MAGIC: Multi-Armed Bandit Guided Iterative Code Generation

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown remarkable capabilities in code generation, yet they often struggle with solution diversity and competition-level problems. In this paper, we introduce MAGIC (**M**ulti-**A**rmed bandit **G**uided **I**terative **C**ode generator), an approach that formalizes plan selection in LLM-based code generation as a Multi-Armed Bandit (MAB) problem, enabling systematic exploration of diverse solution strategies. The method disentangles the generation process into four phases: explicit plan generation, plan selection, code implementation, and code refinement. By treating each potential plan as an arm in the MAB framework, we employ an adapted Upper Confidence Bound (UCB) algorithm that balances the exploration of different solution strategies with the exploitation of promising plans. With the purpose of constraining code refinement to current plans to ensure focused solution space exploitation, we propose to formalize the plans as code skeletons. Experiments on HumanEval, HumanEval+, CodeContest, and APPS demonstrate significant improvements over existing methods, with pass@1 up to 97.0% on HumanEval and 45.5% on CodeContest using GPT-4o. Through variance-based diversity metrics, we show that MAGIC substantially increases solution diversity, particularly benefiting performance on challenging competitive programming tasks.

1 Introduction

In the past few years, large language models (LLMs) have made significant strides in various fields (OpenAI, 2023). Coding, as a critical application area, holds particular importance, making the coding capabilities of LLMs especially noteworthy. Early research efforts (Rozière et al., 2023; Austin et al., 2021) predominantly focused on pre-training and fine-tuning LLMs on large-scale code datasets to improve their coding proficiency. Recently, the rise of general-purpose LLMs with advanced reasoning ability has shifted attention to using these models as agents (Xia et al., 2024; Xi et al., 2023) for iterative code generation and algorithmic design. Furthermore, scaling inference compute, through methods like iterative code refinement with external feedback (Shinn et al., 2024) and tree-structured searching (Song et al., 2024), has proven effective in improving code quality. These advancements highlight the potential for addressing increasingly complex coding challenges (Li et al., 2022).

Current state-of-the-art methods primarily rely on advanced search algorithms to enhance performance. For instance, PG-TD (Zhang et al., 2023b) integrates token-level lookahead search using a planner into the Transformer decoding process, while FunCoder (Chen et al., 2024a) employs dynamic function decomposition to recursively divide tasks into simpler sub-functions, optimizing the search for these sub-components. Additionally, Monte Carlo Tree Search (MCTS) based approaches (Zhou et al., 2024; Hao et al., 2023) effectively balance exploration and exploitation during code generation, demonstrating their utility in navigating complex solution spaces.

Despite the advancements of current search algorithms, several limitations remain. These methods often lack systematically designed structures to handle diverse reasoning processes. For instance, while MCTS approaches (Zhou et al., 2024; Hao et al., 2023) effectively leverage information from ancestor nodes, they may face challenges in utilizing information across different branches. This limitation can lead to overlapping generation across branches, reducing both efficiency and diversity. Furthermore, existing approaches fail to fully leverage the inherent language capabilities of LLMs. Studies (Tang et al., 2023) have shown that while LLMs excel in semantic reasoning, they struggle with symbolic reasoning. Another line of research (Jiang et al., 2023) highlights the advantages of

explicit planning, where a model first generates a structured plan before taking action, leading to notable performance improvements. Collectively, these limitations prevent current methods from fully realizing the potential of LLMs, constraining their ability to handle diverse reasoning tasks and systematically explore solution spaces.

Motivated by these limitations, we propose MAGIC (**M**ulti-**A**rmed bandit **G**uided **I**terative **C**ode generator), a novel framework designed to address the challenges of solution diversity and systematic exploration in LLM-based code generation. Specifically, MAGIC structures the code generation process into four distinct phases: explicit plan generation, plan selection, code implementation, and code refinement. In the plan generation phase, inspired by previous work (Wang et al., 2024), we leverage the strong semantic planning capabilities of LLMs to generate an overall plan in natural language. Beyond generating a textual plan, we also prompt the LLM to produce code skeletons based on the plan, with detailed reasons provided in Sec. 4. To systematically explore diverse solution strategies, we formalize plan selection as a Multi-Armed Bandit (MAB) problem. Each potential plan is treated as an arm in the MAB framework, and we employ an adapted Upper Confidence Bound (UCB) algorithm (Auer et al., 2002) to balance the exploration of new strategies with the exploitation of promising ones. Furthermore, building on previous findings (Chen et al., 2024b; Zhong et al., 2024) that LLMs can refine their outputs when provided with external feedback, we integrate a refinement mechanism into our framework to iteratively enhance the generated solutions.

To evaluate our method, we conduct extensive experiments on a variety of code generation benchmarks, ranging from simple foundational tasks (Chen et al., 2021; Liu et al., 2023) to competition-level challenges (Hendrycks et al., 2021; Li et al., 2022). Notably, MAGIC achieves a 97.0% Pass@1 on HumanEval (Chen et al., 2021) and a 45.5% Pass@1 on CodeContest (Li et al., 2022), setting new SOTA performance on both benchmarks. These results demonstrate that our method excels at handling tasks of varying complexity, showcasing its effectiveness across both simple and highly challenging problems.

Our contributions can be summarized as follows:

- We identify key limitations in existing LLM-based code generation methods, including insufficient solution diversity, repeated sampling of suboptimal paths, and underutilization of LLMs' semantic reasoning capabilities.
- We propose MAGIC, a novel framework that formulates LLM-based code generation as a Multi-Armed Bandit (MAB) problem. MAGIC enables systematic exploration and exploitation of diverse solution strategies, addressing limitations in existing methods.
- Extensive experiments on multiple benchmarks, including HumanEval, HumanEval+, CodeContest, and APPS, demonstrate the effectiveness of MAGIC. Notably, MAGIC achieves state-of-the-art performance, with a 97.0% Pass@1 on HumanEval and a 45.5% Pass@1 on CodeContest.

2 Related Works

2.1 AI for Code

Code-related tasks have become a critical area of focus in artificial intelligence, encompassing a wide range of applications such as code generation, completion, translation, and summarization. Early works explored various aspects of these tasks, leveraging models like CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021) to understand code syntax and semantics, enabling advancements in tasks such as code search and summarization.

The development of large-scale pre-trained models, such as GPT-4 (OpenAI, 2023) and Llama (Touvron et al., 2023) , has significantly expanded the scope of code-related tasks. These general-purpose models excel not only in code generation but also in reasoning and refining code through natural language interfaces. Additionally, fine-tuned models (Ni et al., 2024; Rozière et al., 2023) can achieve further specialization for code challenges. Building upon these advancements, this work investigates new approaches to enhance code-related tasks, leveraging the strengths of LLMs for systematic improvements.

2.2 Language Model Agent

With the advancement of research, there is growing attention on the reasoning, searching, and interaction capabilities of LLMs, which have become foundational to most SOTA approaches. Techniques like Chain-of-Thought (CoT) prompting (Wei et al., 2022) enable step-by-step reasoning by

breaking problems into intermediate steps, while methods like Least-to-Most prompting (Zhou et al., 2023) focus on first decomposing complex problems into simpler subproblems and then sequentially solving them. Leveraging LLMs' strong ability for interacting with external environments and in-context learning, advanced approaches such as ReAct (Yao et al., 2023b) and Self-Edit (Zhang et al., 2023a) refine answers iteratively using external feedback. To explore larger solution spaces, methods like Tree-of-Thought (Yao et al., 2023a) and MCTS-based techniques (Hao et al., 2023; Zhou et al., 2024) utilize tree structures to generate diverse outputs, significantly enhancing LLMs' problem-solving capabilities on tasks requiring extensive exploration.

2.3 Code Agent

Recent advances in coding agents have introduced innovative frameworks to enhance code generation. AgentCoder (Huang et al., 2024) employs a multi-agent system where programmer, test designer, and executor agents collaborate to iteratively refine code based on feedback. LDB (Zhong et al., 2024) enables debugging by segmenting programs into basic blocks and tracking runtime execution, allowing efficient error pinpointing. FunCoder (Chen et al., 2024a) uses a divide-and-conquer strategy with dynamic function decomposition and functional consensus to tackle complex requirements. Recently, SFS (Light et al., 2024) formulates code generation as a black-box optimization problem and introduces an evolutionary search method that enhances diversity and feedback exploitation. All these approaches address limitations in traditional methods and significantly improve coding performance across diverse benchmarks.

3 Problem Formulation

3.1 Plan Space and Code Space Definitions

To formalize the code generation process, we define two key spaces: the **Plan Space** and the **Code Space**.

Plan Space Definition. Let $\mathcal{T}$ denote the Plan Space, which contains all possible plans that can be generated by the LLM for solving a given coding problem x. A plan $T \in \mathcal{T}$ is a structured representation of the steps or strategies required to implement the solution. Each plan T serves as a blueprint for generating code and can be represented as a sequence of high-level actions or a skeleton structure:

$$\mathcal{T} = \{T_1, T_2, \ldots, T_k \mid T_i \sim P_{LLM}(T|x)\}.$$

Code Space Definition. Let $\mathcal{Y}$ denote the Code Space, the set of all possible code implementations that can be generated by the LLM. Formally, we define:

$$\mathcal{Y} = \{y_1, y_2, \ldots, y_k \mid y_i \sim P_{\text{LLM}}(y \mid x)\}$$

Linking Plan Space and Code Space. The Plan Space $\mathcal{T}$ and the Code Space $\mathcal{Y}$ are intrinsically connected in the code generation process. For a given coding problem x, the relationship between plans and code implementations can be expressed as:

$$P(y|x) = \mathbb{E}_{T \sim P(T|x)} \left[P(y|x, T) \right] \quad (1)$$

where $P(T|x)$ is the probability of selecting a plan T given X, and $P(y|x, T)$ is the probability of generating code Y conditioned on both the coding problem x and the selected plan T. This formulation highlights that the overall likelihood of generating code y is influenced by both the distribution over plans and the conditional distribution of code generation.

3.2 Code Refinement Process

The code refinement process can be formalized as an iterative procedure, where at each refinement step j, the current code y_j is improved to produce a refined version y_{i+1}. This process can be expressed as:

$$P(y_{j+1}|x, T) = \mathbb{E}_{y_j \sim P(y_j|x,T)} P(y_{j+1}|y_j, x, T), \quad (2)$$

where $P(y_j|x)$ represents the distribution of code implementations at step j, and $P(y_{j+1}|y_j, x)$ models the conditional probability of generating the next refined version based on the current code y_j and the problem x.

To evaluate the quality of the refined code, we define a correctness function $C : \mathcal{Y} \to [0, 1]$:

$$C(y) = \begin{cases} 1 & \text{if } y \text{ passes all test cases,} \\ \frac{n_p}{n_t} & \text{otherwise,} \end{cases} \quad (3)$$

where n_p is the number of passed test cases, and n_t is the total number of test cases. This correctness function provides a quantitative measure of how well the refined code satisfies the problem requirements.

Figure 1: An overview of MAGIC: When given a problem description, an LLM is prompted to generate multiple diverse plans in natural language. Then MAGIC formalizes plan selection as an MAB problem by treating different plans as bandit arms with different reward distributions. By utilizing UCB-P as the plan-selection metric, the model can effectively search for an optimal code solution in code space with a balance between exploration and exploitation.

4 Methodology

At a high level, we disentangle the code generation process into four key components: plan generation, plan selection, code implementation and code refinement. Below, we explain each component in detail.

4.1 Plan Generation

Inspired by previous works (Tang et al., 2023; Wang et al., 2024), which have shown the benefits of natural language guidance in code generation, we adopt the plan-then-solve approach(Zhou et al., 2023; Jiang et al., 2023; Wang et al., 2023). Instead of embedding the reasoning path directly within the generated code as comments, we explicitly generate standalone plans that serve as blueprints for code implementation. Since only a finite set of discrete plans can be generated, the previous Eq. 1 can be rewritten as:

$$P(y|x) = \sum_{T \in \mathcal{T}} P(T|x)P(y|x,T) \qquad (4)$$

To ensure plan diversity and reduce content overlap, which may lead to redundant sampling, we propose an iterative plan generation strategy. In each iteration, we prompt the LLM to generate a plan that differs from the previously generated plans. Each plan generated in the next round is conditioned on the problem x and all previously generated plans. Formally: $T_{i+1} \sim P(T|x, T_1, T_2, \ldots, T_i)$. This iterative approach ensures that each new plan is informed by previous plans, increasing diversity and reducing redundancy.

In addition to generating textual plans, we also generate a code skeleton based on the plan, which includes function heads and docstrings. This serves two purposes: (1) decomposing the problem into smaller functions helps the LLM better understand and reason about the code, and (2) it provides a structured regularization for the code refinement process, making refinements more robust, as will be discussed in Sec.4.3.

4.2 Plan Selection

As demonstrated in Eq. 4, to maximize the likelihood of obtaining a correct solution, it is crucial to assign greater weight to the most promising plan. Therefore, our objective is to identify and select the optimal plan. Since the plan is already fixed in plan generation phase, we can formalize the plan selection process as a Multi-Armed Bandit (MAB) problem. Building on the pre-defined UCB algorithm, we propose **UCB-P** with an additional penalty term to adapt better to our framework. For the i-th plan, the UCB-P score is given by:

$$\text{UCB-P}_i(t) = \hat{\mu}_i + \lambda_1 \cdot \sqrt{\frac{2\log(t)}{n_i}} - \frac{m}{\lambda_2 n_i}, \quad (5)$$

where $\hat{\mu}_i$ is the estimated reward mean, reflecting the historical performance of the plan and encourages exploitation of plans with higher average rewards. The second term $\sqrt{\frac{2\log(t)}{n_i(t)}}$ can be considered as exploration bonus, plans that have been tried fewer times receive a higher value, encouraging the framework to gather more informa-

tion about them. With both terms, UCB can effectively balance between exploitation (favoring high-reward plans) and exploration (prioritizing less-tried plans). In our problem setting, the reward for each plan is set as the correctness function C, which is defined in Equation. 3.

Penalty term ($-\frac{m}{\lambda_2 n_i}$): During experiments, we observed that in some cases, when the code frequently failed during the refinement phase, the LLM tended to take shortcuts by generating solutions that could easily pass basic public test cases but still failed to fulfill the original requirements. To mitigate this issue, we introduce a penalty term based on the number of refinements m applied to the plan. This penalty term ensures that plans requiring excessive refinements reveive progressively lower UCB-P values, thereby discouraging strategies that merely optimize for passing public test cases rather than comprehensively addressing problem requirements, which can significantly prevent local optimal solutions. Additionally, a plan requiring frequent refinements is indicative of poor quality, as a high-quality plan should guide the LLM toward generating correct solutions with minimal adjustments. The hyperparameter λ_2 scales the impact of the penalty.

Our customized UCB-P formula ensures that the selection of plans is guided by a balance of past performance, the potential for improvement, and the need to avoid over-reliance on refinement, thereby promoting a better exploration of the solution space. The detailed plan selection algorithm can be checked in Appendix.

4.3 Code Implementation and Refinement

Code Implementation: Similar to how plans are conditional on previous plans, each code implementation under a plan should also be conditioned on previous implementations. We adopt similar prompt strategy to ask the LLM tries to generate different code implementation from previous generated code. This can be formalized as: $y_{i+1} \sim P(y_{i+1}|x, T, y_1, y_2, \ldots, y_i)$. This strategy guarantees that even under the same plan, successive code implementations maintain sufficient diversity, enabling exploration of different implementation variants within the plan.

Code Refinement: When the initially generated code cannot fully pass public test cases, we decide to refine it with execution feedback. When refining the code, we shift our focus from maintaining implementation diversity to optimizing the existing code. We observe that if no restrictions are imposed during the code refinement process, the answers generated by the LLM in the refinement phase often lack stability. Specifically, when initial attempts fail to pass the public test cases, the LLM may attempt entirely new implementation, which can lead to significant overlap with answers generated under other plans. This not only undermines the exploitation of the current code but also reduces overall diversity across plans. Algorithms like MCTS (Zhang et al., 2023b; Zhou et al., 2024) also suffer from similar situation, due to the lack of information sharing across different branches.

To handle this limitation, we propose using code skeletons to constrain the search space during the refinement phase. Each code skeleton defines a fixed structure for the solution, which consists of predefined function heads and their description in natural language format. By enforcing these skeletons, the refinement process is restricted to modifying the function implementations while keeping the overall structure fixed. This approach assures that each revision builds on the preceding solutions, retaining the integrity of the current plan and focusing on exploiting its potential.

This approach can be further understood through the concept of trust region optimization (Schulman et al., 2015). By constraining the changes between the previously generated code and the refined code, we guide the LLM to focus its adjustments on specific, bounded regions of the solution space. Specifically, the LLM is allowed to modify only the parts of the code it identifies as incorrect, while the rest remains fixed. This effectively limits the search to a "trust region" around the current solution, preventing disruptive changes that might undermine the stability of the refinement process.

By limiting the magnitude of modifications, this approach encourages the LLM to focus on incremental improvements to the current plan, enhancing its effectiveness while avoiding disruptive changes that could compromise the stability and coherence of the solution.

5 Experiments

5.1 Dataset

For extensive evaluation, we have used four benchmark datasets: two from basic programming, which are **HumanEval** (Chen et al., 2021) and **HumanEval+** (Liu et al., 2023). Another two are **CodeContest** (Li et al., 2022) and **APPS**

Table 1: Performance Across Different Approaches and Datasets.

Model	Framework	Basic Programming		Competitive Programming			
		HE	HE+	CC	APPS-I	APPS-Int	APPS-C
GPT-4o	Direct	89.6%	84.1%	19.4%	73.3%	48.3%	18.3%
	Reflexion	95.1%	87.2%	40.6%	80.0%	56.7%	33.3%
	LATS	95.7%	87.2%	42.4%	81.7%	58.3%	36.7%
	LDB	96.3%	89.0%	41.8%	**86.7%**	56.7%	35%
	MAGIC	**97.0%**	**90.2%**	**45.5%**	85.0%	**60.0%**	**38.3%**
GPT-4o-mini	Direct	87.2%	80.5%	10.9%	63.3%	40.0%	8.3%
	Reflexion	93.9%	84.8%	21.8%	70.0%	50.0%	23.3%
	LATS	94.5%	85.4%	23.6%	70.0%	51.7%	26.7%
	LDB	93.3%	84.8%	19.4%	73.3%	46.7%	25.0%
	MAGIC	**94.5** %	**86.0%**	**27.3%**	**73.3%**	**53.3%**	**26.7%**
Llama3.1-8B-Instruct	Direct	64.6%	57.9%	4.2%	25.0%	1.7%	0.0%
	Reflexion	79.9%	70.1%	11.5%	36.7%	3.3%	3.3%
	LATS	82.3%	72.6%	12.1%	40.0%	6.7%	3.3%
	LDB	82.3%	72.0%	10.9%	41.7%	5.0%	**5.0%**
	MAGIC	**83.5%**	**75.0%**	**12.1%**	**41.7%**	**6.7%**	3.3%

HE = HumanEval, HE+ = HumanEval+, CC = Code Contest, APPS-C = APPS (Competition), APPS-I = APPS (Introduction), APPS-Int = APPS (Interview)

(Hendrycks et al., 2021), from complex competitive programming domains. HumanEval is a dataset of 164 problems that only requires to complete a single given function. HumanEval+ is an advanced version of it, which contains more hidden test cases. While for CodeContest and APPS, they only provide requirements in natrual language, and the LLM needs to output an entire code. CodeContest contains 165 challenging competition level coding problems. For APPS we choose 60 competition-level problems, 60 interview-level problems, and 60 introductory-level problems, which follows the choice of previous works (Olausson et al., 2024).

5.2 Baseline

We introduce the following baselines: **Direct** means instructs the model to generate code directly from the input problem; **Reflexion** utilizes solution's execution feedback to generate self-reflections. The reflections are used to iteratively refine the solution. **LATS** augments language models with Monte Carlo Tree Search to enable structured exploration and planning, incorporating environment feedback and self-reflection to iteratively refine decisions. **LDB** segments generated programs into basic blocks and leverages runtime execution information to iteratively debug and refine code by verifying correctness block by block.

5.3 Implementation Details

Following common practice in code generation evaluation, we applied pass@1 (Chen et al., 2021) as our evaluating metric. For each question, we can select only one code candidate for final evaluation with hidden test cases. To ensure fair comparison accross different approaches, we fix the total number of samples to 40 for each problem across all approaches, excluding zero-shot direct sampling. Notably, for LDB, each sampling round consists of an initial solution generation followed by one potential debugging iteration (counting as 2 samples). To fully utilize sampling budget, we combine LDB with resampling - if the debugged solution fails, we initiate a new round with a fresh solution rather than continuing to debug the same code. For our approach, we set the number of plans to 5, and the refinement chance to 1, we can generate and refine 20 code solutions within the 40-sample budget.

5.4 Main Results

From Table. 1, we observe that our approach achieves superior performance across benchmarks ranging from simple to hard (97.0% pass@1 on HumanEval, 45.5% pass@1 on CodeContest). The improvements are particularly pronounced on complex tasks - for GPT-4o, MAGIC outperforms the direct prompting baseline by 26.1% on CodeContest. Notably, LATS and LDB also demonstrate strong performance. We also notice that while MAGIC excels with larger models, its effectiveness diminishes when applied to Llama-8B, indicating that our method heavily relies on strong models' reasoning ability.

5.5 Analysis of Number of Plans

Model	HumanEval	CodeContest
Ours (5 plans)	97.0%	45.5%
Ours (2 plans)	95.7%	41.8%
Ours (1 plan)	94.5%	38.8%

Table 2: Pass@1 on HumanEval and CodeContest using GPT-4o with different number of plans

In this section, we applied our approach using different number of plans with GPT-4o, aiming to analyze the impact of varying plan counts on the overall performance, as the number of plan is closely related to solution diversity. The experiment results are shown in Table 2. We set the number of plans to 5, 2, and 1. For each plan, the LLM generated code four times and performed one refinement iterations. Our experimental results demonstrate that the impact of multiple plans varies significantly across problem difficulty. For straightforward programming tasks in HumanEval, the performance remains relatively stable regardless of the number of plans used (from 97.0% to 94.5% pass@1). However, on CodeContest's competition-level problems, we observe a substantial performance drop from five-plan to single-plan approaches (from 45.5% vs 38.8% pass@1). This indicates that diverse planning paths become increasingly crucial when tackling complex programming challenges, where exploring multiple solution strategies helps navigate intricate problem spaces and identify optimal implementations.

5.6 Ablation Study

Ablation	HumanEval	CodeContest
Ours	94.5%	27.3%
w/o UCB	94.5%	25.4%
w/o Test Cases	92.7%	18.2%
w/o Constraint	93.3%	23.0%

Table 3: Pass@1 on HumanEval and CodeContest using GPT-4o-mini for ablation study

Our ablation studies reveal several key insights about model performance with GPT-4o-mini. Removing UCB-based plan selection leads to minimal impact on HumanEval (94.5% pass@1) but affects CodeContest performance (25.4% vs 27.3% pass@1), as UCB helps balance exploration and exploitation among promising solution paths. For test case ablation, while HumanEval performance remains stable (92.7%), CodeContest sees a significant drop (18.2%). This disparity occurs because for simple HumanEval problems, the LLM can reason effectively and generate appropriate test cases. However, for complex CodeContest problems, the LLM can only generate basic test cases, missing edge cases critical for competition-level problems. Finally, we remove constraints by no longer prompt LLM to generate diverse plans and code, which leads to performance degradation (93.3% and 23.0% respectively) due to repeated sampling and unstable refinement, which reduces sample efficiency within our fixed computation budget.

6 Code Space Analysis

6.1 Embedding with UniXcoder

UniXcoder (Guo et al., 2022) is a pre-trained model for programming languages. Notably, UniXcoder enhaces code representation by integrating cross-modal contents such as Abstract Syntax Trees (ASTs) and code comments, which makes the model better capture the logic and semantic meaning of the code. In our experiment, we utilize UniXcoder to embed the generated code due to its strong code representation ability.

6.2 Code Diversity

Code diversity refers to the extent to which multiple generated code solutions differ from each other in terms of structure, logic, and implementation details. Higher diversity implies the exploration of varied approaches to solving the same problem, enhancing the likelihood of finding optimal solutions, instead of getting stuck in local optimal solutions.

Variance-Based Diversity: One straightforward way of quantifying code diversity is to measure the variance of the generated code embeddings. A higher variance indicates greater diversity in the solution space, as it reflects a broader distribution of solutions rather than repeated sampling similar implementations. Formally, given a set of embedded code representations $\{e_1, e_2, \ldots, e_N\}$ of coding problem x, we define the code solution variance of problem x as:

$$\text{Var}(x) = \frac{1}{N}\sum_{i=1}^{N} \|e_i - \bar{e}\|^2, \text{where} \quad \bar{e} = \frac{1}{N}\sum_{i=1}^{N} e_i$$

Pairwise Similarity: From another perspective, we can also quantify the code similarity between each code solutions, since is has the opposite meaning to variance, which can also reflect code diversity. For 2 different code embeddings $\{e_i, e_j\}$, we define the pairwise cosine similarity between e_i and e_j as $sim(e_i, e_j)$, which is the cosine similarity between the 2 embeddings. Now, given a coding problem x and its solution set $\{e_1, e_2, \ldots, e_N\}$, we can define the pairwise code solution similarity of problem x as:

$$S(x) = \frac{2}{N(N-1)} \sum_{i<j} sim(e_i, e_j).$$

In out setting, we compute the code embeddings with the GPT-4o-mini's generated code from CodeContest, the results can be shown in Table 4. Based on the experiment results presented in Table. 4, we observe the Resample method exhibits the highest code diversity, since it generates solutions independently without refining any previous code. While this results in greater exploration in code space, the lack of exploitation limits the performance of Resample. Among methods that containing phase to refine existing code, our approach MAGIC achieves the highest diversity, enabling broader exploration of potential optimal solutions while also leveraging well-performing code for effective exploitation.

Method	Variance	Pairwise Similarity
Resample	**0.403**	**0.885**
Reflexion	0.223	0.931
LATS	0.262	0.925
MAGIC	0.336	0.911

Table 4: Embedding Variance for Different Methods

6.3 Code Space Visualization

In this section, we want to analyze how different searching algorithms explore the code space. For a random problem, we sample 40 code solutions with different approaches. Then, we use UniXcoder as the embedding model and apply t-SNE (van der Maaten and Hinton, 2008) to project the embeddings into a 2D space. For the MCTS algorithm, we adopt LATS as the base method. For the resampling approach, we generate code solutions for 40 times in a zero-shot manner while setting the temperature to 1.

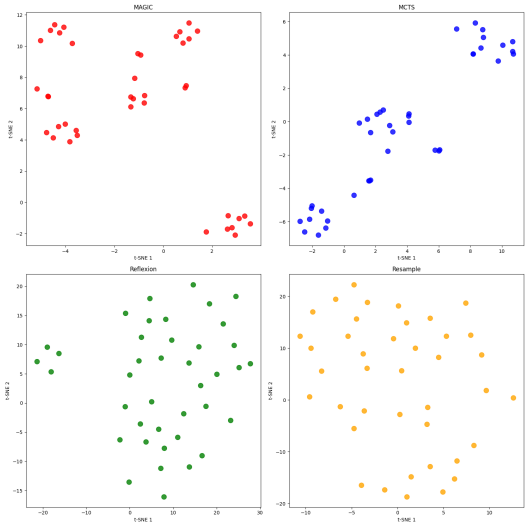

Figure 2: Visualization of Different Approaches' Solution Embeddings.

We discovered that the embedding space for Reflexion and Resample does not exhibit clear clustering properties. This is because Reflexion can only refine code based on previously generated solutions, while Resample generates solutions independently without leveraging prior information. This observation indicates that both methods lack a systematic approach to exploring the solution space. In contrast, for our approach (MAGIC) and the MCTS-based LATS method, the embedding visualization reveals distinct clustering patterns, especially for our approach. This indicates that our method effectively explores the potential optimal solution space in a structured manner.

7 Conclusion

In this paper, we introduced Multi-Armed Bandit Guided Iterative Code Generator (MAGIC), a novel code generation framework that formulates plan selection as a Multi-Armed Bandit (MAB) problem. By leveraging the strong language planning capabilities of LLMs and our customized UCB-P term, MAGIC efficiently avoids redundant sampling and achieves state-of-the-art performance, with notable gains in solution diversity and competitive programming tasks. We also conducted comprehensive ablation studies to understand the contribution factors behind MAGIC's strong performance. Code space analysis reveals MAGIC's property of generating more diverse code. Our findings highlight the importance of structured exploration in LLM-based code generation, offering insights into more efficient and adaptive search strategies.

Limitations

Despite MAGIC's strong performance, several limitations warrant discussion. First, the framework heavily relies on public test cases for code refinement. Without these, LLMs can only generate low quality test cases due to the limited reasoning ability of current models. Additionally, while MAGIC shows substantial improvements with advanced models like GPT-4o, the benefits become less with weaker models like Llama-8B, which indicates MAGIC heavily relies on the reasoning ability of LLMs. Moreover, although we introduce code skeleton into our framework to constrain code refinement, our current implementation does not leverage more advanced property of modularized code, such as conducting unit tests for each function.

References

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024a. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In *NeurIPS*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024b. Teaching large language models to self-debug. In *ICLR*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *EMNLP*, pages 1536–1547.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *ACL*, pages 7212–7225.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. In *EMNLP*, pages 8154–8173.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *Preprint*, arXiv:2105.09938.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv:2312.13010*.

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jonathan Light, Yue Wu, Yiyou Sun, Wenchao Yu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, Wei Cheng, and 1 others. 2024. Scattered forest search: Smarter code space exploration with llms. *arXiv preprint arXiv:2411.05010*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *NeurIPS*.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: Teaching large language models to reason about code execution. In *ICML*.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *ICLR*.

OpenAI. 2023. GPT-4 technical report. *arXiv:2303.08774*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code. *arXiv:2308.12950*.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *ICML*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Jialin Song, Jonathan Raiman, and Bryan Catanzaro. 2024. Effective large language model debugging with best-first tree search. *Preprint*, arXiv:2407.19055.

Xiaojuan Tang, Zilong Zheng, Jiaqi Li, Fanxu Meng, Song-Chun Zhu, Yitao Liang, and Muhan Zhang. 2023. Large language models are in-context semantic reasoners rather than symbolic reasoners. *arXiv:2305.14825*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *arXiv:2302.13971*.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of Machine Learning Research*, pages 2579–2605.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. 2024. Planning in natural language improves llm search for code generation.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *ACL*, pages 2609–2634.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *ACL*, pages 8696–8708.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, and 10 others. 2023. The rise and potential of large language model based agents: A survey. *arXiv:2309.07864*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: deliberate problem solving with large language models. In *NeurIPS*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing reasoning and acting in language models. In *ICLR*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *ACL*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023b. Planning with large language models for code generation. In *ICLR*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *ACL*, pages 851–870.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning acting and planning in language models. *ICML*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *ICLR*.

A MAGIC Algorithm

Algorithm 1 Code Generation with UCB-P Guided Plan Selection

Input: Number of plans k, termination criterion $\mathcal{C}$, coding problem x
Output: Refined code satisfying correctness conditions
Generate k distinct plans $\{T_1, \ldots, T_k\}$ using $P(T \mid x)$.
for $i = 1, \ldots, k$ **do**
    Generate an initial code implementation y_i based on plan T_i.
end for
while $\mathcal{C}$ is not satisfied **do**
    Select a plan T_i based on the greatest UCB value.
    Generate code based on the selected plan T_i.
    if the generated code fails public test cases **then**
        Refine the code implementation.
    end if
    Update the UCB value for plan T_i using the observed reward.
end while

B Apps Tasks Used for Our Evaluation.

This subset of APPS is the same as previous work's (Olausson et al., 2024) subset.

Difficulty	Task IDs
Introductory	4004, 4058, 4063, 4065, 4100, 4108, 4117, 4155, 4164, 4182, 4193, 4195, 4211, 4217, 4241, 4249, 4270, 4275, 4281, 4293, 4333, 4347, 4350, 4356, 4409, 4426, 4431, 4450, 4465, 4484, 4498, 4505, 4507, 4514, 4544, 4553, 4586, 4610, 4662, 4663, 4667, 4677, 4681, 4704, 4716, 4741, 4750, 4786, 4787, 4801, 4855, 4862, 4864, 4870, 4873, 4890, 4897, 4952, 4966, 4984
Interview	0004, 0013, 0033, 0056, 0073, 0074, 0089, 0091, 0124, 0131, 0139, 0162, 0166, 0183, 0186, 0191, 0199, 0205, 0249, 0253, 0268, 0274, 0300, 0304, 0341, 0342, 0413, 0427, 0434, 0466, 0467, 0496, 0501, 0511, 0537, 0564, 0571, 0575, 0579, 0592, 0597, 0626, 0637, 0676, 0704, 0728, 0757, 0765, 0788, 0794, 0804, 0805, 0811, 0829, 0879, 0904, 0915, 0925, 0937, 0948
Competition	3017, 3019, 3054, 3062, 3063, 3066, 3070, 3077, 3083, 3097, 3117, 3135, 3161, 3186, 3209, 3220, 3286, 3287, 3323, 3335, 3353, 3355, 3371, 3375, 3376, 3388, 3404, 3411, 3433, 3441, 3445, 3470, 3481, 3484, 3548, 3557, 3605, 3609, 3634, 3635, 3671, 3679, 3709, 3754, 3769, 3792, 3798, 3799, 3804, 3810, 3819, 3823, 3836, 3843, 3849, 3876, 3913, 3934, 3972, 3974

Table 5: APPS tasks used for our evaluation (60 Introductory, 60 Interview, and 60 Competition problems).

C Example Generated Plan and Code Skeleton

In this section we also provide a basic example of how our generated plan and code skeleton look like. The detailed information can be checked on the next 2 pages.

D AI Usage

ChatGPT is used to correct grammatical errors and polish the paper writing.

```
To solve this problem, we need to efficiently handle two types of operations on a tree
structure with rooms, each having an illusion rate. The operations involve updating
the illusion rate of a room and computing the minimum energy required to travel between
two rooms.

Here's a plan to tackle this problem:

1. **Parse Input:** Start by reading the number of rooms `n` and the number of queries
`q`. Then, read the initial illusion rates for each room. Next, read the `n-1`
tunnels that form the tree structure connecting the rooms. Finally, read the queries
which can either update an illusion rate or ask for the minimum energy path.

2. **Data Structure Selection:**
   - Use an adjacency list to represent the tree. This will help in efficiently
   navigating through the tree.
   - Maintain a list to store the illusion rates of each room, which can be updated in
   constant time.

3. **Handling Queries:**
   - For type 1 queries, update the illusion rate of the specified room.
   - For type 2 queries, calculate the minimum energy required to travel from room `u`
   to room `v`. Since the structure is a tree, use Depth-First Search (DFS) or
   Breadth-First Search (BFS) to find the path and calculate the energy for each
   segment of the path using the given formula: `max(|a_x + a_y|, |a_x - a_y|)`.

4. **Output Results:** For each type 2 query, compute the result and print it.
```

Figure 3: Example generated plan.

```
def main():
    """
    Main function to handle input parsing, processing queries, and outputting results.

    Input:
        - First line: Two integers n (number of rooms) and q (number of queries).
        - Second line: n integers representing the initial illusion rates of the rooms.
        - Next n-1 lines: Each contains two integers indicating a tunnel between
        two rooms.
        - Next q lines: Each contains a query, either type 1 (update illusion rate)
        or type 2 (calculate minimum energy).

    Output:
        - For each type 2 query, print the minimum sum of energy required to move from
        room u to room v.
    """

def update_illusion_rate(illusion_rates, room_index, new_rate):
    """
    Updates the illusion rate of a specific room.

    Input:
        - illusion_rates (List[int]): Current list of illusion rates for all rooms.
        - room_index (int): Index of the room whose illusion rate is to be updated.
        - new_rate (int): New illusion rate for the specified room.

    Output:
        - None. The function updates the illusion_rates list in place.
    """

def calculate_minimum_energy(tree, illusion_rates, start, end):
    """
    Calculates the minimum energy required to travel from start room to end room.

    Input:
        - tree (Dict[int, List[int]]): Adjacency list representing the tree of rooms.
        - illusion_rates (List[int]): List of current illusion rates for each room.
        - start (int): The starting room index.
        - end (int): The ending room index.

    Output:
        - min_energy (int): The minimum energy required to travel from start to end
        room.
    """

def find_path_and_energy(tree, illusion_rates, current, target, visited):
    """
    Helper function to find the path and compute energy using DFS or BFS.

    Input:
        - tree (Dict[int, List[int]]): Adjacency list representing the tree of rooms.
        - illusion_rates (List[int]): List of current illusion rates for each room.
        - current (int): Current room index in the traversal.
        - target (int): Target room index to reach.
        - visited (Set[int]): Set of visited nodes to avoid cycles.

    Output:
        - energy (int): Total energy calculated for the path from current to target.
    """
```

Figure 4: Example generated code skeleton.