SCATTERED FOREST SEARCH: SMARTER CODE SPACE EXPLORATION WITH LLMS

Jonathan Light*¹, Yue Wu², Yiyou Sun³, Wenchao Yu³, Yanchi liu³, Xujiang Zhao³, Ziniu Hu⁴, Haifeng Chen³, Wei Cheng^{⊠3}

¹Rensselaer Polytechnic Institute, ²Princeton University, ³NEC Laboratories America, ⁴XAi lij54rpi.edu

ABSTRACT

We propose a novel approach to scaling LLM inference for code generation. We frame code generation as a black box optimization problem within the code space, and employ optimization-inspired techniques to enhance exploration. Specifically, we introduce SCATTERED FOREST SEARCH to enhance solution diversity while searching for solutions. Our theoretical analysis illustrates how these methods avoid local optima during optimization. Extensive experiments on HumanEval, MBPP, APPS, CodeContests, and Leetcode reveal significant performance improvements. For instance, our method achieves a pass@1 rate of 67.1% on HumanEval+ and 87.2% on HumanEval with GPT-3.5, marking improvements of 8.6% and 4.3% over the state-of-the-art, while also halving the iterations needed to find the correct solution. Furthermore, our method scales more efficiently than existing search techniques, including tree search, line search, and repeated sampling.

1 Introduction

Recent work highlights the effectiveness of scaling inference compute over training compute (Snell et al., 2024; Brown et al., 2024; Gandhi et al., 2024). The most common approach by far is to repeatedly sample from the LLM with the same prompt and filter out the best response using a verifier, also known as best-of-N sampling (Cobbe et al., 2021; Lightman et al., 2023). Methods that leverage feedback from the verifier to revise previous solutions in a line or tree-like fashion have also been explored (Feng et al., 2023; Chen et al., 2024a). Code generation is one such setting where scaling LLM inference through repeated sampling has also been effective (Wang et al., 2024; Chen et al., 2022).

Inference scaling is effective because, given enough attempts, the LLM is likely to sample the correct solution eventually (Snell et al., 2024; Brown et al., 2024). Therefore, generating diverse solutions is crucial for effective exploration. Our experiments show that existing methods such as best-of-N (BoN) and tree search often produce similar solutions, leading to *insufficient exploration of the solution space* (refer to Sec.3.5). Hence, a sampling and testing approach that balances exploration and exploitation

Code space optimization

?

O tests passed

1 tests passed

2 tests passed

3 tests passed

Figure 1: **2D Visualization of Code Space** represents each point as a possible code solution. The goal is to efficiently search this space for the solution with the best performance, defined by the number of unit tests passed, as indicated by the contours above.

testing approach that balances exploration and exploitation can greatly improve inference scaling.

To tackle this issue, we propose framing solution generation as a **black-box optimization problem** (Golovin et al., 2017) (as illustrated in Figure 1), in which validation tests serve as the black box and the LLM functions as the optimizer (Yang et al., 2024). Drawing from optimization theory, we develop SCATTERED FOREST SEARCH (SFS) to efficiently search for code solutions that successfully pass the maximum number of validation tests. Our method: 1) enhances exploration by enabling the LLM to propose diverse search directions, 2) improves exploitation by leveraging feedback and prior search experiences, and 3) initializes various random seed code solutions to ensure broader search coverage.

^{*}Work done during the internship at NEC Laboratories America. [™]Corresponding author.

Specifically, SFS contains three key techniques. SCATTERING is a novel technique that dynamically varies input prompts when sampling from LLMs, driving more diverse and exploratory outputs. In SCATTERING, the LLM suggests different *textual optimization directions and steps*, analogous to *gradients* in numerical optimization, before advancing towards a new solution. During tree search refinement, SCATTERING effectively *perturbs or mutates* previous solutions, resulting in an evolutionary search process. We further propose FORESTING, the tree search equivalent of *multi-start optimization*, where SCATTERING plays a crucial role in *diversifying initialization seeds*, ensuring they are *well-distributed throughout the search space*. This enhances the breadth of exploration while effectively mitigating clearly incorrect solutions, such as those containing syntax errors.

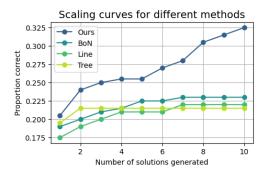


Figure 2: Scaling curve for different search methods. We run each method for 10 iterations total using gpt-3.5-turbo on APPS and report the proportion of problems where the correct solution has been discovered at each iteration.

Additionally, drawing inspiration from ant colony optimization and particle swarm optimization, we introduce SCOUTING to enhance SCATTERING by sharing feedback and experiences across search branches. When one branch discovers positive (or negative) results by following a specific textual direction, this information is relayed to guide future search steps, encouraging or discouraging exploration in that direction. Consequently, SCOUTING improves exploitation by intensifying the search around promising textual directions. We also provide a theoretical explanation demonstrating how our methods enhance exploration and circumvent local search regions when sampling from LLMs.

Our **parameter-free** method is simple yet effective. It does not require additional training

or labelled data, yet achieves great performance. As illustrated in Figure 2, our method is able to discover the correct solution significantly faster than the other approaches. We evaluate our method on five different code generation benchmarks, HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS, CodeContests, and Leetcode, showing increased inference time performance across the board, as well as better inference scaling. Our method also outperforms prior state-of-the-art techniques on HumanEval+ and MBPP+. Additionally, it generates higher *solution diversity* than previous methods, hence balancing exploration and exploitation, resulting in *faster discovery* of correct solutions and better scaling.

To sum up, our contributions are as follows:

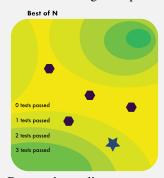
- We analyze the lack of diversity in current LLM code search processes and frame code generation as a black-box optimization problem within the language space, emphasizing the need for a balance between exploration and exploitation.
- We introduce SCATTERED FOREST SEARCH (SFS), composed of techniques SCATTERING
 and FORESTING that enhance the search process by encouraging broader exploration of the
 code space through textual optimization and seed initialization. We also provide a theoretical
 explanation for their effectiveness. Additionally, we present SCOUTING, which further improves
 SFS by exploiting promising search directions.
- We demonstrate that our method significantly enhances the accuracy, scalability, and solution diversity of the search process across widely used code generation benchmarks, including HumanEval, MBPP, Leetcode, APPS, and CodeContests.

2 Background

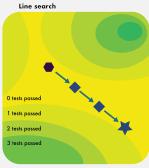
2.1 PROBLEM DESCRIPTION

In the task $x = \langle p, H \rangle$ of program synthesis, the solver is given a prompt p in natural language, which asks the solver to write code for some object s. The goal is to complete the code implementation of s such that it passes all the hidden tests H. The solver is not allowed to see the hidden tests. Sometimes, the solver is also given validation (visible) tests V that they can test their solution s on before they submit it for evaluation on the hidden tests H. They can also generate their own validation tests V.

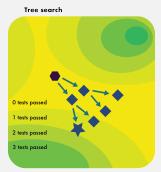
Figure 3: **Overview of prior methods** used for code generation with LLMs. Points represent solutions. Hexagons represent initial solutions. Star represents the final selected solution.



Repeated sampling generates multiple solutions using the LLM without leveraging feedback from previous iterations.



Line search rigidly exploits feedback and cannot revert to a previous solution if a new change worsens the outcome



Tree search is more flexible but still lacks sufficient exploration, as the generated solutions tend to be very similar.

Usually both hidden and validation tests are in the form of a set of assert statements that test the functionalities of s. An example of one such code generation task is shown in Appendix B.

A solution s' is said to be correct if it passes all the hidden tests H. The solver is allowed k submissions $[s]_k$ to the hidden tests. If at least one submission s^* passes all the hidden tests, then the task is considered to be solved. Given a set of tasks \mathcal{X} , the proportion of tasks $\langle p, H \rangle \in \mathcal{X}$ that are solved by the agent is called the **pass@k rate** (Chen et al., 2021).

2.2 Prior methods

A couple of different inference time methods have been tried in prior works to enhance the code generation capabilities of the LLM, which are shown in Figure 3. We elaborate more on the pros and cons of each method under a code space optimization framework below:

Best of N (BoN), or repeated sampling, involves sampling multiple independent solutions $[s]_n, s \sim \text{LLM}(p)$ from a language model using the same prompt p. The best solution s^* is then selected based on a verifier, commonly the number of validation tests passed (Li et al., 2022; Chen et al., 2024a).

Line search begins by sampling an initial seed code $s_0 \sim \text{LLM}(p)$. It then iteratively refines the previous solution s_{i-1} based on its test feedback f_{i-1} (Shinn et al., 2023; Madaan et al., 2023). This iterative self-refinement leverages test execution feedback to guide the model toward sampling successful solutions, i.e. $s_i \sim \text{LLM}(p|s_{i-1}, f_{i-1})$. However, line search is limited by its rigidity – requiring improvements on the most recent solution, even if the latest edits are incorrect. Thus, it struggles to effectively explore the search space and is more likely to get stuck in local optima.

Tree search overcomes the rigidity of line search by generating multiple child solutions for each parent solution, utilizing tree-structured exploration methods such as BFS, DFS, and MCTS (Feng et al., 2023; Chen et al., 2024a; Hao et al., 2023; Yao et al., 2023; Zhou et al., 2024; Tian et al., 2024). Given a parent solution s_i and its feedback f_i , the LLM produces k child solutions s_{i0} , s_{i1} , ..., s_{ik} . Although higher temperatures can produce diverse solutions, in practice, these solutions often resemble each other because they originate from the same prompt (refer to Sec. 3.5). Consequently, tree search still faces challenges in fully exploring the search space.

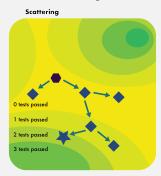
3 METHODOLOGY

Our method incorporates three optimization inspired techniques to enhance both **exploration** and **exploitation** of tree search (MCTS) using LLMs.

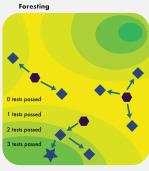
3.1 Tree branch Scattering

In tree search, children solutions of the same parent solution tend to be highly similar to one another since the LLM is given the same prompt to generate the children. We encourage more exploration when generating children solutions by querying the LLM to generate possible improvement directions $[d]_n$ first. The LLM is then instructed to implement a specific direction d_i for each child s_{ij} that it

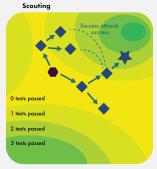
Figure 7: **Core techniques** used by SFS. Points represent solutions. Hexagons represent initial solutions. Star represents the final selected solution.



SCATTERING encourages tree search to **explore** more diverse solutions by using varied directional prompts for each branch or seed solution



FORESTING boosts **exploration** by performing tree search dynamically from multiple random seed solution starting points



SCOUTING shares successful search directions across branches of the search tree, providing general insights to better **exploit** feedback

generates from parent s_i . This helps us explore different, often orthogonal, improvement directions that help us **explore a wider region** of the search space, similar to trust-region methods in numerical optimization (xiang Yuan, 2015). We refer to this technique as SCATTERING the tree branches.

feedback from validation tests

→ propose new textual directions using LLM

choose 1 direction to implement when branching

Example SCATTERING Directions

Thoughts: The feedback suggests that the main problem is that the function is returning the first element of min-k and max-k instead of the entire lists.

Direction 1: Modify the return statement to return min-k and max-k instead of (min-k[0], max-k[0], min-k[-1], max-k[-1]). This will ensure that the function returns the entire lists of minimum and maximum k elements.

Direction 2: Update the function to handle the case when K is greater than the length of the tuple. In this case, return the entire sorted tuple as both the minimum and maximum k elements.

Furthermore, we employ MCTS and utilize the UCT formula (Eq. 1) to dynamically **select directions for exploration**, as illustrated below

$$UCT(s, d) = \widehat{Q}(s, d) + c\sqrt{\frac{\ln\left(\sum_{b} n(s, d)\right)}{n(s, d)}},$$
(1)

where c is the exploration parameter, n(s, d) is the number of visits of direction d at solution s, and $\widehat{Q}(s, d)$ is the estimated q-value which is updated via backpropagation as follows:

$$\widehat{Q}(\boldsymbol{s}_i, \boldsymbol{d}_i)^{(t+1)} \leftarrow (1 - \alpha(n))\widehat{Q}(\boldsymbol{s}_i, \boldsymbol{d}_i)^{(t)} + \alpha(n) \max\{\widehat{Q}(\boldsymbol{s}_i, \boldsymbol{d}_i)^{(t)}, \widehat{Q}(\boldsymbol{s}_{ij}, \boldsymbol{d}_{ij})^{(t+1)}\}, \quad (2)$$

where $\alpha(n)$ is the weighted average parameter that depends on n(s,d). The backpropogation occurs along the entire MCTS simulated trajectory $\tau = [s_0, d_{0i}, s_{0i}, ..., s_{-2}, d_{-1}, s_{-1}]$, where the q-value for the penultimate state s_{-1} is updated using the evaluation value of the final state $\widehat{Q}(s_{-2}, d_{-1})^{(t+1)} \leftarrow v(s_{-1})$ instead. We take the max of $\widehat{Q}(s_i, d_i)^{(t)}$ and the next state's q-value $\widehat{Q}(s_{ij}, d_{ij})$ to ensure that if the next solution is worse, the current solution can be used instead. This approach dynamically selects which direction to explore, prioritizing more promising parent solutions s_j over exploring all directions of a parent solution s_i . Using UCT to select distinct actions has been effective in balancing exploration and exploitation in other settings (Browne et al., 2012).

3.2 Forest search and forest Scattering

Iterative refinement faces the challenge that a very faulty initial seed solution may be difficult to correct effectively during the search process. An intuitive approach to address this issue is to

generate multiple seed solutions and perform tree search from each one. We refer to this technique as FORESTING, in which we generate n seed solutions $[s]_n$ and dynamically select which seed function to evolve from using the UCT formula (Eq. 1). Specifically, for each MCTS simulation, we select seed function s_i with the highest UCT value and conduct the simulation from that point. This closely resembles **random seed initialization**, a widely and effective approach in optimization literature.

Similar to branch SCATTERING, we can also promote more diverse seed solution generation through forest SCATTERING by providing the LLM with different instructions prior to generating solutions. We investigate the impact of various SCATTERING instructions on performance in Sec. 3.5.

Example Forest Seed Instructions

Seed instruction 1: Write the code in a modular and extensible manner, ensuring each function or class has a single responsibility and can be easily extended or modified without impacting other parts of the system. Prioritize clear interfaces and loose coupling between components.

Seed instruction 2: Focus on writing highly efficient code with minimal memory usage and fast execution times. Use data structures and algorithms optimized for performance, and consider edge cases that could lead to bottlenecks. Prioritize speed and resource efficiency over readability.

Seed instruction 3: Prioritize readability and maintainability in your code. Write clear and descriptive comments, use meaningful variable and function names, and structure the code in a way that is easy for others to understand and modify. Follow established coding standards and best practices.

3.3 Branch Scouting

Inspired by traditional optimization techniques like Ant Colony Optimization and Particle Swarm optimization, we utilize insights from other branches—specifically, which improvement directions are effective—to enhance the current solution. Specifically, after generating a new solution d_{-1} by applying improvement d_{-1} to solution d_{-2} , we provide feedback on the new solution f_{-1} to the LLM, inquiring whether the improvement was effective and what general insights it can derive from this. The insights will then be store in global memory, and in future iterations when the LLM is prompted to generate improvement directions, the global insights will be included in the prompt. This approach enables knowledge about effective and ineffective improvement directions to be shared across branches, facilitating more efficient exploitation of the search feedback and thereby enhancing our SCATTERING technique.

Example SCOUTING Insights

Insight 1: Modify the return statement to return \min_{n-k} and \max_{n-k} instead of $(\min_{n-k}[0], \max_{n-k}[0], \min_{n-k}[-1], \max_{n-k}[-1])$. This will ensure that the function returns the entire lists of minimum and maximum k elements.

Insight 2: Update the function to handle the case when K is greater than the length of the tuple. In this case, return the entire sorted tuple as both the minimum and maximum k elements.

use insights to generate directions are directions see if the direction worked or not application worked or not application and update insights based on feedback

3.4 A THEORETICAL PERSPECTIVE

The proposed techniques—SCATTERING and FORESTING—can be analyzed via the Markov chain theory, particularly focusing on the concepts of diverse transition kernels, conductance, and mixing times (Levin & Peres, 2020). We can define the search strategy as a Markov transition kernel P(s,s') which denotes the probability of generating a new solution s' given the current solution s. A chain of self-refined solutions s_0, s_1, s_2, \ldots are generated following the transition kernel P.

In previous methods including line search and tree search, the transition is realized by an LLM π that is conditioned on the previous solution s and the feedback f = F(s), denoted by $\pi(s'|s, f)$. The transition kernel is

$$P_{\text{previous}}(\mathbf{s}, \mathbf{s}') = \pi_c(\mathbf{s}'|\mathbf{s}, F(\mathbf{s})), \tag{3}$$

where we use π_c to emphasize that the LLM π is prompted to output code. We know $\pi(c)$ can be extremely concentrated and outputs highly similar solutions.

With SCATTERING, we first sample an improvement direction d generated by the LLM with prompt $\pi(\cdot|s, F(s))$. And then prompt LLM π to generate the next solution s' given the current solution s,

the feedback f = F(s), and the improvement direction d. The transition kernel is

$$P_{\text{SCATTERING}}(\boldsymbol{s}, \boldsymbol{s}') = \sum_{\boldsymbol{d}} \pi_t(\boldsymbol{d}|\boldsymbol{s}, F(\boldsymbol{s})) \pi_c(\boldsymbol{s}'|\boldsymbol{s}, F(\boldsymbol{s}), \boldsymbol{d}), \tag{4}$$

where π_c is still an extremely concentrated policy and outputs highly similar solutions. However, we have a text-based "reflection" $\pi_t(\mathbf{d}|\mathbf{s}, F(\mathbf{s}))$ that can generate highly diverse improvement directions. In practice, we can observe that indeed the directions are highly diverse.

In Markov chain theory, a diverse transition kernel increases the probability of moving between different regions of the state space (denoted by S), thereby enhancing the **conductance** Φ of the chain:

$$\Phi_P(S) = \frac{\sum_{s \in S, s' \notin S} \mu(s) P(s, s')}{\mu(S)},$$
(5)

where S is a subset and μ is the stationary distribution

Higher conductance improves the spectral gap γ , which is inversely related to the mixing time of the Markov chain (see Cheeger's inequality) (Levin & Peres, 2020), reducing the likelihood of the chain getting trapped in local regions. More formally speaking, for a local region S, the previous methods (equation 3) rely on directly generating new responses that can easily stuck in the local region S ($P_{\text{previous}}(s,s')\approx 0$ when $s\in S$ and $s'\not\in S$, thus the conductance Φ is near 0). While our SCATTERING search (equation 4) can generate highly diverse directions d and lead to new solutions s' out of the local region S ($P_{\text{SCATTERING}}(s,s')>0$ if some d gives correct direction).

3.5 EMPIRICAL VALIDATION

We varied the types of seed instructions used to generate seed code during BoN sampling to validate the effects of increasing solution diversity with SCATTERING. In the Jabberwocky setting, the model was prompted with different lines from the humorous nonsense poem "Jabberwocky" before generating seed code. In the Style setting the model was prompted with different coding style instructions such as writing code in a 'highly modular way' or 'focus on brevity and clarity'. In the Role setting, the model was prompted with different software engineer personalities such as 'You are an innovator.' or 'You are a perfectionist'. All input prompts were LLM generated by gpt-3.5-turbo around a common theme.

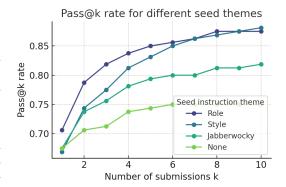


Figure 11: **Pass@k rate** for repeated sampling with different initialization seed types on HumanEval using gpt-3.5-turbo-0613. Increasing seed variety with SCATTERING significantly improves both Pass@k rate and scaling.

We present the pass@k performance of these seed generation styles in Figure 11, with detailed performance metrics in Tables 1 and 19. This includes: 1) the **pass@any** rate, which measures the proportion of problems where the correct solution was found at any point during the search, 2) the mean **validation score**, averaged across all candidate solutions generated (Eq. 6), and 3) the mean **BERT cosine similarity** between embeddings of candidate solution pairs, averaged over all problems. Embeddings were taken using CodeBERT, a pretrained model for understanding code semantically (Feng et al., 2020).

$$\text{Mean BERT cosine similarity} = \frac{1}{|\mathcal{X}|} \sum_{\langle \boldsymbol{p}, \boldsymbol{H} \rangle \in \mathcal{X}} \frac{1}{|\mathcal{S}_{\boldsymbol{p}}|(|\mathcal{S}_{\boldsymbol{p}}|-1)} \sum_{\substack{\boldsymbol{s}, \boldsymbol{s}' \in \mathcal{S}_{\boldsymbol{p}} \\ \boldsymbol{s} \neq \boldsymbol{s}'}} \frac{\text{embed}(\boldsymbol{s}) \cdot \text{embed}(\boldsymbol{s}')}{\|\text{embed}(\boldsymbol{s})\| \|\text{embed}(\boldsymbol{s}')\|}$$

In addition to the metrics we discussed previously, we also include other similarity metrics such as **tf-idf similarity**, which measures the average cosine similarity between tf-idf vectors, the **Levenshtein similarity**, and the **token sequence similarity**. See App. I for more details, including examples.

As shown in Table 1, "Role" and "Style" performed best in discovering the correct solution. Surprisingly, even *unrelated prompts, like lines from nonsense poems, boosted performance*. Overall, all SCATTERING themes reduced solution similarity while maintaining comparable validation scores, demonstrating the effectiveness of using varied inputs during the search process.

Table 1: **Effects of different seed instruction SCATTERING themes.** 10 seed solutions were generated using gpt-3.5-turbo for each theme, and we filtered the seeds using 6 generated validation tests to select the best one to submit for evaluation on the HumanEval benchmark.

Seed theme	pass@1	pass@any	tf-idf sim.	BERT sim.	lev. sim.	seq. sim.	val. score
None	72.5	75.0	0.9013	0.9976	0.8971	0.9361	0.7786
Jabberwocky	74.4	81.9	0.7559	0.9944	0.7749	0.8444	0.7658
Style	79.4	88.1	0.6826	0.9929	0.7119	0.7504	0.7548
Role	81.9	87.5	0.7734	0.9957	0.7907	0.8323	0.7649

4 EXPERIMENTS

We demonstrate that our search method outperforms prior methods by showing that it 1) achieves higher accuracy, 2) finds correct solutions faster and scales better, and 3) explores more diverse solutions without sacrificing exploitation of good ones.

4.1 EVALUATION BENCHMARKS

We evaluate our method on several popular code generation benchmarks. **HumanEval** consists of 164 human-generated Python questions (Chen et al., 2021), and **MBPP** includes 399 problems (Austin et al., 2021). The original sets included only 3 to 8 hidden tests \mathcal{H} , which researchers found inadequate for thoroughly evaluating correctness in edge cases. Additional hidden tests were added, resulting in the **HumanEval+** and **MBPP+** sets (Liu et al., 2024a).

Both **APPS** (Hendrycks et al., 2021) and **CodeContests** (Li et al., 2022) feature challenging code competition problems. From the 10,000 problems in APPS, we randomly sample 200 for evaluation due to budget constraints. We adapt the competition format of both datasets to resemble the HumanEval format for Python evaluation. **Leetcode** (Guo et al., 2024) includes recent problems scraped from the website, ensuring that LLMs released before July 2023 have not been trained on these data.

4.2 ACCURACY

We conduct experiments on a variety of code generation benchmarks as shown in Table 2, where we see that our method achieves a higher pass@1 rate than other search methods and the base accuracy (i.e., evaluating the first solution that the LLM model generates). Methods were given the same search budget (10 solutions), they used 6 self-generated validation tests.

Table 2: **Performance of our method compared to prior search methods.** Pass@1 performance reported here. Both solutions and validation tests were generated using gpt-3.5-turbo.

Method / Benchmark	HumanEval+	MBPP+	Leetcode	APPS	CodeContests
Base	58.5%	64.9%	30.0%	16.0%	1.82%
Line	53.0%	61.2%	28.9%	14.5%	1.21%
Tree (MCTS)	59.8%	65.4%	31.7%	18.0%	2.42%
Best of N	65.2%	64.4%	33.3%	19.5%	1.82%
Ours (SFS)	67.1%	65.7%	36.7%	20.5%	4.24%

We also measure the proportion of problems where the **correct solution was found** (**pass@any**) at any point during the search process, as shown in Table 3. We evaluate on HumanEval and MBPP rather than their plus versions due to computational constraints. HumanEval+ and MBPP+ contain 80x and 35x more tests, respectively, which makes it challenging to verify each generated solution. The pass@any rate is higher than pass@1 because, even if the correct solution is found, inaccurate validation tests may lead the algorithm to submit an alternative solution. Our method achieves significantly higher pass@any rates and demonstrates greater improvements from pass@1 to pass@any, indicating that it *can better leverage a more accurate verifier to filter for the correct solution*. We further explore the impact of noisy validation tests in detail in Sec. 4.6.

Table 3: Pass@any accuracy of our method compared to prior search methods. Both solutions and validation tests were generated using gpt-3.5-turbo. We run each method for 10 iterations.

Method / Benchmark	HumanEval	MBPP	Leetcode	APPS	CodeContests
Line	83.1%	82.9%	33.3%	22.0%	2.99%
Tree (MCTS)	76.9%	79.6%	33.9%	21.5%	2.42%
Best of N	75.6%	77.3%	33.9%	23.0%	3.03%
Ours (SFS)	89.0%	86.1%	39.4%	32.5%	6.06%

We also compare to existing state-of-the-art works as shown in Table 4 and 5. In Table 4 setting, validation tests are self-generated, and cap our solution budget at 40 generations max, same as in prior literature (Zhou et al., 2024). In Table 5, a subset of the ground truth hidden tests are given (3 for HumanEval, 1 for MBPP) (Zhong et al., 2024), and we compare against similar methods under this setting. We see that our method achieves higher performance in both settings.

Table 4: **Comparison to prior works** when ground truth tests are not given. We report Pass@1 performance with GPT-3.5.

Table 5: **Comparison to prior works** when a subset of ground truth tests are given. We report Pass@1 performance with GPT-3.5.

HumanEval

Benchmark

Tacte given

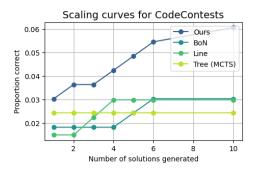
MBPP

Benchmark	HumanEval	MBPP
CoT (Wei et al., 2022)	46.9	54.9
ReAct (Yao et al., 2022)	56.9	67.0
Reflexion (Shinn et al., 2023)	68.1	70.0
ToT (Yao et al., 2023)	54.4	65.8
RAP (Hao et al., 2023)	63.1	71.4
LATS ^a (Zhou et al., 2024)	75.6	79.6
Ours (SFS)	82.5	81.7

Ours (SFS)	87.2	91.3
(Zhong et al., 2024)	82.9	76.0
(Chen et al., 2023) LDB	00.5	, 2.0
SD (+Trace)	80.5	72.6
(Chen et al., 2023)	81.1	74.4
SD (+Expl.)	01.1	74.4
rests given) 3	1

^aran under our setup, see App. D for similar setup

4.3 SCALABILITY



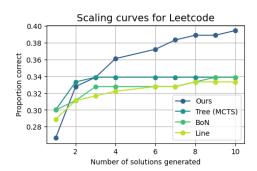


Figure 12: Scaling curves for different search methods on CodeContests (left) and Leetcode (right). We run each method for 10 iterations using gpt-3.5, reporting the proportion of problems where the correct solution is discovered at each iteration. Additional curves can be found in Sec. C

We report the **average number of iterations** (solutions generated) it takes before the search algorithm discovers the correct solution in Table 6. **Iters.** (incl) is the average number of iterations it takes including problems where the algorithm succeeds on the first try. **Iters.** (excl) is the average number of iterations it takes excluding first try successes, where the search algorithm is actually used to find the correct solution. On both metrics, our method demonstrates the ability to *discover the correct solution much more quickly than the other methods*. Moreover, we see in Figure 2 and 12 that our method also *scales better than the other methods* on all datasets.

Table 6: Metrics for different search methods. We run different search methods for 10 iterations each using gpt-3.5-turbo on HumanEval. Our method generates more diverse solutions and discovers the correct solution faster.

Method	pass@1	pass@any	BERT sim.	val. score	iters. (incl)	iters. (excl)
Line	68.1%	83.1%	0.9992	0.795	2.09	7.13
Tree (MCTS)	75.6%	76.9%	0.9998	0.827	2.38	8.09
Best of N	73.8%	75.6%	0.9983	0.774	2.59	9.00
Ours (SFS)	82.5%	89.0%	0.9945	0.813	1.67	5.06

4.4 SOLUTION DIVERSITY

We see in Table 6 that our proposed search method is able to propose more diverse candidate solutions with a lower semantic similarity score, while maintaining high quality search with by generating solutions with high validation scores. This shows that our methods help both *exploration and exploitation without sacrificing too much of one for the other*. Detailed stats shown in App. F.

4.5 ABLATION ON TECHNIQUES

We performed an ablation study on the three introduced techniques as shown in Table 7 (and App. G), all of which enhanced performance and efficiency, with SCATTERING yielding the highest gains.

Table 7: **Ablation metrics for techniques used in our method.** We run search methods for 10 iterations each using gpt-3.5-turbo on HumanEval.

Ablation	pass@1	pass@any	BERT sim.	val. score	iters. (incl)	iters. (excl)
Everything	82.5%	89.0%	0.9945	0.813	1.68	5.06
No SCATTERING	75.6%	78.1%	0.9982	0.802	2.43	8.82
No Foresting	79.4%	86.3%	0.9982	0.817	2.05	6.56
No SCOUTING	81.9%	86.3%	0.9942	0.792	2.12	6.05

Table 8: **Performance on benchmarks when the ground truth tests are given.** We run 10 iterations with our method using gpt-3.5-turbo-0613 on HumanEval. When the tests are accurate, we can achieve even higher performance, even if only a subset is given.

Tests given	pass@1	pass@any	BERT sim.	val. score	iters. (incl)	iters. (excl)
No tests given	82.5%	89.0%	0.9945	0.813	1.68	5.06
3 tests given	87.2%	90.2%	0.9952	0.862	2.34	6.86
All tests given	89.0%	89.0%	0.9949	0.864	2.04	5.88

4.6 VERIFIER (VALIDATION TEST) ACCURACY

Previous work emphasized the importance of a reliable verifier (Liu et al., 2024a; Chen et al., 2022; Zhang et al., 2023a). In our case, we use self-generated validation tests noisy as a black-box verifer. Figure 13 shows the confusion matrix for whether self-generated validation tests accurately predict solution correctness. The false negative rate of 27.5% highlights misalignment between validation and ground truth tests. We performed an ablation study comparing performance when given different numbers of ground-truth tests as validation (Table 8). While finding the best verifier is not our focus, the results show a significant impact of verifier accuracy on performance. There are strong interactions between the search method and evaluation metric—just one ground-truth validation test dramatically improves performance. This demonstrates how well our methods exploit accurate evaluation signals and perform even better when the black-box evaluation is more accurate. Detailed metrics shown in App. H.

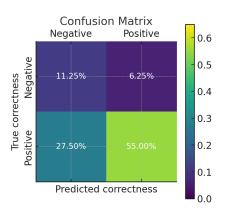


Figure 13: **Confusion matrix for self-generated validation tests** with gpt-3.5-turbo-0613 using our method.

4.7 ABLATION ON MODEL

As shown in Figure 14, weaker models scale better with our method. This supports the trade-off between training and inference compute: better-trained models scale worse with inference compute, and vice versa. Detailed stats in App. E.

5 Additional Related work

Code generation with large language models. Some works focus on either training their own model (Guo et al., 2024) or fine tuning existing models to adapt them towards code related tasks (Roziere et al., 2023; Jain et al., 2023; Roziere et al., 2023). Recent literature has shown a flora of methods to improve LLM code generation performance during inference time, including agentic approaches (Qian et al., 2023; Liu et al., 2024b), multi-agent approaches

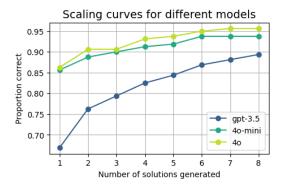


Figure 14: **Scaling curve for different LLM-models.** We run our method on each model for 10 iterations on HumanEval and report the proportion of problems where the correct solution has been discovered at each iteration.

(Tao et al., 2024; Hong et al., 2023; Islam et al., 2024), tool usage (Zhang et al., 2024), and self-revision (Le et al., 2023). Our work compliments these works by showing that increasing solution diversity and exploration can have a large impact on inference scaling.

Solution validation and feedback. Prior work has demonstrated that effective and accurate validation methods significantly boost performance in code generation (Chen et al., 2022; 2024b) and related tasks like math theorem proving (Cobbe et al., 2021; Uesato et al., 2022). The importance of a reliable evaluation metric is critical (Liu et al., 2024a), and incorporating natural language feedback and reflection on execution results further enhances performance (Zhang et al., 2023c; Bai et al., 2022). Our work complements this by introducing new search methods that more effectively discover solutions meeting these validation and evaluation criteria. Combining a strong search method with a robust validation approach leads to improved solution generation performance.

Black box optimization. While many of these methods are domain specific, we adapt many of the insights behind these methods into using LLMs to search over language and code space. Our textual optimization steps can be thought of as 'textual gradients', though we do not use them to conduct backpropogation (Yuksekgonul et al., 2024). There have been some recent works exploring the usage of LLMs as optimizers for different problems (Zhang et al., 2023b; Liu et al., 2024c), including using evolutionary optimization algorithms (Lange et al., 2024; Liu et al., 2024b).

Tree search. Tree search has proven effective in decision-making domains. Previous works have applied tree search to multi-step tasks, such as determining the next reasoning step (Silver et al., 2017; Zhou et al., 2024; Yao et al., 2023; Besta et al., 2024) or generating the next code token (Zhang et al., 2023c). While prior work primarily uses tree search for planning (Jiang et al., 2023; Feng et al., 2023; Bairi et al., 2024), we show that *tree search can also be used for optimization*, functioning as a black-box method for exploring a region rather than a line.

6 Conclusion

We have shown that framing code generation as an optimization task over the code space and applying SCATTERED FOREST SEARCH is highly effective. The simple yet powerful SCATTERING technique, which encourages the LLM to produce more diverse outputs, underscores the importance of exploration in optimization and search processes. This framework and these techniques can offer valuable insights for other code or language generation tasks.

The integration of search-based methods could significantly reduce computational costs while maintaining or enhancing performance, making them attractive for large-scale deployments, especially in real-time applications or those requiring resource efficiency. This framework also lays the groundwork for future research into optimization strategies for language models, potentially leading to more advanced search algorithms, hybrid models, and novel techniques that push the limits of generative models.

REFERENCES

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.
- Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *FSE*, 1:675–698, 2024.
- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *AAAI*, volume 38, pp. 17682–17690, 2024.
- Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. arXiv preprint arXiv:2407.21787, 2024.
- Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.
- Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. Alphamath almost zero: process supervision without process. *arXiv preprint arXiv:2405.03553*, 2024a.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.
- Ziru Chen, Michael White, Raymond Mooney, Ali Payani, Yu Su, and Huan Sun. When is tree search useful for llm planning? it depends on the discriminator. *arXiv preprint arXiv:2402.10890*, 2024b.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. *arXiv* preprint arXiv:2309.17179, 2023.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D Goodman. Stream of search (sos): Learning to search in language. arXiv preprint arXiv:2404.03683, 2024.
- Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. Google vizier: A service for black-box optimization. In *SIGKDD*, pp. 1487–1495, 2017.

- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv* preprint arXiv:2401.14196, 2024.
- Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.
- Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.
- Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv* preprint arXiv:2405.11403, 2024.
- Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators. arXiv preprint arXiv:2311.14904, 2023.
- Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 2023.
- Robert Lange, Yingtao Tian, and Yujin Tang. Large language models as evolution strategies. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 579–582, 2024.
- Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv* preprint arXiv:2310.08992, 2023.
- David A. Levin and Yuval Peres. Markov chains and mixing times. https://pages.uoregon.edu/dlevin/MARKOV/markovmixing.pdf, 2020.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *NeurIPS*, 36, 2024a.
- Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. Large language models as evolutionary optimizers. In 2024 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE, 2024b.
- Shihong Liu, Samuel Yu, Zhiqiu Lin, Deepak Pathak, and Deva Ramanan. Language models as black-box optimizers for vision-language models. In *CVPR*, pp. 12687–12697, 2024c.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *NeurIPS*, 36, 2023.
- Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 6, 2023.

- Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS*, 36, 2023.
- David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling Ilm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *arXiv preprint arXiv:2403.17927*, 2024.
- Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. Toward self-improvement of llms via imagination, searching, and criticizing. *arXiv preprint arXiv:2404.12253*, 2024.
- Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.
- Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 35: 24824–24837, 2022.
- Ya xiang Yuan. Recent advances in trust region algorithms. Mathematical Programming, 2015.
- Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *ICLR*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*, 36, 2023.
- Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.
- Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. arXiv preprint arXiv:2401.07339, 2024.
- Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. Algo: synthesizing algorithmic programs with llm-generated oracle verifiers. arXiv preprint arXiv:2305.14591, 2023a.
- Michael R Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. Using large language models for hyperparameter optimization. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023b.
- Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv* preprint arXiv:2303.05510, 2023c.

Li Zhong, Zilong Wang, and Jingbo Shang. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

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

APPENDIX: SMARTER CODE SPACE EXPLORATION WITH LLMS

A ETHICS STATEMENT

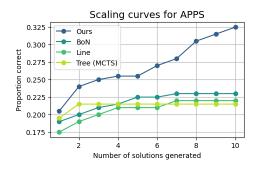
All contributing authors of this paper confirm that they have read and pledged to uphold the ICLR Code of Ethics. Our proposed method is specifically designed for code generation using LLMs. We acknowledge the potential for LLMs to generate code that may inadvertently introduce vulnerabilities or ethical concerns. Our research prioritizes the development of methodologies that emphasize responsible usage, ensuring that generated code adheres to best practices in security and ethics. We recognize that LLMs can perpetuate and amplify biases present in training data. We aim to contribute positively to the field of code generation while addressing the potential challenges and responsibilities that arise from the use of advanced AI technologies.

B CODING PROBLEM EXAMPLE

```
Example code generation prompt, solution, and tests
Prompt: Write a function greatest_common_divisor(a,b) that returns the GCD
of two integers a and b
Validation tests:
assert (greatest common divisor (3,5) == 1)
assert(greatest_common_divisor(25,15) == 5)
assert(greatest_common_divisor(0,3) == 3)
Proposed solution 1:
def greatest_common_divisor(a, b):
    for i in range (min(a, b), 0, -1):
         if a % i == 0 and b % i == 0:
             return i
Test feedback:
assert(greatest_common_divisor(3,5) == 1) #output 1 is
correct
assert (greatest common divisor (25,15) == 5) #output 5 is
correct
assert (greatest common divisor (0,3) == 3) #output None is
incorrect
```

C SCALING

Our method scales well with increased iterations. We show scaling curves for each dataset as shown in Figures 15, 16, 17.



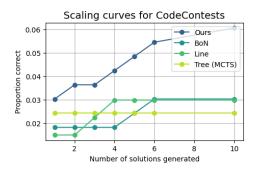
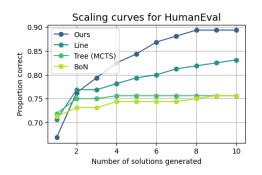


Figure 15: Scaling curves for different search methods on APPS (left) and CodeContests (right). We run each method for 10 iterations using gpt-3.5-turbo, reporting the proportion of problems where the correct solution is discovered at each iteration.



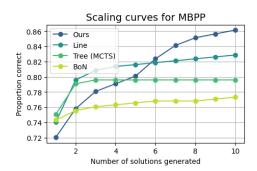


Figure 16: Scaling curves for different search methods on HumanEval (left) and MBPP (right). We run each method for 10 iterations using gpt-3.5-turbo, reporting the proportion of problems where the correct solution is discovered at each iteration.

D PRIOR WORKS BENCHMARK

In the LATS experimental setup, in each iteration after a solution is generated, the algorithm is allowed to check if the new solution is correct on the ground truth tests before proceeding (Zhou et al., 2024). We show the performance of our method in the same setup here, with the same solution budget. We see that given the same setup, our method still achieves higher performance.

Table 9: Comparison to prior works. We report Pass@1 performance with GPT-3.5.

Method / Benchmark	HumanEval	MBPP
CoT (Wei et al., 2022)	46.9	54.9
ReAct (Yao et al., 2022)	56.9	67.0
Reflexion (Shinn et al., 2023)	68.1	70.0
ToT (Yao et al., 2023)	54.4	65.8
RAP (Hao et al., 2023)	63.1	71.4
LATS (Zhou et al., 2024)	83.8	81.1
Ours (Forest SCOUTING)	93.3	91.2

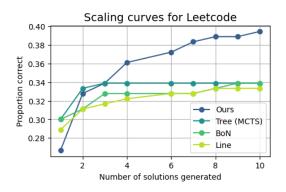


Figure 17: Scaling curves for different search methods on LeetCode. We run each method for 10 iterations using gpt-3.5-turbo, reporting the proportion of problems where the correct solution is discovered at each iteration.

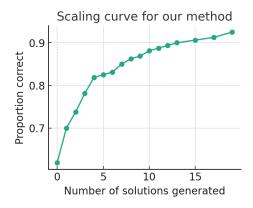


Figure 18: **High budget scaling performance for our method**. Proportion of problems where the correct solution was discovered by our method on HumanEval with gpt-3.5-turbo0613 with 20 iterations.

E ABLATION ON MODELS

In this section, we provide a detailed ablation study on the performance of different base language models, focusing on metrics relevant to the HumanEval benchmark. Table 10 displays the results of applying various search methods across three models: gpt-3.5, gpt-4o-mini, and gpt-4o. For each model, we report key performance indicators such as pass@1, pass@any, BERT similarity, and validation score, along with the number of iterations (inclusive and exclusive). Furthermore, Table 11 provides an extended comparison of metrics, including similarity measures like TF-IDF and Levenshtein, alongside error rates (false positives/negatives) and true classification rates, helping to evaluate model behavior in more detail across different performance aspects.

Table 10: Ablation on models. We run search methods for 10 iterations each on HumanEval

Model	pass@1	pass@any	BERT sim.	val. score	iters. (incl)	iters. (excl)
gpt-3.5 gpt-4o-mini	82.5% 89.4%	89.0% 93.8%	0.9945 0.9876	0.813 0.845	1.68 0.84	5.06 5.83
gpt 40 mini gpt-40	90.6%	95.6%	0.9915	0.894	0.68	4.95

Table 11: Performance metrics for different base LLM models on HumanEval. We ran gpt-3.5-turbo-0613 for 10 iterations for each model.

Metric	gpt-3.5	gpt-4o-mini	gpt-4o
Pass@Any	89.0%	93.8%	95.6%
Pass@1	82.5%	89.4%	90.6%
Validation score	0.813	0.845	0.894
BERT sim.	0.9945	0.9876	0.9915
TF-IDF sim.	0.743	0.643	0.691
Levenshtein sim.	0.721	0.631	0.665
Token seq. sim.	0.765	0.719	0.705
False positive rate	6.25%	4.38%	4.38%
False negative rate	27.50%	33.13%	17.50%
True positive rate	55.00%	56.25%	73.13%
True negative rate	11.25%	6.25%	5.00%
Iters. (incl)	1.68	0.84	0.68
Iters. (excl)	5.06	5.83	4.95

F PERFORMANCE METRICS FOR DIFFERENT DATASETS

We display detailed performance metrics for MBPP (Table 13), HumanEval (Table 12), CodeContests (Table 16), Leetcode (Table 14), and APPS (Table 15).

The performance metrics across various datasets, including MBPP, HumanEval, CodeContests, Leetcode, and APPS, offer a comprehensive comparison of different search methods used to evaluate large language model (LLM) performance. Each table highlights the results of different strategies, such as Line, Tree-based search (MCTS), Best of N, and our proposed method. Key metrics, such as Pass@1, Pass@Any, validation score, and various similarity metrics (e.g., BERT, TF-IDF, and Levenshtein), illustrate the effectiveness of each search method in terms of accuracy and relevance.

Additionally, false positive and false negative rates provide insight into the errors made by each approach, while true positive and negative rates reflect the precision and recall of each method. These metrics give a holistic view of model performance, especially in scenarios involving iterative search, with the number of iterations (both inclusive and exclusive) further contextualizing the computational complexity of each method.

Table 12: Performance metrics for different search methods on HumanEval. We ran gpt-3.5-turbo-0613 for 10 iterations for each method.

Metric	Line	Tree (MCTS)	Best of N	Ours
Pass@Any	83.1%	76.9%	75.6%	89.0%
Pass@1	68.1%	75.6%	73.8%	82.5%
Validation score	0.795	0.827	0.774	0.813
BERT sim.	0.9992	0.9998	0.9983	0.9945
TF-IDF sim.	0.495	0.502	0.936	0.743
Levenshtein sim.	0.480	0.496	0.925	0.721
Token seq. sim.	0.476	0.500	0.962	0.765
False positive rate	8.1%	5.0%	3.8%	6.3%
False negative rate	16.9%	23.1%	28.1%	27.5%
True positive rate	51.3%	52.5%	45.6%	55.0%
True negative rate	23.8%	19.4%	22.5%	11.3%
Iters. (incl)	2.09	2.61	2.59	1.68
Iters. (excl)	7.13	8.87	9.00	5.06

Table 13: Performance metrics for different search methods on MBPP. We ran gpt-3.5-turbo-0613 for 10 iterations for each method.

Metric	Line	Tree (MCTS)	Best of N	Ours
Pass@Any	82.9%	79.6%	77.3%	86.1%
Pass@1	73.6%	76.1%	76.6%	78.3%
Validation score	0.784	0.774	0.729	0.729
BERT sim.	0.9991	0.9998	0.9993	0.9956
TF-IDF sim.	0.473	0.521	0.960	0.747
Levenshtein sim.	0.455	0.519	0.956	0.759
Token seq. sim.	0.450	0.520	0.974	0.785
False positive rate	7.6%	5.8%	6.0%	6.5%
False negative rate	18.9%	26.2%	33.0%	21.7%
True positive rate	54.7%	49.9%	43.6%	56.7%
True negative rate	18.9%	18.1%	17.4%	15.1%
Iters. (incl)	1.91	2.29	2.36	1.91
Iters. (excl)	7.35	9.20	9.20	6.85

Table 14: Performance metrics for different search methods on Leetcode. We ran gpt-3.5-turbo-0613 for 10 iterations for each method.

Metric	Line	Tree (MCTS)	Best of N	Ours
Pass@Any	33.3%	33.9%	33.9%	39.4%
Pass@1	28.9%	33.3%	33.3%	36.7%
Validation score	0.352	0.354	0.372	0.387
BERT sim.	0.9933	0.9998	0.9936	0.9961
TF-IDF sim.	0.840	0.934	0.889	0.589
Levenshtein sim.	0.813	0.926	0.902	0.694
Token seq. sim.	0.818	0.925	0.897	0.652
False positive rate	1.1%	0.6%	0.6%	5.0%
False negative rate	12.2%	27.2%	26.7%	24.4%
True positive rate	8.3%	6.1%	6.7%	9.4%
True negative rate	78.3%	66.1%	66.1%	61.1%
Iters. (incl)	6.78	7.31	6.74	6.42
Iters. (excl)	9.54	10.44	9.63	8.75

G TECHNIQUE ABLATION

In this section, we present the results of an ablation study to assess the impact of different components of our method on performance, as shown in Table 17. We evaluate four variations: the full method with all components enabled ("Everything"), and three ablations where we remove key techniques—SCATTERING, FORESTING, and SCOUTING—one at a time.

The ablation results highlight how critical each technique is for achieving high performance. Removing SCATTERING leads to a noticeable drop in Pass@Any, decreasing from 89.0% to 78.1%, while removing FORESTING has a milder effect, with Pass@Any only falling to 86.3%. Notably, FORESTING removal significantly affects similarity scores, with a drop in TF-IDF similarity from 0.743 to 0.419. Similarly, removing SCOUTING results in a slight reduction in performance metrics, but Pass@1 remains comparable to the full method, indicating its robustness.

Table 15: Performance metrics for different search methods on APPS. We ran gpt-3.5-turbo-0613 for 10 iterations for each method.

Metric	Line	Tree (MCTS)	Best of N	Ours
Pass@Any	22.0%	21.5%	23.0%	32.5%
Pass@1	14.5%	18.0%	19.5%	20.5%
Validation score	0.232	0.211	0.205	0.232
BERT sim.	0.9886	0.9997	0.9984	0.9946
TF-IDF sim.	0.837	0.943	0.825	0.523
Levenshtein sim.	0.817	0.936	0.843	0.626
Token seq. sim.	0.813	0.933	0.840	0.567
False positive rate	6.5%	3.0%	3.0%	7.0%
False negative rate	7.5%	17.5%	18.5%	16.0%
True positive rate	2.0%	1.5%	2.0%	2.5%
True negative rate	84.0%	78.0%	76.5%	74.5%
Iters. (incl)	7.93	8.66	7.82	7.30
Iters. (excl)	9.61	10.75	9.65	9.18

Table 16: Performance metrics for different search methods on Codecontests. We ran gpt-3.5-turbo-0613 for 10 iterations for each method.

Metric	Line	Tree (MCTS)	Best of N	Ours
Pass@Any	3.0%	2.4%	3.0%	6.1%
Pass@1	1.21%	2.42%	1.82%	4.24%
Validation score	0.091	0.078	0.051	0.086
BERT sim.	0.9838	0.9997	0.9980	0.9950
TF-IDF sim.	0.871	0.980	0.770	0.502
Levenshtein sim.	0.856	0.981	0.800	0.603
Token seq. sim.	0.846	0.982	0.766	0.502
False positive rate	2.2%	1.2%	0.6%	1.2%
False negative rate	0.7%	2.4%	1.8%	1.2%
True positive rate	0.0%	0.0%	0.0%	0.0%
True negative rate	97.0%	96.4%	97.6%	97.6%
Iters. (incl)	9.74	10.73	9.75	9.53
Iters. (excl)	9.89	11.00	9.93	9.83

H VERIFIER ACCURACY AND GROUND-TRUTH VALIDATION TESTS

In this section, we assess the impact of incorporating ground-truth validation tests on verifier accuracy, as shown in Table 18. We compare three configurations: no ground-truth tests ("None"), using 3 ground-truth tests, and utilizing all available tests. These variations allow us to explore how different amounts of external validation influence the performance of our verifier.

The results demonstrate a clear improvement in Pass@1 and validation scores when ground-truth tests are introduced. With no external tests, the Pass@1 rate is 82.5%, which increases to 87.2% when 3 tests are used and further to 89.0% with all tests. Similarly, the validation score improves from 0.813 to 0.862 and 0.864, respectively. These enhancements reflect the verifier's increased accuracy when more reliable validation signals are available.

Table 17: Performance metrics for different ablations on techniques for our method. We ran gpt-3.5-turbo-0613 for 10 iterations for each ablation on HumanEval.

Metric	Everything	No SCATTERING	No FORESTING	No SCOUTING
Pass@Any	89.0%	78.1%	86.3%	86.3%
Pass@1	82.5%	75.6%	79.4%	81.9%
Validation score	0.813	0.802	0.817	0.792
BERT sim.	0.9945	0.9982	0.9982	0.9942
TF-IDF sim.	0.743	0.926	0.419	0.740
Levenshtein sim.	0.721	0.911	0.427	0.729
Token seq. sim.	0.765	0.945	0.409	0.756
False positive rate	0.063	0.063	0.050	0.063
False negative rate	0.275	0.244	0.244	0.281
True positive rate	0.550	0.513	0.550	0.538
True negative rate	0.113	0.181	0.156	0.119
Iters. (incl)	1.68	2.43	2.05	2.12
Iters. (excl)	5.06	8.82	6.56	6.05

Table 18: Performance metrics when given different amounts of ground truth tests. 6 self-generated tests were used for validation in 'None'. We ran gpt-3.5-turbo-0613 for 10 iterations for each setting.

Metric	None	3 tests	All tests
Pass@Any	89.0%	90.2%	89.0%
Pass@1	82.5%	87.2%	89.0%
Validation score	0.813	0.862	0.864
BERT sim.	0.9945	0.9952	0.9949
TF-IDF sim.	0.743	0.767	0.762
Levenshtein sim.	0.721	0.739	0.729
Token seq. sim.	0.765	0.758	0.757
False positive rate	6.3%	9.8%	1.8%
False negative rate	27.5%	2.4%	3.7%
True positive rate	55.0%	78.0%	85.4%
True negative rate	11.3%	9.8%	9.1%
Iters. (incl)	1.68	2.34	2.04
Iters. (excl)	5.06	6.86	5.88

I SEED SCATTERING THEME

We calculate the validation score as shown below:

Mean validation score =
$$\frac{1}{|\mathcal{X}|} \sum_{\langle p, H \rangle \in \mathcal{X}} \frac{1}{|\mathcal{S}_p|} \sum_{s \in \mathcal{S}_p} \text{proportion of validation tests passed}(s)$$
 (6)

We explore four themes: "None", "Jabberwocky", "Style", and "Role" to assess their impact on various evaluation metrics, including Pass@1, validation score, and similarity measures (BERT, TF-IDF, Levenshtein, and token sequence). Additionally, we provide false positive and negative rates, as well as true positive and negative rates, to further analyze the effects of the seed themes on performance in Table 19.

We also present examples of the SCATTERING seed instructions used for each theme and the metaprompt used to generate those instructions.

Table 19: Performance Metrics for varies seed instruction themes.

Seed instruction theme	None	Jabberwocky	Style	Role
Pass@1	72.50%	74.38%	79.38%	81.88%
Validation score	0.7786	0.7658	0.7548	0.7649
BERT similarity	0.9976	0.9944	0.9929	0.9957
TF-IDF similarity	0.9013	0.7559	0.6826	0.7734
Levenshtein similarity	0.8971	0.7749	0.7119	0.7907
Token sequence similarity	0.9361	0.8444	0.7504	0.8323
False positive rate	0.0500	0.0438	0.0500	0.0438
False negative rate	0.3000	0.3063	0.3500	0.3438
True positive rate	0.4250	0.4375	0.4438	0.4750
True negative rate	0.2250	0.2125	0.1563	0.1375
Pass@Any	75.00%	81.88%	88.13%	87.50%

Metaprompt: Roles of a Software Engineer

What are different possible roles that a software engineer can have, and what are the characteristics of each role?

You are a problem solver. You are analytical, logical, detail-oriented. You thrive on tackling complex problems and finding efficient solutions, enjoy the challenge of debugging and often see issues as puzzles to be solved, and are methodical in your approach and persistent in your efforts to overcome obstacles.

You are an innovator. You are creative, visionary, adaptable. You are always looking for new ways to apply technology. You are not just interested in how things work but also in how they can be improved or transformed. You enjoy pioneering new techniques and technologies and are comfortable with experimentation and risk-taking.

You are a communicator. You are interpersonal, collaborative, empathetic. You excel in environments where teamwork and collaboration are key. You are skilled at explaining complex technical details in simpler terms and bridging the gap between technical teams and non-technical stakeholders. You value relationships and work well in roles that require negotiation and coordination.

Metaprompt: Instructions for Writing Code in Different Styles

What are ten different possible instructions you can give to a software engineer before they write code, instructing them to write code in three different styles.

Write the code in a highly modular way, breaking down functionality into small, reusable components. Each function or class should have a single responsibility, and avoid large monolithic structures.

Use an object-oriented approach where each concept is modeled as a class. Leverage inheritance, encapsulation, and polymorphism to create a flexible, scalable design.

Write the code in a functional programming style, avoiding mutable state and side effects. Use pure functions, higher-order functions, and recursion where appropriate.

Metaprompt: Jabberwocky Poem by Lewis Carroll

Recite the poem "Jabberwocky" by Lewis Carroll.

'Twas brillig, and the slithy toves. Did gyre and gimble in the wabe:

All mimsy were the borogoves, And the mome raths outgrabe.

Beware the Jabberwock, my son! The jaws that bite, the claws that catch!