only one of the solutions for the electrical outlets problem that contains comments.

The ACGs are trained to write code like humans, so that humans can

understand and use it, and they do succeed at that. All generated solutions looks like they are written by a human. This makes it no harder to understand, debug, and change this code than any other code written by another person.

```java
// Initialize variables to hold the minimum,
// maximum, and average temperature for each
// week and for the entire measurement period
double[] minTemps = new double[numWeeks];
double[] maxTemps = new double[numWeeks];
double[] avgTemps = new double[numWeeks];
double overallMin = Double.MAX_VALUE;
double overallMax = Double.MIN_VALUE;
double overallSum = 0.0;
```

Listing 5.1: Variable declaration in CodePal's Java solution for the temperature problem

```c
for (i = 0; i < num_containers; i++){
  for (j = i+1; j < num_containers; j++){
    for (k = j+1; k < num_containers; k++){
      for (l = k+1; l < num_containers; l++){
        for (m = l+1; m < num_containers; m++){
          for (n = m+1; n < num_containers; n++){
            sum = container[i] + container[j]  +
 ↪  container[k] + container[l] + container[m] +
 ↪  container[n];
            if (sum == 150) {
              count++;
            }
```

Listing 5.2: Nested loops in GitHub Copilots second C solution for the AOC problem

```
n = int(input())
  people = []
    for i in range(n):
      name, age, weight, height, home_town =
       ↪  input().split()
        people.append([name, int(age), int(weight),
         ↪  int(height), home_town])
    people.sort(key=lambda x: (x[3], x[2], x[1],
     ↪  x[4], x[0]))
    for person in people:
        print(*person)
```

Listing 5.3: GitHub Copilots full second Python solution for the sorting problem

Chapter 6

Discussion and Conslusions

This chapter starts with answering the research question of this thesis. The chapter then continues with evaluation of the work done, including problems, methods of testing the generated code, and the reliability of the results. At last, we discuss any future research on this topic.

6.1 Answering the Research Question

The research question in the thesis is:

RQ. What type of problems can be solved accurately with automatically generated code?

When examining the data in the results, it seems that two key factors play a part in how well the chosen ACGs can solve a problem. One of these factors is the level of complexity required by the solution, and the other one is the problem description.

Problems that are straightforward in their description and do not require a complex solution seem to be accurately solved by automatically generated code. The problem with the highest average accuracy was the sorting problem, where the automatically generated code had an accuracy of 100% on 8 out of

the 9 solutions. The electrical outlets problem was the only problem to get 0% accuracy on all 9 solutions. This supports the conclusion that problems that lack a deeper level of complexity are more likely to be correctly solved by an ACG, while complex problems seem to be difficult for the ACGs to get right.

The drawn conclusion is that the problems that can be accurately solved with automatically generated code are problems with direct descriptions and known algorithms. For problems that are less direct, it would be beneficial for the programmer to identify the underlying problem before feeding it to the ACGs.

6.2 Thoughts on various topics

The following section presents an evaluation on the different topics involved in the thesis regarding the selection of the problems, the testing method, and the reliability of the tests.

6.2.1 Problem Selection

The selection of problems was difficult due to the many factors to take into account, when formulating the prompt from which the ACGs produce executable code. As mentioned earlier, the problem description seems to play a big role in what solution was suggested by the ACGs. A mere change of words in a sentence could produce different suggested solutions. If for example a prompt asking for a minimum value or shortest path changes to instead look for maximum value or longest path, the code might be very different. It is difficult, if not impossible, to argue for using the best possible prompt to describe a problem to the ACGs, especially since part of the evaluation is connected to the ACGs ability to interpret natural language. For each prompt in the tests, we tried to determine whether the prompt would be clear enough to understand from the perspective of being computer science students.

The problems were chosen to span a wide range of common programming challenges, considered to be relevant to students studying computer science. However, since only a few problems were chosen, it is hard to say if the results from these problems are sufficient to claim that ACGs problem-solving

capabilities are in line with that of a computer science student. We purposely chose problems to include what is to be considered fundamental programming such as sorting, and also problems that are considered complex. The results show that in this sense, the ACGs produce the expected result.

Based on the results, there is a difference in accuracy depending on what language the ACG operates in, regardless of the problem. The tested ACGs could, given the same problem and description, pinpoint a correct algorithm in one language but fail to do the same in another programming language. This is likely related to the data that the ACG is trained on.

6.2.2 Testing Method

The testing method consisted of randomly generating data within a desired range as input to each ACG solution and verifying the response output given by the solution. It should be noted that the amount of tested instances were not exhaustive in the way that all possible problem instances within the set range were tested. It is however clear, from the results, that the solutions that had a perfect accuracy likely would have the same accuracy even if there were an exhaustive number of instances tested. This argument holds for the solutions that had an accuracy of 0%.

There is also some critique to be made on our choice to randomize the problem instances for every test. If the tests for the boxes problem were to be run again, the results may differ marginally due to the randomness of the instances. The accuracy will most likely not change significantly, but there are arguments to be made that it is not optimal with such a testing method. A better alternative could have been to randomize the problem instances first, and then used these on every test. This would make it so that all solutions were tested on the same instances, and the results of the test would not change if performed multiple times.

6.2.3 Reliability

These test results are to be expected, should the tests be reproduced. When prompting the ACGs, it was clear that for the same problem prompt, the ACG produced the same code even though prompted several times. For CodePal specifically, there was even a noticeable difference in the speed for generating an answer when asking to solve a problem recently solved, perhaps this is due to the produced answer being kept in a database for fast retrieval to the same prompts. GitHub Copilot on the other hand suggested up to 10 different solutions to any prompt, in which the order of suggested solutions could differ from prompt to prompt, but the first suggested solutions were the same.

It is, however, a good question whether the used ACGs are learning dynamically and will store the prompt, solutions and, the actions taken on it as a way to improve what future code to generate. It is also possible that GitHub Copilot can read other files in the same project as the current file, and reuse parts of their code. If this is the case, then the very same prompts might lead to different results should the test be reproduced. For reproduction of the results, the code and presented results can be found in our GitHub repository.

Lastly, some of the problem descriptions were of a more complex nature and when these problems were given to the tested ACGs, they clearly misunderstood them. One of the difficulties when using ACGs is the lack of transparency. It is difficult for the user to rewrite the formulation to perhaps get a better result, e.g. an algorithm that solves the problem, due to the fact that the communication is rather one-way. When the ACGs produce code, they do not elaborate on whether it understood the assignment and, if not, what parts of the description it did not understand. This lack of transparency makes the ACGs less viable as a trustworthy tool for users that either have little or no coding experience, because the only way to understand what an ACG has difficulties with is to analyze the suggested code to search for misinterpretations.

6.3 Future Work

The ability to automatically generate code is sought after to say the least. The GitHub Copilot extension currently have close to 6 million downloads [42] and

IntelliCode, an AI-assisted tool utilizing machine learning to help developers with code completion through automatic code generation, have close to 30 million downloads [43]. This is in no way a measurement for the accuracy of automated code, but should be seen as a market demand for using code generators to help software development. This makes it also important to evaluate these tools.

To validate and more accurately define what problems ACGs can solve, it is suggested that more research is put into how prompts should be formulated to produce the best possible answer. Other improvements may include that of ACGs simply getting better at understanding natural language, as this would wage in on the prompts not needing to be as precisely defined to produce viable solutions.

Future work for this thesis could consist of using more problems and generating several different prompts for each problem. In that way, each prompt could be rewritten to better capture the ACGs ability to solve problems, based on the problems' formulation. As noted in chapter 5, the tested ACGs handled problems that were wrapped in a scenario differently from problems that were not. Another way could be to first carry research in the form of a questionnaire, to gather more data on which prompts that are the most understandable and clear to humans.

More ACGs exist than those included in this thesis, some of which may be more accurate in other programming languages. These could also be tested with more time. Some ACGs are open source projects, such as PolyCoder [44]. The code generation engine in these open source projects would be run on local machines, and not accessed through plugins or web-interfaces as the ACGs used in this thesis. Running these engines would require hardware with more computing power than we currently have.

References

[1] K. R. Chowdhary, "Natural language processing," in *Fundamentals of Artificial Intelligence*. Springer Nature India, 2020, pp. 603–649.

[2] E. Kalliamvakou, *Research: Quantifying github copilot's impact on developer productivity and happiness*, Sep. 2022. [Online]. Available: https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/ (visited on 02/06/2023).

[3] U. B. of Labor Statistics, *Software developers, quality assurance analysts, and testers : Occupational outlook handbook*, Feb. 2023. [Online]. Available: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm (visited on 02/06/2023).

[4] J. Kleinberg and T. Éva, "Chapter 8. np and computational intractability," in *Algorithm design*. Pearson/Addison-Wesley, 2014, pp. 451–531.

[5] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality," *Proceedings of the 38th International Conference on Software Engineering*, 2016. DOI: 10.1145/2884781.2884840. (visited on 02/06/2023).

[6] IBM, *What is natural language processing?* 2022. [Online]. Available: https://www.ibm.com/topics/natural-language-processing (visited on 02/16/2023).

[7] E. Kavlakoglu, *Nlp vs. nlu vs. nlg: The differences between three natural language processing concepts*, Nov. 2020. [Online]. Available: https://www.ibm.com/blogs/watson/2020/11/nlp-vs-nlu-vs-nlg-the-differences-between-three-natural-language-processing-concepts/ (visited on 02/24/2023).

[8] P. M. Nadkarni, L. Ohno-Machado, and W. W. Chapman, "Natural language processing: An introduction," *J. Amer. Med. Inform. Assoc*, vol. 18, no. 5, pp. 544–551, Sep. 2011.

[9] N. Karumanchi, "Introduction," in *Data Structures and algorithms made easy in Java: Data Structure and algorithmic puzzles*. CareerMonk Publications, 2020, pp. 16–17.

[10] F. Galjic, *Programmeringsprinciper i Java*, swe, 1. uppl. Lund: Studentlitteratur, 2013, ISBN: 9789144094427.

[11] J. Kleinberg and T. Éva, "Chapter 2. basics of algorithm analysis," in *Algorithm design*. Pearson/Addison-Wesley, 2014, pp. 29–71.

[12] K. Erciyes, "Introduction to the theory of computation," in *Discrete Mathematics and Graph Theory: A Concise Study Companion and Guide*. Cham: Springer International Publishing, 2021, pp. 197–218, ISBN: 978-3-030-61115-6. DOI: 10.1007/978-3-030-61115-6_10. [Online]. Available: https://doi.org/10.1007/978-3-030-61115-6_10.

[13] M. R. Garey and D. S. Johnson, *Computers and intractability*. freeman San Francisco, 1979, vol. 174.

[14] G. Hart-Davis, *Python* (Teach yourself visually), eng. Hoboken, New Jersey: John Wiley & Sons Inc., 2022, ISBN: 9781119860273.

[15] G. Rajagopalan, *A Python Data Analyst's Toolkit: Learn Python and Python-Based Libraries with Applications in Data Analysis and Statistics*, eng. Berkeley, CA: Apress L. P, 2020, ISBN: 9781484263983.

[16] *What is java technology and why do i need it?* [Online]. Available: https://www.java.com/en/download/help/whatis_java.html (visited on 03/28/2023).

[17] H. Schildt, "The history and philosophy of java," in *Java: A beginner's guide, ninth edition*. McGraw-Hill Education, 2022.

[18] J. O. Ogala and D. V. Ojie, "Comparative analysis of c, c++, c# and java programming languages," *Global Scientific Journals*, vol. 8, no. 5, pp. 1899–1903, May 2020.

[19] Oracle, *Java timeline*, 2020. [Online]. Available: https://www.oracle.com/java/moved-by-java/timeline/#2020 (visited on 04/05/2023).

[20] B. W. Kerninghan and D. M. Ritchie, in *The C programming language*. Prentice Hall, 1998, pp. 8–8.

[21] N. Parlante, *Essential c*, eng, 2003. [Online]. Available: http://cslibrary.stanford.edu/101/EssentialC.pdf (visited on 04/12/2023).

[22] Sep. 2022. [Online]. Available: https://www.mentofacturing.com/Vincent/implementations.html (visited on 04/13/2023).

[23] S. Cass, *Top programming languages 2022*, Nov. 2022. [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2022 (visited on 04/13/2023).

[24] *A tour of c#*, Feb. 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (visited on 04/13/2023).

[25] *Github copilot · your ai pair programmer*, 2022. [Online]. Available: https://github.com/features/copilot.

[26] *Openai codex*, 2022. [Online]. Available: https://openai.com/blog/openai-codex (visited on 03/09/2023).

[27] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, "The robots are coming: Exploring the implications of openai codex on introductory programming," *Australasian Computing Education Conference*, 2022. DOI: 10.1145/3511861.3511863.

[28] [Online]. Available: https://codepal.ai/user/history/item/644a770733c5e34e01788c5f (visited on 04/28/2023).

[29] J. Vanian, *Chatgpt and generative ai are booming, but the costs can be extraordinary*, Mar. 2023. [Online]. Available: https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html.

[30] S. S. Sundaram, S. Gurajada, M. Fisichella, S. S. Abraham, *et al.*, "Why are nlp models fumbling at elementary math? a survey of deep learning based word problem solvers," *arXiv preprint arXiv:2205.15683*, 2022.

[31] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of github copilot's code generation," *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022. DOI: 10.1145/3558489.3559072.

[32] M. Chen *et al.*, *Evaluating large language models trained on code*, 2021. DOI: 10.48550/ARXIV.2107.03374. [Online]. Available: https://arxiv.org/abs/2107.03374.

[33] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, *Do users write more insecure code with ai assistants?* 2022. DOI: 10.48550/ARXIV.2211.03622. [Online]. Available: https://arxiv.org/abs/2211.03622.

[34] *Quantitative data collection methods*. [Online]. Available: https://research-methodology.net/research-methods/quantitative-research/ (visited on 04/13/2023).

[35] *Qualitative data collection methods*. [Online]. Available: https://research-methodology.net/research-methods/qualitative-research/ (visited on 04/13/2023).

[36] [Online]. Available: https://www.kth.se/student/kurser/kurs/ID1018?periods=6&startterm=20202&l=en (visited on 05/17/2023).

[37] [Online]. Available: https://www.kth.se/student/kurser/kurs/DD2350?periods=6&startterm=20222&l=en (visited on 05/17/2023).

[38] J. Kleinberg and T. Éva, "Chapter 6. dynamic programming," in *Algorithm design*. Pearson/Addison-Wesley, 2014, pp. 451–531.

[39] E. Wastl, *Advent of code: About*, 2015. [Online]. Available: https://adventofcode.com/2015/about.

[40] E. Wastl, *Advent of code*, 2015. [Online]. Available: https://adventofcode.com/2015/day/17.

[41] J. Kleinberg and T. Éva, "Chapter 4. greedy algorithms," in *Algorithm design*. Pearson/Addison-Wesley, 2014, pp. 157–251.

[42] [Online]. Available: https://marketplace.visualstudio.com/items?itemName=GitHub.copilot.

[43] [Online]. Available: https://marketplace.visualstudio.com/items?itemName=VisualStudioExptTeam.vscodeintellicode.

[44] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022. DOI: 10.1145/3520312.3534862.

TRITA-EECS-EX- 2023:0000