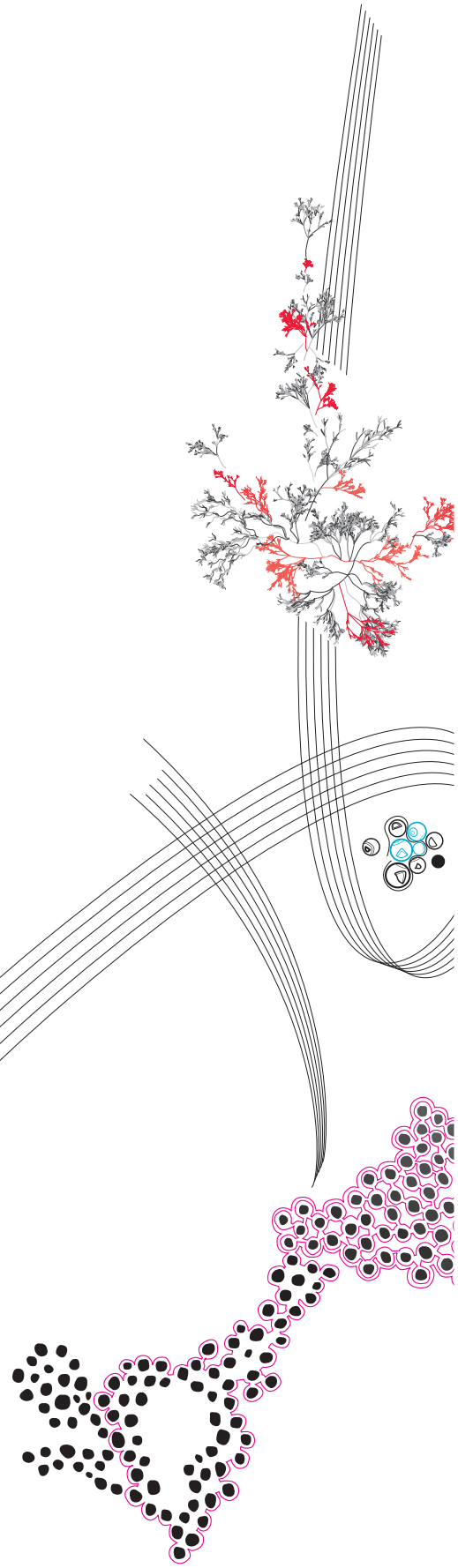


MSc Computer Science
Final Project

Generating Patch Ingredients for Search-based Program Repair using Code Language Models

Oebele Lijzenga

Supervisors:
Iman Hemati Moghadam
Vadim Zaytsev
Shenghui Wang

August, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

As software systems grow in size, more bugs occur, which are usually resolved manually. Manual bug localization and fixing is a costly time-consuming process, and hinders the development of new software. Search-based automated program repair (APR) techniques attempt to fix bugs in programs by searching a search space of patches using an evolutionary algorithm. Patches are constructed from code elements elsewhere in the program, also referred to as the *redundancy assumption*. As a result however, search-based APR techniques are not capable of fixing bugs if the required patch ingredients are not present elsewhere in the code. Previous work has attempted to treat this problem, but has failed to produce additional patch ingredients in a cost-effective manner. This study proposes ARJACLM, a novel search-based APR technique based on ARJA, which uses pre-trained code language models (CLM) to generate patch ingredients on-the-fly. Moreover, an extensive evaluation of the code generation capabilities of 20 CLMs is performed to determine which CLMs are most cost-effective, and are suitable for use in APR techniques. Results show that the performance of ARJACLM is improved by 59% when CLMs are used. Furthermore, CLM-based patch ingredients are of higher quality than their redundancy-based counterparts, and ARJACLM performs best when redundancy-based patch ingredients are omitted as a result. Moreover, the results expose several challenges involved with incorporating CLMs into a search-based technique, and provide directions for future research.

Contents

Chapter 1

Introduction

The escalating size and complexity of modern software systems have given rise to an increasing number of software bugs [1]. The consequences of this trend are not merely inconvenient, with reports estimating that bug localization and fixing costs billions of dollars annually around the world [2]. This financial burden underscores the critical need for efficient bug fixing strategies to reduce impact on software development projects.

The manual resolution of bugs, a common practice in the software development life-cycle, poses significant challenges. Manually fixing bugs is error-prone and time-consuming, especially for software systems that are already in active use. Developers are estimated to spend 50% of their time fixing bugs [3]. This considerable time investment hampers productivity and has motivated researchers to investigate automated program repair (APR). Many APR techniques have been proposed over the years, and substantial progress has been made in fixing more bugs [4]. Furthermore, state-of-the-art APR techniques are able to fix more complex bugs compared to their early counterparts [5].

APR techniques can be categorized into four classes, namely constraint-based, template-based, search-based, and learning-based. Constraint-based techniques fix bugs by synthesizing patches that satisfy functional constraints obtained from the test suite [6]. Constraint-based techniques can effectively synthesize patches for complex logic expressions, but fail to patch high-level constructs like function calls since synthesis techniques only produce arithmetic and first order logic expressions [7]. Template-based techniques leverage predefined repair templates to fix common programming errors. Template-based APR techniques have been demonstrated to be highly effective [8], but are less flexible due to the predefined nature of repair templates.

Search-based APR techniques use a heuristic function to guide a search algorithm to generate patches in an iterative manner. Patches are commonly constructed from code elements elsewhere in the code, also referred to as the *redundancy assumption*. The redundancy assumption provides a limited source of patch ingredients which makes exploration of the search space of possible patches feasible. However, search-based APR techniques based on the redundancy assumption can only produce patches consisting of code elsewhere in the project as a result, constraining their bug fixing capabilities.

To address these limitations, Yuan et al. [9] attempt to generate novel patch ingredients for a search-based technique by mutating existing redundancy-based patch ingredients. Moreover, Li et al. [10] treat patch ingredient limitations by generating novel patch ingredients using learning-based APR technique SequenceR [11]. Despite these efforts, both techniques for generating additional patch ingredient yield only a slight performance improvement, but with a substantially increased computational cost as a side-effect.

Learning-based APR techniques based on pre-trained code language models (CLM) have recently emerged as the state-of-the-art APR techniques [12, 13]. Such models leverage the strong code understanding capabilities of large state-of-the-art code language models to generate patches, and circumvent the difficulties involved in training a new task-specific neural network for the purpose of APR. Zhang

et al. [13] achieve state-of-the-art performance by leveraging CLMs to generate patch ingredients for a template-based APR technique.

Despite their impressive capabilities, the use of CLMs is limited in several ways. Deep leaning models can only consider a limited amount of code, preventing them from analyzing entire programs at once, which diminishes the quality of generated code. Secondly, CLMs suffer from hallucinations, which frequently results in the generation of syntactically or semantically incorrect code [14, 15, 16]. Finally, CLMs only possess generic code generation capabilities, and no explicit APR capabilities. Instead, CLMs must be integrated into an APR technique, which leverages CLMs to generate code at specific locations, with appropriate context, to construct patches [12, 13, 17, 18].

Search-based APR techniques and CLMs exhibit complementary strengths and weaknesses, suggesting potential synergies in their integration. Search-based APR techniques are limited by the quality and availability of patch ingredients. CLMs, through guided code generation tasks,can potentially address this limitation by providing a diverse set of high-quality patch ingredients. The integration of these two approaches may amplify their capabilities, while mitigating individual weaknesses. For instance, search-based techniques' ability to consider the context of the buggy program during the repair process could compensate for CLMs' current inability to fully incorporate program context. This integration presents an opportunity to leverage the strengths of both approaches: the systematic exploration capability of search-based methods and the generative power of CLMs.

This study investigates the effectiveness of integrating CLM-generated patch ingredients into a search-based APR technique. The contributions of this study are as follows. First, in Chapter 3, we propose ARJACLM, an novel search-based APR technique which leverages a CLM to generate patch-ingredients on-the-fly. ARJACLM aims to combine the strengths of both search-based and learning-based techniques, while mitigating their respective weaknesses.

Second, in Chapter 4, we empirically evaluate code infill generation capabilities of 20 CLMs. CLMs are being developed at a rapid pace. As a result, the cost and performance characteristics of well-known CLMs are unclear. Moreover, the quality of code generated by CLMs differs per task, so an evaluation of CLMs specifically for the purpose of APR is required to determine which models are most suitable for ARJACLM. We perform a controlled experiment to determine the efficiency and effectiveness of code infill generation capabilities of 20 CLMs.

Chapter 5 evaluates the bug fixing capabilities of ARJACLM, guided by four research questions. First, we systematically evaluate various values for five key parameters of ARJACLM. Previous work has omitted the systematic evaluation of various parameter values due to the high computational cost of evaluating APR techniques. Nevertheless, determining the most effective parameter setting for ARJACLM is crucial to the assessment of bug fixing capabilities of the search-based technique. Furthermore, two of the evaluated parameters specifically control the integration of CLM-based patch ingredients into ARJACLM, allowing for quantitative analysis of their impact on bug fixing performance.

Besides studying the bug fixing capabilities of ARJACLM, Chapter 5 evaluates the trade-off between its cost and performance. The impact of CLM-based patch ingredients on the efficiency of search-based APR techniques is currently unknown. Moreover, the efficiency of APR techniques strongly influences their practical value, and further industry adaptation.

Finally, we study the quality of CLM-based patch ingredients. The potential for CLM-based patch ingredients to out-perform the redundancy-assumption has been demonstrated in other techniques [8]. Nevertheless, the value of CLM-based patch ingredients may strongly differ depending on their purpose and integration technique. In Chapter 5, we compare the value of CLM-based and redundancy-based patch ingredients to the repair capabilities of ARJACLM.

Building on these findings, Chapter 6 provides direction for future research into search-based APR techniques based on the presented results. Finally, Chapter 7 concludes the study by summarizing our keys findings and their implications.

Chapter 2

Background and Related Work

This chapter provides an overview of automated program repair techniques, and efforts to incorporate large language models.

2.1 Automated Program Repair

The goal of automated program repair (APR) is to repair defects in programs in an automated manner. This study focuses on test-based APR, a branch of APR research where a test suite with at least one failing test exhibits the symptoms of a bug in a program. Test-based APR techniques aim to construct patches for the buggy program such that the patched program passes the entire test suite, and the patch is one that a human developer could have written.

APR techniques typically follow a common structure. An off-the-shelf fault-localization technique is used to find *likely buggy statements*, typically based on coverage of passing and failing test cases. The APR technique generates *candidate patches* which typically target likely buggy statements. Subsequently each candidate patch is compiled. If this succeeds then the patch is a *compileable patch*. If the patched program passes the test suite, the patch is considered a *plausible patch*. If a plausible patch is semantically and syntactically equivalent to the developer patch then it is a *correct patch*.

The primary focus of APR research is the generation of higher quality candidate patches. Patch generation techniques can be categorized into one of the classes discussed below.

- **Constraint-based** patch generation techniques typically use constraint-solving or synthesis techniques to synthesize candidate patches [4]. For example, Nopol [6] fixes bugs in if-then-else statements by feeding runtime information of patch execution into an *satisfiability modulo theory* solver. The result of the solver is translated into a code patch.

- **Template-based** techniques use predefined repair templates to fix bugs. These repair templates are usually hand-crafted based on common bug patterns, and are therefore less flexible than their counterparts. Nevertheless, the template-based APR tool TBar[8] currently stands out as the best performing APR tool employing a traditional (i.e. non learning-based) approach.

- **Search-based** patch generation techniques use heuristic functions to converge towards plausible patches. For example, GenProg [19] and ARJA [9] generate and combine patches iteratively and rank them using a fitness score.

- **Learning-based** patch generation techniques use a machine-learning or deep-learning model to generate patches from buggy code. Recoder [5], CURE [20] and KNOD [21] train deep-learning models to translate buggy code to fixed code. AlphaRepair [12], GAMMA [13] and SarGaM [17] generate patches using general-purpose pre-trained code language models. All current state-of-the-art APR techniques use deep-learning models for patch generation.

This study proposes a novel APR technique that combines search-based and learning-based APR techniques. The following sections discuss these techniques in more detail.

2.2 Search-based APR

Search-based APR techniques use a search algorithm to explore a space of possible patches, and can typically be distinguished based on three components. First, the candidate generation technique dictates the creation of new candidate patches or the mutation of existing ones, thereby defining the search space. Second, a heuristic function is used to rank candidate patches and guides the search algorithm to generate better patches. The heuristic function is an indicator of the quality of the candidate patch. Most search-based techniques use a heuristic function based on the number of passing test cases for this purpose. Finally, the search strategy determines how the search space is explored, utilizing the heuristic function and patch generation technique.

A well-known example of a search-based technique is a genetic algorithm [22], which leverages natural selection to optimize for one or more objectives. In genetic algorithms, a subset of the search-space, referred to as the *population*, is improved in generations. A genetic algorithm starts with instantiating the initial population, where individuals are initialized with random properties, or properties obtained using some predefined initialization logic. In each generation, new individuals are created by applying mutation and crossover operations to existing individuals. Mutation modifies one or more properties of a single individual. Crossover combines two individuals to obtain two offspring individuals which inherit properties from their parents. Subsequently, new individuals are evaluated according to a fitness function, which is a heuristic function for the quality of the individual. At the end of each generation, a selection method is used to determine which individuals in the population are replaced with new individuals. Generations are usually evaluated until either an adequate individual is found, or a predefined generation limit is reached.

Having outlined the key components of search-based APR techniques, we now turn our attention to examining related work in this area.

2.2.1 GenProg

GenProg [19] was the first search-based APR technique to find test-adequate patches using a genetic algorithm. In GenProg, each individual represents a patch which modifies the buggy code. Mutation operations modify patches by introducing more modifications to the code, and crossover generates offspring patches which inherit code modifications from their parents. The fitness of individuals is determined based on the result of compiling and executing tests on the patched code. Patches which result in more passing test cases receive a higher fitness score. The genetic algorithm of GenProg terminates if a test-adequate solution is found, or a predefined generation limit is reached.

Evaluation of the fitness function is the most time-consuming part of GenProg, as it requires compilation of the source code and execution of test cases. Thus, the number of patch evaluations must be constrained where possible. GenProg employs two measures for this purpose. First, GenProg uses fault-localization to attach a weight to each statement based on coverage of passing and failing tests. Statements that are covered by many failing tests cases receive a higher suspiciousness score, and are more likely to be targeted by a patch as a result. This mechanism results in the prioritization the modification of statements which are likely to be the cause of the bug.

The second measure constrains what code can be added to the buggy program, also referred to as *patch ingredients*. Arbitrarily generated patch ingredients are too likely to be incorrect, and would result in a large search space of low quality, which cannot be feasibly searched due to the high cost of evaluating individuals. Instead, GenProg re-uses statements from elsewhere in the buggy program under the assumption that ingredients to a bugfix are already present elsewhere in the program. This

is commonly referred to as the *redundancy assumption* or the *plastic surgery hypothesis* [23, 24]. Patch ingredients obtained from the redundancy assumption are referred to as *donor code*.

Figure 2.1 shows the representation of patches used for the genetic algorithm in GenProg. Each patch is a sequence of modifications to the buggy program, which consists of an operation (*replace*, *insert*, *delete*), the statement which is modified, and an ingredient statement in case of a *replace* or *insert* operation. Crossover of two patches is performed by selecting a random cut point in the sequence of modifications which make up each patch. Two offspring patches are obtained by swapping modifications after the cut point between the two parents. Mutation modifies an existing patch by adding an additional modification which uses a randomly selected operation, target statement and ingredient statement if applicable.

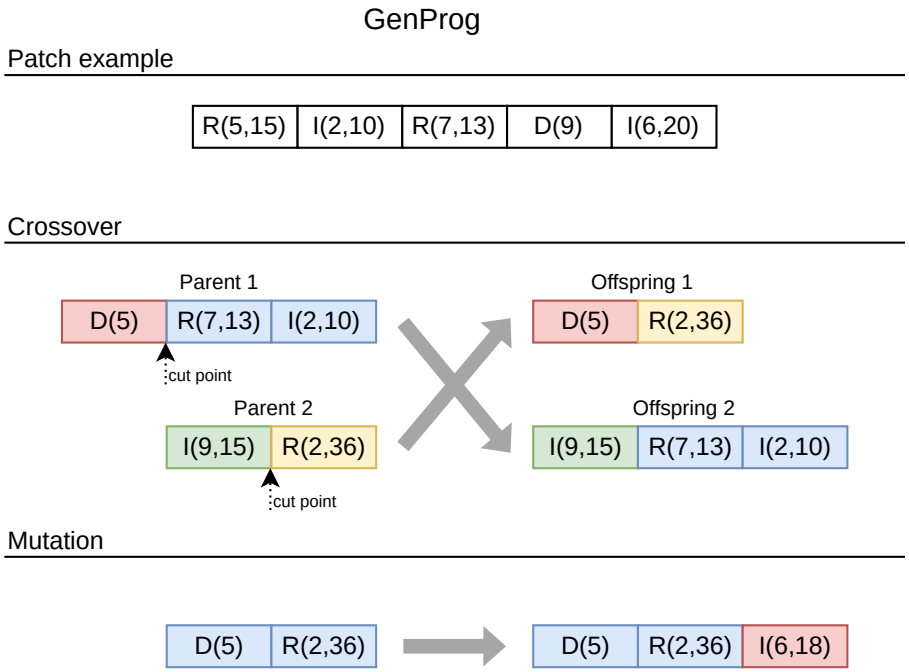


FIGURE 2.1: The GenProg patch representation. The numbers inside parentheses are identifiers for statements in the buggy program.

Although the approach of GenProg was ground breaking at the time, empirical studies have found some shortcomings. For example, GenProg frequently generates patches with an order of magnitude more changes than necessary [19]. Thus, parts of the search space are explored that are clearly undesirable, resulting in prolonged execution times, and lower patch quality. GenProg mitigates this problem by reducing each test-adequate patch to the least amount of code modifications such that the test suite passes. As a result, patch elements which do not contribute to making the test suite pass are removed.

The patch reduction technique of GenProg significantly reduces the size of generated patches. However, Qi et al. [25] observe that GenProg mostly generates patches that remove functionality instead of repairing it. This might be a negative side-effect of the patch reduction step. Moreover, GenProg has been shown to work better with random search instead of the genetic algorithm [26], which demonstrates that the search-based approach used in GenProg does not effectively guide the repair process. Finally, a large-scale experiment [27] has shown that jGenProg [28], an implementation of GenProg for Java, finds correct patches for only five out of 224 bugs from the Defects4J 1.2 [29] benchmark.

2.2.2 ARJA

ARJA [9] is another APR technique based on a genetic algorithm. ARJA improves upon GenProg in several ways. First, ARJA uses two objectives instead of one, also referred to as multi-objective search. Similar to GenProg, the first objective is based on the number of passing tests, while the second is based on the amount of code modified by the patch. A multi-objective genetic algorithm based on NSGA-II [30] is used to search for test-adequate patches that perform few modifications to the code, under the hypothesis that guiding genetic search to explore shorter patches yields more test-adequate and correct patches.

In this study, we extend ARJA. Therefore, in the next sections, we will delve deeper into ARJA, providing a comprehensive discussion to lay the groundwork for understanding our extension to this technique.

2.2.2.1 NSGA-II

The genetic algorithm of ARJA is based on NSGA-II, and uses an objective based on the number of passing tests, and another objective based on the number of modifications performed by the patch. This objective based on patch size guides the genetic algorithm to explore patches that are shorter, mitigating exploration of large patches which are unlikely to be correct as discussed in Section 2.2.1.

NSGA-II is a multi-objective optimization algorithm that addresses many common pitfalls in genetic algorithms. It strikes an effective balance between convergence towards a Pareto-optimal set of solutions, and maintaining population and solution diversity. A selection method based on a diversity metric and Pareto dominance is used for this purpose. An individual dominates another if all of its fitness scores are at least as good as that of the other, and at least one fitness score is better. Individuals that are dominated by fewer individuals in the population, and achieve a higher diversity score, are more likely to be selected as a parent for crossover, or to be transferred to the next generation.

The diversity metric of NSGA-II is based on the crowding distance of an individual with respect to the adjacent individuals in its Pareto front. A Pareto front is a set of individuals which are not dominated by other individuals, other than those of previous fronts. Figure 2.2a provides a visual representation of non-dominated sorting. Pareto fronts are obtained by computing the initial Pareto front, which are all individuals which are not dominated by other individuals. Subsequent Pareto fronts consist of individuals which are not dominated by other individuals, other than those of previous Pareto fronts.

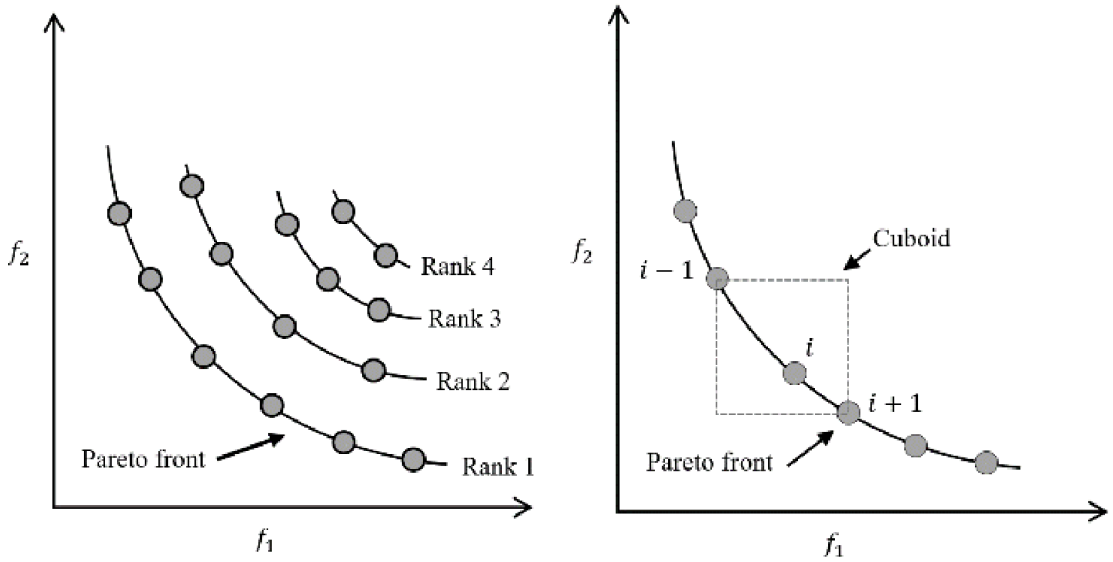
Crowding distance of an individual is the difference between fitness scores of the adjacent individuals in the Pareto front. Figure 2.2b shows an example of the crowding distance calculation of individual i, based on the difference in fitness of adjacent individuals $i-1$ and $i+1$ with respect to objectives f_1 and f_2. The crowding distance of i, denoted by $c(i)$, is computed as:

$$c(i) = f_1(i+1) - f_1(i-1) + f_2(i-1) - f_2(i+1) \tag{2.1}$$

A high crowding distance indicates that an individual covers a relatively unexplored part of the search-space, and is therefore prioritized in selection.

Finally, NSGA-II leverages elitism to mitigate the potential loss of fit individuals due to crossover and mutation. Elitism ensures that one or more best performing solutions are always transferred to the next generation. Elitism speeds up genetic algorithms and helps accelerate convergence towards Pareto-optimal solutions [32, 33].

Similar to GenProg, ARJA uses tournament selection to select individuals for crossover, mutation and population replacement. Tournament selection is a widely used selection method where a set of individuals is randomly chosen from the population, with or without replacement. The individuals take part in a tournament, and the fittest individual wins.



(A) Non-dominated sorting

(B) Crowding distance calculation

FIGURE 2.2: Non-dominated sorting and crowing distance calculation in NSGA-II [31].

2.2.2.2 Search-space Optimizations

ARJA operates under the redundancy assumption, using donor code as patch ingredients. In addition, ARJA introduces a novel screening procedure which prevents the use of undesirable donor code or edit operations by filtering them with respect to each *likely buggy statement* (LBS). For example, a *continue* statement should not be used as an ingredient for an LBS which is not contained inside a loop. Table 2.1 shows the rules and rationale for ingredient filtering.

Moreover, donor code is screened based on whether symbol references in donor code are valid when placed at the location of an LBS. Table 2.2 shows the rules used to filter edit operations per LBS. Finally, Table 2.3 shows rules which preclude the use of specific combinations of ingredients and edit operations. Altogether, these measures aim to improve the quality of the search space by preventing code anti-patterns, which reduces execution time of the genetic algorithm and yield higher quality patches.

2.2.2.3 Generating Additional Patch Ingredients

Besides improving the quality of the existing search-space, ARJA is used to evaluate a novel *type matching* technique for generating additional patch ingredients. The introduced type matching technique replaces variable and method references in redundancy-based patch ingredients which are not in scope at the location of an LBS. Variable and method references are replaced with symbols that are in scope, and satisfy typing requirements. Type matching allows ARJA to fix bugs even if the required patch ingredients are not present elsewhere in the code. Instead, redundancy-based patch ingredients only need to contain the required code patterns, and the appropriate code symbols can be obtained using type matching.

Despite relieving the constraints of donor code availability on ARJA, empirical evaluation shows that type matching does not improve the performance of ARJA on real-world bugs. Type matching introduces many new patch ingredients which increases the size of the search-space. The authors note that search ability of the genetic algorithm of ARJA is possibly not strong enough to handle the large search space determined by type matching [9].

No.	Rule	Rationale
1	The `continue` statement can be used as the ingredient only for a likely-buggy statement in the loop.	The keyword continue cannot be used out of a loop (i.e., `for`, `while` or `do-while` loop).
2	The `break` statement can be used as the ingredient only for a likely-buggy statement in the loop or a `switch` block.	The keyword `break` cannot be used out of a loop (i.e., `for`, `while` or `do-while` loop) or a `switch` block.
3	A `case` statement can be used as the ingredient only for a likely-buggy statement in a `switch` block having the same enumerated type.	The keyword `case` cannot be used out of a `switch` block, and the value for a `case` must be the same enumerated type as the variable in the `switch`.
4	A `return`/`throw` statement can be used as the ingredient only for a likely-buggy statement in a method declaring the compatible `return`/`throw` type.	Avoid returning/throwing a value with non-compatible type from a method.
5	A `return`/`throw` statement can be used as the ingredient only for a likely-buggy statement that is the last statement of a block.	Avoid the unreachable statements.
6	A VDS can be used as the ingredient only for another VDS having the compatible declared type and the same program or disrupting the program too much.	Avoid using an edit operation with no effect on the program or disrupting the program too much.

TABLE 2.1: ARJA rules for disabling specific ingredients

No.	Rule	Rationale
1	Do not delete a variable declaration statement (VDS).	Deleting a VDS is usually very disruptive to a program, and keeping a redundant VDS usually does not influence the correctness of a program
2	Do not delete a `return`/`throw` statement which is the last statement of a method not declared void.	Avoid returning no value from a method that is not declared `void`

TABLE 2.2: ARJA rules for disabling specific operations

No.	Rule	Rationale
1	Do not replace a statement with the one having the same AST.	Avoid using an edit operation with no effect on the program.
2	Do not replace a VDS with the other kinds of statements.	Avoid disrupting the program too much.
3	Do not insert a VDS before a VDS.	The same with No. 1.
4	Do not insert a `return`/`throw` statement before any statement.	Avoid the unreachable statements.
5	Do not replace a `return` statement (with return value) that is the last statement of a method with the other kinds of statements.	Avoid returning no value from a method that is not declared `void`.
6	Do not insert an assignment statement before an assignment statement with the same left-hand side.	The same with No. 1.

TABLE 2.3: ARJA rules for disabling specific operations on specific ingredients

2.2.2.4 Patch Representation

ARJA introduces a novel patch representation to address the limited flexibility of the crossover operation in GenProg. The crossover operation of GenProg cannot construct offspring consisting of edits which inherit properties from both parents. For example, a test-adequate patch might be found by constructing an offspring patch with an edit which consists of the edit operation of one parent, and the ingredient of the other parent. The patch representation and crossover operation used by GenProg poses a limitation on the way that promising elements of patches can be combined to obtain better patches.

The novel patch representation used by ARJA is shown in Figure 2.3. A more fine-grained patch representation is leveraged to permit independent crossover of elements of each patch edit. Each patch consists of three concatenated arrays, where each element of each array corresponds to an LBS. The first array contains Booleans which indicates whether the corresponding LBS is modified by this patch. The second array contains the respective patch operation, either *replace*, *insert* or *delete*. The third array references the patch ingredient. Edit operations and ingredients are only applied if the edit is enabled, indicated by the respective Boolean in the first array. The patch example of Figure 2.3 represents a patch which replaces the first LBS with ingredient statement 15, and deletes the third LBS.

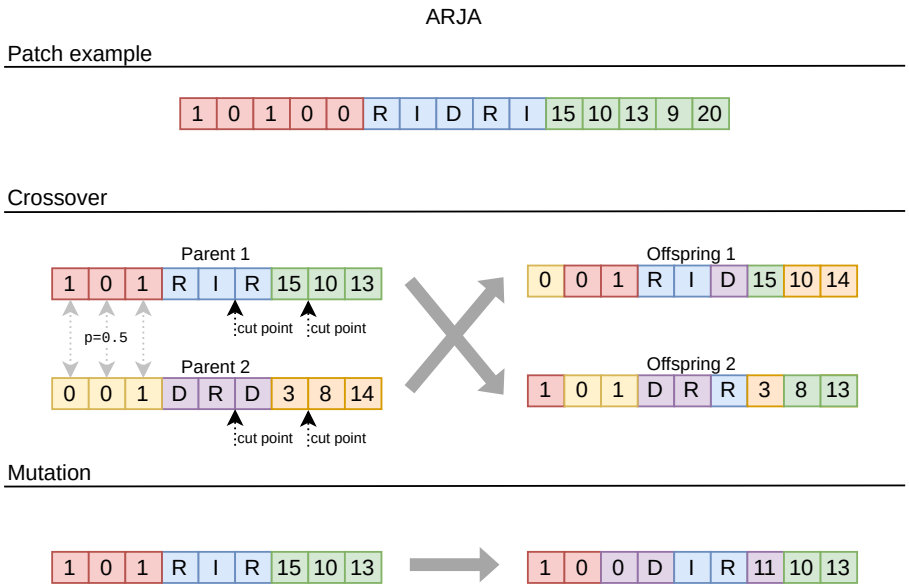


FIGURE 2.3: The ARJA patch representation. The numbers are identifiers for statements in the buggy program.

A fine-grained crossover operation leverages the novel patch representation using two separate cut points for patch operations and ingredients. This allows patch operation and ingredients to be swapped between parents independently to obtain offspring patches. As a result, edits of offspring patches can inherit the patch operation and ingredient for a single LSB from different parents. Crossover of the first array, which indicates which LBSs are modified, is performed by swapping individual values between parents with a probability of 0.5.

The mutation operation of ARJA applies a predefined mutation probability to each each element of the three arrays of a patch. If a patch element is modified, an alternative value is selected. Elements of the first array are mutated by flipping the Boolean values. Patch operations and patch ingredients are mutated by randomly selecting an alternative operation or ingredient.

2.2.2.5 Performance

ARJA is evaluated on the Defects4J 1.2 [29] benchmark and fixes 18 out of 224 bugs whereas GenProg fixes five. Computation time for ARJA and GenProg are roughly similar. Moreover, an ablation study demonstrates that both the multi-objective search and rules for donor code improve the number of bugs fixed by ARJA. In addition, the patch length objective helps ARJA generate shorter patches.

2.3 Deep Learning

Learning-based APR techniques leverage deep learning to generate patches using neural machine translation (NMT). Large bugfix training sets are used to train a deep learning model to translate buggy code into fixed code. Various learning-based APR techniques have been proposed in the past, leveraging the latest NMT architectures to fix more bugs [34, 11, 35, 36, 37, 5].

For example, Recoder [5] leverages a deep-learning model based on the transformer architecture [38] to generate patches. The transformer architecture has gained a lot of attention recently, being leveraged in well known language models like GPT. Transformer models are highly effective at text translation and generation tasks due to their ability to capture complex relationships between parts of the input. Recoder demonstrated state-of-the-art performance, fixing 51 out of 224 bugs from the Defects4J 1.2 benchmark.

Despite promising results, significant challenges are involved in training NMT models. The construction of high quality training sets for APR is a very time-consuming. The quality of these datasets is typically evaluated by validating only a sample of the dataset manually [39]. The amount of manual effort involved in constructing high-quality datasets currently limits the effectiveness of NMT-based APR techniques [4]. Additionally, NMT models frequently generate code containing syntax or semantic errors [10]. Besides, expertise in deep-learning is required to construct state-of-the-art NMT models, hindering the application deep-learning techniques to the domain of APR research.

Recent APR research has attempted to overcome the challenges of constructing custom application-specific NMT models, leveraging pre-trained general-purpose code language models instead. The following section will delve into these models in greater detail.

2.3.1 Code Language Models

Large language models (LLM) leverage the power of transformer models to understand and generate human-like text. These models are trained on vast amounts of textual data and can perform a wide range of tasks. Building on this success, researchers have developed code language models (CLM) trained specifically for code generation, which can perform a wide range of tasks like generating code snippets [40, 41, 42, 43, 44], suggesting code improvements [40, 44, 45], and summarizing [40, 44, 45, 46] or translating code [40, 44, 47, 48]. CLMs are trained on large unlabeled corpuses of publicly available code with generic training objectives like predicting the next token (i.e. piece of text) in a given sequence.

In contrast to task-specific neural networks like Recoder [5], CLMs are designed to be applied to many downstream tasks. The large amount of available training data, and the usability of CLMs for many downstream tasks has motivated researchers to invest extensive computational and human resources into constructing large powerful CLMs, some of which are publicly available. Clearly, the pre-trained nature of CLMs and their strong code generation capabilities pose a valuable avenue for APR research.

CLMs can be applied to various code generation tasks depending on the training objectives used to construct the model. Figure 2.4 shows various common code generation tasks. Mask prediction, or fill-in-the-blank style code generation, is used to generate code infills for existing code, while open-ended code completion is used to complete existing code. Alternatively, natural language prompts can be used to instruct CLMs to provide specific instructions on generating or modifying code.

Prompt-based code generation is only effective for CLMs with strong natural language understanding, and extensive effort may be required to engineer an effective prompt format. Moreover, previous research has shown that additional task-specific training (i.e. fine-tuning) may be required to achieve effective prompt-based code generation [18]. On the contrary, many CLMs are trained on mask prediction and open-ended code generation tasks, providing this functionality out-of-the-box as a result.

```
Input:
def sum_of_even_numbers(numbers):
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:
            <mask>
    return even_sum

Output:
even_sum += num
```

(A) Mask prediction

```
Input:
def sum_of_even_numbers(numbers):
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:



Output:
            even_sum += num
    return even_sum
```

(B) Open-ended code completion

```
Input:
Write a function which returns the sum of
even numbers in a list



Output:
def sum_of_even_numbers(numbers):
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:
            even_sum += num
    return even_sum
```

(C) Code generation prompt

```
Input:
Buggy line: even_sum *= num
Context:
def sum_of_even_numbers(numbers):
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:
            even_sum *= num
    return even_sum
The fixed code is:

Output:
def sum_of_even_numbers(numbers):
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:
            even_sum += num
    return even_sum
```

(D) Code modification prompt

FIGURE 2.4: Two code generation techniques

2.3.2 AlphaRepair

Xia and Zhang [12] propose AlphaRepair which uses CodeBERT [43] for mask prediction to fix bugs in a zero-shot setting (i.e. without fine-tuning). AlphaRepair fixes 50 out of 395 bugs from the Defects4J 1.2 benchmark.

AlphaRepair uses three strategies to mask LBSs as shown in Figure 2.5. Patches are constructed by using CodeBERT to generate infills for the masked code. The *complete mask* strategy either replaces a line with mask tokens or inserts a line of mask tokens before or after a line of code. The *partial mask* strategy replaces the start or end of a line with mask tokens. The *template mask* strategy replaces commonly buggy code elements with mask tokens. For example, the arguments to a function call or (parts of) the condition of an if-statement are replaced with mask tokens.

CodeBERT is one of the first widely used CLMs, and its use in AlphaRepair poses several chal-

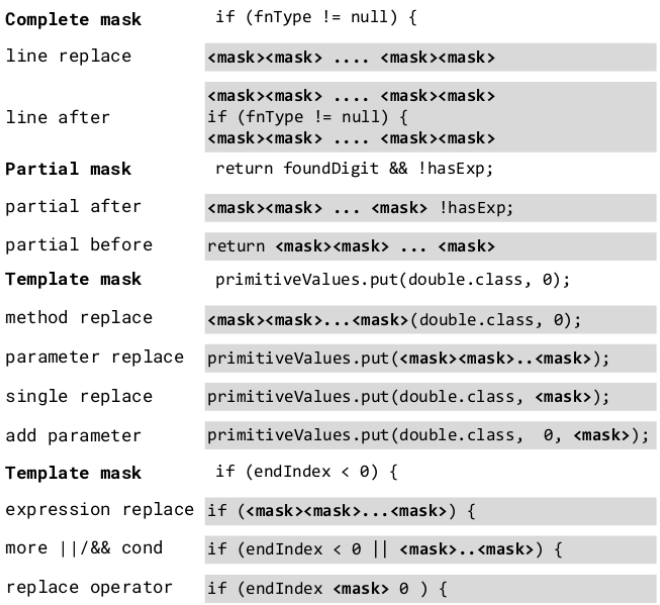


FIGURE 2.5: Mask strategies employed by AlphaRepair [12].

lenges. First, CodeBERT generates exactly one token for each mask token in the provided prompt. Therefore, AlphaRepair must guess the number of tokens to generate beforehand to generate token sequences. To obtain the most probable code infills of arbitrary length, AlphaRepair generates infills for all line lengths up to L+10 tokens where L is the number of tokens in the original line of code. This mask prediction method is significantly slower than using a CLM capable of generating an arbitrary number of tokens for a single mask token [13].

Another challenge comes from the difference between the training objective of CodeBERT and its application. AlphaRepair uses CodeBERT with prompts containing sequences of consecutive mask tokens whereas it is trained to predict tokens for masks which are sparse. As a result, CodeBERT cannot leverage the immediate context surrounding each mask token as it consists of other mask tokens, reducing the quality of generated tokens. To treat this problem, AlphaRepair re-ranks generated token sequences afterwards. Re-ranking is performed by querying CodeBERT for the likelihood of each individual token in a line and uses these results to compute a joint score. The score for each token obtained during re-ranking is more accurate than the score obtained during initial token prediction as the full context of the generated sequence is available.

The final challenge in using CodeBERT for mask sequence prediction lies in applying *beam search*. Beam search is a technique for sampling token sequences from CLMs which involves sampling multiple token sequences in parallel to avoid falling into local optima. CodeBERT only predicts a single token at a time for mask predict task. Thus, a custom beam search implementation must be provided to generate token sequences using CodeBERT. This increases the burden on researchers to develop efficient and correct sampling strategies

AlphaRepair shows that CLMs can be used to construct effective APR techniques in a straight forward manner. However, CodeBERT is an early CLM which is constrained in its usage patterns, and many larger CLMs capable of generating high-quality infills of arbitrary length have been published since.

2.3.3 GAMMA

Zhang et al. [13] propose GAMMA, a hybrid APR technique leveraging a CLM to generate patch ingredients for a template-based APR technique. Template-based APR techniques use repair tem-

plates, some of which require patch ingredients to be completed. Traditional template-based techniques like TBar [8] leverage the redundancy assumption to obtain patch ingredients. Donor code is not always available however, which significantly reduces the effectiveness of template-based APR techniques [8, 49, 50]. Instead, GAMMA leverages CLMs to generate patch ingredients for repair templates. GAMMA is evaluated using three CLM's, namely UniXcoder [47], CodeBERT [43] and ChatGPT. UniXcoder and CodeBERT are used in a zero-shot setting.

UniXcoder is a CLM trained for mask prediction and is capable of an arbitrary length token sequence for a single mask token. Therefore, UniXcoder is much more convenient for generating an arbitrary number of tokens when compared to CodeBERT as discussed in Section 2.3.2. Zhang et al. [13] show that GAMMA with CodeBERT performs similar to GAMMA with UniXcoder, but the CodeBERT version runs much slower due to limitations discussed in Section 2.3.2.

ChatGPT is the third CLM used in the evaluation of GAMMA. ChatGPT (based on GPT-3.5 at the time) is asked to provide the 250 most likely replacements for the mask token and is given the buggy code with the fix template and the original buggy line as context. This version of GAMMA performs significantly worse than the other two. Zhang et al. [13] note that ChatGPT is not publicly available and therefore it is unclear how it is trained and how it can best be leveraged. Furthermore, the prompt design could be further improved to give ChatGPT a better understanding of the mask prediction task and its context.

Empirical evaluation of GAMMA shows that it achieved state-of-the-art bug-fixing performance at the time. GAMMA demonstrates that CLMs are an effective source for patch ingredients, and patch ingredients generated by CLMs can be of higher quality than their redundancy-based counterparts. Thus, the performance of GAMMA justifies further research into incorporating CLMs into traditional, redundancy-based APR techniques.

2.3.4 ARJANMT

Li et al. [10] propose ARJANMT, a hybrid APR technique which augments ARJA by introducing additional patch ingredients generated by the learning-based APR technique SequenceR [11]. Search-based APR techniques like GenProg and ARJA are incapable of fixing bugs when the necessary donor code to construct a correct patch is not present elsewhere in the buggy program. ARJANMT overcomes this limitation by using SequenceR to generate novel patch ingredients which are used to supplement the existing redundancy-based patch ingredients. ARJANMT is evaluated on a subset of the Defects4J 1.2 benchmark. ARJANMT finds correct patches for 50 bugs, whereas ARJA finds 47. In addition, ARJANMT fixes four bugs that other evaluated APR techniques do not fix.

ARJANMT demonstrates that patch ingredients generated by deep-learning techniques can improve the performance of traditional search-based APR techniques. Nevertheless, the improvement in bug fixing capabilities of ARJANMT over AJRA is limited as it is significantly constrained in two ways. First, ARJANMT generates all patch ingredients beforehand based on the initial buggy program, so patch ingredients generated by SequenceR are not based on the patched code into which they are integrated, reducing the relevance of learning-based patch ingredients as a result.

Another limitation of ARJANMT is the use of SequenceR for generating patch ingredients. SequenceR is based on a neural network trained specifically for APR, for which only limited training data and computational resources are available. Section 2.3.1 discusses the advantages of CLMs compared to domain-specific learning-based techniques for code generation. CLMs are potentially capable of generating patch ingredients of higher quality.

This study proposes and evaluates a novel hybrid search-based APR technique which leverages CLMs to generate additional patch ingredients, and aims to address the limitations of ARJANMT.

Chapter 3

Proposed Technique

The previous chapter has presented past research into search-based APR techniques. Such techniques have become more sophisticated over time. Multi-objective search, advanced search space reduction measures and improved patch representations and mutation operations are leveraged to more effectively guide a genetic algorithm to find bug-fixing patches. Nevertheless, the effectiveness of search-based APR techniques remains constrained by the redundancy assumption, preventing the repair of bugs which require the introduction of novel code into the project.

Yuan et al. [9] aim to generate additional patch ingredients by modifying existing redundancy-based ingredients, while Li et al. [10] leverage a learning-based APR technique to generate new ingredients. As discussed in Chapter 2, these efforts have failed to produce promising results. Yet overcoming this limitation remains crucial for advancing the bug-fixing capabilities of search-based APR techniques.

Recently, advancements in APR research have been made by leveraging pre-trained general-purpose code language models (CLM) to generate code. The capability of these models to generate high-quality code ad-hoc in a flexible manner have allowed for advancements in state-of-the-art APR techniques [12, 13]. In addition, GAMMA [13] improves the performance of template-based APR technique TBar [8] by using CLMs to generate patch ingredients. GAMMA demonstrates that bug fixing capabilities of existing redundancy-based APR techniques can be augmented with patch ingredients generated by CLMs.

This study aims to research the potential for leveraging CLMs to overcome the limitations posed upon search-based APR techniques by the redundancy assumption. This chapter proposes ARJACLM, a novel search-based APR technique based on ARJA which leverages CLMs to generate additional patch ingredients on-the-fly. The novelty of ARJACLM lies in the use of CLMs to generate patch ingredients on-the-fly in a zero-shot setting. Moreover, several improvements are made to the search-based technique of ARJA in general. The soure-code of AJRACLM is publicly available on GitHub[1].

This chapter provides a comprehensive overview of ARJACLM and explains the underlying rationale of notable design decisions. Subsequently, Chapter 4 evaluates the code generation capabilities of various CLMs to determine which CLMs are suitable for generating patch ingredients for ARJACLM. Chapter 5 assesses the effectiveness and efficiency of ARJACLM, and discusses the impact of changing various configuration parameters.

[1] https://github.com/olijzenga/ARJACLM/tree/main

3.1 Overview

The focus of this study is to evaluate the effectiveness of incorporating CLMs into a search-based APR technique. ARJACLM uses ARJA, the state-of-the-art search-based technique at the time of writing, as the baseline search-based technique to facilitate this research. Various modifications are made to effectively incorporate CLMs.

Figure 3.1 shows how ARJACLM works in a nutshell. The input of ARJACLM consists of a buggy Java program, one or more LBSs obtained from fault localization and a JUnit test suite of which at least one test case fails for the buggy program. LBSs are labeled with weights indicating the likelihood that the statement is buggy, as determined by an external out-of-the-box fault localization technique. ARJACLM leverages a multi-objective genetic algorithm to find test-adequate patches based on a subset of the entire test suite. In the preparation phase, the input is prepared for the genetic algorithm. In the post-processing phase, patches which fail to pass the full test suite are dropped. The output of ARJACLM is a set of unique patches which satisfy the entire test suite. The upcoming sections will delve into the specifics of each phase, providing a comprehensive understanding of the repair process.

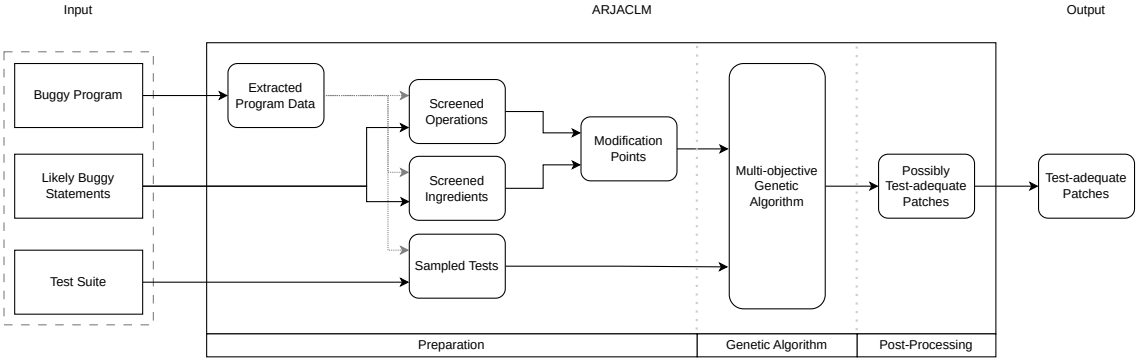


FIGURE 3.1: A data dependency overview of ARJACLM

3.2 Preparation

The preparation phase transforms the input such that it can be used by the genetic algorithm. The input of the genetic algorithm is a test suite, and a set of *modification points*. Modification points represent statements in the buggy program which can be modified by a patch. Figure 3.2 shows an example of a modification point. Each modification point has a unique index, a suspicousness score, the *target statement* which can be modified, and the *patch operations* and redundancy-based *patch ingredients* which can be used to modify the target statement. The suspiciousness score ranges from 0 to 1, where 1 indicates a very suspicious statement.

Modification points are constructed based on LBSs provided in the input. The most suspicious LBSs are used to construct modification points such that each modification point corresponds to an LBS. The statement of an LBS is the target statement of the modification point, and the weight of the LBS is used as the suspiciousness score. Redundancy-based patch ingredients are extracted from the project source code, and both patch operations and ingredients are screened with respect the LBS, as discussed in subsequent sections. The number of modification points is determined by the n_{max} parameter of ARJACLM.

Index	0
Suspicousness	0.733
Target	`int x = 0;`
Operations	`insert`
	`replace`
	`delete`
Redundancy Ingredients	`y = 1;`
	`int x = 5;`
	`if (x > 3) { return -1; }`
	`continue;`

FIGURE 3.2: An example of a modification point in ARJACLM

3.2.1 Operation Screening

Operation screening determines which patch operations are suitable for modifying an LBS in the buggy program, and are therefore used for its respective modification point. The patch operation screening rules of ARJA, as shown in Table 2.2, are used for this purpose. Operation screening increases the efficiency of the search space by preventing the use of the *delete* patch operation for modification points where the deletion of the target statement is likely to yield code that does not compile.

3.2.2 Ingredient Screening

Ingredient screening filters patch ingredients based on their compatibility with the code surrounding an LBS. Patch ingredients for a modification point are those compatible with the corresponding LBS. ARJACLM leverages the ingredient screening rules from ARJA, as shown in Table 2.1, for this purpose. These rules focus on preventing incorrect use of context-specific statements like `continue`, `break` and `throw`.

Besides rule-based ingredient screening, patch ingredients are screened for compatibility at the location of an LBS. Patch ingredients that reference code symbols which are not visible at the location of an LBS, or violate type constraints, are rejected. Ingredient symbol screening validates references to local variables, instance variables, static variables, classes, and methods. Arguments for method invocations are also screened for visibility and compatibility with a method signature.

In comparison to ARJA, the ingredient symbol screening procedure in ARJACLM validates more code symbols, and is more strict as a result. Table 3.1 shows a comparison of types of patch ingredients supported by the ingredient symbol screening procedures of ARJA and ARJACLM. Symbol visibility checking is the ability to determine whether a symbol is in scope at a location in the code, while type compatibility checking determines whether code symbols do not violate type constraints. ARJACLM is capable of resolving visibility of all code symbols within a patch ingredient, whereas ARJA is only capable of doing so for method invocations which make up the entire statement. For example, ARJA can screen `f(x, y);`, but not `z = f(x, y);` as the method invocation is nested in an assignment statement. Neither ARJA nor ARJACLM can evaluate type compatibility of results of unary operators, binary operators and literals. Finally, both ARJA and ARJACLM ignore code elements that they are not capable of screening, rather than rejecting the entire ingredient. Thus, false positives can arise if code symbols which violate visibility or type constraints are contained within code elements not supported by the screening procedure. ARJACLM mitigates this problem using more extensive ingredient screening which covers more code elements.

Extensive ingredient symbol screening as used in ARJACLM further improves the efficiency of the

	Symbol Visibility		Type Compatibility	
Statement	ARJA	ARJACLM	ARJA	ARJACLM
`x = y;`	✓	✓	✓	✓
`x = y + 1;`	✓	✓	✗	✗
`f(x, y);`	✓	✓	✓	✓
`z = f(x, y);`	✗	✓	✗	✓
`z = f(x, 1);`	✗	✓	✗	✗
`f(x).y();`	✗	✓	✗	✓
`f(x).y(z);`	✗	✓	✗	✓

TABLE 3.1: A comparison of types of patch ingredients supported by ingredient symbol screening of ARJA and ARJACLM. A checkmark is provided only if the respective technique can handle all symbols in the input.

search space by preventing compilation errors in patched code due to semantic errors. Nevertheless, extensive ingredient screening can lead to an increase in the number of *false negatives* in ingredient screening, reducing the amount of available, valid, patch ingredients. False negatives arise from the complex nature of resolving and type matching of code symbols, especially for usages of external code packages. This problem is partially mitigated in ARJACLM by the use of CLMs, which substantially reduces the reliance of redundancy-based patch ingredients. Moreover, Chapter 5 demonstrates that a sufficient amount of redundancy-based patch ingredients exist for ARJACLM to be effective without CLMs.

3.2.3 Test Filtering

The test suite of a buggy program consist of negative tests, which establish the symptoms of a bug, and positive tests which demonstrate the correctness of the remaining code. In most cases however, not all positive tests are required to determine that a patch does not break working parts of the program. For example, if a patch modifies module x, and module y does not depend on x, then the behavior of module y is remains unchanged. Consequently, the results of tests for module y remain unchanged, and execution of these tests can be omitted when evaluating this patch.

Most of the execution time of search-based APR techniques consists of the evaluation of patches. Previous research has explored test filtering techniques to speed up fitness evaluation. For example, GenProg [51] is capable of randomly sampling a predefined percentage of positive tests. A new sample of the positive tests is obtained for each fitness evaluation. This technique introduces risk that a positive test is omitted which evaluates functionality that is altered by a patch. Moreover, GenProg caches fitness evaluation results to avoid re-evaluation of previously evaluated patches. Therefore, patch evaluation results obtained with inadequate test suites are re-used throughout the entire execution of the genetic algorithm.

ARJA filters positive tests based on their code coverage. Positive tests are used for fitness evaluation only if they cover at least one statement of a class that may be modified by the genetic algorithm. This test filtering technique substantially reduces the number of positive tests and guarantees that all relevant tests are executed.

ARJACLM leverages a straight forward test filtering technique. Test cases located in the same package as a negative test are always used. Moreover, positive tests are randomly selected until a fraction of all positive tests are sampled. The r_{pos} parameter is a configurable parameter of ARJACLM and determines how many tests are sampled. Test sampling is only performed once, instead of sampling tests for each variant, as ARJACLM leverages a test cache which mitigates the purpose of repeated test sampling. This test sampling technique improves upon GenProg by prioritizing test cases which are

likely related to the buggy code based on their location, and avoids the use of a complex coverage based sampling technique.

3.3 Genetic Algorithm

The genetic algorithm of ARJACLM evolves patches which modify modification points in the code, using the sampled test suite to improve the population using natural selection. ARJACLM leverages a multi-objective genetic algorithm based on NSGA-II [30]. This section presents the genetic algorithm in detail.

3.3.1 Patch Representation

In genetic search, individuals in a population are mutated, combined and selected to obtain individuals of higher fitness. Individuals in ARJACLM correspond to patches which modify the buggy program. A patch consists of *edits*. Each edit modifies the target statement of a modification point using a patch operation and ingredient. Edits can be either enabled or disabled. The enabled edits of a patch are the modifications applied to the buggy program by the patch.

Figure 3.3 shows the data structure used by the genetic algorithm to represent patches. Similar to ARJA, each patch consist of three separate arrays, where element i of each array corresponds to the edit for modification point with index i. The first array contains Booleans indicating whether edit i is enabled, and the second and third array contain the respective patch operation (*insert*, *replace* or *delete*) and patch ingredient.

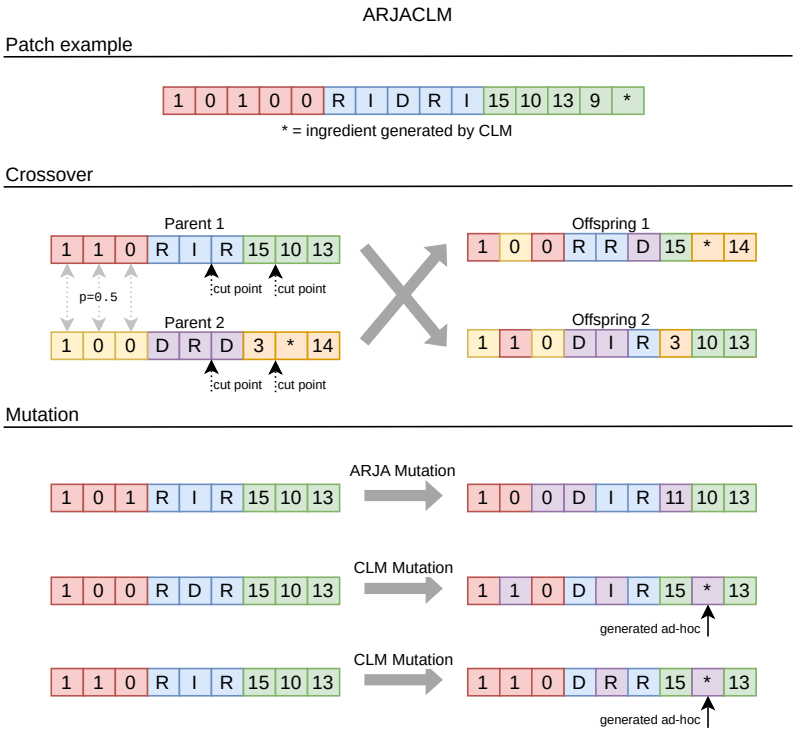


FIGURE 3.3: The patch representation of ARJACLM. '*' is used to indicate an arbitrary patch ingredient generated by a CLM.

The proposed patch representation introduces two novel aspects. First, contrary to the public im-

plementation of ARJA [2], new patch ingredients can be introduced into patches of ARJACLM on-the-fly. As a result, patch edits are not limited to redundancy-based patch ingredients obtained beforehand. Instead, novel patch ingredients generated ad-hoc by a CLM can be introduced in a straight-forward manner. Second, patch ingredients can consist of an arbitrary number of statements instead of exactly one, providing CLMs with more freedom in the amount of code that is generated.

3.3.2 Population Initialization

The initial population for genetic search determines which patches are evolved to construct test-adequate patches. Population initialization aims to construct an initial population which consists of diverse individuals that are likely to be of high quality. A diverse initial population allows the genetic algorithm to explore various parts of the search space. An initial population with low quality individuals however, can make genetic search fall into the local optimum easily, which diminishes its effectiveness [52].

ARJACLM constructs the initial population randomly in the same manner as ARJA. Each edit is initialized with a randomly selected patch operation and redundancy-based ingredient of its respective modification point, using a uniform distribution. Edits are initialized as enabled with a probability of $suspciousness \times \mu$, where *suspiciousness* is the suspiciousness score of the respective modification point, and μ is a configurable parameter of ARJACLM. As a result, more suspicious statements are more likely to be modified by patches in the initial population, which guides genetic search to explore the modification of more suspicious code.

3.3.3 Crossover and Mutation

Crossover and mutation operations generate new individuals based on individuals in the current population. Crossover operations combine two parent individuals to obtain two offspring individuals with the goal of obtaining offspring patches which inherit good properties of their parents. Mutation operations aim to introduce new patch elements into the population by modifying an individual. Figure 3.3 shows the crossover and mutation operations of ARJACLM.

Following ARJA, crossover is applied $N/2$ times per generation, producing N offspring patches, where N is equal to the population size, which is a configurable parameter of ARJACLM. Mutation is applied to each offspring patch.

3.3.3.1 Crossover

ARJACLM uses the crossover operation of ARJA. Crossover is performed separately for the edit enabled/disabled status, patch operations and patch ingredients. The patch enabled/disabled status is not swapped between two parents if their value is the same for two respective edits. If the value differs, the values are swapped with a probability of 0.5. Patch operations of two parents are combined using single-point crossover. A cut point is randomly selected, which is a point in the list of patch edits. Only patch operations of edits after the cut point are swapped between parents. Single-point crossover is also used to combine patch ingredients, using a separate, newly selected, cut point.

3.3.3.2 Mutation

Two mutation operations are used in ARJACLM. First, the *redundancy mutation* is based on the mutation of ARJA, and is used to enable or disable edits, and to modify the patch operation and redundancy-based ingredient for an edit. With probability p_{mut}, the patch operation and ingredient of each edit is replaced with a randomly selected alternative of the modification point. The *CLM mutation* functions slightly different from the redundancy mutation and leverages a CLM to generate patch ingredients.

[2] https://github.com/yyxhdy/arja

The redundancy mutation in ARJACLM differs from mutation in ARJA with respect to enabling and disabling edits. The mutation operation of ARJA enables and disables patch edits with the same probability, resulting in a tendency to generate patches where half of the edits are enabled. In practice, the genetic algorithm is executed with many patch edits, so modification of many code locations yields lower quality patches [9, 19]. ARJACLM uses different mutation probabilities for this reason. Disabled edits are enabled with probability p_{mut}, but enabled edits are disabled with probability $p_{mut} \times (|\{e \in edits \mid e \text{ is enabled}\}| + 1)$. As a result, the redundancy mutation converges towards patches with a lower number of enabled edits, which are more likely to be similar to developer-written patches.

Besides the redundancy mutation which is limited to applying redundancy-based patch ingredients, ARJACLM introduces a novel *CLM mutation* operation which leverages a CLM to generate patch ingredients on-the-fly. As discussed in Section 2.3.1, CLMs can be prompted to generate code at arbitrary locations. The CLM mutation generates a new patch ingredient at the location of an LBS and applies it to the respective patch edit.

Similar to the redundancy mutation, the CLM mutation enables and disables edits based on the p_{mut} probability. Moreover, a patch ingredient is generated and applied to an edit if this edit was enabled in the previous step, or with probability p_{mut}. When a patch ingredient is generated by the CLM mutation, either the *insert* or *replace* operation is selected randomly. This operation determines which code location the patch ingredient is generated for. Generating a patch ingredient using a CLM is costly however. Therefore edits for which ingredients are generated by the CLM mutation are enabled to ensure that CLM patch ingredients are always used.

The value for p_{mut} is equal to m_{mut}/n, where n is the number of modification points, and m_{mut} is a multiplier for the mutation probability used to scale the basic $1/n$ mutation probability. The $1/n$ mutation probability ensures that the overall number of mutations remains roughly the same for various amounts of modification points.

ARJACLM generates patch ingredients using mask prediction. Mask prediction is the most straightforward method for obtaining infills for arbitrary code locations. The prompt format for mask prediction provides full control over the target location of generated code. The generated patch ingredients can therefore be generated and incorporated into the CLM mutation in a straight forward manner. The C parameter of AJRACLM determines how many lines of code context surrounding the mask token are provided to the CLM. More specifically, up to $C/2$ lines of context from the target source file are provided both before and after the mask token.

Whenever mutation is performed in ARJACLM, the mutation operation is randomly selected. CLM mutation is selected with probability p_{clm}, which is a configurable parameter of ARJACLM. The redundancy mutation is selected otherwise. As previously discussed, the CLM mutation enables and disables patch edits in the same way as the redundancy assumption. Moreover, mutation of patch operations and ingredients occurs at roughly the same frequency for both mutation operations. A similar mutation rate is therefore achieved for both mutation operations, such that ARJACLM does not rely on either mutation to modify specific patch elements. As a result, ARJACLM can be an effective APR technique for any value of p_{clm}, permitting the evaluation of a wide range of CLM mutation probabilities to determine the value CLM-based patch ingredients. Note that even for $p_{clm} = 1$, some patches using redundancy-based patch ingredients are evaluated, as population initialization solely leverages redundancy-based ingredients as discussed in Section 3.3.2.

Screening for patch ingredients and operations as discussed in Section 3.2.2 and Section 3.2.1 is not applied to patch ingredients generated by a CLM. As a result, CLMs are provided with more control over what code can be generated. If a CLM is adequately powerful, this can yield unique new patches of high quality for complex code elements.

3.3.3.3 Operation and Ingredient Screening

Patch operation and ingredient screening as discussed in Section 3.2.2 and Section 3.2.1 prevent the construction of patches for which the patch ingredient or operation introduces a code anti-pattern by itself. However, anti-patterns can also occur as the result of a combination of a patch ingredient and a specific operation. ARJACLM leverages a patch representation similar to ARJA, where patch ingredients and operations are considered independent values. As a result, mutation and crossover in ARJACLM can give rise to patch edits containing code anti-patterns.

ARJACLM leverages the rules for screening combinations of patch ingredients and operations from ARJA, as shown in Table 2.3. In ARJACLM, ingredient and operation screening is performed for new individuals obtained through crossover and mutation. Patch edits which exhibit a code anti-pattern are rectified by randomly selecting an alternative redundancy-based patch ingredient such that no anti-patterns occur. The patch is discarded if no such ingredient is available. This screening technique treats the construction of patches containing anti-patterns without diminishing the mutation rate of the genetic algorithm.

3.3.4 Fitness Evaluation

The multi-objective genetic algorithm used in ARJACLM is based on the minimization of the two objectives of ARJA, namely the weighted failure rate (denoted by f_1), and patch size (denoted by f_2). The weighted failure rate is given by Equation 3.1, where T_n and T_p are the set of negative and sampled positive tests respectively, and w is a configurable parameter of ARJACLM where $0 \leq w \leq 1$.

$$f_1(x) = \frac{|\{t \in T_n \mid x \text{ fails } t\}|}{|T_n|} \times (1 - w) + \min(1, \frac{|\{t \in T_p \mid x \text{ fails } t\}|}{5}) \times w \tag{3.1}$$

The weighted failure rate f_1 differs from ARJA in its computation of the failure rate for positive tests. ARJACLM divides the number of failed positive tests by 5, instead of the number of positive tests, as T_p is usually much larger than T_n, and varies strongly between buggy programs. As a result, f_1 provides a more sensitive and consistent heuristic for the failure ratio of positive tests. Patches where $f_1(x) = 0$ are test-adequate.

The patch size is given by Equation 3.2, which is the number of enabled edits of the patch.

$$f_2(x) = |\{e \in x \mid e \text{ is enabled}\}| \tag{3.2}$$

The genetic algorithm of ARJACLM simultaneously minimizes f_1 and f_2, optimizing for test-adequate patches which perform few modifications on the program. Patches which either fail to compile, exceed the test execution timeout or have zero enabled edits, are assigned fitness $+\infty$ for both objectives.

3.3.5 Selection

Genetic algorithms use a selection algorithm to determine which individuals are selected for crossover and mutation, and to select individuals for replacement in the population. Following ARJA, tournament selection is used for this purpose. In ARJACLM, each tournament consists of two randomly selected individuals. In order of occurrence, a patch wins the tournament if:

1) It compiles, and the other does not

2) It is test-adequate, and the other is not

3) It is dominated by fewer individuals than the other

4) Its crowding distance is greater than that of the other

A tournament winner is selected randomly if these criteria fail to determine a winner.

3.3.6 Replacement

Each generation of a genetic algorithm ends with the replacement of individuals in the population with new individuals obtained from crossover and mutation. Genetic algorithms based on NSGA-II perform replacement based on its selection method and elitism. Elitism guarantees that the $e$fittest individuals of the current population are carried over to the next generation. Elite count $e$is a configurable parameter of AJRACLM. Tournament selection as discussed in Section 3.3.5 is used to select the remainder of the new population.

3.4 Post-validation

The genetic algorithm of ARJACLM stops once G generations have been evaluated, which is a configurable value. The test-adequacy of patches is determined by the genetic algorithm based on a sample of the entire test suite, as discussed in Section 3.2.3. In the post-validation phase, solutions generated by the genetic algorithm are evaluated using the entire test suite of the buggy program. Patches which fail to pass the entire test suite are removed from the set of test-adequate solutions.

This chapter has presented ARJACLM, a novel APR technique which leverages patch ingredients generated by CLMs to overcome limitations of existing search-based APR techniques. Chapter 4 studies the effectiveness of many well-known CLMs for mask prediction, and studies their behavior under various sampling configurations. Two effective CLMs with different cost characteristics are selected for the evaluation of ARJACLM in Chapter 5.

3.5 Contributions

This chapter has so far presented ARJACLM, which is based on ARJA, but involves some key differences. This section summarizes the differences between ARJACLM and ARJA. First, AJRACLM leverages a more straight forward test filtering approach, partially selecting positive tests at random to reduce the execution time. The coverage-based test filtering technique of ARJA provides more reliable filtering, but increases the burden of implementing the technique. Ingredient screening of ARJACLM is more strict compared to ARJA, which omits screening of complex ingredients, resulting in more, but lower quality, redundancy-based patch ingredients.

The patch representation of ARJACLM is more flexible than that of AJRA, as it allows for introducing additional arbitrary patch ingredients on-the-fly. This flexibility is crucial to incorporating CLM-based patch ingredients. Such ingredients are introduced into the population by a novel mutation operation which functions similar to the existing redundancy-based mutation of ARJA, but leverages CLM-generated patch ingredients instead. Moreover, both the redundancy-based and CLM-based mutation operations of ARJACLM dynamically adjust the probability of disabling patch edits based on the number of edits already present in the patch. This mechanism avoids the exploration of patches with an excessive number of code edits.

ARJACLM leverages a fitness function which increases the weight given to the failure rate of positive tests. ARJA records the maximum loss as the result of failing positive test cases if all (usually hundreds or thousands) positive test cases fail, whereas in ARJACLM, maximum loss for positive tests

is obtained when five positive tests fail. ARJACLM is much more sensitive to positive test failures as a result.

Chapter 4

Evaluation and Selection of Code Language Models

Code language models have recently been developed at a rapid pace, and are trained using various general tasks, datasets and training methodologies. Moreover, CLMs come in various sizes, which influences the quality of generated code and the cost of applying them. As a result, it is currently unknown which CLMs are most effective at generating code infills. Evaluating ARJACLM with a wide range of CLMs is too costly, so a CLM with strong code generation capabilities is used for this purpose. In this chapter, the code infilling capabilities of 20 CLMs are evaluated to determine which CLMs are most suitable for evaluating ARJACLM.

Several works have attempted to evaluate code generation capabilities of CLMs, generating code for a benchmark of code generation tasks, and evaluating the correctness of the generated code. Recently, Jiang et al. [53] evaluate APR capabilities of CLMs via code infilling, and use APR benchmarks with test suites to determine the correctness of generated code. In addition, the study evaluates the efficiency of CLMs by comparing the cost of applying each CLM to the number of bugs fixed, where cost is expressed in the number of parameters of each CLM, the generation time per fix and GPU memory usage.

Despite reporting several key metrics for selecting CLMs, the evaluation of Jiang et al. has several limitations. CodeT5 [40], PLBART [44], CodeGen [54] and InCoder [46] are thoroughly evaluated, but numerous novel CLMs have emerged since. Moreover, the evaluation solely relies on the beam search sampling strategy with a single beam size. Nucleus sampling is not evaluated, limiting our understanding of the impact of sampling methods and parameters on performance and cost of CLMs.

This chapter addresses the discussed limitations with respect to code infilling capabilities of CLMs by evaluating performance and cost of a wide range of well-known CLMs using both beam search and nucleus sampling with a broad spectrum of parameters. Moreover, a novel benchmark is proposed for evaluating CLMs' capabilities of generating infills consisting of a single line of code. The results are used to improve understanding of performance and cost characteristics of CLMs, and to determine which CLMs are used to evaluate the proposed APR technique.

4.1 Selection of CLMs

CLMs are identified through an informal snowballed exploration of academic papers, surveys [58, 53] and online resources, guided by three inclusion criteria. First, each CLM must be publicly available and capable of being executed locally to facilitate the acquisition of cost metrics and the configuration of the sampling technique. Thus, CLMs like ChatGPT and Codex are excluded. Second, CLMs must provide explicit mask prediction capabilities, as it is the code generation technique leveraged by ARJACLM. Therefore, CLMs like GPT-NeoX [59] and CodeGeeX2 [48] are excluded. Third, CLMs must

Name	Variants	Functional	Evaluated
CodeBERT [43]	125M	Yes	No
CodeGen [54]	250M, 2B, 6B, 16B	Yes	No
CodeGen 2 [45]	1B, 3.7B, 7B, 16B	Yes	Yes
CodeGen 2.5 [45]	7B	No	No
Incoder [46]	1B, 6B	Yes	Yes
CodeT5 [40]	small (60M), base (220M), large (737M)	Yes	Yes
PLBART [44]	base (140M), large (400M)	Yes	Yes
Refact [55]	1.6B	Yes	Yes
StarCoder [41]	15.5B	Yes	Yes
StarCoderPlus [41]	15.5B	No	No
SantaCoder [42]	1.1B	Yes	Yes
CodeLLama [56]	7B, 13B	Yes	Yes
CodeLLama-Instruct [56]	7B, 13B	Yes	Yes
CodeShell [57]	7B	Yes	Yes
UniXCoder [47]	125M	Yes	Yes

TABLE 4.1: An overview of CLMs considered for evaluation. Variants of CLMs are denoted by the number of parameters, expressed in millions (M) or billions (B).

not consume over 48G of VRAM when 16-bit quantization is used, which results in the exclusion of CLMs like CodeLLaMA 34B and 70B [56].

Table 4.1 lists all identified CLMs that match the inclusion criteria. For the sake of completeness, this table also includes CLMs which satisfy the inclusion criteria but were not evaluated for other reasons. Among the chosen CLMs, we encountered challenges with CodeGen 2.5 and StarCoderPlus, which failed to produce meaningful results during preliminary testing. Moreover, earlier versions of CodeGen are excluded since they are superseded by CodeGen 2 [54, 45]. CodeBERT is omitted from the evaluation due to its impracticality in generating code sequences of arbitrary lengths, as discussed in Section 2.3.2.

4.2 Experiment Design

The code infill generation capabilities of CLMs are evaluated by generating code infills for a benchmark set containing Java infilling tasks. Each infill is tests using the test suite provided with each infill task. We construct a novel single-line code infilling benchmark based on the HumanEval-Java [53]. HumanEval-Java is an APR benchmark obtained by translating the code completion benchmark HumanEval [60] from Python to Java. Jiang et al. manually introduce a bug into each completed Java file, resulting in an APR benchmark with 163 buggy files, each accompanied with the correct version of the file and a test suite consisting of 6.3 test cases on average.

We construct a novel single-line code infilling benchmark by masking a random line in each correct Java file which has been modified with respect to its buggy variant, ensuring that behaviorally relevant code is masked. Thus, a code infilling benchmark is obtained which consists of 163 Java files with a masked line of code, and a test suite for evaluating the correctness of each infill. Figure 4.1 shows an infilling task from the constructed benchmark, and Figure 4.2 shows a single test case of the corresponding test suite.

This experiment evaluates the performance and cost of CLMs using two sampling techniques—beam search and nucleus sampling—across a predefined range of parameters. Beam search is a deterministic sampling method which generates token sequences by exploring multiple sequences of infills at once to avoid local optima. The *beam size* determines how many sequences are explored in parallel. A higher beam size can yield higher quality results, at the cost of increased VRAM usage and compu-

```java
package humaneval.buggy;

import java.util.ArrayList;
import java.util.List;

// Implement a function that takes an non-negative integer and returns an array of
// the first n integers that are prime numbers and less than n.
// for example:
// count_up_to(5) => [2,3]
// count_up_to(11) => [2,3,5,7]
// count_up_to(0) => []
// count_up_to(20) => [2,3,5,7,11,13,17,19]
// count_up_to(1) => []
// count_up_to(18) => [2,3,5,7,11,13,17]

public class COUNT_UP_TO {
    public static List<Integer> count_up_to(int n) {
        List<Integer> primes = new ArrayList<Integer>();

        for (int i = 2; i < n; i += 1){
            boolean is_prime = true;
            <mask>
                if (i % j == 0) {
                    is_prime = false;
                    break;
                }
            }
            if (is_prime) primes.add(i);
        }
        return primes;
    }
}
```

FIGURE 4.1: A sample from the CLM infilling benchmark

```java
package humaneval;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class TEST_COUNT_UP_TO {
    @org.junit.Test(timeout = 3000)
    public void test_0() throws java.lang.Exception {
        List<Integer> result = humaneval.buggy.COUNT_UP_TO.count_up_to(5);
        org.junit.Assert.assertEquals(
            result, new ArrayList<Integer>(Arrays.asList(2, 3))
        );
    }
}
```

FIGURE 4.2: Part of the test suite for a sample of the CLM infilling benchmark

tation time. Nucleus sampling is a statistical sampling method which explores a single token sequence, selecting subsequent tokens from the *top-p* most likely tokens of the cumulative probability distribution. Each subsequent token is randomly selected based on the probability of the token, such that probable tokens are more likely to be selected. The *temperature* parameter scales the probabilities of tokens before selection. Temperatures between 0 and 1 lead to more predictable results, while temperatures over 1 lead to less predictable results. A temperature of 1 yields unmodified probabilities.

Beam search is evaluated with beam sizes 5 and 10, and nucleus sampling is evaluated with all combinations of top-p $\in \{0.2, 0.4, 0.6, 0.8\}$ and temperature $\in \{0.1, 0.4, 0.7, 1.0, 1.3, 1.6, 1.9\}$. These parameter ranges are selected based on the results of informal preliminary testing, which showed no benefit in evaluating top-p $= 1.0$, or temperature ≥ 2.0. Moreover, beam size ≥ 10 led to excessive VRAM usage without demonstrating noticeably improved results to justify the cost. Thus a total of 2 and 28 sampling configurations are evaluated for beam search and nucleus sampling respectively. It should be noted that UniXCoder and Refact are only evaluated using beam search and nucleus sampling respectively, as only these sampling methods are provided out-of-the-box in the versions published on HuggingFace[1].

Result	Description
No result	Occurs if the CLM generates less than the number of requested infills.
Compilation failure	The infilled code fails to compile.
Test compilation failure	Test code fails to compile. Can occur if the function signature in the infilled code is modified.
Test failure	One or more test cases failed.
Test timeout	Test execution has exceeded its configured timeout.
Success	The source and test code compile successfully and all test cases pass.

TABLE 4.2: List of possible evaluation outcomes for a code infill result.

All experiments are performed using a single NVidia A40 with 48GB of VRAM, using 16-bit quantization. Each CLM is applied to the single-line infilling benchmark once for each sampling configuration, in a zero-shot setting i.e. without fine-tuning. Five infills are generated for each infill task. Subsequently, each infill is evaluated by compiling and testing the infilled code, which yields to one of the results shown in Table 4.2.

Several metrics are employed to measure the performance and cost of CLMs. The performance of CLMs is evaluated based on the following metrics. The *N=1* performance of a CLM is the number of infill tasks for which the first generated infill is test-adequate. The *N=5* performance is the number of infill tasks for which *any* of the five infills is test-adequate. Moreover, we report the compilation rate of infills applied to the respective masked file. Compilation rate is based on all five infills for the 163 infill tasks:

$$\text{compilation rate} \;=\; \text{total number of compileable infills} \;/\; (163 * 5) \tag{4.1}$$

Finally, we measure the number of unique infills generated for each task. Infills are considered duplicate if they are syntactically equal, disregarding additional white-space and indentation.

The cost of applying each sampling configuration of each CLM is determined based on the following metrics. Firstly, the number of parameters of each CLM indicates its size, which impacts the VRAM usage and computation time needed to generate infills. Second, the total time needed to generate all infills for all infill tasks is measured for each CLM, along with peak VRAM usage.

For each CLM, we discuss the best results obtained across all sampling configurations. The best result is determined based on the N=1 performance. It is explicitly mentioned when N=5 performance is reported instead.

[1] https://huggingface.co/

CLM	Size	Fixes N=1	Fixes N=5	Fixes Difference	Beam Size	Top p	Temp-erature	Compilation Rate
UniXCoder	125M	26	34	+8 (30.8%)	5			46.3%
Refact	1.6B	112	121	+9 (8%)		0.6	0.1	95.3%
SantaCoder	1.1B	101	107	+6 (5.9%)		0.6	0.7	97.1%
CodeShell	7B	125	132	+7 (5.6%)		0.6	0.7	96.1%
StarCoder	15.5B	136	141	+5 (3.7%)		0.8	0.7	97.8%
PLBART Base	140M	41	67	+26 (63.4%)	10			70.1%
PLBART Large	400M	42	76	+34 (81%)	10			80.3%
CodeT5 Small	60M	22	38	+16 (72.7%)	10			31.3%
CodeT5 Base	222M	50	61	+11 (22%)	5			46.9%
CodeT5 Large	737M	69	69	-		0.4	0.1	94.4%
CodeLLaMA 7B	7B	126	135	+9 (7.1%)		0.6	1	98.0%
CodeLLaMA 13B	13B	137	143	+6 (4.4%)		0.6	0.7	98.6%
CodeLLaMA-Instruct 7B	7B	74	116	+42 (56.8%)		0.8	0.7	62.1%
CodeLLaMA-Instruct 13B	13B	76	132	+56 (73.7%)		0.8	1.3	62.0%
CodeGen2 1B	1B	2	3	+1 (50%)		0.4	1.3	1.7%
CodeGen2 3.7B	3.7B	8	9	+1 (12.5%)		0.4	1.3	36.8%
CodeGen2 7B	7B	44	46	+2 (4.5%)		0.6	0.4	48.0%
CodeGen2 16B	16B	124	126	+2 (1.6%)		0.8	0.7	82.0%
Incoder-1B	1B	63	74	+11 (17.5%)	5			54.7%
Incoder-6B	6B	70	82	+12 (17.1%)	5			63.1%

TABLE 4.3: Per model, infill task success rate for the N=1 and N=5, resource usage and the sampling settings for the best benchmark results for N=1.

4.3 Experiment Results

Table 4.3 shows the results for the best performing sampling configuration for each CLM. Appendix A provides the complete benchmark results of all sampling configurations for each CLM. This section primarily examines the overall performance of CLMs based on N=1 and N=5 performance.

4.3.1 Performance and CLM Size

Figure 4.3 shows the N=1 performance of all evaluated CLMs with respect to their size. A broad spectrum of cost and performance values can be observed. For example, CodeT5 is the smallest CLM (60 million parameters) and generates 22 correct infills, while CodeGen2-16B is the largest (16 billion parameters) and generates correct infills for 124 out of 163 tasks. Notably, UniXCoder, CodeT5, PLBART, SantaCoder, Refact, CodeShell, CodeLLaMA, StarCoder and CodeGen2-16B perform best with respect to their size. A positive relationship between size and performance can be observed for these CLMs, demonstrating that the investment of more computational resources can yield better performance. Performance gain for larger CLMs diminishes however. For example, SantaCoder is 18x larger than CodeT5 Small and performs 359% better, but StarCoder is 14x larger than SantaCoder and only performs 35% better.

CodeLLaMA-Instruct 7B and 13B produce 74 and 76 correct infills, respectively, while the base CodeLLaMA model generates 126 and 136 correct infills. CodeLLaMA-Instruct is a version of CodeLLaMA fine-tuned to more accurately respond to human instructions. It is plausible that the fine-tuning step has diminished the mask prediction capabilities of CodeLLaMA-Instruct compared to its base model.

CodeGen2 1B, 3.7B and 7B generate 2, 8 and 44 correct infills respectively, whereas CodeGen2 16B

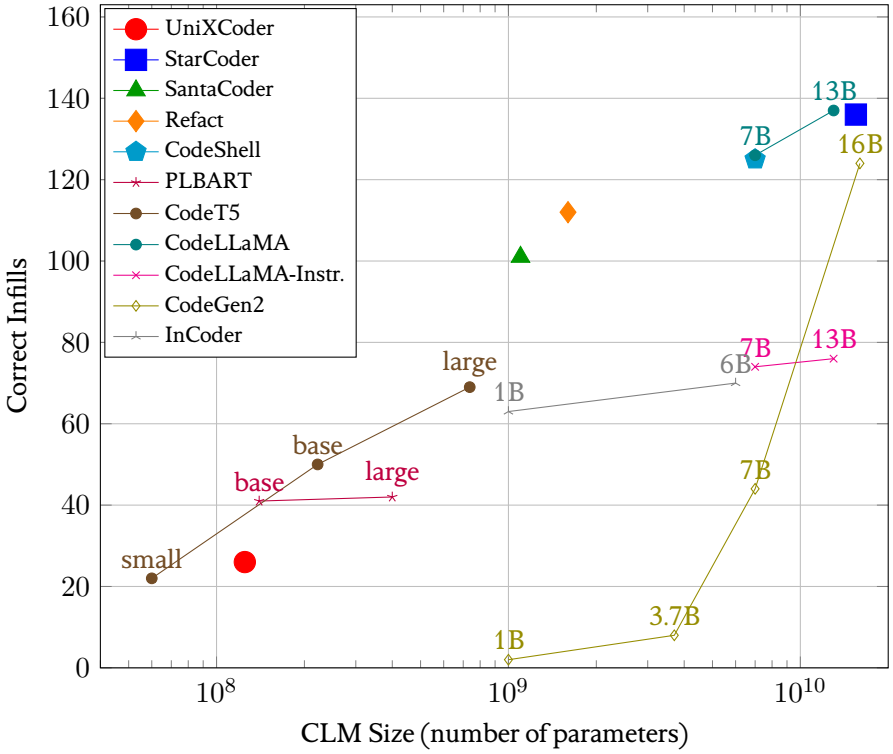


FIGURE 4.3: N=1 performance of CLMs vs their size.

generates 124 correct infills. Both preliminary evaluation and empirical evaluation demonstrate that CodeGen2 1B and 3.7B are ineffective at code infilling tasks.

4.3.2 Performance and Cost

We compare the performance of CLMs to their cost in terms of total infill time and peak recorded VRAM usage. Nucleus sampling and beam search have different cost characteristics, which hinders direct comparison of results obtained from different sampling techniques. For this reason, the we instead report the performance and cost of each CLM based on the metrics obtained for the best performing nucleus sampling configuration (wrt. N=1) for the model. Thus, the reported N=1 performance differs from previously reported performance for CLMs which perform best under a beam search configuration. Moreover, UniXCoder is excluded from this comparison as it is only evaluated using beam search.

Figure 4.4 shows the N=1 performance of CLMs compared to VRAM usage. VRAM usage of CLMs is mostly proportional to their size, the only exception being CodeGen2-1B, which uses more memory than its 7B and 16B counterparts while generating far fewer correct infills.

Figure 4.5 plots the performance of CLMs compared to the total infill generation time. CodeT5 Small and Base take 19 and 69 seconds respectively, meanwhile PLBART, SantaCoder and Refact take 107 to 188 seconds, and CodeT5 Large, CodeLLaMA, Codeshell and StarCoder take 243 to 307 seconds. PLBART, CodeLLaMA-Instruct and CodeGen2 are outperformed by their counterparts with similar performance. CodeT5, SantaCoder, Refact, CodeLLaMA, CodeShell and StarCoder demonstrate a positive relationship between time cost and performance. Nevertheless, the gain in performance obtained from increased costs diminishes for larger CLMs.

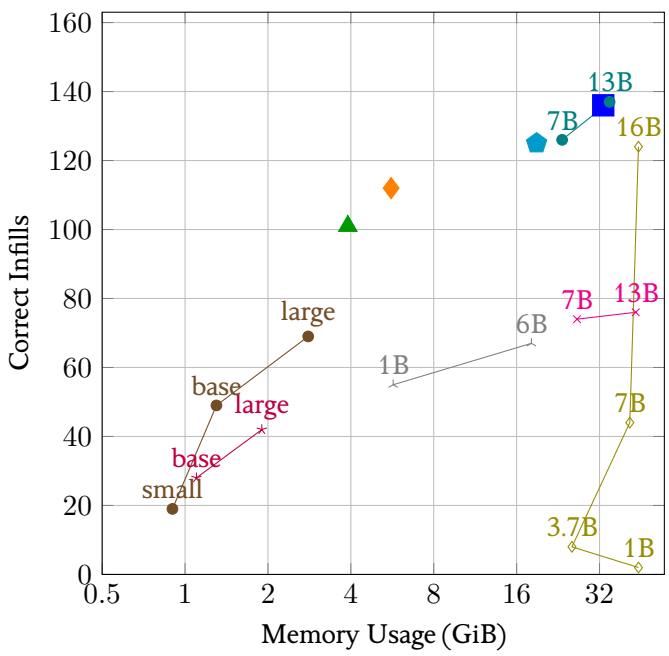


FIGURE 4.4: N=1 performance of CLMs vs memory usage.

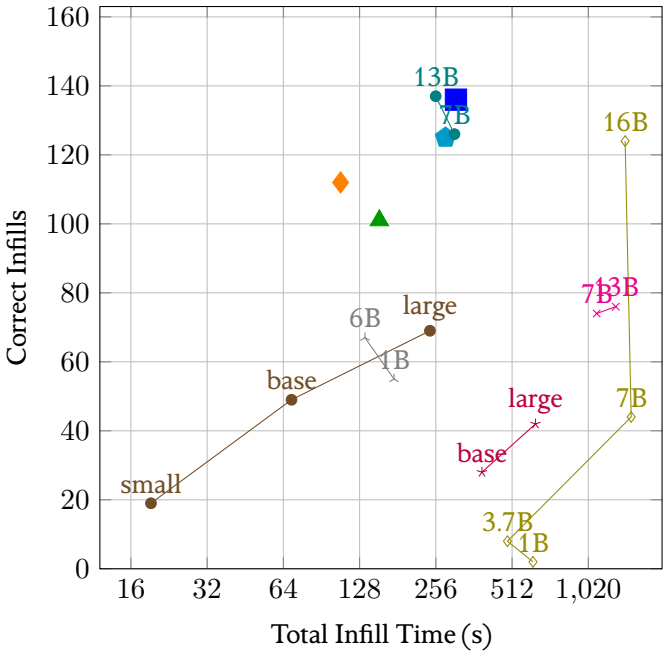


FIGURE 4.5: N=1 performance of CLMs vs infill time.

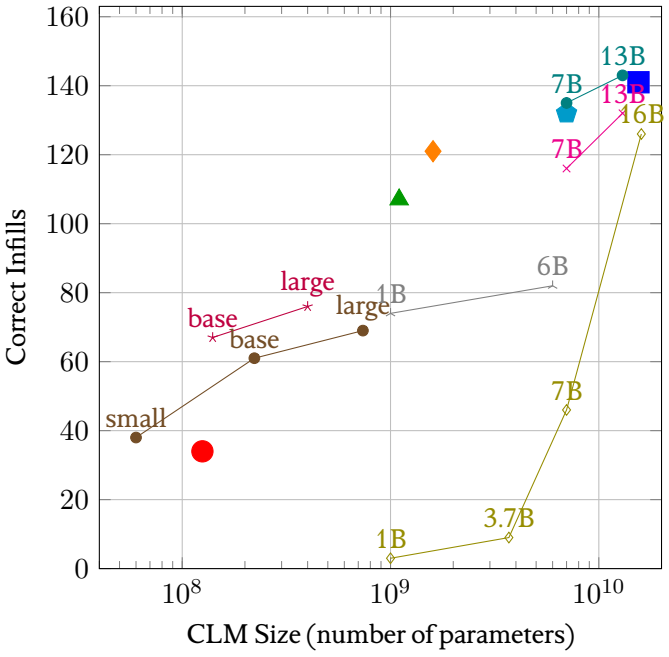


FIGURE 4.6: N=5 performance of CLMs vs their size.

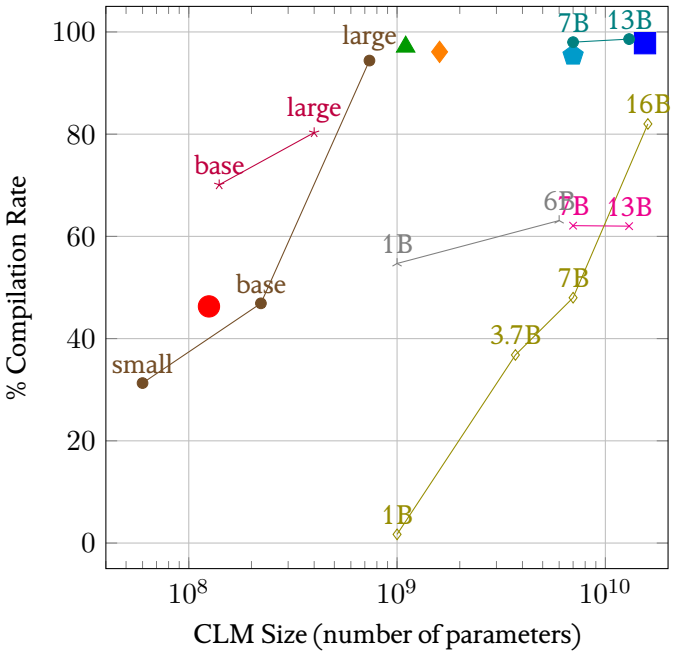


FIGURE 4.7: Compilation rates (%) for infills generated by CLMs.

4.3.3 CLM Size and Compilation Failures

Negative evaluations of code infills are mostly comprised of compilation errors and test failures. Figure 4.7 plots the compilation rate of infills generated by CLMs with respect to their size. Infills generated by CodeT5 Large, SantaCoder and Refact compile successfully 94.4% to 97.1% of the time, while compilation rates for larger models like CodeShell, CodeLLaMA 7B and 13B and StarCoder range from 96.1% to 98.6%. Compilation rates for CLMs smaller than CodeT5 Large quickly diminish, ranging from 80.3% for PLBART Large to 1.7% for CodeGen2-1B.

4.3.4 N=1 and N=5 Success Rate

The previously discussed results present the overall performance of each CLM as the number of infill tasks for which the first infill yielded a correct result, i.e. N=1. We hypothesize that the first generated infill is of the highest quality, and only the first infill should be considered for ARJACLM to obtain an efficient search space. Table 4.3 shows a comparison of N=1 and N=5 performance for each CLM, and Figure 4.6 plots the N=5 performance. On average, the N=5 performance of CLMs is increased by 27% compared to N=1. There are some notable outliers however.

CodeLLaMA Instruct 7B and 13B perform 56.8% and 73.7% better respectively for N=5. An informal manual analysis of several generated infills shows that CodeLLaMA Instruct generates high-quality infills similar to CodeLLaMA, but frequently fails to halt token generation at the appropriate moment, which yields infills containing superfluous code, disturbing the syntax or behavior of the program. Nevertheless, CodeLLaMA Instruct exhibits much improved N=5 performance as the probability of halting token generation at the proper location at least once increases when more infills are generated.

PLBART Base and Large and CodeT5 Small perform 63.4%, 81% and 72.7% better respectively for N=5. In contrast to CodeLLaMA Instruct, these CLMs do not fail to appropriately halt token generation. Instead, correct subsequent infills are frequently a slight variation of the first, incorrect, infill. It must be noted that PLBART Base and Large and CodeT5 Small demonstrate the best performance for both N=1 and N=5 under beam search, which is capable of generating a diverse set of infills consistently, improving the probability of generating correct code at least once.

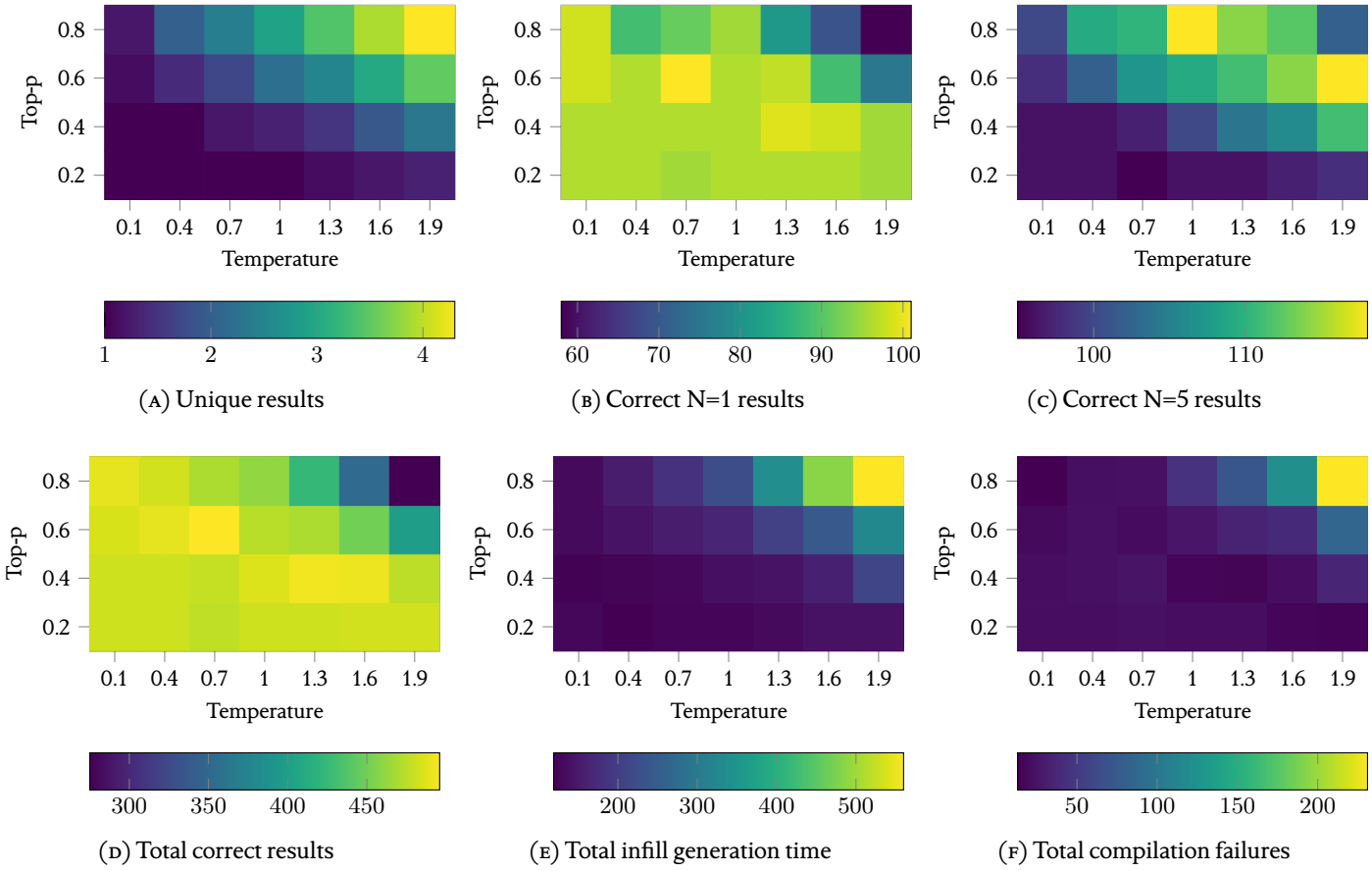
4.3.5 Nucleus Sampling and Beam Search

Some cost characteristics of nucleus sampling and beam search are well known. For example, beam search utilizes more VRAM than nucleus sampling and is usually more time consuming. The most suitable sampling method depends on the model and its application. Table 4.3 shows the best performing sampling settings for each CLM. Again, note that UniXCoder and Refact were only evaluated using beam search and nucleus sampling respectively.

4.4 Infill Diversity and Quality

The evaluation results presented so far, mostly focused on determining the most effective CLMs with regards to the quality of the first infill they generate. Nevertheless, generating additional, unique, infills of high quality is a desirable trait for CLMs in general.

Diverse results are easily obtained using beam search, as over 4.5 average unique infills can be observed for most CLMs when beam search is applied. For example, SantaCoder produces 4.8 and 4.7 unique infills on average for beam sizes 5 and 10 respectively. Meanwhile, the conducted experiments show that all CLMs produce downwards of two unique infills per task for the majority of evaluated nucleus sampling configurations. However, higher result diversity can be achieved with nucleus sampling using high values for top-p and temperature.



(A) Unique results (B) Correct N=1 results (C) Correct N=5 results

(D) Total correct results (E) Total infill generation time (F) Total compilation failures

FIGURE 4.8: Heatmap of Santacoder infilling metrics pertaining result diversity.

This section studies the trade-off between patch quality and diversity for nucleus sampling based on results obtained for SantaCoder. Similar results can be observed for most CLMs. Figure 4.8a shows the average number of unique infills generated by SantaCoder per infill task for all evaluated top-p and temperature values. The number of average unique infills range from 1.0 for low top-p temperature values, while high values for both parameters yield up to 4.3.

Figure 4.8b shows the number of successful N=1 results for the evaluated nucleus sampling configurations. The results demonstrate that the best N=1 results are obtained using moderate parameter values. Performance is strongly diminished for high top-p and temperature values. All other sampling configurations demonstrate a performance loss of less than 20% with respect to the best performing parameters.

Figure 4.8c shows number of successful N=5 results. The sensitivity of N=5 performance is similar to that of N=1. However, N=5 performance is not strongly diminished for high diversity sampling configurations. Notably, the best N=5 performance is obtained by striking a specific balance between moderate to high top-p and temperature values.

Figure 4.8d shows the total number of correct results generated out of all five generated infills per task. The results mostly correspond with N=1 performance, but with more clearly diminished results for high-diversity configurations.

Figure 4.8e shows the time needed to complete all infilling tasks. More time is needed to generate infills for high-diversity sampling configurations. Furthermore, Figure 4.8f shows that infills generated under high-diversity configurations are less likely to be compileable.

The case study of infill diversity on SantaCoder demonstrates that diversity can come at the cost of

result quality when nucleus sampling is used. Depending on the application, a balance must be struck between quality and diversity to obtain the desired result. Similar patterns to those presented for SantaCoder can be observed for Refact, CodeT5 Large, StarCoder, CodeLLaMA and CodeShell.

4.5 Notable Infill Tasks

Some infill tasks which are not fixed by any CLM, or are fixed by exactly one CLM. For N=1, there are 13 tasks for which no CLM produced a test-adequate result, 6 of those could not even be fixed by any model at $N = 5$: `CAR_RACE_COLLISION`, `CHECK_IF_LAST_CHAR_IS_A_LETTER`, `DECODE_SHIFT`, `FIND_ZERO`, `GET_ROW`, `IS_EQUAL_TO_SUM_EVEN`, `MAKE_A_PILE`, `MIN_SUBARRAY_SUM`, `MULTIPLY`, `STARTS_ONE_ENDS`, `STRING_SEQUENCE`, `TRIANGLE_AREA_2` and `TRUNCATE_NUMBER`.

For $N = 5$, CodeLLaMA Instruct 7B managed to fix `TRIANGLE_AREA_2`, CodeT5 Large fixed `MIN_SUBARRAY_SUM`, and PLBART Large managed to fix `TRUNCATE_NUMBER`. Several CLMs fixed `CHECK_IF_LAST_CHAR_IS_A_LETTER`, `DECODE_SHIFT`, `FIND_ZERO` and `STRING_SEQUENCE`. This result shows that despite promising statistics, some relatively simple infill tasks cannot be completed by CLMs.

4.6 Conclusion

This evaluation studies the effectiveness of code infilling capabilities of 20 CLMs with various characteristics. CodeT5, SantaCoder, Refact, CodeShell, CodeLLaMA and StarCoder are the most effective CLMs at generating infills with respect to their cost and size, covering a wide range from 70 million up to 15.5 billion parameters. In addition, 94.4% of infills generated by CodeT5 Large, SantaCoder, Refact, CodeShell, CodeLLaMA and StarCoder were compileable. Finally, larger CLMs are slower, require more VRAM, and diminishing returns can be observed in the effectiveness of CLMs with respect to their size and cost.

The experiments show that effective CLMs exist in a wide range of sizes, and can be selected based on a budget of computational resources. Both Refact and SantaCoder are effective at a low cost, while CodeLLaMA 13B and StarCoder are the most effective CLMs at a higher cost. Larger CLMs like SantaCoder and CodeLLaMA 13B provide more effective code infilling capabilities than their smaller, more efficient, counterparts. AJRACLM is evaluated with a larger CLM to determine its strongest overall bug fixing capabilities. We arbitrarily select CodeLLaMA 13B for this purpose.

The overall difference between N=1 and N=5 performance across all CLMs is 27% on average. This performance improvement is only limited to 8% for the CodeLLaMA 13B however. This improvement does not justify a drastic increase in search space size. Therefore, Chapter 5 evaluates ARJACLM by only considering the first infill generated for each task.

The empirical evaluation of infilling capabilities of CLMs is adequate for obtaining the most suitable CLMs for evaluating ARJACLM. The evaluation is limited in some ways however, which make the results less generalizable.

First, each CLM is evaluated exactly once for each sampling configuration using a fixed seed. The performance these experiments can be influenced by randomness, which could be reduced by executing each benchmark multiple times using different seeds.

Second, generated infills were not manually validated. Infills which pass the test suite may still be incorrect, as it is often infeasible to evaluate all scenarios. It is currently unclear to what extent CLMs generate infills which are test-adequate but incorrect nevertheless.

Third, infilling is evaluated by replacing an existing line of code from HumanEval-Java with a mask token. Thus, all infilling tasks pertain scenario where code should be present in place of the mask token. However, ARJACLM leverages CLMs to generate code at arbitrary suspicious code locations, where

additional code is not always desirable. Thus, the CLM evaluation setting differs from the intended purpose in ARJACLM.

Finally, only a limited range of sampling parameters were evaluated. Table 4.3 shows that several CLMs perform best using nucleus sampling options which are the minimum or maximum of the evaluated ranges for top-p and temperature. It follows that some CLMs might perform better with sampling parameters outside of the evaluated range. Moreover, a more fine-grained range of parameter values can yield slightly better performance for many CLMs.

Chapter 5

Evaluation of ARJACLM

This chapter evaluates ARJACLM CodeLLaMA 13B, and addresses the following research questions:

RQ1: What is the performance of ARJACLM in terms of bug fixing?

To address this research question, the overall performance of ARJACLM needs to be determined. To this end, we perform an systematic investigation to determine the most effective configuration of ARJACLM, and measure its overall performance with CodeLLaMA.

RQ2: What is the performance of ARJACLM compared to other search-based techniques?

A comparison of ARJACLM with other search-based techniques reveals the extent of the performance improvement provided by CLMs.

RQ3: What is the time and memory efficiency of ARJACLM?

The cost of ARJACLM is expected to be higher when using CLMs. We investigate whether the cost of ARJACLM can be justified by its performance.

RQ4: What is the quality of CLM generated patch ingredients?

Evidence on the quality of CLM-based patch ingredients compared to their redundancy-based patch counterparts can be useful for a wide range of APR techniques.

This remainder of this chapter discusses the experimental evaluation of ARJACLM and presents its results.

5.1 Defects4J

The research questions are answered based on experiments on Defects4J 2.0 [29], which is a collection of bugs mined from real-world open-source Java projects, commonly used to evaluate APR techniques [5, 8, 9, 12, 19, 17, 18, 51, 61, 62]. Defects4J 2.0 consists of 835 bugs in total from 17 different projects. For each bug, a test suite is provided with positive test cases, and at least one negative test case which demonstrates the bug. Moreover, tools are provided to facilitate the compilation of the buggy program and the execution of its test suite.

Not all bugs of Defects4J are evaluated for this experiment. Table 5.1 shows how many bugs of each project are evaluated. In total 398 out of 835 bugs of Defects4J are evaluated. Many bugs are excluded from the evaluation for two reasons. First, many test suites of bugs contain *flaky* tests. Flaky tests fail

Project	#Total Bugs	# Pre-processed Bugs	# Evaluated Bugs
Chart	26	26	20
Cli	39	38	20
Closure	174	0	0
Codec	18	18	18
Collections	4	0	0
Compress	47	45	20
Csv	16	0	0
Gson	18	18	18
JacksonCore	26	0	0
JacksonDatabind	112	0	0
JacksonXml	6	0	0
Jsoup	93	91	20
JxPath	22	0	0
Lang	64	38	20
Math	106	99	20
Mockito	38	0	0
Time	26	25	20
Total	**835**	**398**	**176**

TABLE 5.1: Overview of the number of bugs per project of Defects4J, and the number of evaluated bugs.

to produce consistent test results due to randomness, or time-based behavior, for example. Defects4J provides tooling which for obtaining consistent test results across an entire test suite by filtering results of flaky tests, and providing a more consistent execution environment. However, this execution method is not compatible with ARJACLM and GZoltar [63], which is the fault localization technique used for this evaluation. Both tools execute tests directly, and are not compatible with external execution tools. As a result, all flaky tests have to be manually found by the authors, which is a time-consuming process. Therefore not all bugs are evaluated.

Another reason for excluding many Defects4J bugs from this experiment is the inability to directly compile and run all buggy projects. While Defects4J provides tools to simplify the compilation of buggy projects, direct compilation is necessary to allow for instrumentation of GZoltar to be included in the program artifact. On the other hand, direct compilation fails for some bugs on certain system configurations due to the need for specific settings. These settings are provided by Defects4J's compilation tool but are otherwise cumbersome to apply.

Finally, to reduce the substantial computational cost involved with evaluating configurations of ARJACLM on complex real-world bugs, only a subset of the Defects4J bugs that were successfully pre-processed are evaluated. For each project, a maximum of 20 bugs are evaluated. Consequently, ARJACLM is evaluated on 176 bugs in total.

5.2 Evaluation Protocol

For RQ1 we investigate which parameters are optimal for ARJACLM. Table 5.2 shows the default parameters used for this purpose. These default values are based on those of AJRA, and on results for preliminary experimentation. Experiments are performed for specific parameter values in a structured manner, where each experiment evaluates the performance of one modified parameter with respect to a basic parameter setting.

We identify two key parameters which substantially affect the incorporation of CLMs into ARJACLM. Table 5.3 shows these parameters, and their respective evaluated values. p_{clm} and C control how

often the CLM is leveraged, and how much code context is provided respectively. The evaluation of various parameter values provides key insights into effective generation and integration of CLM-based patch ingredients.

Experiments for p_{clm} and C are performed independently. Note that C is evaluated up 800 lines of code to stay within the 16k token context limit of CodeLLaMA. The overall best result of these experiments represents the overall performance of ARJACLM. Moreover, AJRACLM is evaluated without CLMs to determine the relative performance gain of ARJACLM when CLMs are introduced. Metrics obtained for this result is used result is used to answer RQ1-4

Parameter	Description	Default Value
N	Population size	40
G	Maximum number of generations	20
γ_{min}	Suspiciousness threshold	0.1
n_{max}	Maximum number of modification points	40
μ	Scale for number of enabled edits in the initial population	0.06
w_{pos}	Positive test weight	0.33
e	Elite count	1
m_{mut}	Mutation probability multiplier	0.1
p_{mut}	Mutation probability	m_{mut}/n
p_{clm}	CLM mutation ratio	0.4
C	Number of lines of context for mask predict prompts	100

TABLE 5.2: The parameter setting for ARJACLM in the experiments.

Parameter	Evaluated Values
p_{clm}	$\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$
C	$\{100, 200, 400, 800\}$

TABLE 5.3: The values evaluated for parameters of ARJACLM.

For each experiment, the following metrics are collected:

1) "#Fixed Bugs": number of bugs from Defects4J for which one or more test-adequate patch was produced.
2) "#Evaluated Patches": total number of unique evaluated patches.
3) "Total Time": total time elapsed.
4) "CLM Time": time spent generating CLM infills
5) "VRAM": peak VRAM usage.
6) "CLM Parse Rate": rate at which CLM ingredients are syntactically correct and therefore can be successfully parsed as Java code.
7) "CLM Compilation Rate": compilation rate of patches involving at least one CLM ingredient. Note that CLM ingredients are not used in patches if they are not syntactically correct. As a result, patch ingredients are guaranteed to be syntactically correct, and the compilation rate only reflects the rate of semantic correctness of the code.
8) "Redundancy Compilation Rate": compilation rate of patches involving only redundancy-based ingredients. Also see the note on CLM compilation rate, as redundancy-based patch ingredients are also guaranteed to be syntactically correct.

All experiments performed on ARJACLM are executed three times due to the stochastic nature of the technique. Additional trails could provide more significant results, but this is not feasible due to the

high cost involved in the evaluation of APR techniques on Defects4J. Nevertheless, three trials should provide adequate metrics for the performance of ARJACLM. For all metrics, the average value across all trials is reported as the overall result.

All experiments were performed on 2.1 GHz Intel Xeon Silver 4216 processor machines with 32GB memory and a single NVidia A40 GPU with 48GB VRAM. The CPU is shared with other users, but 16 physical cores are allocated for up to 15 parallel patch evaluations, preventing starvation of CPU capacity. The GPU is exclusively dedicated to the APR process, and therefore is not shared with other users. All experiments with CodeLLaMA use 8-bit quantization as providing more context results in higher VRAM usage, and without quantization, the available 48GB of VRAM is inadequate for $C \in \{200, 400, 800\}$ with CodeLLaMA.

5.3 Experimental Results

This section presents the results in order to address RQ1-4.

5.3.1 RQ1: Establishing the performance of ARJACLM

ARJACLM_n represents ARJACLM without the use of CLMs. ARJACLM_n fixes 25.0 bugs on average. Table 5.4 shows the results for experiments on ARJACLM for p_{clm} and C. ARJACLM performs best for high values of p_{clm}. For $p_{clm} = 1.0$, a performance improvement of 34.4% is observed with respect to $p_{clm} = 0.0$, at a 309% increased computation time. This result shows that CLM-based patch ingredients yield a more effective search-based APR technique compared to the redundancy assumption.

Parameter	Value	#Fixed Bugs	Total Time (in hours)
p_{clm}	0.0	29.0	21.6
	0.2	30.3	37.1
	0.4	33.0	53.3
	0.6	33.7	66.3
	0.8	36.0	78.7
	1.0	**39.0**	88.3
C	100	31.8	27.6
	200	32.6	30.5
	400	**35.6**	33.0
	800	33.4	37.2

TABLE 5.4: Results for the evaluation of ARJACLM.

The results for C show that providing more context is only valuable to a limited extent. More context should allow CLMs to generate better code based on more information on code patterns and available code symbols. Despite this, the best performance for ARJACLM is observed when roughly half of the available context size is used. Moreover, a larger context size results in longer execution time. The results provide evidence that CodeLLaMA cannot effectively leverage its full context size in the setting of ARJACLM. It is currently unclear whether this observation is a result of the limited capability of CodeLLaMA to deal with larger contexts, or whether the setting of ARJACLM prevents them from doing so.

Overall, the best result for ARJACLM is obtained for $p_{clm} = 1.0$, generating test-adequate patches for 39.0 out of 176 bugs on average. Thus, ARJACLM fixes 39 bugs whereas ARJACLM_n fixed 25 bugs, noting an improvement in bug fixing capabilities of 56%.

5.3.2 RQ2: Comparison against other search-based techniques

We compare ARJACLM performance on Defects4J against ARJA and GenProg. ARJA is evaluated on 224 bugs of Defects4J 1.0 from the JFreeChart, Joda-Time, Commons Lang and Commons Math projects [9]. Motwani et al. [51] evaluate GenProg on the entirety of Defects4J 1.0, which includes all bugs from the aforementioned projects. ARJACLM is evaluated on 20 bugs for each of the aforementioned projects. We compare the number of bugs fixed on these 80 repair tasks for ARJACLM, ARJA and GenProg.

ARJA is evaluated on Defects4J with a single trial, while GenProg is evaluated with 30 trials. We report the result of the single ARJA trial. GenProg fixes 20.7 bugs on average per trial, and the best performing trial fixes 23 bugs. We report the result of the best GenProg trial to avoid understating its capabilities. ARJA and GenProg are evaluated with time limits of four and three hours respectively. Our experiments use a time limit of one hour, but very few additional patches would be found with a greater time limit.

Project	ARJACLM_n	ARJACLM	GenProg	ARJA
JFreeChart	7.0	9.7	4	8
JodaTime	1.0	3	1	4
Commons Lang	2.0	5	2	3
Commons Math	4.0	5.3	4	5
Total	14	23	11	20

TABLE 5.5: A comparison of ARJACLM, GenProg and ARJA.

ARJACLM_n outperforms GenProg, but does not outperform AJRA. ARJACLM is a novel search-based APR technique implementation, and more extensive effort is required to replicate the search-based APR capabilities of ARJA. Nevertheless, ARJACLM_n is capable of finding test-adequate patches for buggy programs, and provides a framework for evaluating the incorporation of CLMs. ARJACLM slightly outperforms ARJA on the limited set of repair tasks.

5.3.3 RQ3: Efficiency of AJRACLM

	#Fixed	#Eval-	VRAM	Time per Bug (minutes)				CLM Time per Bug (minutes)			
Technique	Bugs	uations	(GiB)	Min	Median	Max	Avg	Min	Median	Max	Avg
ARJACLM_n	14	563.4	-	0.6	6.6	30.5	7.9	-	-	-	-
ARJACLM	23	334.6	21.3	0.8	31.8	64.8	32.3	0	26.8	60.6	26.9

TABLE 5.6: A comparison of cost of ARJACLM variants per bug.

Table 5.6 shows the cost of ARJACLM. As previously discussed results have shown, introducing CLMs yields better resultsFor ARJACLM_n, the average computation time per bug is 7.9 minutes, while ARJACLM takes 32.3 (+308%) minutes on average. ARJACLM spent 26.9 minutes generating ingredients using CodeLLaMA.

We also note that ARJACLM evaluates fewer bugs than ARJACLM_n. The results indicate that fewer patch evaluations are required to find test-adequate patches when CLMs are applied. As a result, the CPU cost of ARJACLM is lower than that of ARJACLM_n. Moreover, the average execution time of 32.3 minutes per bug of ARJACLM can be adequate for practical applications given the time and financial cost of manual bug fixing.

5.3.4 RQ4: Quality of CLM-based patch ingredients

	CLM		Redundancy
Technique	Parse Rate	Compilation Rate	Compilation Rate
ARJACLM$_n$	-	-	60.6%
ARJACLM	43.5%	68.2%	60%

TABLE 5.7: The quality of CLM and redundancy-based patch ingredients.

Table 5.7 compares the quality of CLM and redundancy-based patch ingredients. The results for ARJACLM$_n$ show that 60.6% of patches consisting only of redundancy-based patch ingredients compile successfully. For ARJACLM, 43.5% of patch ingredients generated by CodeLLaMA are syntactically correct, and 68.2% of patches containing a CLM-based patch ingredient successfully compile.

The syntactic correctness of CLM-based patch ingredients is disappointing compared to the results of Chapter 4, where over 98.6% of infills generated by CodeLLaMA were compileable. The setting of CLMs in ARJACLM differs from the CLM evaluation in three key ways. First, ARJACLM provides partial context of code surrounding the each infill location due to context size limitations imposed by CLMs, whereas HumanEval-Java consists of small, complete Java classes. Second, the infill tasks of ARJACLM for Defects4J bugs are more complex than those evaluated in HumanEval-Java, and infills might be of lower quality as a result. Finally, the context of infill tasks may contain buggy code, whereas infill tasks for HumanEval-Java contain strictly correct code.

An informal, manual analysis of syntactically incorrect infills was performed to determine the cause of syntax errors. In some cases, we observed that CodeLLaMA attempts to complete a function rather than provide an infill for the mask token provided inside of it. This occurred in the case where there was a bug elsewhere in the code. In this case CodeLLaMA attempts to complete the code, and does so in a correct manner. However, this hallucination produces a code completion rather than an infill, which results in syntax errors when the generated code is inserted into the existing code.

```
Prompt:
if (x > y) {
    <mask>
}

Result:
    return x;
} else {
    return y;
```

FIGURE 5.1: An example of an infill which is syntactically correct, but alters the structure of the surrounding code.

Another reason for many syntax errors for CLM-based patch ingredients is a constraint posed by the patch representation of ARJACLM. In ARJACLM, each patch ingredient consists of one or more statements. Some infills provided by CodeLLaMA are syntactically correct, but alter the structure of the surrounding code, and cannot be parsed as a separate sequence of statements. Figure 5.1 shows an example of such an infill. ARJACLM considers such infills syntactically incorrect, as only complete statements can be incorporated into the patch representation of ARJACLM. A more flexible patch representation designed specifically for the incorporation of learning-based techniques could overcome this constraint.

The compilation rate of patches containing at least one (syntactically correct) CLM-based patch ingredients is more promising, as the use of at least one CLM-based patch ingredient yields a 12.5% higher compilation rate compared to patches consisting only of redundancy-based ingredients. This supports our findings in Table 5.4, which shows that CLM patch ingredients contribute to better APR capabilities compared to redundancy-based ingredients. We note that the compilation rate of patches in general is expected to be lower than the compilation rates of infills from Chapter 4, as the search-based technique combines and mutates patches with limited consideration of the relation between separate patch ingredients and patch operations.

5.4 Threats to Validity

This section discusses validity concerns regarding this experimental evaluation. First, 176 bugs of Defects4J 2.0 are used to evaluate the performance of ARJACLM. The limited number of bugs in which ARJACLM is evaluated poses a significant limitation on the significant of the obtained results. The available computational resources currently limit our ability to provide more significant results.

Another validity concern regards the training of CLMs on large repositories of public data. Jiang et al. [53] note that projects from Defects4J are possibly included in training sets for CLMs. As a result, CLMs could demonstrate better performance on Defects4J bugs than on real-world bugs. It is currently unclear to which extent Defects4J data is included in the training sets of CLMs, and what performance impact might be observed as a result. CLMs are not trained on pairs of buggy and fixed code however, limiting the impact of CLM training data on experimental results.

This study primarily uses the number of test-adequate patches as a metric to determine the performance of ARJACLM. The number of correct patches are usually substantially lower than the number of correct patches, as demonstrated by many experimental studies on APR techniques [12, 13, 19, 21, 17, 27, 51]. The number of correct patches is commonly obtained through independent manual evaluation of patches by several individuals other than the authors. The setting of this study does not provide a straight-forward avenue for such an evaluation.

Another validity concern regarding the test-adequate patches metric is the variance between the results of experiment trials. ARJA is evaluated on Defects4J using only a single trial [9], and the authors note that more trials are needed to provide more representative performance metrics. GenProg is evaluated based on 20 trials instead. ARJACLM is evaluated based on three trials. Additional trials would reduce the variance of overall results, and provide more convincing performance metric and allow for better selection of optimal parameters.

The primary metric used to represent the cost of ARJACLM is its total execution time. The experiments are performed on a shared CPU however. Adequate cores are allocated to ARJACLM to prevent starvation of computational resources. Nevertheless, other processes running on the same CPU can affect the speed of the entire CPU, resulting in inconsistent execution times.

Finally, the validity of this empirical study is limited by the novel ARJACLM technique and its implementation. ARJACLM_n is intended to replicate ARJA, but fails to provide the same bug fixing capabilities. Therefore, this study does not provide direct evidence that CLMs can augment state-of-the-art search-based techniques. Future research should investigate whether the relative performance gain of ARJACLM over ARJACLM_n can be replicated for ARJA.

Chapter 6

Future Work

This chapter provides four avenues of potential future work regarding hybrid search-based APR techniques which leverage CLM-based patch ingredients.

First, this study demonstrates that CLM-based patch ingredients can significantly improve the performance of search-based APR techniques. Nevertheless, ARJACLM$_n$ does not have the same repair capabilities as ARJA, and it is possible that the impact of CLM-based patch ingredients on a more capable search-based technique gives rise to different results. Future work should investigate whether the same performance improvement can be achieved for a state-of-the-art search-based technique.

Second, our results show that the patch representation of AJRACLM is not entirely suitable for leveraging CLM-generated patch ingredients. Patch ingredients in ARJACLM represent entire statements, and as a result, infills generated by CLMs cannot be incorporated into ARJACLM if they alter the structure of surrounding statements. A more flexible patch representation could enable the use of more CLM-generated patch ingredients, and provides CLMs with more freedom in code generation.

The third avenue of future work regards the improvement of the quality of CLM-based patch ingredients. The results for RQ4 show that the majority of CLM-based patch ingredients are syntactically incorrect, while our evaluation of CLM demonstrates that many CLMs are capable of generating compileable code over 95% of the time. Moreover, the case study of syntactically incorrect infills shows that many incorrect infills are not caused by the lack of infill capabilities of CLMs, but are the result of the constraints posed upon CLMs by the mask predict format. Improving quality of CLM-based patch ingredients is a promising avenue of research and higher quality ingredients would further improve the performance and efficiency of ARJACLM.

Fine-tuning is a promising method for improving the quality of infills for ARJACLM. Jiang et al. [53] demonstrate that fine-tuning can improve overall infill quality. In addition, it can provide CLMs with the ability to leverage context of the buggy code line to improve infill quality specifically for APR techniques. The existing buggy code line provides valuable context to the existing code and allows powerful CLMs like CodeLLaMA to generate even more suitable infills.

Besides mask prediction, search-based APR techniques could leverage alternative code generation methods which provide CLMs with more control over the code edit location. For example, CLMs with strong natural language capabilities can be prompted to edit the code location which it deems most suspicious. Such a code generation task omits the constraint imposed by mask prediction tasks upon CLMs.

A third avenue of research could be to leverage more infills per code location. ARJACLM generates only single infill for each infill task, under the hypothesis that subsequent infills are likely of lower quality. In Chapter 4, we observe only an 8% performance improvement from $N = 1$ (i.e. a single infill) to $N = 5$ (i.e. the best of five infills). This hypothesis however does not consider that ARJACLM rejects CLM-based patch ingredients which are syntactically incorrect. It is possible that subsequent infills might provide additional value in such scenarios. Moreover, additional infills might be valuable for

highly suspicious statements, and additional cost of generating more infills would be mostly mitigated if they are generated only in specific cases.

Finally, to further augment search-based APR techniques, CLMs can be leveraged in more ways. CLMs are not only capable of generating code. In addition, CLMs can provide a score for the occurrence probability of code. This capability could be leveraged to screen patch ingredients, provide an additional search objective, or determine where code should be added or removed at a specific location. The impact of these potential improvements are mostly unexplored at the time of writing, and might give rise to a new sequence of research on hybrid search-based APR techniques.

Chapter 7

Conclusion

Traditional search-based APR techniques are constrained in their repair capabilities by the availability of high-quality patch ingredients. Bugs can only be fixed by such techniques if the ingredients to a patch are present elsewhere in the code. Previous work has attempted to generate additional patch ingredients by modifying redundancy-based ingredients, or by leveraging a learning-based APR technique, but these efforts have not yielded significant improvements. We study the potential to generate additional patch ingredients using pre-trained code language models (CLMs). ARJACLM is proposed to empirically evaluate the impact of CLM-based patch ingredients on a search-based APR technique based on ARJA.

CLMs are used to generate patch ingredients for ARJACLM on-the-fly in a zero-shot setting. We perform an extensive comparative analysis of well-known CLMs to determine which state-of-the-art CLMs are most effective with respect to their cost. We find that SantaCoder, Refact, CodeLLaMA and StarCoder effectively generate high-quality code infills with respect to their resource usage, where SantaCoder and Refact are smaller models, while CodeLLaMA and StarCoder are larger and more resource intensive. Smaller CLMs like CodeT5 provide even stronger infilling capabilities with respect to their cost, but they perform poorly overall, frequently generating code that fails to compile. A high level of infill quality is required as search-based APR techniques are only effective for efficient search spaces, and therefore smaller CLMs like CodeT5 Large are excluded.

We systematically evaluate ARJACLM with various parameters and obtain an effective set of parameters. Using these parameters, we obtain the overall performance of ARJACLM without CLM-based patch ingredients, and with ingredients generated by CodeLLaMA. The results demonstrate that ARJACLM out-performs ARJACLM without any CLM. We conclude that CLMs substantially improve the performance of search-based APR techniques. In addition, larger, more capable, CLMs provide a stronger performance improvement than their smaller counterparts. Moreover, ARJACLM performs best when only CLM-based patch ingredients are used. CLM-based patch ingredients are clearly of higher quality than their redundancy-based counterparts. This conclusion provides clear direction to future research into search-based APR techniques, which can avoid the cumbersome process of obtaining and screening redundancy-based donor code.

The performance of ARJA and GenProg on 80 bugs of Defects4J are compared to the performance of ARJACLM and ARJACLM_n on those same bugs. ARJA out-performs ARJACLM_n, but ARJACLM out-performs ARJA. More extensive effort is required to provide ARJACLM_n with the same bug-fixing capabilities of ARJA. Nevertheless, ARJACLM_n out-performs GenProg, and as a result shows that a significant performance improvement can be achieved for search-based APR techniques which perform at the level of well-known existing techniques. Future work should investigate the augmentation of ARJACLM_n to achieve similar performance to ARJA, such that the impact of CLMs on state-of-the-art search-based techniques can be more extensively investigated.

The cost of three ARJACLM variants are evaluated with respect to their cost. CLM-based vari-

ants introduce a higher computational cost per bug. Nevertheless, a substantial improvement in bug fixing capabilities is observed. Finally, we evaluate the quality of both redundancy and CLM-based patch ingredients. Results show than CLM-based patch ingredients are often not parseable, and frequently do not compile. Nevertheless, CLM-based patch ingredients more frequently compile than their redundancy-based counterparts. Moreover, the experimental results demonstrate that CLM-based patch ingredients play a stronger role in fixing bugs than redundancy-based ones. Still, future work should investigate ways to more effectively harness the strong infilling capabilities of CLMs, which are sometimes constrained in their bug fixing capabilities due to the strict mask prediction prompt format. Improved code generation methods could even further improve the performance of APR techniques leveraging CLM-based patch ingredients.

To conclude, CLMs can be used to effectively improve the performance of search-based techniques, albeit at a higher computational cost, but more extensive effort should be invested into evaluating its effect on state-of-the-art techniques. Moreover, we conclude that CLM-based patch ingredients are of higher quality than their redundancy-based counterparts, and many avenues of future research exist to further improve the quality of CLM-based ingredients.

Appendices

Appendix A

CLM Evaluation Results

This appendix contains the results for all benchmarks performed for each CLM. Each table contains the time spent on mask prediction in seconds, beam size, top-p. temperature, the number of unique results generated and N=5 performance (N=5 Suc). Moreover, the following metrics are provided for the first generated infill (N=1) and in total for all generated infills: number of correct infills (Suc), test failures (TFail), compilation failures (CFail), test compilation failure (TCFail), test timeout (TTo) and no result (NoRes).

A.1 UniXCoder

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1 Suc	TFail	CFail	TCFail	TTo	NoRes	Total Suc	TFail	CFail	TCFail	TTo	NoRes
5			107.3	4.6	4.2	34	26	65	72	0	0	0	71	266	391	0	0	87
10			137.7	12.4	4.7	35	25	66	72	0	0	0	80	307	428	0	0	0

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
	0.2	0.1	99.2	5.6	1	102	102	51	9	1	0	0	510	258	42	5	0	0
	0.4	0.1	96.2	5.6	1.2	104	103	50	9	1	0	0	511	262	37	5	0	0
	0.6	0.1	107.6	5.6	1.8	121	112	43	7	1	0	0	536	241	33	5	0	0
	0.8	0.1	112.6	5.6	1.9	119	104	54	4	1	0	0	523	255	32	5	0	0
	0.2	0.4	96.1	5.6	1	102	102	51	9	1	0	0	510	258	42	5	0	0
	0.4	0.4	97	5.6	1.2	104	103	50	9	1	0	0	511	262	37	5	0	0
	0.6	0.4	109.2	5.6	1.9	119	104	52	6	1	0	0	516	255	39	5	0	0
	0.8	0.4	114.2	5.6	1.9	121	107	49	6	1	0	0	520	249	41	5	0	0
	0.2	0.7	96.3	5.6	1.1	101	101	53	8	1	0	0	505	265	40	5	0	0
	0.4	0.7	100.8	5.6	1.3	104	100	55	7	1	0	0	505	270	35	5	0	0
	0.6	0.7	103.6	5.6	1.8	111	99	56	7	1	0	0	501	279	30	5	0	0
	0.8	0.7	119.5	5.6	2.1	116	97	56	8	1	1	0	496	264	44	5	6	0
	0.2	1	87.2	5.6	1.1	106	106	48	8	1	0	0	530	241	39	5	0	0
	0.4	1	93.1	5.6	1.3	107	105	50	7	1	0	0	524	248	38	5	0	0
	0.6	1	113.8	5.6	1.8	113	104	53	5	1	0	0	517	261	32	5	0	0
	0.8	1	145.6	5.6	2.7	126	90	62	10	1	0	0	487	273	50	5	0	0
	0.2	1.3	91.5	5.6	1.1	102	101	52	9	1	0	0	507	258	45	5	0	0
	0.4	1.3	123.1	5.6	1.7	110	105	53	4	1	0	0	506	269	36	4	0	0
	0.6	1.3	147.1	5.6	2.4	123	100	52	10	1	0	0	495	261	53	4	2	0
	0.8	1.3	171.8	5.6	3.1	119	87	56	18	1	1	0	441	284	79	5	6	0
	0.2	1.6	120.2	5.6	1.3	104	102	52	8	1	0	0	510	261	39	5	0	0
	0.4	1.6	168.9	5.6	2.1	120	99	53	10	1	0	0	491	268	48	5	3	0
	0.6	1.6	209.7	5.6	3	121	91	55	15	1	1	0	455	287	65	4	4	0
	0.8	1.6	360.9	5.6	3.7	118	72	61	29	1	0	0	384	271	153	3	4	0
	0.2	1.9	120.9	5.6	1.3	107	106	50	6	1	0	0	526	249	35	5	0	0
	0.4	1.9	201.9	5.6	2.5	116	91	58	13	1	0	0	466	270	72	5	2	0
	0.6	1.9	272.6	5.6	3.5	119	83	55	22	1	2	0	403	287	116	4	5	0
	0.8	1.9	1052.2	5.6	4.3	91	55	37	68	1	2	0	264	238	303	4	6	0

A.3 SantaCoder

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			495.2	4	4.8	108	91	45	26	1	0	0	284	310	213	5	3	0
10			656	5.1	4.7	98	79	38	45	1	0	0	231	231	348	5	0	0
	0.2	0.1	125.6	3.9	1	96	96	62	4	1	0	0	480	310	20	5	0	0
	0.4	0.1	119.9	3.9	1	96	96	62	4	1	0	0	480	310	20	5	0	0
	0.6	0.1	128.8	3.9	1.1	98	98	62	2	1	0	0	483	309	18	5	0	0
	0.8	0.1	129.5	3.9	1.2	100	98	62	2	1	0	0	488	309	13	5	0	0
	0.2	0.4	118.5	3.9	1	96	96	62	4	1	0	0	480	310	20	5	0	0
	0.4	0.4	122.5	3.9	1	96	96	61	5	1	0	0	480	308	22	5	0	0
	0.6	0.4	139.6	3.9	1.4	102	96	61	5	1	0	0	488	300	22	5	0	0
	0.8	0.4	153.4	3.9	2	109	88	69	5	1	0	0	481	308	21	5	0	0
	0.2	0.7	123.6	3.9	1	95	95	63	4	1	0	0	475	313	22	5	0	0
	0.4	0.7	125.8	3.9	1.2	97	96	62	4	1	0	0	477	309	24	5	0	0
	0.6	0.7	153.1	3.9	1.7	107	101	57	4	1	0	0	496	295	19	5	0	0
	0.8	0.7	180.9	3.9	2.4	110	91	65	7	0	0	0	469	318	22	4	2	0
	0.2	1	125.1	3.9	1	96	96	62	4	1	0	0	480	310	20	5	0	0
	0.4	1	135.6	3.9	1.3	100	96	62	4	1	0	0	485	309	16	5	0	0
	0.6	1	164.2	3.9	2.2	109	96	60	6	1	0	0	473	312	25	5	0	0
	0.8	1	221.4	3.9	2.9	118	95	61	6	1	0	0	460	304	44	5	2	0
	0.2	1.3	128.6	3.9	1.1	96	96	62	4	1	0	0	480	310	20	5	0	0
	0.4	1.3	140	3.9	1.5	104	99	61	2	1	0	0	492	303	15	5	0	0
	0.6	1.3	203.1	3.9	2.5	111	97	56	9	1	0	0	469	305	34	5	2	0
	0.8	1.3	336.4	3.9	3.4	114	81	66	15	1	0	0	422	318	71	4	0	0
	0.2	1.6	139.3	3.9	1.2	97	96	64	2	1	0	0	481	313	16	5	0	0
	0.4	1.6	157.3	3.9	1.9	106	98	58	6	1	0	0	490	300	20	5	0	0
	0.6	1.6	240.9	3.9	3	114	88	66	7	1	1	0	449	323	38	3	2	0
	0.8	1.6	482.6	3.9	3.9	112	69	63	30	0	1	0	351	330	122	3	9	0
	0.2	1.9	137.8	3.9	1.3	98	95	64	3	1	0	0	481	315	14	5	0	0
	0.4	1.9	210	3.9	2.3	111	95	58	8	1	1	0	474	300	35	4	2	0
	0.6	1.9	325	3.9	3.5	118	75	70	17	1	0	0	397	330	84	4	0	0
	0.8	1.9	559.8	3.9	4.3	102	58	62	40	1	2	0	275	298	231	3	8	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			837	19.2	4.7	127	113	26	23	1	0	0	317	190	301	5	2	0
10			1087.3	23	4.6	116	102	21	39	1	0	0	251	140	416	5	3	0
	0.2	0.1	230.7	18.9	1	121	121	37	4	1	0	0	605	185	20	5	0	0
	0.4	0.1	225.3	18.9	1	121	121	37	4	1	0	0	605	185	20	5	0	0
	0.6	0.1	239.2	18.9	1.2	126	123	35	4	1	0	0	609	180	21	5	0	0
	0.8	0.1	240.1	18.9	1.2	127	124	33	5	1	0	0	613	173	24	5	0	0
	0.2	0.4	234.6	18.9	1	121	121	37	4	1	0	0	605	185	20	5	0	0
	0.4	0.4	239.9	18.9	1	121	121	37	4	1	0	0	605	183	22	5	0	0
	0.6	0.4	257.2	18.9	1.3	127	122	34	6	1	0	0	609	173	28	5	0	0
	0.8	0.4	292.8	18.9	1.7	133	122	36	4	1	0	0	609	170	30	5	1	0
	0.2	0.7	233	18.9	1	122	122	36	4	1	0	0	610	180	20	5	0	0
	0.4	0.7	247.4	18.9	1.1	123	122	36	4	1	0	0	609	178	22	5	1	0
	0.6	0.7	279.3	18.9	1.6	132	125	30	6	1	1	0	622	160	27	5	1	0
	0.8	0.7	326.2	18.9	2.2	140	121	33	7	1	1	0	599	176	33	5	2	0
	0.2	1	233.5	18.9	1	122	122	36	4	1	0	0	610	180	20	5	0	0
	0.4	1	254.1	18.9	1.3	127	124	34	4	1	0	0	620	166	23	5	1	0
	0.6	1	311.9	18.9	1.9	135	122	33	6	1	1	0	609	162	36	4	4	0
	0.8	1	373.6	18.9	2.5	139	109	39	13	1	1	0	571	194	41	5	4	0
	0.2	1.3	243.3	18.9	1	122	122	36	4	1	0	0	610	177	23	5	0	0
	0.4	1.3	266	18.9	1.4	127	124	33	4	1	1	0	613	166	29	5	2	0
	0.6	1.3	321.5	18.9	2.2	138	118	36	9	0	0	0	588	187	35	3	2	0
	0.8	1.3	470.3	18.9	2.9	145	115	37	10	1	0	0	559	202	48	5	1	0
	0.2	1.6	245.8	18.9	1.1	121	121	37	4	1	0	0	604	185	21	5	0	0
	0.4	1.6	302.7	18.9	1.7	130	123	33	6	1	0	0	608	158	44	5	0	0
	0.6	1.6	408.1	18.9	2.6	140	110	44	8	0	1	0	553	202	54	4	2	0
	0.8	1.6	680.8	18.9	3.6	136	93	45	24	1	0	0	468	223	116	5	3	0
	0.2	1.9	292.4	18.9	1.2	125	124	34	4	1	0	0	617	172	21	5	0	0
	0.4	1.9	409.1	18.9	2	133	123	34	5	1	0	0	593	186	31	5	0	0
	0.6	1.9	690.7	18.9	3.1	141	108	34	21	0	0	0	522	194	95	3	1	0
	0.8	1.9	1009.8	18.9	4	129	80	43	40	0	0	0	400	233	181	1	0	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			854.5	33	4.6	138	131	20	11	1	0	0	435	170	205	5	0	0
10			1439.2	36.8	4.7	134	119	22	21	1	0	0	315	116	379	5	0	0
	0.2	0.1	264.4	33	1	131	131	28	3	1	0	0	655	140	15	5	0	0
	0.4	0.1	263	33	1	131	131	28	3	1	0	0	655	140	15	5	0	0
	0.6	0.1	263.6	33	1.1	131	131	28	3	1	0	0	653	142	15	5	0	0
	0.8	0.1	261.4	33	1.2	133	129	30	3	1	0	0	655	142	13	5	0	0
	0.2	0.4	263.7	33	1	131	131	28	3	1	0	0	655	140	15	5	0	0
	0.4	0.4	264.6	33	1	131	131	28	3	1	0	0	655	140	15	5	0	0
	0.6	0.4	274.4	33	1.2	136	130	29	3	1	0	0	657	140	13	5	0	0
	0.8	0.4	285.2	33	1.5	138	131	29	2	1	0	0	657	146	6	5	1	0
	0.2	0.7	263.3	33	1	131	131	28	3	1	0	0	655	140	15	5	0	0
	0.4	0.7	281.3	33	1.1	131	131	27	4	1	0	0	655	136	19	5	0	0
	0.6	0.7	294	33	1.5	136	130	29	3	1	0	0	658	140	12	5	0	0
	0.8	0.7	307.1	33	1.8	141	136	23	3	1	0	0	653	144	13	5	0	0
	0.2	1	267.6	33	1	131	131	28	3	1	0	0	655	140	15	5	0	0
	0.4	1	288.4	33	1.2	133	132	27	3	1	0	0	656	136	18	5	0	0
	0.6	1	315.9	33	1.8	144	133	28	1	1	0	0	658	143	9	5	0	0
	0.8	1	344.2	33	2.3	141	125	31	3	1	3	0	631	158	17	5	4	0
	0.2	1.3	270.2	33	1.1	131	131	28	3	1	0	0	655	139	16	5	0	0
	0.4	1.3	298.1	33	1.4	134	127	31	3	1	1	0	654	140	15	5	1	0
	0.6	1.3	341.2	33	2	143	131	28	3	1	0	0	635	165	10	5	0	0
	0.8	1.3	439	33	2.7	146	120	37	5	1	0	0	612	165	33	5	0	0
	0.2	1.6	273.5	33	1.1	131	131	28	3	1	0	0	655	139	16	5	0	0
	0.4	1.6	299	33	1.5	140	131	29	2	1	0	0	656	145	9	5	0	0
	0.6	1.6	389.3	33	2.4	144	121	40	1	1	0	0	616	164	28	5	2	0
	0.8	1.6	649.3	33	3.3	138	112	38	12	1	0	0	543	199	69	4	0	0
	0.2	1.9	284.7	33	1.2	131	131	27	4	1	0	0	655	135	20	5	0	0
	0.4	1.9	335.6	33	1.9	140	130	27	4	1	1	0	634	159	15	5	2	0
	0.6	1.9	559.2	33	2.9	143	116	38	8	1	0	0	574	177	59	5	0	0
	0.8	1.9	857.2	33	3.7	137	98	33	31	1	0	0	470	177	163	3	2	0

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			418.7	1.2	5	56	41	90	30	1	1	0	112	407	288	5	3	0
10			417.3	1.4	5	67	41	93	28	1	0	0	129	442	239	5	0	0
	0.2	0.1	383.5	1.1	1	26	26	87	48	1	1	0	130	435	240	5	5	0
	0.4	0.1	362.6	1.1	1	26	26	87	48	1	1	0	130	435	240	5	5	0
	0.6	0.1	390.2	1.1	1.3	26	25	86	50	1	1	0	125	427	253	5	5	0
	0.8	0.1	384.1	1.1	1.6	31	26	87	48	1	1	0	127	423	256	5	4	0
	0.2	0.4	382.8	1.1	1	26	26	87	48	1	1	0	130	435	240	5	5	0
	0.4	0.4	391.8	1.1	1.1	26	26	86	49	1	1	0	129	429	247	5	5	0
	0.6	0.4	386.9	1.1	1.9	32	27	80	54	1	1	0	131	429	246	5	4	0
	0.8	0.4	394.9	1.1	2.9	37	25	83	54	1	0	0	128	426	256	5	0	0
	0.2	0.7	394.6	1.1	1	26	26	86	49	1	1	0	130	429	246	5	5	0
	0.4	0.7	390.4	1.1	1.4	27	24	85	53	1	0	0	128	423	255	5	4	0
	0.6	0.7	387.4	1.1	2.7	33	24	81	56	1	1	0	122	434	251	5	3	0
	0.8	0.7	419.9	1.1	3.8	43	23	88	51	1	0	0	121	425	264	5	0	0
	0.2	1	367.4	1.1	1.1	26	26	86	50	1	0	0	129	433	248	5	0	0
	0.4	1	375.6	1.1	2.1	32	24	88	50	1	0	0	126	427	257	5	0	0
	0.6	1	389.7	1.1	3.5	44	25	85	52	1	0	0	131	412	263	5	4	0
	0.8	1	400.3	1.1	4.4	43	24	70	68	1	0	0	103	387	317	5	3	0
	0.2	1.3	394.7	1.1	1.3	26	25	87	49	1	1	0	127	426	252	5	5	0
	0.4	1.3	389.2	1.1	2.9	34	28	85	49	1	0	0	131	424	252	5	3	0
	0.6	1.3	384.6	1.1	4.2	44	25	73	62	1	2	0	118	397	292	4	4	0
	0.8	1.3	440.2	1.1	4.6	43	22	63	76	1	1	0	98	311	395	4	7	0
	0.2	1.6	384.7	1.1	1.7	28	24	84	53	1	1	0	122	425	259	5	4	0
	0.4	1.6	400.5	1.1	3.8	43	23	79	59	1	1	0	118	395	295	5	2	0
	0.6	1.6	408.7	1.1	4.5	42	19	74	69	1	0	0	94	342	374	5	0	0
	0.8	1.6	435.5	1.1	4.8	32	13	39	111	0	0	0	64	196	555	0	0	0
	0.2	1.9	385.4	1.1	2.3	31	23	90	49	1	0	0	121	422	267	5	0	0
	0.4	1.9	412.4	1.1	4.3	48	25	66	71	1	0	0	119	347	341	4	4	0
	0.6	1.9	403	1.1	4.8	29	15	43	104	0	1	0	67	244	498	3	3	0
	0.8	1.9	484.4	1.2	5	19	7	15	141	0	0	0	31	83	700	1	0	0

A.7 PLBART Large

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			716.4	2	5	73	42	99	20	1	1	0	162	448	195	4	6	0
10			746.4	2.8	5	76	42	98	21	1	1	0	173	475	156	3	8	0
	0.2	0.1	667.7	1.9	1	38	38	91	31	1	2	0	190	455	155	5	10	0
	0.4	0.1	672	1.9	1	38	38	91	31	1	2	0	190	455	155	5	10	0
	0.6	0.1	642.8	1.9	1.2	40	37	93	30	1	2	0	188	460	152	5	10	0
	0.8	0.1	655.6	1.9	1.6	41	37	90	32	1	3	0	186	449	162	5	13	0
	0.2	0.4	671.7	1.9	1	38	38	91	31	1	2	0	190	455	155	5	10	0
	0.4	0.4	650.9	1.9	1.1	38	37	92	31	1	2	0	186	454	160	5	10	0
	0.6	0.4	663.9	1.9	1.9	46	34	91	33	1	4	0	182	453	162	5	13	0
	0.8	0.4	664.3	1.9	3	47	39	95	27	1	1	0	175	467	160	5	8	0
	0.2	0.7	641.8	1.9	1	38	38	91	31	1	2	0	190	455	155	5	10	0
	0.4	0.7	649.6	1.9	1.4	42	39	89	32	1	2	0	191	448	158	5	13	0
	0.6	0.7	661.7	1.9	2.7	48	33	93	34	1	2	0	177	468	159	5	6	0
	0.8	0.7	675.3	1.9	3.7	62	38	82	40	1	2	0	184	446	171	5	9	0
	0.2	1	640.3	1.9	1.1	38	37	92	31	1	2	0	188	457	155	5	10	0
	0.4	1	655.5	1.9	1.9	42	36	95	29	1	2	0	187	464	149	5	10	0
	0.6	1	679	1.9	3.6	56	33	97	31	1	1	0	173	469	161	5	7	0
	0.8	1	676.2	1.9	4.3	65	34	89	37	1	2	0	164	438	196	5	12	0
	0.2	1.3	645.7	1.9	1.2	42	40	89	31	1	2	0	195	448	156	5	11	0
	0.4	1.3	632.9	1.9	2.9	51	42	93	26	1	1	0	188	461	151	5	10	0
	0.6	1.3	656.6	1.9	4.1	59	32	90	38	1	2	0	164	455	185	5	6	0
	0.8	1.3	684	1.9	4.7	53	28	72	61	1	1	0	126	394	285	5	5	0
	0.2	1.6	614.7	1.9	1.5	43	39	92	29	1	2	0	194	452	154	5	10	0
	0.4	1.6	644.7	1.9	3.6	53	32	89	40	1	1	0	172	456	174	5	8	0
	0.6	1.6	664.6	1.9	4.4	55	24	88	50	1	0	0	129	411	265	5	5	0
	0.8	1.6	682.7	1.9	4.8	38	21	42	98	1	1	0	85	238	485	3	4	0
	0.2	1.9	665.6	1.9	2	44	38	90	31	1	3	0	180	461	160	5	9	0
	0.4	1.9	692.2	1.9	4.1	55	37	87	38	0	1	0	159	418	229	4	5	0
	0.6	1.9	913.5	1.9	4.8	44	19	55	86	0	3	0	86	287	433	4	5	0
	0.8	1.9	968.8	1.9	4.9	21	3	31	129	0	0	0	37	128	649	1	0	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			33.1	0.9	4.6	36	22	60	81	0	0	0	67	191	555	0	2	0
10			34.7	1	4.8	38	22	61	80	0	0	0	67	187	557	0	4	0
	0.2	0.1	24	0.9	1	18	18	51	94	0	0	0	90	255	470	0	0	0
	0.4	0.1	19.7	0.9	1	18	18	51	94	0	0	0	90	255	470	0	0	0
	0.6	0.1	18.8	0.9	1.1	18	18	52	93	0	0	0	90	254	471	0	0	0
	0.8	0.1	18.4	0.9	1.2	18	18	50	95	0	0	0	82	252	481	0	0	0
	0.2	0.4	18.5	0.9	1	18	18	51	94	0	0	0	90	255	470	0	0	0
	0.4	0.4	21	0.9	1.1	18	18	50	95	0	0	0	89	246	480	0	0	0
	0.6	0.4	19	0.9	1.5	18	15	53	95	0	0	0	76	261	478	0	0	0
	0.8	0.4	22.4	0.9	1.9	22	17	47	99	0	0	0	76	240	499	0	0	0
	0.2	0.7	18.6	0.9	1.1	18	17	51	95	0	0	0	89	254	472	0	0	0
	0.4	0.7	20.4	0.9	1.4	19	17	50	96	0	0	0	85	252	478	0	0	0
	0.6	0.7	19.2	0.9	2.1	22	19	51	92	0	1	0	80	236	495	0	4	0
	0.8	0.7	31.3	0.9	2.7	23	18	47	97	0	1	0	76	216	520	0	3	0
	0.2	1	27.1	0.9	1.3	19	19	49	95	0	0	0	90	246	479	0	0	0
	0.4	1	27.3	0.9	2	20	18	47	98	0	0	0	83	235	497	0	0	0
	0.6	1	30.2	0.9	2.8	22	14	48	101	0	0	0	73	215	527	0	0	0
	0.8	1	18.9	0.9	3.4	21	11	40	110	0	2	0	67	186	558	0	4	0
	0.2	1.3	21.7	0.9	1.6	19	17	47	99	0	0	0	84	244	487	0	0	0
	0.4	1.3	19.6	0.9	2.6	21	19	44	98	0	2	0	83	206	521	0	5	0
	0.6	1.3	25.4	0.9	3.6	20	11	41	111	0	0	0	62	170	583	0	0	0
	0.8	1.3	23.4	0.9	4.1	19	11	32	120	0	0	0	56	123	636	0	0	0
	0.2	1.6	20.6	0.9	2	19	16	45	102	0	0	0	78	221	516	0	0	0
	0.4	1.6	20.7	0.9	3.3	26	16	39	108	0	0	0	71	188	552	0	4	0
	0.6	1.6	34.6	0.9	4.1	19	12	30	120	0	1	0	59	135	617	0	4	0
	0.8	1.6	28.3	0.9	4.6	19	12	25	126	0	0	0	45	115	654	0	1	0
	0.2	1.9	19.5	0.9	2.3	19	16	43	104	0	0	0	72	201	542	0	0	0
	0.4	1.9	23.2	0.9	3.9	19	10	35	117	0	1	0	55	154	604	0	2	0
	0.6	1.9	26.9	0.9	4.5	23	9	25	128	0	1	0	53	119	638	0	5	0
	0.8	1.9	30	0.9	4.8	15	7	21	134	0	1	0	33	80	700	0	2	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			57.8	1.5	4.8	61	50	57	54	0	2	0	151	227	427	1	9	0
10			76.4	1.7	4.9	61	50	58	54	0	1	0	145	233	429	1	7	0
	0.2	0.1	80.3	1.3	1	48	48	59	54	0	2	0	240	296	270	0	9	0
	0.4	0.1	55	1.3	1	48	48	59	54	0	2	0	240	295	270	0	10	0
	0.6	0.1	68.9	1.3	1.1	50	49	60	53	0	1	0	244	296	270	0	5	0
	0.8	0.1	58.7	1.3	1.2	50	49	59	54	0	1	0	245	283	281	0	6	0
	0.2	0.4	68.7	1.3	1	48	48	59	54	0	2	0	240	295	270	0	10	0
	0.4	0.4	68.7	1.3	1	48	48	60	53	0	2	0	240	299	268	0	8	0
	0.6	0.4	70.5	1.3	1.3	49	48	61	53	0	1	0	237	286	287	0	5	0
	0.8	0.4	63.5	1.3	1.7	49	47	58	57	0	1	0	225	279	305	1	5	0
	0.2	0.7	57.6	1.3	1	48	48	59	54	0	2	0	240	295	270	0	10	0
	0.4	0.7	70.5	1.3	1.1	48	48	60	54	0	1	0	238	302	270	0	5	0
	0.6	0.7	66.9	1.3	1.7	48	45	61	56	0	1	0	222	293	295	0	5	0
	0.8	0.7	66.1	1.3	2.3	50	45	68	48	0	2	0	205	294	307	0	9	0
	0.2	1	54.5	1.3	1	48	48	59	54	0	2	0	240	295	270	0	10	0
	0.4	1	61.1	1.4	1.4	47	47	61	54	0	1	0	234	293	282	0	6	0
	0.6	1	67.6	1.3	2.4	51	48	54	60	0	1	0	214	264	330	1	6	0
	0.8	1	86.1	1.3	2.9	49	43	53	67	0	0	0	199	252	359	1	4	0
	0.2	1.3	55.4	1.3	1.1	48	48	60	53	0	2	0	240	298	267	0	10	0
	0.4	1.3	61	1.3	1.8	50	46	59	57	0	1	0	223	289	296	0	7	0
	0.6	1.3	72.6	1.3	2.9	51	42	54	66	0	1	0	199	250	360	1	5	0
	0.8	1.3	77.2	1.3	3.6	50	41	55	65	1	1	0	183	233	392	2	5	0
	0.2	1.6	68.7	1.3	1.2	48	48	59	54	0	2	0	236	303	269	0	7	0
	0.4	1.6	69	1.3	2.4	49	43	55	64	0	1	0	207	270	334	0	4	0
	0.6	1.6	87.6	1.4	3.5	53	44	53	65	0	1	0	186	236	389	0	4	0
	0.8	1.6	68.6	1.4	4.3	54	34	45	84	0	0	0	146	212	455	0	2	0
	0.2	1.9	54.9	1.3	1.5	49	49	61	52	0	1	0	235	309	266	0	5	0
	0.4	1.9	59.8	1.3	2.9	52	40	55	67	0	1	0	194	235	381	0	5	0
	0.6	1.9	69.4	1.3	4	47	33	52	76	1	1	0	142	219	449	1	4	0
	0.8	1.9	78.8	1.4	4.6	42	19	40	104	0	0	0	90	160	562	0	3	0

A.10 CodeT5 Large

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			108.8	3.1	4.2	75	65	86	11	0	1	0	190	353	266	0	6	0
10			137.5	4.9	4.2	76	66	84	12	0	1	0	200	357	252	0	6	0
	0.2	0.1	229.7	2.8	1	68	68	85	9	0	1	0	340	425	45	0	5	0
	0.4	0.1	242.5	2.8	1	69	69	84	9	0	1	0	343	422	45	0	5	0
	0.6	0.1	234.3	2.8	1	69	68	85	9	0	1	0	339	420	49	0	7	0
	0.8	0.1	220.1	2.8	1.1	68	68	85	9	0	1	0	330	424	55	0	6	0
	0.2	0.4	233.8	2.8	1	68	67	85	9	0	2	0	338	423	45	0	9	0
	0.4	0.4	310.4	2.8	1	68	67	84	9	0	3	0	333	423	50	0	9	0
	0.6	0.4	284	2.8	1.2	67	66	83	11	0	3	0	328	417	63	0	7	0
	0.8	0.4	328.1	2.8	1.4	67	64	86	11	0	2	0	301	415	88	0	11	0
	0.2	0.7	318.2	2.8	1	69	66	81	9	0	7	0	337	419	45	0	14	0
	0.4	0.7	367.3	2.8	1.1	67	64	82	12	0	5	0	324	412	65	0	14	0
	0.6	0.7	293.1	2.8	1.4	65	61	88	12	0	2	0	301	409	92	0	13	0
	0.8	0.7	332.2	2.8	1.8	67	60	91	10	0	2	0	284	395	118	0	18	0
	0.2	1	438.2	2.8	1	67	63	82	9	0	9	0	330	416	45	0	24	0
	0.4	1	304.6	2.8	1.1	69	67	84	10	0	2	0	335	413	60	0	7	0
	0.6	1	254	2.8	1.7	65	60	85	15	0	3	0	298	395	104	0	18	0
	0.8	1	375.5	2.8	2.3	67	54	74	16	0	19	0	266	361	127	0	61	0
	0.2	1.3	332.8	2.8	1	68	64	82	10	0	7	0	316	397	49	0	53	0
	0.4	1.3	298	2.8	1.3	68	61	77	12	0	13	0	286	361	77	0	91	0
	0.6	1.3	381.3	2.8	2	68	54	77	13	0	19	0	270	346	119	0	80	0
	0.8	1.3	372.1	2.8	2.4	67	55	80	13	0	15	0	267	361	155	0	32	0
	0.2	1.6	366.3	2.8	1.1	68	65	75	12	0	11	0	301	373	56	0	85	0
	0.4	1.6	502	2.8	1.7	68	52	55	16	0	40	0	264	331	93	0	127	0
	0.6	1.6	549.9	2.8	2.4	59	34	57	15	0	57	0	138	186	154	0	337	0
	0.8	1.6	632.9	2.8	3	49	25	31	25	0	82	0	113	154	197	0	351	0
	0.2	1.9	626.3	2.8	1.1	65	33	41	11	0	78	0	256	328	59	0	172	0
	0.4	1.9	537.8	2.8	2	68	42	59	14	0	48	0	234	299	107	0	175	0
	0.6	1.9	520.2	2.8	2.7	59	37	55	17	0	54	0	202	304	179	0	130	0
	0.8	1.9	330.2	2.8	3.5	63	46	72	30	0	15	0	200	280	253	0	82	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			574.4	31.4	4.8	138	120	29	13	1	0	0	413	198	194	5	5	0
10			1168	43.8	4.8	127	112	28	22	1	0	0	310	152	348	5	0	0
	0.2	0.1	214	21	1	121	120	39	2	1	1	0	603	191	10	5	6	0
	0.4	0.1	210.5	21	1	121	120	39	2	1	1	0	601	190	10	5	9	0
	0.6	0.1	222.9	22	1.1	125	123	36	2	1	1	0	617	178	10	5	5	0
	0.8	0.1	234.6	22	1.3	128	123	35	2	1	2	0	616	179	10	5	5	0
	0.2	0.4	180	21	1	121	120	39	2	1	1	0	604	191	10	5	5	0
	0.4	0.4	183.8	21	1	121	121	38	2	1	1	0	605	190	10	5	5	0
	0.6	0.4	196.7	22	1.3	126	123	36	2	1	1	0	616	180	10	5	4	0
	0.8	0.4	235.2	23.4	1.7	128	119	41	2	1	0	0	605	189	12	5	4	0
	0.2	0.7	197.4	21	1	121	121	38	2	1	1	0	605	190	10	5	5	0
	0.4	0.7	187.4	21	1.1	121	121	38	2	1	1	0	605	190	10	5	5	0
	0.6	0.7	234.9	22	1.6	132	125	32	3	1	2	0	613	180	14	5	3	0
	0.8	0.7	259.7	23.4	2.1	137	123	31	6	1	2	0	618	171	16	5	5	0
	0.2	1	180.5	21	1	121	121	38	2	1	1	0	605	190	10	5	5	0
	0.4	1	192.2	21	1.2	122	118	39	2	1	3	0	601	192	10	5	7	0
	0.6	1	303.1	23.4	1.8	135	126	31	4	1	1	0	618	175	11	5	6	0
	0.8	1	266.5	20.9	2.4	141	123	35	4	1	0	0	602	190	17	5	1	0
	0.2	1.3	182.4	21	1.1	121	121	38	2	1	1	0	605	190	10	5	5	0
	0.4	1.3	205.7	21	1.4	125	123	36	2	1	1	0	602	194	10	5	4	0
	0.6	1.3	261	23.4	2.2	139	123	35	3	1	1	0	618	172	17	5	3	0
	0.8	1.3	331	20.9	3	141	111	44	3	1	4	0	566	215	24	5	5	0
	0.2	1.6	181	21	1.1	121	120	39	2	1	1	0	604	191	10	5	5	0
	0.4	1.6	223.9	21	1.6	129	123	35	3	1	1	0	601	191	16	5	2	0
	0.6	1.6	302.1	20.9	2.6	142	119	35	7	1	1	0	583	195	27	5	5	0
	0.8	1.6	437.1	23.4	3.5	130	102	45	14	1	1	0	490	231	86	5	3	0
	0.2	1.9	178.7	21	1.2	121	120	39	2	1	1	0	601	193	12	5	4	0
	0.4	1.9	256.2	21	2	134	120	36	6	1	0	0	602	191	15	4	3	0
	0.6	1.9	391.2	21.5	3.1	142	108	46	8	0	1	0	524	232	48	4	7	0
	0.8	1.9	770.7	21	4	123	70	47	45	1	0	0	375	218	218	4	0	0

A.12 CodeLLaMA 13B

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			886.5	44.2	4.8	140	128	21	13	1	0	0	459	170	179	5	2	0
	0.2	0.1	237.1	34.8	1	134	134	26	2	1	0	0	670	130	10	5	0	0
	0.4	0.1	236.2	34.8	1	134	134	26	2	1	0	0	670	130	10	5	0	0
	0.6	0.1	242	34.8	1.1	136	135	26	1	1	0	0	673	132	5	5	0	0
	0.8	0.1	255.4	34.8	1.2	137	135	24	3	1	0	0	673	129	8	5	0	0
	0.2	0.4	242.2	34.8	1	134	134	26	2	1	0	0	670	130	10	5	0	0
	0.4	0.4	244.2	34.8	1	134	134	25	3	1	0	0	670	127	13	5	0	0
	0.6	0.4	256.7	34.8	1.3	138	133	28	1	1	0	0	668	137	5	5	0	0
	0.8	0.4	279	34.8	1.6	140	133	27	1	1	1	0	664	140	4	5	2	0
	0.2	0.7	239.5	34.8	1	134	134	26	2	1	0	0	670	130	10	5	0	0
	0.4	0.7	247.1	34.8	1.1	134	134	26	2	1	0	0	669	130	11	5	0	0
	0.6	0.7	255.5	34.8	1.5	143	137	24	1	1	0	0	671	132	6	5	1	0
	0.8	0.7	311.2	34.8	1.9	145	129	31	2	1	0	0	658	142	10	4	1	0
	0.2	1	249.8	34.8	1	134	134	26	2	1	0	0	670	131	9	5	0	0
	0.4	1	265.7	34.8	1.2	134	133	27	2	1	0	0	665	136	9	5	0	0
	0.6	1	289.7	34.8	1.7	144	133	28	0	1	1	0	673	130	3	5	4	0
	0.8	1	387.7	39.2	2.4	146	130	30	2	1	0	0	639	153	16	4	3	0
	0.2	1.3	245.2	34.8	1.1	134	134	25	3	1	0	0	670	129	11	5	0	0
	0.4	1.3	258.6	34.8	1.3	139	133	28	1	1	0	0	674	128	8	5	0	0
	0.6	1.3	331.8	34.8	2	143	135	25	1	1	1	0	645	154	6	5	5	0
	0.8	1.3	413.4	41.6	2.8	148	121	34	7	1	0	0	612	177	19	5	2	0
	0.2	1.6	251.7	34.8	1.1	134	134	26	2	1	0	0	667	133	10	5	0	0
	0.4	1.6	281.2	34.8	1.5	142	134	26	2	1	0	0	667	137	4	5	2	0
	0.6	1.6	357.9	34.8	2.5	147	127	31	3	1	1	0	634	159	13	5	4	0
	0.8	1.6	745.2	44	3.4	147	108	39	14	0	2	0	531	209	67	4	4	0
	0.2	1.9	308.5	34.8	1.1	134	134	27	1	1	0	0	668	136	6	5	0	0
	0.4	1.9	357.4	34.8	1.8	145	136	23	3	1	0	0	651	146	13	5	0	0
	0.6	1.9	585.7	41.6	2.8	147	113	42	6	1	1	0	591	184	32	4	4	0
	0.8	1.9	772.9	39.2	4	128	84	44	33	0	2	0	423	236	145	2	9	0

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			1298.1	37.1	4.8	69	29	14	119	1	0	0	105	72	634	4	0	0
10			1654	43.7	4.8	58	29	14	120	0	0	0	93	57	662	3	0	0
	0.2	0.1	685.7	25.3	1	68	68	27	67	1	0	0	340	135	335	5	0	0
	0.4	0.1	698.8	25.3	1	68	68	27	67	1	0	0	340	135	335	5	0	0
	0.6	0.1	742	26.5	1.4	70	66	24	72	1	0	0	334	123	353	5	0	0
	0.8	0.1	792.6	26.5	1.9	76	67	25	70	1	0	0	328	119	363	5	0	0
	0.2	0.4	695	25.3	1	68	68	27	67	1	0	0	340	135	335	5	0	0
	0.4	0.4	683.7	25.3	1.1	71	70	26	66	1	0	0	347	133	330	5	0	0
	0.6	0.4	781.1	26.5	2.2	80	62	26	74	1	0	0	326	129	357	3	0	0
	0.8	0.4	885.6	26.6	3.1	97	66	24	72	1	0	0	335	124	348	4	4	0
	0.2	0.7	798	25.3	1.1	69	69	26	67	1	0	0	342	132	336	5	0	0
	0.4	0.7	809.4	26.5	1.4	74	70	26	66	1	0	0	350	127	333	5	0	0
	0.6	0.7	991.8	26.5	3	89	63	31	68	0	1	0	322	133	353	4	3	0
	0.8	0.7	1103.7	26.5	3.7	116	74	23	65	0	1	0	357	147	304	3	4	0
	0.2	1	797.4	25.3	1.3	69	68	27	67	1	0	0	341	132	337	5	0	0
	0.4	1	847.6	26.5	2	79	68	30	65	0	0	0	345	137	331	2	0	0
	0.6	1	1054.4	26.5	3.5	102	70	26	66	1	0	0	339	129	343	3	1	0
	0.8	1	1258.5	26.5	4	125	71	28	62	1	1	0	359	152	298	2	4	0
	0.2	1.3	798.9	25.3	1.4	70	69	25	68	1	0	0	344	133	333	5	0	0
	0.4	1.3	860.8	26.5	2.4	83	65	26	71	0	1	0	345	132	334	3	1	0
	0.6	1.3	1107.8	26.5	3.8	114	60	31	71	1	0	0	352	156	301	4	2	0
	0.8	1.3	1235.2	26.5	4.2	131	67	33	63	0	0	0	337	174	302	1	1	0
	0.2	1.6	1340.2	25.3	1.6	74	70	24	68	1	0	0	354	132	325	4	0	0
	0.4	1.6	1625.6	26.6	2.9	88	67	26	70	0	0	0	270	196	344	3	2	0
	0.6	1.6	2071	26.5	4	122	67	37	57	1	1	0	346	175	285	4	5	0
	0.8	1.6	2378.8	26.5	4.5	127	64	40	58	0	1	0	311	193	300	4	7	0
	0.2	1.9	1333.5	25.3	1.8	77	71	22	69	1	0	0	359	125	326	5	0	0
	0.4	1.9	1698	26.6	3.3	105	68	29	64	1	1	0	347	150	308	3	7	0
	0.6	1.9	1253.6	26.6	4.3	131	66	39	57	1	0	0	344	177	289	2	3	0
	0.8	1.9	1303.7	26.5	4.6	111	55	39	69	0	0	0	260	184	367	2	2	0

A.14 CodeLLaMA Instruct 13B

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			1521.8	42.5	4.8	64	30	16	117	0	0	0	118	67	629	1	0	0
	0.2	0.1	975.5	43.5	1	62	62	21	80	0	0	0	310	105	400	0	0	0
	0.4	0.1	982.6	43.5	1	62	62	21	80	0	0	0	310	105	400	0	0	0
	0.6	0.1	1067.2	43.4	1.5	65	58	22	83	0	0	0	281	112	422	0	0	0
	0.8	0.1	1100.3	43.7	2	69	56	23	84	0	0	0	274	106	435	0	0	0
	0.2	0.4	1001.6	43.4	1	62	62	21	80	0	0	0	310	105	400	0	0	0
	0.4	0.4	976.8	43.5	1.2	63	63	20	80	0	0	0	308	100	407	0	0	0
	0.6	0.4	1119.3	43.4	2.4	73	56	20	87	0	0	0	278	94	443	0	0	0
	0.8	0.4	1216	43.6	3.5	92	58	16	89	0	0	0	290	88	436	1	0	0
	0.2	0.7	997.9	43.5	1	62	62	21	80	0	0	0	310	105	400	0	0	0
	0.4	0.7	973.1	43.5	1.6	63	60	21	82	0	0	0	297	103	415	0	0	0
	0.6	0.7	1196.2	43.4	3.3	84	59	19	84	1	0	0	282	100	430	2	1	0
	0.8	0.7	1282	43.4	4	117	63	18	82	0	0	0	344	85	385	0	1	0
	0.2	1	1028.1	43.5	1.2	63	62	21	80	0	0	0	309	105	401	0	0	0
	0.4	1	1035.3	43.4	2.2	64	59	18	86	0	0	0	288	101	426	0	0	0
	0.6	1	1257.1	43.4	3.7	99	65	26	72	0	0	0	316	97	400	1	1	0
	0.8	1	1418.8	43.3	4.2	125	72	26	64	1	0	0	326	132	353	3	1	0
	0.2	1.3	975.5	43.4	1.5	62	60	21	82	0	0	0	300	95	420	0	0	0
	0.4	1.3	1048.1	43.4	2.9	80	59	20	83	1	0	0	307	88	418	2	0	0
	0.6	1.3	1266.2	43.4	4.1	116	65	21	77	0	0	0	330	102	380	3	0	0
	0.8	1.3	1312.6	43.3	4.3	132	76	31	56	0	0	0	361	144	308	2	0	0
	0.2	1.6	965.1	43.5	1.7	63	59	21	83	0	0	0	291	101	423	0	0	0
	0.4	1.6	1131.8	43.4	3.5	99	62	12	89	0	0	0	326	80	405	3	1	0
	0.6	1.6	1253.6	43.3	4.1	125	71	25	65	0	2	0	348	129	334	1	3	0
	0.8	1.6	1350.8	43.3	4.5	132	70	25	67	1	0	0	344	157	307	4	3	0
	0.2	1.9	1156.2	43.5	2.1	66	59	20	84	0	0	0	291	105	419	0	0	0
	0.4	1.9	1318.3	43.3	3.9	110	65	23	75	0	0	0	334	108	372	1	0	0
	0.6	1.9	1326.4	43.3	4.3	130	67	28	68	0	0	0	337	160	316	0	2	0
	0.8	1.9	1407.2	43.3	4.6	125	66	32	65	0	0	0	289	169	355	0	2	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			693.9	44.2	3.9	1	1	0	162	0	0	0	5	0	810	0	0	0
10			1643.8	44.1	4.1	1	1	0	162	0	0	0	5	0	810	0	0	0
	0.2	0.1	574.1	43.8	1	1	1	4	158	0	0	0	5	20	790	0	0	0
	0.4	0.1	561.2	42.5	1	1	1	4	158	0	0	0	5	20	790	0	0	0
	0.6	0.1	550.5	43.8	1.3	1	1	3	159	0	0	0	5	18	792	0	0	0
	0.8	0.1	563.6	43.6	1.9	1	1	4	158	0	0	0	5	19	791	0	0	0
	0.2	0.4	558.1	43	1	1	1	4	158	0	0	0	5	20	790	0	0	0
	0.4	0.4	585.2	44.1	1.3	1	1	3	159	0	0	0	5	15	795	0	0	0
	0.6	0.4	573.5	44	3.4	1	1	5	157	0	0	0	5	13	797	0	0	0
	0.8	0.4	595.8	44	4.3	2	1	5	157	0	0	0	5	15	795	0	0	0
	0.2	0.7	573.6	44.3	1.4	1	1	4	158	0	0	0	5	20	790	0	0	0
	0.4	0.7	625.4	43.2	3.5	1	1	2	160	0	0	0	5	8	802	0	0	0
	0.6	0.7	583.9	44	4.6	1	1	1	161	0	0	0	5	4	806	0	0	0
	0.8	0.7	613.2	44	4.9	2	1	3	159	0	0	0	6	5	804	0	0	0
	0.2	1	605	44	2.5	1	1	3	159	0	0	0	5	12	798	0	0	0
	0.4	1	632.4	44.1	4.4	2	1	0	162	0	0	0	6	5	804	0	0	0
	0.6	1	656.9	41.7	5	1	1	3	159	0	0	0	4	13	798	0	0	0
	0.8	1	506.1	44.1	5	2	1	1	161	0	0	0	5	11	799	0	0	0
	0.2	1.3	626.3	43.3	3.6	1	1	1	161	0	0	0	5	5	805	0	0	0
	0.4	1.3	618.7	44.3	4.8	3	2	1	160	0	0	0	7	7	801	0	0	0
	0.6	1.3	651.8	43.9	5	1	1	3	159	0	0	0	5	20	790	0	0	0
	0.8	1.3	443.9	43.9	5	1	1	5	157	0	0	0	5	17	792	0	1	0
	0.2	1.6	569.4	42.3	4.2	2	1	0	162	0	0	0	6	4	805	0	0	0
	0.4	1.6	601.8	43.3	5	2	1	4	158	0	0	0	6	14	795	0	0	0
	0.6	1.6	558.9	43.4	5	1	1	3	159	0	0	0	3	21	791	0	0	0
	0.8	1.6	397.4	44.1	5	1	1	2	160	0	0	0	4	18	793	0	0	0
	0.2	1.9	634.7	44	4.7	2	1	2	160	0	0	0	6	10	799	0	0	0
	0.4	1.9	557.2	42.7	5	2	1	6	155	1	0	0	6	17	791	1	0	0
	0.6	1.9	449.5	43.5	5	2	1	2	160	0	0	0	5	13	797	0	0	0
	0.8	1.9	340.2	44.1	5	1	1	1	161	0	0	0	5	17	793	0	0	0

A.16 CodeGen2 3.7B

Beam Size	Top p	Temp	Mask Time	VRAM	N=5 Res	AnySuc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			694	23.1	3.6	5	5	21	137	0	0	0	12	112	691	0	0	0
10			1421.4	43.8	3.9	7	5	22	136	0	0	0	15	103	697	0	0	0
	0.2	0.1	462.7	23.2	1	6	6	65	91	0	1	0	30	325	455	0	5	0
	0.4	0.1	460.3	23.2	1	6	6	65	91	0	1	0	30	325	455	0	5	0
	0.6	0.1	457.6	25.2	1.3	6	6	65	91	0	1	0	30	323	457	0	5	0
	0.8	0.1	448.8	25.2	1.8	7	7	62	93	0	1	0	32	315	463	0	5	0
	0.2	0.4	466.5	23.2	1.1	6	6	66	90	0	1	0	30	328	452	0	5	0
	0.4	0.4	455.5	23.2	1.7	6	6	65	91	0	1	0	29	320	462	0	4	0
	0.6	0.4	452.7	25.2	3	6	5	63	94	0	1	0	25	318	467	0	5	0
	0.8	0.4	401.4	25.2	3.8	6	6	64	92	0	1	0	24	312	476	0	3	0
	0.2	0.7	474.5	23.2	1.7	6	6	63	93	0	1	0	30	315	465	0	5	0
	0.4	0.7	437.2	32.1	3.1	8	8	65	89	0	1	0	34	319	458	0	4	0
	0.6	0.7	487	22.1	4.1	7	6	59	97	0	1	0	24	299	488	0	4	0
	0.8	0.7	443.8	25.4	4.6	6	5	57	99	0	2	0	21	272	516	0	6	0
	0.2	1	470.5	32.1	2.4	6	6	65	91	0	1	0	30	316	465	0	4	0
	0.4	1	474.4	25.4	3.9	7	6	58	98	0	1	0	29	308	473	0	5	0
	0.6	1	446.8	33.5	4.7	7	5	51	106	0	1	0	23	266	521	0	5	0
	0.8	1	451.9	30.8	4.9	6	3	52	108	0	0	0	16	247	548	0	4	0
	0.2	1.3	461.6	26.1	3.1	7	6	64	92	0	1	0	31	308	473	0	3	0
	0.4	1.3	490.5	25.4	4.4	9	8	54	100	0	1	0	31	268	513	0	3	0
	0.6	1.3	437.7	22.1	4.9	7	4	51	107	0	1	0	21	251	538	0	5	0
	0.8	1.3	432.9	28.8	4.9	9	6	40	117	0	0	0	21	192	599	0	3	0
	0.2	1.6	508.1	27.4	3.8	7	7	61	94	0	1	0	27	311	472	0	5	0
	0.4	1.6	441.5	34.4	4.6	7	5	48	109	0	1	0	26	257	529	0	3	0
	0.6	1.6	440.7	25.4	4.9	8	5	46	111	0	1	0	22	230	557	0	6	0
	0.8	1.6	400.6	30.8	5	9	4	36	123	0	0	0	22	185	604	0	4	0
	0.2	1.9	410.6	25.4	4.1	8	7	62	93	0	1	0	27	306	478	0	4	0
	0.4	1.9	434.7	28.3	4.9	7	4	46	111	0	2	0	21	237	550	0	7	0
	0.6	1.9	392.5	22.7	5	9	6	34	122	0	1	0	22	198	592	0	3	0
	0.8	1.9	426.7	24.7	5	9	3	33	127	0	0	0	16	155	639	0	5	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			1699.3	37.6	3.5	47	40	27	95	1	0	0	122	129	559	2	3	0
10			3456.8	43.7	3.6	46	41	29	92	1	0	0	122	133	556	2	2	0
	0.2	0.1	1529	38.3	1	42	42	33	87	1	0	0	210	165	435	5	0	0
	0.4	0.1	1535.7	38.3	1	42	42	33	87	1	0	0	210	165	435	5	0	0
	0.6	0.1	1509.4	38.3	1.3	42	42	35	84	1	1	0	209	172	427	5	2	0
	0.8	0.1	1494	38.3	1.7	43	43	34	85	1	0	0	210	164	434	5	2	0
	0.2	0.4	1537.8	38.3	1	42	42	33	87	1	0	0	210	165	435	5	0	0
	0.4	0.4	1503.5	38.3	1.4	42	42	34	85	1	1	0	210	179	419	5	2	0
	0.6	0.4	1510.7	41.2	2.4	46	44	40	78	1	0	0	210	181	418	5	1	0
	0.8	0.4	1607.5	41.2	3.1	43	38	42	82	1	0	0	175	177	457	5	1	0
	0.2	0.7	1565.7	38.3	1.2	42	42	34	86	1	0	0	210	172	428	5	0	0
	0.4	0.7	1529.1	41.2	2.2	42	42	35	85	1	0	0	210	178	422	5	0	0
	0.6	0.7	1561.6	41.2	3.2	40	36	39	87	1	0	0	182	180	447	5	1	0
	0.8	0.7	1469.7	39.8	3.6	44	39	45	77	1	1	0	179	185	443	5	3	0
	0.2	1	1544.1	38.3	1.7	42	42	33	87	1	0	0	210	163	434	5	3	0
	0.4	1	1526.5	40.3	2.8	43	43	35	84	1	0	0	206	187	416	5	1	0
	0.6	1	1557.7	34.7	3.5	40	38	43	81	1	0	0	183	189	435	5	3	0
	0.8	1	1538.5	38.9	4	45	38	41	83	1	0	0	172	172	466	3	2	0
	0.2	1.3	1517.7	38.3	2.1	42	42	38	82	1	0	0	209	184	417	5	0	0
	0.4	1.3	1495.2	41.2	3.3	43	38	42	82	1	0	0	198	188	423	5	1	0
	0.6	1.3	1414.8	41.7	3.9	42	37	51	74	1	0	0	166	192	450	5	2	0
	0.8	1.3	1379.7	41.9	4.2	46	35	46	80	1	1	0	146	175	488	2	4	0
	0.2	1.6	1534.7	41.2	2.5	43	42	35	84	1	1	0	210	179	419	5	2	0
	0.4	1.6	1526.6	38	3.5	42	39	42	81	1	0	0	190	187	429	5	4	0
	0.6	1.6	1503.8	34.4	4.2	47	33	39	90	1	0	0	145	167	499	3	1	0
	0.8	1.6	1367.4	37	4.4	41	32	40	90	1	0	0	128	159	521	3	4	0
	0.2	1.9	1522.4	41	2.8	44	42	35	85	1	0	0	212	175	422	5	1	0
	0.4	1.9	1329.9	38.7	3.7	44	37	44	81	1	0	0	184	178	447	5	1	0
	0.6	1.9	1425.7	34.4	4.3	41	31	44	85	1	2	0	135	177	496	3	4	0
	0.8	1.9	1506.7	37	4.7	34	26	39	96	1	1	0	94	145	572	2	2	0

Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	N=1						Total					
							Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
	0.2	0.1	1384.5	44.3	1	116	116	44	2	1	0	0	580	220	10	5	0	0
	0.4	0.1	1388.8	44.3	1	116	116	44	2	1	0	0	580	220	10	5	0	0
	0.6	0.1	1392.7	44.3	1.1	116	116	44	2	1	0	0	573	218	19	5	0	0
	0.8	0.1	1391.4	44.3	1.2	115	114	47	1	1	0	0	561	216	33	5	0	0
	0.2	0.4	1388.4	44.3	1	116	116	44	2	1	0	0	580	220	10	5	0	0
	0.4	0.4	1406.5	44.3	1	116	116	44	2	1	0	0	580	220	10	5	0	0
	0.6	0.4	1490.9	44.3	1.3	117	116	44	2	1	0	0	559	211	40	5	0	0
	0.8	0.4	1509.1	44.3	1.6	117	115	45	2	1	0	0	542	177	91	5	0	0
	0.2	0.7	1383	44.3	1	116	116	44	2	1	0	0	580	220	10	5	0	0
	0.4	0.7	1387.8	44.3	1	116	116	44	2	1	0	0	580	213	17	5	0	0
	0.6	0.7	1504.4	44.3	1.5	118	115	46	1	1	0	0	547	176	87	5	0	0
	0.8	0.7	1429.5	44.3	1.9	126	124	38	0	1	0	0	542	126	144	3	0	0
	0.2	1	1406.4	44.3	1	116	116	44	2	1	0	0	580	220	10	5	0	0
	0.4	1	1438.3	44.3	1.2	117	115	45	2	1	0	0	577	199	34	5	0	0
	0.6	1	1468.1	44.3	1.8	119	118	44	0	1	0	0	537	146	127	5	0	0
	0.8	1	1454.2	44.3	2.4	115	113	47	2	1	0	0	460	128	223	4	0	0
	0.2	1.3	1475.4	44.3	1	116	116	44	2	1	0	0	579	217	14	5	0	0
	0.4	1.3	1443.6	44.3	1.3	120	120	41	1	1	0	0	579	181	50	5	0	0
	0.6	1.3	1434.3	44.3	2.1	117	115	46	1	1	0	0	503	148	159	5	0	0
	0.8	1.3	1535	44.3	2.9	115	105	55	2	1	0	0	437	126	251	1	0	0
	0.2	1.6	1483	44.3	1.1	116	116	44	2	1	0	0	580	214	16	5	0	0
	0.4	1.6	1433.8	44.3	1.6	116	115	47	0	1	0	0	549	177	84	5	0	0
	0.6	1.6	1593.6	44.3	2.6	111	108	51	3	1	0	0	449	126	235	5	0	0
	0.8	1.6	1736.9	44.3	3.3	107	105	54	3	1	0	0	375	106	333	1	0	0
	0.2	1.9	1506.5	44.3	1.2	118	117	43	2	1	0	0	581	195	34	5	0	0
	0.4	1.9	1375.8	44.3	1.8	116	112	46	4	1	0	0	527	151	132	5	0	0
	0.6	1.9	1657.7	44.3	3	109	99	60	3	1	0	0	399	132	282	2	0	0
	0.8	1.9	2195.6	44.3	3.8	92	84	58	20	1	0	0	300	112	401	2	0	0

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			187.5	6.2	3.9	74	63	60	39	1	0	0	196	221	343	3	0	52
10			229.1	9.5	3.9	70	62	60	41	0	0	0	191	196	376	0	0	52
	0.2	0.1	158.3	5.7	1	52	52	70	41	0	0	0	241	327	201	0	0	46
	0.4	0.1	155.2	5.7	1	52	52	70	41	0	0	0	241	327	201	0	0	46
	0.6	0.1	166.3	5.7	1.3	52	51	69	42	1	0	0	237	323	204	5	0	46
	0.8	0.1	157.1	5.7	1.6	55	55	69	39	0	0	0	241	302	232	0	0	40
	0.2	0.4	156.3	5.7	1	52	52	70	41	0	0	0	241	327	201	0	0	46
	0.4	0.4	158.7	5.7	1.2	54	54	70	39	0	0	0	244	323	210	0	0	38
	0.6	0.4	168.4	5.7	2	56	51	70	42	0	0	0	231	277	268	0	0	39
	0.8	0.4	168.4	5.7	2.7	59	54	77	32	0	0	0	228	272	277	1	0	37
	0.2	0.7	155.4	5.7	1.1	54	54	70	39	0	0	0	248	322	199	0	0	46
	0.4	0.7	162.1	5.7	1.8	58	54	69	40	0	0	0	244	288	237	0	0	46
	0.6	0.7	156.1	5.7	2.8	61	52	76	34	1	0	0	229	255	285	1	0	45
	0.8	0.7	174.9	5.7	3.3	65	55	64	43	1	0	0	233	217	321	1	0	43
	0.2	1	164.7	5.7	1.3	53	53	67	43	0	0	0	238	305	234	0	0	38
	0.4	1	163.5	5.7	2.4	58	50	70	43	0	0	0	234	270	284	0	0	27
	0.6	1	177.9	5.7	3.4	63	51	67	45	0	0	0	214	207	345	1	0	48
	0.8	1	201.1	5.7	3.8	65	48	68	46	1	0	0	207	216	363	1	0	28
	0.2	1.3	163.5	5.7	1.7	56	53	67	43	0	0	0	239	282	254	0	0	40
	0.4	1.3	169.4	5.7	3	64	52	69	40	0	2	0	234	229	311	1	2	38
	0.6	1.3	193.5	5.7	3.7	67	52	67	42	0	2	0	210	213	344	1	3	44
	0.8	1.3	219.5	5.7	4.1	56	34	63	63	1	2	0	154	187	431	1	3	39
	0.2	1.6	162.1	5.7	2	56	52	70	41	0	0	0	232	276	261	0	3	43
	0.4	1.6	182.5	5.7	3.5	64	52	72	38	1	0	0	220	231	325	2	0	37
	0.6	1.6	240.1	5.7	4	61	46	58	55	1	3	0	175	165	420	1	7	47
	0.8	1.6	507.1	5.8	4.4	47	32	42	87	1	1	0	113	122	535	2	2	41
	0.2	1.9	163.2	5.7	2.6	59	54	65	44	0	0	0	235	234	310	1	0	35
	0.4	1.9	197.6	5.7	3.8	65	51	65	42	1	4	0	203	198	364	2	5	43
	0.6	1.9	221.7	5.7	4.3	55	44	46	72	0	1	0	134	129	515	0	2	35
	0.8	1.9	302	5.7	4.6	34	26	33	103	0	1	0	70	95	615	0	1	34

							N=1						Total					
Beam Size	Top p	Temp	Mask Time	VRAM	UniqRes	N=5 Suc	Suc	TFail	CFail	TCFail	TTo	NoRes	Suc	TFail	CFail	TCFail	TTo	NoRes
5			170.4	21.5	3.9	82	70	66	26	1	0	0	217	266	279	4	0	49
10			220.1	30.5	4	80	69	61	32	1	0	0	212	251	299	4	0	49
	0.2	0.1	142.6	18.1	1	63	63	69	30	1	0	0	297	323	146	5	0	44
	0.4	0.1	141.7	18.1	1	63	63	69	30	1	0	0	297	323	146	5	0	44
	0.6	0.1	136.7	18.1	1.1	65	64	67	31	1	0	0	299	309	155	5	0	47
	0.8	0.1	151.9	18.1	1.4	68	63	65	33	1	1	0	295	280	177	5	7	51
	0.2	0.4	151.1	18.1	1	63	63	69	30	1	0	0	297	323	146	5	0	44
	0.4	0.4	149.4	18.1	1.1	65	65	69	28	1	0	0	298	316	148	5	0	48
	0.6	0.4	134.3	18.1	1.7	73	67	69	26	1	0	0	297	286	182	5	0	45
	0.8	0.4	139.6	18.1	2.4	75	64	66	32	1	0	0	272	284	210	5	0	44
	0.2	0.7	142	18.1	1	63	63	69	30	1	0	0	297	323	146	5	0	44
	0.4	0.7	151.1	18.1	1.6	71	66	71	25	1	0	0	301	300	165	5	0	44
	0.6	0.7	140.7	18.1	2.3	78	63	70	29	1	0	0	277	261	218	5	0	54
	0.8	0.7	160	18.1	3	80	63	71	28	1	0	0	270	255	244	5	0	41
	0.2	1	160.7	18.1	1.2	63	62	69	31	1	0	0	293	311	158	5	0	48
	0.4	1	147.7	18.1	2	69	61	72	29	1	0	0	275	262	222	5	0	51
	0.6	1	162.7	18.1	3.1	78	57	73	32	1	0	0	256	254	254	5	0	46
	0.8	1	197.5	18.1	3.6	80	52	69	42	0	0	0	239	244	288	3	0	41
	0.2	1.3	144.3	18.1	1.4	66	65	71	26	1	0	0	288	313	163	5	0	46
	0.4	1.3	146.8	18.1	2.6	76	66	72	24	1	0	0	269	269	233	5	0	39
	0.6	1.3	176.3	18.1	3.4	81	58	70	34	1	0	0	256	239	277	4	0	39
	0.8	1.3	227.9	18.1	4	77	50	63	49	1	0	0	209	213	361	2	0	30
	0.2	1.6	143.2	18.1	1.8	71	63	69	30	1	0	0	284	292	189	5	0	45
	0.4	1.6	175.4	18.1	3.1	81	63	71	28	1	0	0	259	246	269	5	0	36
	0.6	1.6	201.3	19	3.9	78	61	57	44	0	1	0	226	202	349	1	2	35
	0.8	1.6	301.5	18.1	4.3	68	40	60	62	1	0	0	154	148	465	1	5	42
	0.2	1.9	250.9	18.1	2.2	72	62	70	28	1	2	0	273	287	201	5	8	41
	0.4	1.9	191.9	18.1	3.5	73	56	78	28	1	0	0	230	248	290	5	0	42
	0.6	1.9	322.4	17.9	4.2	71	46	57	58	1	1	0	188	175	416	2	2	32
	0.8	1.9	374	18.8	4.6	50	29	34	97	0	3	0	105	108	571	0	3	28

Bibliography

[1] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. "Automatic Software Repair: A Survey". In: *IEEE Transactions on Software Engineering* (2019). DOI: 10.1109/TSE.2017.2755013.

[2] Herb Krasner. *Cost of Poor Software Quality in the U.S.: A 2022 Report*. 2022.

[3] Tom Britton, Lisa Jeng, Graham Carver, Tomer Katzenellenbogen, and Paul Cheak. *Reversible Debugging Software 'Quantify the time and cost saved using reversible debuggers"*. 2020.

[4] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. *A Survey of Learning-based Automated Program Repair*. 2023. DOI: 10.48550/arXiv.2301.03270.

[5] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. *A Syntax-Guided Edit Decoder for Neural Program Repair*. 2022. DOI: 10.48550/arXiv.2106.08253.

[6] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs". In: *IIEEE Trans. Software Eng.* (2017). DOI: 10.1109/TSE.2016.2560811.

[7] Thomas Durieux and Martin Monperrus. "DynaMoth: dynamic code synthesis for automatic program repair". In: *Proceedings of the 11th International Workshop on Automation of Software Test*. ACM, 2016. DOI: 10.1145/2896921.2896931.

[8] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. "TBar: Revisiting Template-based Automated Program Repair". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019. DOI: 10.1145/3293882.3330577.

[9] Yuan Yuan and Wolfgang Banzhaf. "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming". In: *IIEEE Trans. Software Eng.* (2020). DOI: 10.1109/TSE.2018.2874648.

[10] Dongcheng Li, W. Eric Wong, Mingyong Jian, Yi Geng, and Matthew Chau. "Improving Search-Based Automatic Program Repair With Neural Machine Translation". In: *IEEE Access* (2022). DOI: 10.1109/ACCESS.2022.3164780.

[11] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair". In: *IIEEE Trans. Software Eng.* (2019). DOI: 10.1109/TSE.2019.2940179.

[12] Chunqiu Steven Xia and Lingming Zhang. *Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning*. 2022. DOI: 10.48550/arXiv.2207.08281.

[13] Quanjun Zhang, Chunrong Fang, Tongke Zhang, Bowen Yu, Weisong Sun, and Zhenyu Chen. *GAMMA: Revisiting Template-based Automated Program Repair via Mask Prediction*. 2023. DOI: 10.48550/arXiv.2309.09308.

[14] Lei Huang et al. *A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions*. 2023. DOI: 10.48550/arXiv.2311.05232.

[15] Joseph Spracklen, Raveen Wijewickrama, A. H. M. Nazmus Sakib, Anindya Maiti, and Murtuza Jadliwala. *We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs*. 2024. DOI: 10.48550/arXiv.2406.10279.

[16] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. *Exploring and Evaluating Hallucinations in LLM-Powered Code Generation*. 2024.

[17] Changshu Liu, Pelin Cetin, Yogesh Patodia, Saikat Chakraborty, Yangruibo Ding, and Baishakhi Ray. *Automated Code Editing with Search-Generate-Modify*. 2023. DOI: 10.48550/arXiv.2306.06490.

[18] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. *CIRCLE: Continual Repair across Programming Languages*. 2022. DOI: 10.48550/arXiv.2205.10956.

[19] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. "GenProg: A Generic Method for Automatic Software Repair". In: *IEEE Transactions on Software Engineering* (2012). DOI: 10.1109/TSE.2011.104.

[20] Nan Jiang, Thibaud Lutellier, and Lin Tan. "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021. DOI: 10.1109/ICSE43902.2021.00107.

[21] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. *KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair*. 2023. DOI: 10.48550/arXiv.2302.01857.

[22] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2015. DOI: 10.1007/978-3-662-44874-8.

[23] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. "The plastic surgery hypothesis". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014. DOI: 10.1145/2635868.2635898.

[24] Matias Martinez, Westley Weimer, and Martin Monperrus. "Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. 2014. DOI: 10.1145/2591062.2591114.

[25] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015. DOI: 10.1145/2771783.2771791.

[26] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. "The Strength of Random Search on Automated Program Repair". In: (2014).

[27] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset". In: *Empir Software Eng* (2017). DOI: 10.1007/s10664-016-9470-4.

[28] Matias Martinez and Martin Monperrus. "ASTOR: a program repair library for Java (demo)". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016. DOI: 10.1145/2931037.2948705.

[29] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: a database of existing faults to enable controlled testing studies for Java programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014. DOI: 10.1145/2610384.2628055.

[30] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Trans. Evol. Computat.* (2002). DOI: 10.1109/4235.996017.

[31] Shanu Verma, Millie Pant, and Vaclav Snasel. "A Comprehensive Review on NSGA-II for Multi-Objective Combinatorial Optimization Problems". In: *IEEE Access* (2021). DOI: 10.1109/ACCESS.2021.3070634.

[32] Günter Rudolph. "Evolutionary search for minimal elements in partially ordered finite sets". In: *Evolutionary Programming VII*. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998. DOI: 10.1007/BFb0040787.

[33] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. "Comparison of Multiobjective Evolutionary Algorithms: Empirical Results". In: *Evolutionary Computation* (2000). DOI: 10.1162/106365600568202.

[34] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. *On Learning Meaningful Code Changes via Neural Machine Translation*. 2019. DOI: 10.48550/arXiv.1901.09102.

[35] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. "DeepFix: Fixing Common C Language Errors by Deep Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* (2017). Number: 1. DOI: 10.1609/aaai.v31i1.10742.

[36] Saikat Chakraborty and Baishakhi Ray. *On Multi-Modal Learning of Editing Source Code*. 2021. DOI: 10.48550/arXiv.2108.06645.

[37] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. *SeqTrans: Automatic Vulnerability Fix via Sequence to Sequence Learning*. 2022. DOI: 10.48550/arXiv.2010.10805.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017. DOI: 10.48550/arXiv.1706.03762.

[39] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. *An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation*. 2019. DOI: 10.48550/arXiv.1812.08693.

[40] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. DOI: 10.48550/arXiv.2109.00859.

[41] Raymond Li et al. *StarCoder: may the source be with you!* 2023. DOI: 10.48550/arXiv.2305.06161.

[42] Loubna Ben Allal et al. *SantaCoder: don't reach for the stars!* 2023. DOI: 10.48550/arXiv.2301.03988.

[43] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. DOI: 10.48550/arXiv.2002.08155.

[44] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. *Unified Pre-training for Program Understanding and Generation*. 2021. DOI: 10.48550/arXiv.2103.06333.

[45] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. *CodeGen2: Lessons for Training LLMs on Programming and Natural Languages*. 2023. DOI: 10.48550/arXiv.2305.02309.

[46] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. *InCoder: A Generative Model for Code Infilling and Synthesis*. 2023. DOI: 10.48550/arXiv.2204.05999.

[47] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. *UniXcoder: Unified Cross-Modal Pre-training for Code Representation*. 2022. DOI: 10.48550/arXiv.2203.03850.

[48] Qinkai Zheng et al. "CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X". In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 2023. DOI: 10.1145/3580305.3599790.

[49] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F. Bissyandé. "Where were the repair ingredients for Defects4j bugs?" In: *Empir Software Eng* (2021). DOI: 10.1007/s10664-021-10003-7.

[50] Kui Liu, Jingtang Zhang, Li Li, Anil Koyuncu, Dongsun Kim, Chunpeng Ge, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. "Reliable Fix Patterns Inferred from Static Checkers for Automated Program Repair". In: *ACM Trans. Softw. Eng. Methodol.* (2023). DOI: 10.1145/3579637.

[51] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. "Quality of Automated Program Repair on Real-World Defects". In: *IEEE Transactions on Software Engineering* (2022). DOI: 10.1109/TSE.2020.2998785.

[52] Xinwu Yang, Guizeng You, Chong Zhao, Mengfei Dou, and Xinian Guo. *An Improved multi-objective genetic algorithm based on orthogonal design and adaptive clustering pruning strategy*. 2019. DOI: 10.48550/arXiv.1901.00577.

[53] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. "Impact of Code Language Models on Automated Program Repair". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023). DOI: 10.1109/ICSE48619.2023.00125.

[54] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. DOI: 10.48550/arXiv.2203.13474.

[55] Klimov and Vakhreev. *Applying All Recent Innovations To Train a Code Model*. 2023.

[56] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. DOI: 10.48550/arXiv.2308.12950.

[57] Rui Xie, Zhengran Zeng, Zhuohao Yu, Chang Gao, Shikun Zhang, and Wei Ye. *CodeShell Technical Report*. 2024. DOI: 10.48550/arXiv.2403.15747.

[58] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. *Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code*. 2023. DOI: 10.48550/arXiv.2311.07989.

[59] Sid Black et al. *GPT-NeoX-20B: An Open-Source Autoregressive Language Model*. 2022. DOI: 10.48550/arXiv.2204.06745.

[60] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. DOI: 10.48550/arXiv.2107.03374.

[61] Yuan Yuan and Wolfgang Banzhaf. "A hybrid evolutionary system for automatic software repair". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 2019. DOI: 10.1145/3321707.3321830.

[62] Wenkang Zhong, Chuanyi Li, Kui Liu, Tongtong Xu, Tegawendé F. Bissyandé, Jidong Ge, Bin Luo, and Vincent Ng. *Practical Program Repair via Preference-based Ensemble Strategy*. 2023. DOI: 10.1145/3597503.3623310.

[63] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. "GZoltar: an eclipse plug-in for testing and debugging". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012. DOI: 10.1145/2351676.2351752.