Integrating Large Language Models and Reinforcement Learning for Non-linear Reasoning

YOAV ALON, University of Bristol, United Kingdom
CRISTINA DAVID, University of Bristol, United Kingdom

Large Language Models (LLMs) were shown to struggle with long-term planning, which may be caused by the limited way in which they explore the space of possible solutions. We propose an architecture where a Reinforcement Learning (RL) Agent guides an LLM's space exploration: (1) the Agent has access to domain-specific information, and can therefore make decisions about the quality of candidate solutions based on specific and relevant metrics, which were not explicitly considered by the LLM's training objective; (2) the LLM can focus on generating immediate next steps, without the need for long-term planning. We allow non-linear reasoning by exploring alternative paths and backtracking. We evaluate this architecture on the program equivalence task, and compare it against Chain of Thought (CoT) and Tree of Thoughts (ToT). We assess both the downstream task, denoting the binary classification, and the intermediate reasoning steps. Our approach compares positively against CoT and ToT.

CCS Concepts: • **Software and its engineering → Software functional properties**; • **Computing methodologies → Machine learning approaches**.

Additional Key Words and Phrases: LLM, Graph Neural Networks, Reinforcement Learning, Code Generation

ACM Reference Format:
Yoav Alon and Cristina David. 2025. Integrating Large Language Models and Reinforcement Learning for Non-linear Reasoning. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE044 (July 2025), 21 pages. https://doi.org/10.1145/3715761

1 Introduction

Large Language Models (LLMs) have recently been successfully applied to a wide range of tasks, including code related tasks [8, 11, 29, 32, 49]. When looking at the search strategies they employ, LLMs are still mostly restricted to linear search during inference. For instance, the popular Chain of Thought (CoT) approach [46] attempts to break the reasoning process into smaller steps, denoted by intermediate thoughts of the LLM, thus enabling the LLM to perform logical reasoning. Although multiple thought candidates corresponding to different paths through the search space exist at each step, the LLM always commits to one option, linearly building a solution. Several works have shown that this space exploration strategy may lead to difficulties with long-term proof planning [5, 39]. In particular, when multiple valid deduction steps are available, LLMs are unable to systematically explore the different options.

Given the very large search space exhibited by many problems, exploration of different paths and backtracking may be beneficial. This problem was recently addressed by an approach coined Tree of Thoughts (ToT) [47], which allows LLMs to make decisions by considering multiple different

Authors' Contact Information: Yoav Alon, University of Bristol, Bristol, United Kingdom, jt20948@bristol.ac.uk; Cristina David, University of Bristol, Bristol, United Kingdom, cristina.david@bristol.ac.uk.

ACM 2994-970X/2025/7-ARTFSE044
https://doi.org/10.1145/3715761

reasoning paths and self-evaluating choices to decide the next action. Although ToT was shown to enhance LLM's problem-solving abilities for some tasks, it was not investigated in the context of code-related tasks. Especially for code generation, the LLM's ability to self-evaluate is particularly challenging due to the mismatch between training and inference metrics, which was highlighted by several works [7, 25]. While models are trained using a next-token prediction objective, which maximizes the likelihood of the next ground-truth token, the code generated at inference is usually evaluated on its ability to compile and pass available unit tests.

An objective of this paper is to examine the reasoning abilities of CoT and ToT for a code-related task (involving code generation). This poses a challenge, as evaluating the accuracy of intermediate reasoning steps is inherently difficult. Most prior studies assess reasoning abilities indirectly by measuring performance on downstream tasks, such as mathematical problem-solving [9, 13]. The only work we are aware of that explicitly considers intermediate reasoning steps uses a natural language question-answering task, where examples are derived from a synthetic world model represented in first-order logic [39]. However, it is unclear how to apply this approach to code-related tasks.

Instead, we focus on the well-established problem of answering *program equivalence* queries, long studied in programming languages (via symbolic reasoning) [12, 16, 24, 35]: given programs A and B, we ask whether they are semantically equivalent, meaning that they have the same I/O behaviour. The reasoning required to prove equivalence typically involves identifying a sequence of semantics-preserving transformations that convert program A into program B. This can be represented as a series of mutated programs $A, A_1, \ldots, A_n$, where each such program A_i preserves the behavior of the source program A, and shows increasing syntactical similarity to the target B compared to the previous step A_{i-1}, until finally reaching B itself. For illustration, Figure 1 provides two equivalent programs A and B (both solutions of a Codeforces problem), and Figure 2 provides the sequence of semantics preserving transformations leading from A to B.

Program equivalence is a good candidate for evaluating intermediate reasoning steps because, as we frame it, these steps involve code generation, and their accuracy can be evaluated in a clear, structured manner. Specifically, we assess the accuracy of intermediate steps across three dimensions: syntactical correctness, functional correctness (preserving the behavior of the source program A), and syntactical similarity to the target program B. When programs A and B are equivalent, we know that the final program in the sequence of transformations should match B.

In addition to assessing the reasoning processes of CoT and ToT, we introduce and evaluate a novel architecture that expands upon the non-linear solution exploration concept of ToT. However, instead of relying on the LLM for evaluating candidate reasoning steps, this task is delegated to a Reinforcement Learning (RL) Agent equipped with domain-specific knowledge. This approach aims to overcome the limitations observed in LLM reasoning with CoT and ToT by placing the RL Agent in charge of the critical task of proof planning.

Similar to ToT, a reasoning tree of intermediate reasoning steps is lazily built: at each step, given the current node, the LLM is queried to generate a number of n subsequent candidate intermediate steps; the Agent uses domain-specific information to decide whether to continue exploring one of them, or to backtrack to the previous node; for the latter, an unexplored child of that node may be subsequently chosen for exploration by the Agent, or it may backtrack further. This architecture aims to take advantage of the Agent and LLM's respective strengths:

- The Agent has access to domain-specific information, and can therefore make decisions about the quality of the candidates based on specific and relevant metrics, which were not explicitly considered by the LLM's training objective.

Program *A*: Problem 266B, solution 7604802

```
n,t=list(map(int,input().split()))
a=input()
k=0
for i in range(t):
    a=a.replace('BG','GB')
print(a)
```

Program *B*: problem 266B, solution 13328325

```
n,t = list(map(int,input().split()))
s = input()
while t>0:
    s = s.replace("BG","GB")
    t -= 1
print(s)
```

Fig. 1. Example of two equivalent programs A and B denoting two solutions for the Codeforces Problem 266B: The programming task requires simulating the dynamic rearrangement of a queue consisting of boys and girls in a school canteen. Initially ordered as they entered, the boys in the queue allow the girls directly behind them to move forward each second. The programmer must create an algorithm to predict the final order of the queue after a specified number of seconds based on the initial arrangement and the defined movement rules.

- The LLM can focus on generating immediate next steps, without the need for long-term planning.

For the program equivalence task, the RL Agent leverages domain-specific information by utilising a syntax checker, running unit tests, and applying code similarity metrics.

Key findings. Our experiments led to several key findings, which are discussed in detail in the rest of the paper:

(1) For the program equivalence task, where multiple intermediate reasoning steps are needed, CoT often struggles to maintain accuracy of these steps. In our setting, this means that the intermediate programs frequently fail to maintain the same behaviour as the source program, and that, for equivalent programs, the final program in the transformation sequence has limited similarity to the target program. This aligns with prior research, which concludes that when multiple valid deduction paths exist, LLMs often fail to systematically explore the different options resulting in poor long term planning [39].

(2) While ToT does better than CoT with respect to both maintaining the behaviour of the source program, as well as increasing similarity with the target program, the results are still poor, especially for the latter aspect. We hypothesise that this is caused by the fact that the LLM has to self-evaluate programs in order to decide the next action. However, this evaluation requires domain-specific information, which was not part of the LLM's training objective.

(3) The Agent architecture outperforms CoT and ToT in the evaluation of both the intermediate reasoning and the downstream task.

Contributions. We make the following key contributions:

- We designed an RL and LLM cooperative approach for the exploration of the solution space, suitable for tasks that require long-term planning, and where the success depends on domain-specific criteria that were not explicitly considered during the LLM's training.

Program A

```
n,t=list(map(int,input().split()))
a=input()
k=0
for i in range(t):
    a=a.replace('BG','GB')
print(a)
```

Program A_1: variable renaming a to s

```
n,t=list(map(int,input().split()))
s=input()
k=0
for i in range(t):
    s=s.replace('BG','GB')
print(s)
```

Program A_2: remove redundant $k = 0$

```
n,t=list(map(int,input().split()))
s=input()
for i in range(t):
    s=s.replace('BG','GB')
print(s)
```

Program A_3: for to while conversion

```
n,t=list(map(int,input().split()))
s=input()
while t>0:
    s=s.replace('BG','GB')
    t -=1
print(s)
```

Program A_4 (same as program B): replacing quotes

```
n,t= list(map(int,input().split()))
s = input()
while t>0:
    s = s.replace("BG","GB")
    t -=1
print(s)
```

Fig. 2. Sequence of semantics preserving transformation from program A to program B

- We evaluated our architecture on program equivalence queries, where we investigated both the downstream task, which is the classification result, and the intermediate proof steps. For this task, our architecture compared positively against CoT and ToT prompting.
- To the best of our knowledge, we performed the first investigation of a non-linear space exploration for a code-related task, and showed that it improves results compared to linear solution building.

Roadmap. We start by describing our RL Agent architecture in Sec. 2, followed by evaluating it and comparing it with CoT and ToT in Sec 3 and Sec. 4. We then discuss threats to validity and related works in Sec. 5 and Sec.6, respectively.

2 Our Approach

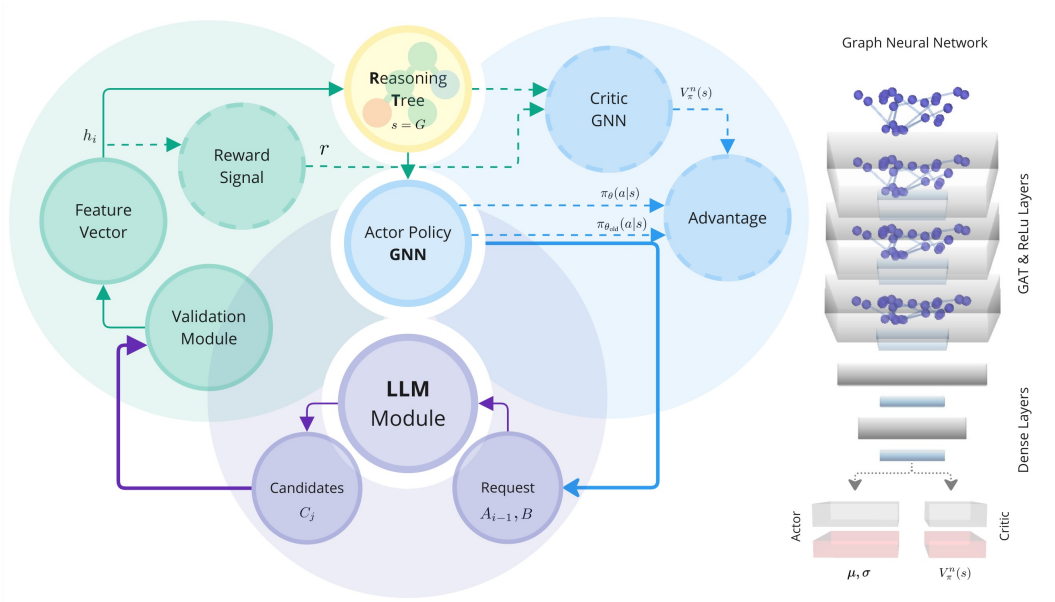

Fig. 3. The proposed architecture (left) is composed of four components: the LLM Module (purple), the Validation Module (green), the Reasoning Tree (yellow), and the Agent (blue). The Graph Neural Network architecture (right) is used for the actor and critic. Dashed lines indicate components used exclusively during training, whereas solid lines indicate those utilised in both training and inference. (Best viewed in color).

When solving a task, our approach generates the intermediate reasoning steps $A, A_1, \cdots A_n$ by lazily building a reasoning tree of such steps. For the equivalence task, the nodes in the reasoning tree are mutated programs (i.e. mutated versions of the source program A), with edges corresponding to the semantics-preserving transformations that lead from a parent program to a child program. The root of the reasoning tree is the source program A. Examples of semantics-preserving transformations are variable renaming, changing the order of a function's arguments, modifications to control flow statements (e.g. switching the order of the branches of an `if` statement – of course, with the corresponding negation of the guard, switching a `while` loop for a `for` one), substitution of data structures.

The overall structure of our architecture consists of four components (See Figure 3), namely the LLM Module, the Validation Module, the Reasoning Tree and the RL Agent, which interact as described next:

At each step, given the current mutated program A_{i-1}, the **LLM Module** queries the LLM, requesting n candidates for the next mutated program A_i obtained by applying a semantics-preserving transformation to A_{i-1}. Next, the **Validation Module** is responsible for assessing the characteristics of each of the n candidates. For each candidate, all the corresponding assessments are combined into a feature vector. The feature vectors are then added to the **Reasoning Tree**, as the children of the node currently explored (the actual programs are stored separately). Lastly, the **RL Agent** guides the exploration of the reasoning tree and decides which node to explore next. Namely, the RL Agent decides whether to continue exploring one of the currently generated children or to

backtrack to the previous node; for the latter, an unexplored child of that node may be subsequently chosen for exploration by the Agent, or it may backtrack further.

2.1 LLM Module

The LLM module is responsible for sending requests to the LLM and collecting the candidate mutated programs. Initially, the prompt includes the source program A and the target program B. In subsequent iterations i>1, the prompt is based on the most recent mutated program A_{i-1} and the target program B. Each time, we request n candidates.

2.2 Validation Module

This module's objective is to take advantage of domain-specific information to compute metrics for the generated candidates, which were not explicitly part of the LLM's training objective. For the current problem, we are interested in syntactical correctness, functional correctness, code similarity, and granularity of the applied transformation. Next, we explain how they are computed.

For **Syntactical Correctness** of a program, we check whether its Abstract Syntax Tree (AST) is structurally valid. The resulting output is either 1 (i.e., the program is syntactically correct) or 0 (i.e., the program is not syntactically correct).

We measure the **Functional Correctness** of a program as the percentage of unit tests that it successfully passes. This measures the code's functional correctness under the predefined test cases.

We use CodeBLEU to assess **Code Similarity** between two programs. CodeBLEU is specifically designed for source code as it combines the strength of BLEU [33] through the n-gram match with information about the code syntax via ASTs and code semantics via data flow. Intuitively, when computing the code similarity between a candidate and the target program, we want it to provide an indication of how many semantics-preserving transformations are still required to get from the candidate program to the target (as opposed to simply indicating the number of different syntactic tokens). Thus, we chose CodeBLEU as opposed to something like BLEU. CodeBLEU also shows a high correlation with human evaluation. It gives an output score between 0 and 100, where a CodeBLEU score of 100 means that the two programs perfectly match.

Given two programs, where the second was obtained from the first one by applying a transformation, **Granularity** is meant to estimate the coarseness of the said transformation. For this purpose, we use the Jaccard index [48], which compares the ASTs of the two programs by evaluating the overlap between their respective sets of nodes. The Jaccard coefficient takes values between 0 and 1, with 0 indicating no overlap and 1 complete overlap between the sets. For our use case, a larger value corresponds to a finer grained transformation. Granularity is of interest to us as we hypothesise that smaller transformations would result in better reasoning abilities by breaking the overall task into small, manageable subtasks. This could also help the user to understand the transformation steps and gain trust in the solution.

Let's assume we have generated mutated programs $A_1, \cdots, A_{i-1}$, and we are now querying the LLM for program A_i. For each candidate C_j, $j = \overline{1,n}$ returned by the LLM, we compute:

- $v(C_j)$**:** Syntactical correctness of C_j.
- $\rho(C_j)$**:** Functional correctness of C_j.
- $\sigma(C_j, A)$**:** Code similarity to the source program A.
- $\sigma(C_j, A_{i-1})$**:** Code similarity to the last generated program A_{i-1} (i.e., the parent node in the reasoning tree). This reflects changes over time.
- $\sigma(C_j, B)$**:** Code similarity to the target program B. This serves as an indication of how many semantics-preserving transformations are still required to get from C_j to B.
- $g(C_j, A_{i-1})$**:** Granularity of the last applied transformation.

$g(C_j, B)$**:** Jaccard index of the current candidate and the target program. This provides syntactic information in the form of the dissimilarity of the ASTs corresponding to the two programs.

All these metrics are then compiled into a feature vector h_j, which the Agent uses as a foundation for making decisions:

$$h_j = [v(C_j), \rho(C_j), \sigma_(C_j, A), \sigma(C_j, A_{i-1}), \sigma(C_j, B), g(C_j, A_{i-1}), g(C_j, B)] \quad (1)$$

In our experiments, we investigated the influence weights for each feature in a feature vector, and determined that the most important are syntactical and functional correctness, as well as similarity to the target program, which is very intuitive.

2.3 Reasoning Tree

The reasoning tree G is lazily built as explained next. At each step, given the current node corresponding to mutated program A_i (initially corresponding to the source program A, which is the parent node of the reasoning tree), we generate n candidates, C_j, $j = \overline{1,n}$. At this point, the reasoning tree G is updated by adding an edge from the current node A_i to each candidate C_j.

Although, for simplicity, we may refer to the nodes in the reasoning tree as the mutated programs, these nodes only store the corresponding feature vectors, while the actual programs are stored separately.

2.4 Agent Module

The RL Agent's role is to determine the most effective sequence of semantics-preserving transformations of source program A such that it becomes as similar as possible to target program B.

Action space. The action performed by the Agent guides the traversal of the reasoning tree G. At each step, given the current node corresponding to mutated program A_i, and the newly generated n candidates, C_j, $j = \overline{1,n}$, the RL Agent can pick from $n+1$ available actions: it can pick to explore one of the C_j candidates or to backtrack to its immediate ancestor node, A_{i-1}. If the latter option is picked, then, in the next iteration, the Agent will be able to pick to explore one of the still unexplored children of A_{i-1} (i.e., a sibling of the current node) or further backtrack. Backtracking is disabled for the root node of the reasoning tree.

RL architecture. We make use of the actor-critic framework [37], where an Agent (the "actor") learns a policy to make decisions, and a value function (the "critic") evaluates the actions taken by the actor by estimating their value or quality. The policy is denoted by $\pi_\theta(s = G, a)$ which represents the probability of taking action a in state s. The value function (corresponding to the critic), denoted by $V(s)$, estimates the expected reward starting from state s. After training, at inference time, the actor Agent will use the learned policy to make decisions about the exploration of the reasoning tree.

In our setting, G serves as the state s for the RL framework. This is necessary in order to enable the possibility of backtracking, where the next action depends on the history of states (remember that, at each step, there is an option to jump back to the parent).

To effectively extract features from the current reasoning tree G, we employ a Graph Attention Network (GAT) [45], which is a variant of a Graph Neural Network (GNN) that leverages attention mechanisms. GATs have been empirically shown to outperform similar GNN models for applications involving code [3]. At a high level, the method of feature extraction from the graph through graph convolution relies on two primary actions: first, gathering the characteristics of neighboring nodes for each individual node; second, updating each node using trainable weights and adjustable

attention mechanisms. For more details, please consult [45]. As shown in Figure 3 (right), our model features three GAT layers, as referenced in [45]. The feature vectors in the initial GAT layer correspond to those computed by the Validation Module, denoted as $h_i^{(0)}$. The first hidden layer outputs features into a space $h_i^{(1)}$, with a dimensionality set to 30. Similarly, for the next layer. The critic's output dimension is set to 1 to provide a single estimate of the state value. Meanwhile, the actor policy's output has a dimensionality of 2, representing μ and σ for a multivariate distribution. A normalized sample from this distribution is transformed into a discrete action within a space of $(n+1)$ options.

Training of the Agent. The training process for the Agent begins with an initial random policy, $\pi_\theta(s = G, a)$. During training, we utilize experience replay, and the reward signal r is subject to discounting, encouraging the actor policy to prioritize decisions that are optimal over the long term. Training is specifically conducted on equivalent programs and the reward signal is designed to encourage the creation of a sequence of semantics-preserving transformations from A to B. As criteria for long-term goals, we expect the semantics preserving transformations to eventually rewrite A into B. The discounted future reward, denoted as D_t, is calculated as

$$D_t = \sum_{k=0}^{n} \gamma^k R_{t+k+1} \tag{2}$$

where t denotes the number of steps until the Agent achieves the final goal, and γ is the discount factor, $0 < \gamma \leq 1$. For low values of γ, the Agent considers immediate rewards, while a gamma close to one assigns greater weights to future rewards. We continue training the Agent until the policy stabilizes and shows signs of convergence for the validation set. We employ transfer learning to adjust the Agent to a different base LLM, which uses significantly fewer training episodes than needed for complete training.

As a note, we initially experimented with providing additional rewards for things like applying finely-grained transformations. However, it proved more effective to allow the policy to autonomously determine the importance of each available option.

2.5 Termination Condition

Defining a termination condition is crucial for the practical deployment of our algorithm. We stop the generation of new mutated programs when one of the following conditions is met:

- A maximum number of steps m have been taken where the generated programs either didn't pass any of the unit tests or the similarity to the target program has not improved. This allows the quality of the generated programs to temporarily decrease/not improve.
- A maximum number of steps p have been taken overall.

2.6 Adapting the Architecture to a Different Task

While we only applied the Agent architecture to the specific problem of answering program equivalence queries, we believe it may be beneficial to other tasks that require long-term proof planning (i.e. several intermediate reasoning steps), and where the success depends on criteria that were not explicitly considered during the LLM's training. In order to adapt the framework for a new task, the following steps are needed:

(a) Determining the nature of the intermediate reasoning steps;
(b) Establishing the metrics for evaluating the intermediate reasoning steps and the external checkers to compute these metrics. These metrics denote the features associated with each node in the reasoning tree;

(c) Re-training the RL Agent to optimise these metrics.

For example, when considering the task of dynamic slicing [23] (i.e. given a program and a set of unit tests, the goal is to generate the smallest slice of the program that still ensures the unit tests pass correctly), the reasoning process can be described as follows: (a) Each reasoning step produces a version of the original program with some instructions removed; (b) The evaluation metrics are syntactic correctness and the number of passed unit tests, with checkers similar to those used in this work. For program repair [14]: (a) Each reasoning step involves a partially repaired version of the original program; (b) The evaluation metrics are syntactic correctness and the number of passed unit tests, again utilising checkers similar to those employed in our current work.

Extending our approach to a different domain, such as Sudoku solving [44] involves the following first two steps: (a) Each iteration produces a partial board where additional numbers have been filled in beyond the initial configuration; (b) The evaluation metric verifies that the current partial board adheres to Sudoku's rules, ensuring a valid progression toward a complete solution.

3 Experimental Evaluation

3.1 Experimental Setup

We run our experiments on a machine AMD Ryzen 9 6900HX Processor withNvidia GeForce RTX 3070 TI with 8GB RAM. All LLMs are accessed through APIs provided by third party services.

We developed a custom implementation of the policy-gradient algorithm in PyTorch, incorporating graph attention convolutions from the Deep Graph Library (DGL) for both the actor and critic. This approach enables us to effectively integrate graph-based data with RL frameworks.

For our experiments, we use $n = 10$, meaning that we request ten candidates from the LLM. For the termination condition, we use $m = 3$ and $p = 10$.

The experiments are conducted twice, and the average metrics from both runs are then calculated and presented.

3.2 Dataset

We make use of a subset of the AlphaCode dataset [6], which consists of programming challenges of varying lengths and difficulties extracted from several coding challenge platforms. Each challenge includes a natural language description and unit tests, along with multiple solutions in Python 2 and C++ 4.3.2. In this work, we focus on Python, so we only use those solutions. Moreover, we only picked those challenges that had at least 50 unit tests. The resulting dataset consists of 7,764 programming problems.

The average number of lines of code and unique tokens (e.g., instruction, variable name) for each program significantly varies by difficulty level. For instance, easy challenges from CodeChef samples have an average of 15 lines of code and 144 unique tokens, while hard challenges average 40 lines of code and 204 unique tokens.

We reserve 75% of the dataset for the training of the RL Agent. However, given that we only train until the performance on the validation set stabilises, we observe that the policy begins to stabilize as early as after 200 to 400 requests to the LLM (not episodes). Also, we employ transfer learning to adjust an Agent trained for a certain LLM to a different one, which uses significantly fewer training episodes. Additionally, we keep 10% of the dataset for validation purposes. That leaves us with 1164 programming problems for evaluation with a mean of 48.96 solutions per programming problem with an average of 26.08 lines of code and 110.71 unique tokens. For each problem, we take the combination of all the solutions to form pairs of equivalent programs.

In our investigation of intermediate reasoning steps (Sec. 4.1) and the additional Agent evaluation (Sec. 4.3), we focus exclusively on pairs of equivalent programs. This allows us to evaluate

the reasoning process more effectively, as the objective is clear: apply a sequence of semantics-preserving transformations to the source program A until it becomes the target program B. For the investigation of the downstream task (Sec. 4.2), we also add pairs of non-equivalent programs. For this purpose, we mutate one program in all equivalent pairs, such that each pair is no longer equivalent – typically through straightforward alterations like changing "+1" to "+2".

3.3 Considered LLMs

In our evaluation, we consider the following LLMs: GPT-3.5 [17], GPT-4 [31], and GPT-4 Turbo [41]. GPT-3.5, an earlier iteration in the GPT series, is known for its robust natural language processing abilities, though it occasionally struggles with nuanced context understanding. GPT-4 offers more accurate and context-aware responses. GPT-4 Turbo, a streamlined variant of GPT-4, is designed for faster response times. We use the default values for the hyper-parameters of the LLMs (e.g. for GPT: Temperature: 1, Top_p-nucleus sampling: 0.9, Frequency Penalty: 0, Presence Penalty: 0).

3.4 Prompting

For the Agent architecture, we experimented with several prompts and chose the following template: *Given programs {A} and {B}, transform the first program so that it becomes syntactically more similar to the second program while retaining its semantics. Apply only one atomic transformation. Provide only the source code without your comments.* This prompt is used by the LLM module in Fig. 3 to obtain candidate programs at each intermediate reasoning step. For the experiments in Sec. 4.2, where we evaluate the downstream task given the generated intermediate reasoning, we provide the sequence of generated transformations in the prompt.

For CoT, we used the prompt suggestion in [19]. To trigger the ToT reasoning, we used the ToT prompt proposed in [15]: *"Imagine n different experts are answering this question. All experts will write down 1 step of their thinking, then share it with the group. Then all experts will go on to the next step, etc. If any expert realizes they're wrong at any point, then they leave. The question is ..."*. We experimented with a few values for the number of experts, n, and we found that the optimal performance is achieved with three agents. When the number exceeds three, the LLM often groups agents together, leading to a severe decline in performance. We tested various LLM settings, such as adjusting the maximum output tokens, but did not manage to improve the results.

4 Experimental Results

We structure our results around three key areas. First, we evaluate and compare the intermediate reasoning steps generated for the CoT prompt, ToT prompt and the Agent architecture. Next, we examine how these intermediate steps contribute to answering the downstream program equivalence queries. Finally, we conduct a deeper analysis of the Agent architecture to identify its most critical components and to assess whether a simpler architecture could achieve similar performance.

4.1 Evaluation of the Intermediate Reasoning Steps

In order to assess the intermediate steps taken by the Agent architecture, CoT prompting and ToT prompting, for each of them we measure: Syntactical Correctness, Functional Correctness, Code Similarity to the target program, and Granularity where appropriate. For Syntactical and Functional Correctness, we are interested in both mean and final values: (1) **mean** represents the average metric calculated across all intermediate mutated programs within the sequence of mutated programs explored for each benchmark. For syntactical correctness we compute the percentage of programs that are syntactically correct; (2) **final** denotes the metric associated with the last mutated program generated for each benchmark. The numbers in the table are the average of mean

Table 1. Comparison between the Agent architecture and CoT with respect to the intermediate reasoning steps (best results in bold font).

Model	Syntactical Correctness (%)		Functional Correctness (%)		Code Similarity	Granularity
	mean	final	mean	final	final	mean
GPT-3.5 + **CoT**	75.96	76.92	31.90	34.62	46.05	0.3879
GPT-3.5 + **Agent**	**81.25**	**78.4**	**72.34**	**52.15**	**87.15**	**0.4892**
GPT-4 + **CoT**	97.06	97.06	65.08	61.03	48.56	88.05
GPT-4 + **Agent**	**98.75**	**99.2**	**82.03**	**70.45**	**95.45**	**91.24**
GPT-4-turbo + **CoT**	96.90	97.24	69.38	64.21	51.62	86.44
GPT-4-turbo + **Agent**	**98.54**	**99.01**	**85.47**	**78.81**	**95.74**	**90.39**

Table 2. Comparison between the Agent architecture and ToT with respect to the intermediate reasoning steps (best results in bold font).

Model	Syntactical Correctness (%)	Functional Correctness (%)	Code Similarity
GPT-3.5 + **ToT**	**96.34**	50.09	62.38
GPT-3.5 + **Agent**	78.4	**52.15**	**87.15**
GPT-4 + **ToT**	96.12	68.76	60.52
GPT-4 + **Agent**	**99.2**	**70.45**	**95.45**
GPT-4-turbo + **ToT**	96.05	71.42	63.2
GPT-4-turbo + **Agent**	**99.33**	**74.57**	**94.39**

and final values across all the benchmarks. The **final** values for Syntactical Correctness in the table denote the percentage of final programs that are syntactically correct.

For ToT, given that the experts explore candidates concurrently, which they share with the rest after each step, it's difficult to determine which intermediate steps were actually useful and should be considered when computing the mean metrics. Thus, Table 2 only contains the values for the final mutated program (and no granularity results).

All the results used for this evaluation are captured in Table 1 and Table 2.

RQ1(a). What are the intermediate reasoning capabilities of CoT and ToT prompting for the program equivalence task? Both CoT and ToT provide poor quality intermediate reasoning steps. While they are generally able to generate intermediate programs that are syntactically correct, the functional correctness drops, meaning that the transformations frequently fail to maintain the semantics of the original program. Even more concerning, the similarity between the final mutated program and the target program B is very low. This is particularly true for CoT, where it ranges between 46.05% and 51.62% for the different LLMs. ToT does a bit better with respect to code similarity, ranging between 60.52% to 63.2%.

These findings support the hypothesis that CoT in particular has issues with long-term planning, which has been suggested by existing works for other problem domains [39] – generating a sequence of semantically equivalent mutated programs that are progressively more similar to a target program requires long range planning. For ToT, we hypothesise that its relatively poor reasoning is due to the fact that the intermediate candidates need to be assessed based on metrics that are different from the ones used during training. Namely, syntactical correctness, functional correctness and code similarity are different from the next-token prediction objective, which maximizes the likelihood of the next ground-truth token.

RQ1(b). How do the reasoning abilities of the Agent architecture compare against CoT and ToT for the program equivalence task? The Agent architecture outperforms both CoT and ToT for the majority of the recorded metrics, with the biggest improvements obtained for functional

correctness and code similarity. This implies that the intermediate programs generated by the Agent architecture are more likely to maintain the semantics of the source program.

The only instance where the Agent was outperformed occurred when GPT-3.5+ToT generated syntactically correct programs more effectively than GPT-3.5+Agent. However, our Agent configuration significantly excelled in generating programs that preserve the semantics of the original code (functional correctness) and exhibit higher syntactic similarity to the target program (code similarity).

Compared to CoT, for the Agent configuration, the code similarity metric of the final program improves on average with 41.1% for GPT-3.5, 46.89% for GPT-4, and 44.12% for GPT-4-turbo. For ToT, the improvements of this metric are 24.77% for GPT-3.5, 34.93% for GPT-4, and 31.19 for GPT-4-turbo. This means that the final programs generated by the Agent architecture are much more similar to the target program than those generate by the CoT and ToT prompts. This supports our hypothesis that being able to explore different paths through the search space, as well as having access to domain specific information (i.e. syntax checker, unit tests, and code similarity metric) leads to better performance.

As an additional remark, the Granularity column in Table 1 suggests that the Agent architecture tends to take finer grained intermediate steps than CoT. It seems reasonable that smaller steps may result in better overall reasoning, and it is in line with out initial hypothesis.

RQ1 Answer: When assessing the intermediate reasoning steps, both CoT and ToT prompting generate low-quality outputs. The resulting intermediate and final mutated programs frequently fail to maintain functional correctness, and the final mutated programs show reduced code similarity to the target program. This suggests possible limitations with long-term proof planning for CoT and ToT prompting. In comparison, our Agent architecture surpasses both CoT and ToT, likely due to its non-linear reasoning abilities and access to domain-specific information when assessing candidate intermediate steps (i.e. programs).

4.2 Evaluation of the Downstream Task

In this section, we investigate the actual answers given to the equivalence queries in four configurations: (1) asking the LLM whether the two given programs are equivalent, without requesting any intermediate reasoning, (2) CoT, (3) ToT, and (4) the Agent architecture. The results are given in Table 3. As metrics, we consider recall, precision, the F1 score and the area under the receiver operating characteristic (ROC AUC). For all the metrics, higher values correspond to higher performance of the model.

RQ2(a). How do CoT prompting and ToT prompting perform with respect to answering equivalence queries? The results for no intermediate reasoning, CoT and ToT are all very close, with the order varying for different models (for GPT-3.5 and GPT-4, CoT and ToT are doing slightly better than no reasoning, whereas for GPT-4-turbo, no reasoning performs somewhat better). This is surprising given that intermediate reasoning is generally expected to facilitate solving downstream tasks. One explanation might be that the intermediate reasoning elicited by CoT and ToT for this problem is too poor to actually help.

RQ2(b). How does the Agent architecture answer equivalence queries compared to CoT and ToT prompting? The Agent architecture outperforms all the other configurations suggesting that, when accurate, intermediate reasoning does improve downstream task outcomes.

Table 3. Comparison between no intermediate reasoning, the Agent architecture, CoT, and ToT with respect to answering equivalence queries (best results in bold font).

Model	F1	ROC AUC	Precision	Recall
GPT-3.5 + **no reasoning**	0.4969	0.4938	0.4938	0.5000
GPT-3.5 + **CoT**	0.5644	0.5563	0.5542	0.5750
GPT-3.5 + **ToT**	0.5542	0.5375	0.5349	0.5750
GPT-3.5 + **Agent**	**0.5839**	**0.5812**	**0.5802**	**0.5875**
GPT-4 + **no reasoning**	0.6135	0.6062	0.6024	0.6250
GPT-4 + **CoT**	0.6234	0.6375	0.6486	0.6000
GPT-4 + **ToT**	0.6194	0.6313	0.6400	0.6000
GPT-4 + **Agent**	**0.7248**	**0.7438**	**0.7826**	**0.6750**
GPT-4-turbo + **no reasoning**	0.6829	0.6750	0.6667	0.7000
GPT-4-turbo + **CoT**	0.6506	0.6375	0.6279	0.6750
GPT-4-turbo + **ToT**	0.6135	0.6062	0.6024	0.6250
GPT-4-turbo + **Agent**	**0.7470**	**0.7375**	**0.7209**	**0.7750**

Table 4. Agent with and without backtracking

Metric	w/o backtracking	w backtracking
Syntactical Correctness	76.64%	**79.29%**
Functional Correctness	57.93%	**71.22%**
Code Similarity	53.20%	**76.03%**

RQ2 Answer: When answering the program equivalence task, the Agent architecture outperforms all the other configurations (no intermediate reasoning, CoT prompting, ToT prompting), suggesting that better intermediate reasoning may lead to better outcomes for the downstream task.

4.3 Additional Agent Evaluation

In this section, we further analyse of the Agent architecture to identify its most critical components and to assess whether a simpler architecture could achieve similar performance. We assess the Agent's performance with respect with its intermediate reasoning abilities, as this is the main objective of our architecture. In order to limit cost (both dollar and compute), these experiments are only performed for GPT-3.5.

RQ3(a). How does the Agent's backtracking ability affect its performance? To answer this RQ, we perform an ablation study aimed at determining how much of the Agent's effectiveness is attributed to its ability to backtrack. For this purpose, we retrain the model with backtracking disabled (while retaining the other possible actions). Results are given in Table 4. The Agent without backtracking abilities results in poorer metric values than the one with backtracking.

Additionally, we investigate how often the agent chooses to backtrack. Figure 4 denotes a histogram of the frequency of each action taken by the Agent, with and without backtracking. When backtracking was present, it accounted for 21.13% of actions, while the LLM's default decision (i.e., picking Candidate 1) was just selected in 13.45% of the cases. Additionally, the data suggests a mild preference for picking candidates that are ranked higher by the LLM, visible by a slightly left-skewed distribution. The ability to backtrack appears to slightly reduce this preference.

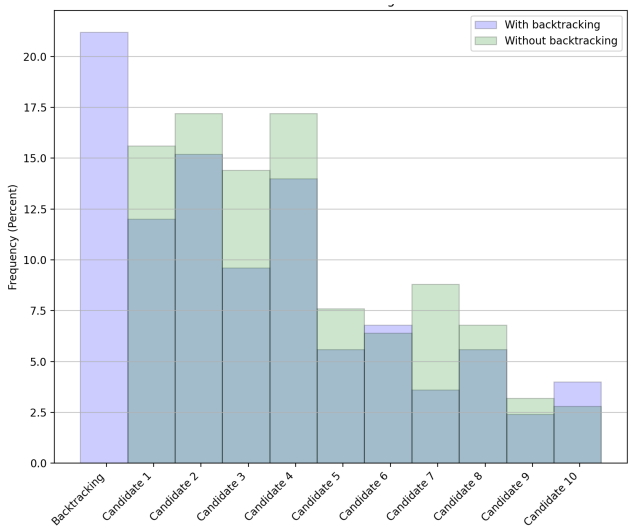

Fig. 4. Histogram comparing the frequency distribution of discrete actions taken by two types of Agents: with and without backtracking. The initial action corresponds to backtracking (when available), whereas the other actions correspond to picking one of the candidate mutated programs. (Best viewed in color).

Table 5. Comparison between the Agent architecture and greedy approaches.

Model	**Syntactical Correctness (%)**		**Semantic Correctness (%)**		**Code Similarity**	**Granularity**
	mean	final	mean	final	final	mean
GPT-3.5 + Policy 1	67.43	62.93	51.28	48.10	48.18	12.64
GPT-3.5 + Policy 2	74.36	78.30	52.98	53.33	57.40	32.40
GPT-3.5 + Policy 3	76.54	73.38	66.09	64.36	65.06	43.65
GPT-3.5 + **Agent**	**81.25**	**78.40**	**72.34**	**52.15**	**87.15**	**48.92**

RQ3(a) Answer: Removing backtracking from our model significantly reduced its reasoning effectiveness, as shown by a reduction in all metrics in Table 4. ‘

RQ3(b). How does the Agent’s learned policy compare against a greedy approach? Rather than learning a policy, here we experiment with a few cheaper greedy approaches where space exploration is rule-based. In particular, Policy 1 favors syntactically correct candidates, Policy 2 prioritizes functional correctness, and Policy 3 chooses the candidate with the highest sum of the normalised metrics for syntactical correctness, functional correctness, and code similarity to the target program.

Table 5 presents a summary of the results, showing that the neural Agent outperforms all the greedy approaches (among which Policy 3 is the most effective), showing that training may be beneficial for the current problem.

RQ3(a) Answer: The neural agent significantly outperforms all the greedy approaches.

RQ3(c) How does the Agent’s performance vary when the source and the target programs use different programming concepts (e.g., data structures, control flow structures)? The same programming task can be solved using different programming concepts (e.g., different data

Table 6. Comparison of source and target pairs implementing similar and different programming concepts

Metric	**Same** (AB) final	**Different** (AC) final
Syntactical Correctness (%)	**94.48**	89.39
Functional Correctness (%)	**82.75**	69.12
Code Similarity	**86.50**	77.83

structures, different control flow structures). It's natural to presume that generating transformations between a source and a target program becomes more complex when the two programs make use of divergent strategies – such programs correspond to Type-4 code clones [4].

For illustration, below we present two distinct solutions for the Codeforces problem 255B: The programming task involves designing an algorithm that manipulates a string consisting of "x" and "y" by first swapping any adjacent "y" and "x" found starting at the start of the string, then removing any adjacent "x" and "y" pairs, repeatedly, until no more such operations can be applied. The algorithm prioritizes swapping over removal and processes the string from the beginning, ultimately printing the modified string as its result.

```
count = { 'x': 0, 'y': 0 }
for i in raw_input():
    count[i] += 1

print(('x' if count['x'] > count['y'] else 'y') * abs(count['x'] - count['y
    ']))
```

Listing 1. Solution: 2869941. This solution utilizes a dictionary to track the frequency of x and y in the string, allowing for a comparison of their counts to determine which character is more prevalent and by what margin.

```
r = []
for x in raw_input():
  if not len(r) or r[-1] == x:
    r.append(x)
  else:
    r.pop()
print ''.join(r)
```

Listing 2. Solution: 2793525. This solution employs a list as a stack to dynamically add or remove characters while iterating through the string, effectively canceling out adjacent 'xy' or 'yx' pairs. This stack approach enables a sequential evaluation and modification of the string, retaining only those characters that do not form cancelable pairs.

We aim to investigate how the performance of the model with the Agent fluctuates when applied to programs using different programming concepts. For this purpose, we randomly selected a subset of 143 Codeforces challenges from our dataset. From each challenge, we manually selected three Python solutions, A, B, and C, such that A and B use similar programming concepts, whereas C uses either a different data structure or a different control flow structure. Then, in our experiments, we compute program transformations for AB (A is the source program, and B is the target) and AC (A is the source program, and C is the target). Again, tthis experiment was only performed for GPT-3.5.

In Table 6, we present a breakdown of the evaluation results for AB and AC. We assess the metrics for the final transformed program. Overall, the analysis suggests that program pairs implementing

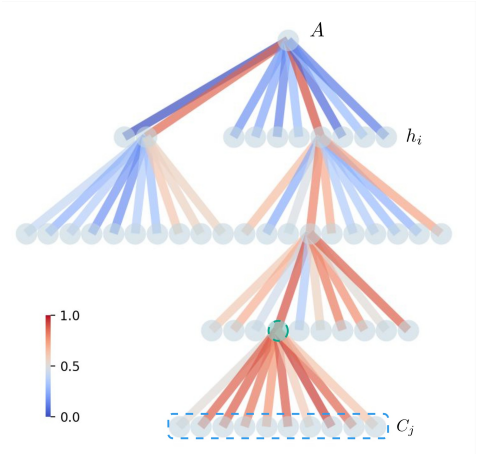

Fig. 5. Visualisation of a reasoning tree displaying graph attention on edges, where high attention scores are represented by red and low attention scores are depicted by blue edges. (Best viewed in color).

similar programming concepts tend to achieve higher scores across all evaluated metrics compared to pairs using different concepts. For programs in the AC group, the syntactical correctness of the final program decreases by 5.09% on average, functional correctness by 13.63% on average, and code similarity to the target program by 8.67% on average.

RQ3(c) Answer: Compared to programs using similar programming concepts, for programs implementing different concepts, syntactical correctness of the final program decreases by 5.09% on average, functional correctness by 13.63% on average, and code similarity to the target program by 8.67% on average.

RQ3(d). What does the Agent learn? We randomly picked a reasoning tree and conducted an analysis of the attention scores obtained from the graph attention layers when the Agent is exploring candidate programs from the current node (the green node), as depicted in Figure 5. We observed that recent candidates and candidates that have been selected in the past receive more attention. Also, it doesn't seem to be the case that the candidates top-ranked by the LLM always attract higher attention scores.

In a separate analysis focused on the feature vectors, we investigated the influence weights for each feature in a vector h_i, and determined that the most important are syntactical and functional correctness, as well as similarity to the target program, which is very intuitive.

RQ6 Answer: Our analysis found that recent candidates in the Reasoning Tree receive the most attention and that the policy prioritizes syntactical and functional correctness, as well as similarity to the target program.

5 Threats to Validity

Our main result is that an RL agent trained with domain-specific information can help guide the LLM's space exploration, and thus produce better results when answering equivalence queries. We discuss threats to the validity of this conclusion.

The number of benchmarks, as well as their length and number of tokens, could limit the generalisation of our findings. Moreover, we only experimented with a limited number of LLMs, and we can't guarantee that our results apply to others. We only applied the Agent architecture to the problem of answering equivalence queries. Thus, we can't say whether the same improvement would be obtained for other tasks. We judge functional correctness with respect to unit tests. However, unit tests may not fully capture a program's behaviour, which means that we may over-report functional correctness. Especially for CoT and ToT, different prompts may results in better results. To reduce the danger, we used prompts proposed in the literature.

Future Work. We aim to explore the application of a reasoning tree that stores embeddings related to the program itself instead of solely relying on computed features from these embeddings. We also plan to apply our Agent architecture to other tasks.

6 Related Work

6.1 External Reasoning and Space Exploration for LLMs

While, initially, LLMs were used as black-box, monolithic entities, recently, there has been a shift towards architectures that foster some form of logical reasoning as part of the problem-solving process, sometimes by leveraging additional, possibly non-neural systems. For instance, Karpas et al. propose a neuro-symbolic architecture, dubbed the Modular Reasoning, Knowledge, and Language system, denoting a flexible architecture with multiple neural models, complemented by discrete knowledge and reasoning modules [18]. Lazaridou et al. embed the model within a simple program that uses Google Search during inference without the need for a dedicated retrieval-augmented model. The CoT approach aims to improve the ability of LLMs to perform complex reasoning by generating a series of intermediate reasoning steps [46].

Other works are focused on decision-making and space exploration, given that LLMs were shown to have difficulty with proof planning when using a linear search strategy. In particular, when multiple valid deduction steps are available, they are not able to systematically explore the different options [39]. Several works attempt to ameliorate this. Schlag et al. introduced LLM programs, where an LLM is embedded in a classic program to carry out more complex tasks [40]. Then, this method decomposes the main problem recursively into subproblems until they can be solved by a single query to the model. LLM+P [28] delegates the actual planning process to a classical planner in order to solve long-horizon robot planning problems. In [36], Princis et al. use best-first search to guide the LLM's exploration of the solution space, but focus exclusively on the generation of SQL queries. Tree of Thoughts integrates thought sampling and value feedback, enabling effective search inside a reasoning tree [47]. It implicitly builds in decision-making and planning. A similar architecture is proposed by Long, who augments an LLM with additional modules, including a prompter Agent, a checker module, a memory module, and a ToT controller.

The approaches based on ToT are the closest to our work. However, there are several differences. While [47] uses DFS/BFS/beam search, we make use of an RL Agent. Instead of using the LLM to evaluate the potential of different candidates, the Agent has access to domain-specific knowledge, which in our case are a syntax checker, unit tests, codeBLEU and Jaccard metrics. Conversely, from [30], we propose an architecture with a different way of exploring the reasoning tree, which only stores the feature vectors. Moreover, none of the existing works investigates the ToT space exploration for code related tasks.

6.2 Reinforcement Learning and LLMs

In recent years, RL has become a common paradigm for fine-tuning LLMs. Many models are fine-tuned with RL, e.g. OpenAI's GPT-4 [31], Google's Gemini [2], and Anthropic's Claude 3 [1]. While

such fine-tuning generally makes use of Proximal Policy Optimization, in [7], Chang et al. focus on more efficient RL algorithms for fine-tuning LLMs on downstream tasks with predefined rewards (e.g. Bleu [34], or reward learned from human preference feedback). As opposed to focusing on the use of RL during training, CodeRL [26] uses it to integrate a critic network that evaluates the functional correctness of the generated programs, enabling the refinement and repair of output programs based on their functional correctness during test time. Conversely to these works, we use RL to train an agent that guides the solution space exploration during LLM-based code generation.

6.3 Program Equivalence

Program equivalence has been studied for a long time. While many of the approaches are based on symbolic reasoning [12, 24], neural approaches have also emerged in recent years. For instance, Kommrusch et al. propose a neural network architecture based on a transformer model to generate proofs of equivalence (expressed as a sequence of rewrites) between program pairs [21]. As opposed to our work, they consider a very conservative set of rewrite rules that are guaranteed to be semantics-preserving, and they only target straight-line programs. The authors previously applied a similar architecture to show equivalence between dataflow graphs with a graph-to-sequence neural model [20]. The approach aims to automate the verification of dataflow graph equivalence by transforming the graphs into sequences and leveraging neural networks for classification. They also presented a method for proving equivalence between complex mathematical expressions [22] using graph-to-sequence neural models. The approach involves representing expressions as graphs and transforming them into sequences, which are then processed by neural networks to classify their equivalence.

A very related software engineering task is code clone detection, i.e. detecting duplicate or similar code. While many techniques and tools have been proposed over the years for code clone detection (see surveys [38, 42]), they might work better for clones with high degree of syntactical similarity (Type-1, Type-2, and Type-3 with high syntactical similarity), but have weaker performance for clones with lower syntactical similarity (Type-3 and Type-4) [43]. More recently, neural approaches have been proposed for the related problem of code clone detection (see survey [27]). Generally, these work leverage neural networks to generate a vector representation for each code fragment and then detect clones by computing the similarities between the vector representations of two code fragments. Dou et al. [10] examines the effectiveness of LLMs in identifying code clones, and reveals that their performance improves with CoT prompting and vector embeddings. As opposed to this work, we are not focusing on code clone detection but rather investigating the exploration of the search space in the context of program transformation and program equivalence. Instead of directly outputting a classification judgment of equivalence, we investigate the ability of the LLM to generate a sequence of rewrites, transforming a source program into the target.

7 Conclusions

We proposed an architecture where an RL agent trained with domain-specific information (i.e., access to a syntax checker, a unit tests checker, and a code similarity checker) can help guide the LLM's space exploration in a non-linear manner. For the task of program equivalence, we compared the reasoning elicited by our architecture against CoT and ToT style prompts, as well as a greedy approach, and reported positive results.

8 Data Availability

The implementation of our prototype tool is available at https://github.com/yoavalon/LLM-Agent.

References

[1] 2024. Anthropic Claude 3. https://www.anthropic.com/news/claude-3-family

[2] 2024. Gemini. https://blog.google/technology/ai/google-gemini-ai/#sundar-note

[3] Yoav Alon and Cristina David. 2022. Using graph neural networks for program termination. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 910–921. https://doi.org/10.1145/3540250.3549095

[4] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Software Eng.* 33, 9 (2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[5] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *CoRR* abs/2303.12712 (2023). https://doi.org/10.48550/ARXIV.2303.12712 arXiv:2303.12712

[6] Ethan Caballero, . OpenAI, and Ilya Sutskever. 2016. Description2Code Dataset. https://doi.org/10.5281/zenodo.5665051

[7] Jonathan D. Chang, Kianté Brantley, Rajkumar Ramamurthy, Dipendra Misra, and Wen Sun. 2023. Learning to Generate Better Than Your LLM. *CoRR* abs/2306.11816 (2023). https://doi.org/10.48550/ARXIV.2306.11816 arXiv:2306.11816

[8] Rudrajit Choudhuri, Dylan Liu, Igor Steinmacher, Marco Gerosa, and Anita Sarma. 2024. How Far Are We? The Triumphs and Trials of Generative AI in Learning Software Engineering. (2024). https://doi.org/10.1145/3597503.3639201

[9] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *CoRR* abs/2110.14168 (2021). arXiv:2110.14168 https://arxiv.org/abs/2110.14168

[10] Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. 2023. Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey. arXiv:2308.01191 [cs.SE]

[11] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. *CoRR* abs/2405.11514 (2024). https://doi.org/10.48550/ARXIV.2405.11514 arXiv:2405.11514

[12] Benny Godlin and Ofer Strichman. 2013. Regression verification: proving the equivalence of similar programs. *Softw. Test. Verification Reliab.* 23, 3 (2013), 241–258. https://doi.org/10.1002/STVR.1472

[13] Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, David Peng, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Shafiq R. Joty, Alexander R. Fabbri, Wojciech Kryscinski, Xi Victoria Lin, Caiming Xiong, and Dragomir Radev. 2022. FOLIO: Natural Language Reasoning with First-Order Logic. *CoRR* abs/2209.00840 (2022). https://doi.org/10.48550/ARXIV.2209.00840 arXiv:2209.00840

[14] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *CoRR* abs/2303.18184 (2023). https://doi.org/10.48550/ARXIV.2303.18184 arXiv:2303.18184

[15] Dave Hulbert. 2023. Using Tree-of-Thought Prompting to boost ChatGPT's reasoning. https://github.com/dave1010/tree-of-thought-prompting. https://doi.org/10.5281/ZENODO.10323452

[16] Guillaume Iooss, Christophe Alias, and Sanjay V. Rajopadhye. 2014. On Program Equivalence with Reductions. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8723)*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer, 168–183. https://doi.org/10.1007/978-3-319-10936-7_11

[17] Md Mahir Asef Kabir, Sk Adnan Hassan, Xiaoyin Wang, Ying Wang, Hai Yu, and Na Meng. 2023. An empirical study of ChatGPT-3.5 on question answering and code maintenance. arXiv:2310.02104 [cs.SE] https://doi.org/10.48550/arXiv.2310.02104

[18] Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tennenholtz. 2022. MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *CoRR* abs/2205.00445 (2022). https://doi.org/10.48550/ARXIV.2205.00445 arXiv:2205.00445

[19] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[20] Steve Kommrusch, Théo Barollet, and L Pouchet. 2020. Equivalence of Dataflow Graphs via Rewrite Rules Using a Graph-to-Sequence Neural Model. *ArXiv* abs/2002.06799 (2020), null. https://www.semanticscholar.org/paper/8bd94d

48eb7a2706391c6cbf11fddf8440099cf9

[21] Steve Kommrusch, Monperrus Martin, and L Pouchet. 2021. Self-Supervised Learning to Prove Equivalence Between Straight-Line Programs via Rewrite Rules. *IEEE Transactions on Software Engineering* null (2021), null. https://doi.org/10.1109/tse.2023.3271065

[22] Steven J Kommrusch, Théo Barollet, and L Pouchet. 2021. Proving Equivalence Between Complex Expressions Using Graph-to-Sequence Neural Models. *ArXiv* abs/2106.02452 (2021), null. https://www.semanticscholar.org/paper/6437aab9075d2a06d277146db10b4f4135432212

[23] Bogdan Korel and Janusz W. Laski. 1990. Dynamic slicing of computer programs. *J. Syst. Softw.* 13, 3 (1990), 187–195. https://doi.org/10.1016/0164-1212(90)90094-3

[24] Shuvendu K. Lahiri, Andrzej S. Murawski, Ofer Strichman, and Mattias Ulbrich. 2018. Program Equivalence (Dagstuhl Seminar 18151). *Dagstuhl Reports* 8, 4 (2018), 1–19. https://doi.org/10.4230/DAGREP.8.4.1

[25] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html

[26] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs.LG]

[27] Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, and Dae-Kyoo Kim. 2022. Deep learning application on code clone detection: A review of current knowledge. *J. Syst. Softw.* 184 (2022), 111141. https://doi.org/10.1016/J.JSS.2021.111141

[28] Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *CoRR* abs/2304.11477 (2023). https://doi.org/10.48550/ARXIV.2304.11477 arXiv:2304.11477

[29] Changshu Liu, Shizhuo Dylan Zhang, and Reyhaneh Jabbarvand. 2024. CodeMind: A Framework to Challenge Large Language Models for Code Reasoning. *CoRR* abs/2402.09664 (2024). https://doi.org/10.48550/ARXIV.2402.09664 arXiv:2402.09664

[30] Jieyi Long. 2023. Large Language Model Guided Tree-of-Thought. *CoRR* abs/2305.08291 (2023). https://doi.org/10.48550/ARXIV.2305.08291 arXiv:2305.08291

[31] OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, , et al. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[32] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pougeum Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. (2024).

[33] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. https://doi.org/10.3115/1073083.1073135

[34] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Pierre Isabelle, Eugene Charniak, and Dekang Lin (Eds.). Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[35] Andrew M. Pitts. 2000. Operational Semantics and Program Equivalence. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures (Lecture Notes in Computer Science, Vol. 2395)*, Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva (Eds.). Springer, 378–412. https://doi.org/10.1007/3-540-45699-6_8

[36] Henrijs Princis, Cristina David, and Alan Mycroft. 2025. Enhancing SQL Query Generation with Neurosymbolic Reasoning. In *The 39th Annual AAAI Conference on Artificial Intelligence*. https://openreview.net/forum?id=sbHQu2EPsm

[37] Andrew Barto Richard Sutton. 2018. *Reinforcement Learning* (second ed.). MIT Press.

[38] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

[39] Abulhair Saparov and He He. 2023. Language Models Are Greedy Reasoners: A Systematic Formal Analysis of Chain-of-Thought. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=qFVVBzXxR2V

[40] Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen-tau Yih, Jason Weston, Jürgen Schmidhuber, and Xian Li. 2023. Large Language Model Programs. *CoRR* abs/2305.05364 (2023). https://doi.org/10.48550/ARXIV.2305.05364 arXiv:2305.05364

[41] Kimya Khakzad Shahandashti, Mithila Sivakumar, Mohammad Mahdi Mohajer, Alvine B. Belle, Song Wang, and Timothy C. Lethbridge. 2024. Evaluating the Effectiveness of GPT-4 Turbo in Creating Defeaters for Assurance Cases. arXiv:2401.17991 [cs.SE]

[42] Utkarsh Singh, Kuldeep Kumar, and DeepakKumar Gupta. 2021. A Study of Code Clone Detection Techniques in Software Systems. In *Proceedings of the International Conference on Paradigms of Computing, Communication and Data Sciences*, Mayank Dave, Ritu Garg, Mohit Dua, and Jemal Hussien (Eds.). Springer Singapore, Singapore, 347–359.

[43] Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 131–140. https://doi.org/10.1109/ICSM.2015.7332459

[44] M. Thenmozhi, Palash Jain, Sai Anand R, and Saketh Ram B. 2017. Analysis of Sudoku Solving Algorithms. *International Journal of Engineering and Technology* 9, 3 (2017). https://www.enggjournals.com/ijet/docs/IJET17-09-03-043.pdf

[45] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML]

[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[47] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html

[48] Morteza Zakeri-Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges. arXiv:2306.16171 [cs.SE]

[49] Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, and Daniel Kroening. 2024. Scalable, Validated Code Translation of Entire Projects using Large Language Models. *CoRR* abs/2412.08035 (2024). https://doi.org/10.48550/ARXIV.2412.08035 arXiv:2412.08035

Received 2024-09-12; accepted 2025-01-14