RETHINKING CODE SIMILARITY FOR AUTOMATED ALGORITHM DESIGN WITH LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

The recent advancement of Large Language Models (LLMs) has revolutionized the algorithm design patterns. A new paradigm, LLM-based Automated Algorithm Design (LLM-AAD), has emerged to generate code implementations for high-quality algorithms. Unlike the traditional expert-driven algorithm development, in the LLM-AAD paradigm, ideas behind the algorithm are often implicitly embedded within the generated code. Therefore, measuring similarity for algorithms may help identify whether a generated algorithm is innovative or merely a syntactic refinement of an existing code implementation. However, directly applying existing code similarity metrics to algorithms raises a critical limitation: they do not necessarily reflect the similarity between algorithms.

To address this, we introduce a novel perspective that defines algorithm similarity through the lens of its problem-solving behavior. We represent the problem-solving trajectory of an algorithm as the sequence of intermediate solutions progressively generated by the algorithm. The behavioral similarity is calculated by the resemblance between two problem-solving trajectories. Our approach focuses on how an algorithm solves a problem, not just its code implementation or final output. We demonstrate the utility of our similarity measure in two use cases. (i) Improving LLM-AAD: Integrating our similarity measure into a search method demonstrates promising results across two AAD tasks, proving the effectiveness of maintaining behavioral diversity in the algorithm search. (ii) Algorithm analysis. Our similarity metric provides a new perspective for analyzing algorithms, revealing distinctions in their problem-solving behaviors.

1 INTRODUCTION

The recent emergence of a new paradigm, *Large Language Model-based Automated Algorithm Design* (LLM-AAD), has drawn increasing attention due to its potential to automatically generate code implementations for expert-level algorithms. Unlike the traditional expert-driven development process, where ideas and design logic of algorithms are explicitly proposed before coding, in LLM-AAD, these ideas are often implicitly embedded within the generated code. Therefore, measuring algorithm similarity between their code may help identify whether an algorithm is innovative rather than a syntactic refinement of an existing code implementation.

Traditionally, similarity between code snippets has been extensively studied in software engineering, with widespread applications in diverse software engineering tasks such as code search (Keivanloo et al., 2014), clone detection (Roy et al., 2009), and evaluation (Dong et al., 2024). Current methods for measuing code similarity can be categorized into two primary paradigms. The first class analyzes programs in a static setting without executing them, calculating similarity based on various code features. These include token-based metrics (e.g., *BLEU*(Papineni et al., 2002), *ROUGE* (Lin, 2004)), structure-based metrics (e.g., *Abstract Syntax Tree* (Ren et al., 2020)), and embedding-based methods (e.g., *CodeScore* (Dong et al., 2024)). In contrast, execution-based metrics focus on the similarity of the execution output (Roziere et al., 2020), which assesses whether generated programs produce the same outputs as reference implementations on a suite of test cases.

However, simply applying existing code similarity metrics for algorithm similarity raises a critical gap: they do not necessarily reflect the similarity of algorithms. Figure 1 (a) demonstrates an example in which breadth-first search (BFS) and depth-first search (DFS) are used to perform a binary tree

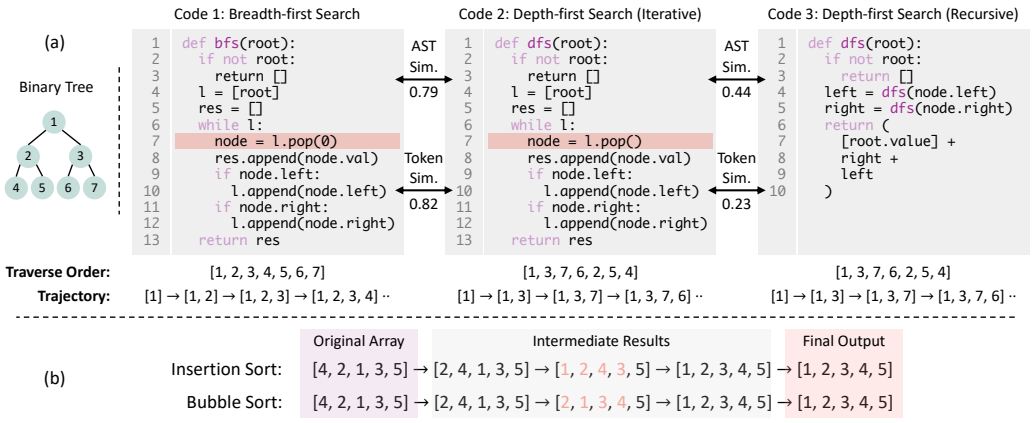


Figure 1: Examples demonstrating existing code similarity metrics are not capable of reflecting the similarity of algorithms. **(a)** Algorithms with similar code implementation may hold different ideas and design logic (Code 1 and Code 2), while different code implementations may share consistent design logic (Code 2 and Code 3). **(b)** Merely checking the output of two outputs cannot distinguish between two distinct algorithms.

traversal. The first code snippet (BFS) shares nearly identical implementation to the second code (Iterative DFS), with a similarity score of 0.79 in AST similarity and 0.82 in token similarity, both higher than that between the second and the third code snippets. However, the third code snippet is substantially an implementation variant of DFS, which is supposed to be identical to the second code in terms of design logic. Similarly, as shown in Figure 1 (b), calculating the similarity of the final outputs fails to distinguish between the insertion sort and the bubble sort algorithm, as their final outputs are exactly the same. Both examples collectively showcase that existing metrics are not sufficient for reflecting algorithm similarity.

To address this gap, we introduce ***BehaveSim***, a novel similarity metric from the behavioral perspective: We define algorithm similarity as the resemblance between two problem-solving trajectories, where a trajectory consists of the intermediate or partial solutions that are progressively generated by an iterative algorithm. A brief example is demonstrated in Figure 1 (b), where the insertion sort and the bubble sort exhibit different problem-solving trajectories (according to the sequence of their intermediate results), reflecting their distinction in problem-solving behavior. This behavioral view focuses on how an algorithm solves a problem, not just its code implementation or final output.

We demonstrate two direct use cases of *BehaveSim* in this paper: ❶ Improving LLM-AAD. Recent advances have suggested that promoting diversity is crucial for guiding the search toward high-quality algorithms (Romera-Paredes et al., 2024; Novikov et al., 2025; Wang et al., 2024). *BehaveSim* enables direct control over the diversity among generated algorithms. We integrate *BehaveSim* in a multi-island-based search method, demonstrating its superiority over existing state-of-the-art methods across two AAD tasks, proving the effectiveness of maintaining behavioral diversity in algorithm search. ❷ Algorithm Analysis. By analyzing generated algorithms from AAD runs, our findings indicate that *BehaveSim* can identify algorithms with distinct implementations while presenting identical behaviors, and algorithms with similar code structure may possess disparate behavior. This reveals the potential of *BehaveSim* to discover novel algorithms in problem-solving behavior.

Our primary contributions are:

1. We introduce the concept of algorithm similarity, demonstrating the necessity of measuring algorithm similarity from the perspective of its problem-solving behavior.

2. We introduce *BehaveSim*, a tangible way to measure behavior similarity for algorithms. We conduct extensive empirical studies to validate its effectiveness, showing that it can distinguish algorithms with disparate behaviors where existing code similarity metrics fail.

3. We demonstrate two direct usages of *BehaveSim* in LLM-AAD and algorithm analysis.

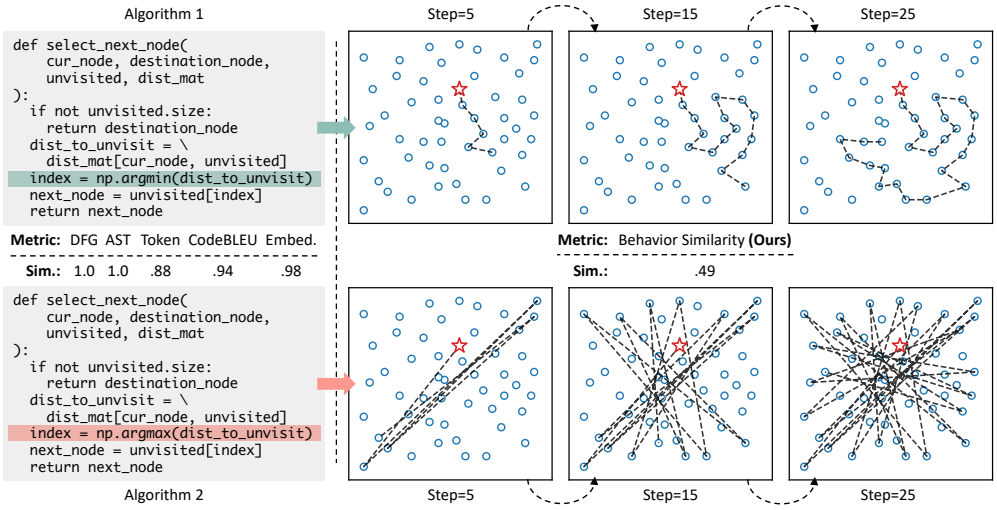


Figure 2: Problem-solving behaviors on the traveling salesman problem (TSP) for two algorithms with highly similar code. The only distinction in their implementations lies in the use of `argmin()` vs. `argmax()`, which leads to profoundly different behaviors. Nevertheless, existing similarity metrics still assign them a high degree of similarity, failing to reveal their behavioral distinction.

2 Revisiting Code Similarity Metric

This section provides an overview of existing code similarity measures and performs an empirical study to understand if they are capable of measuring behavioral similarity.

2.1 Existing Code Similarity Measure

Measuring code similarity is crucial for diverse software engineering tasks such as code evaluation (Dong et al., 2024), code clone detection (Roy et al., 2009; Svajlenko et al., 2014; Mou et al., 2015), and code search and reuse (Holmes & Murphy, 2005; Keivanloo et al., 2014). A common paradigm is to directly calculate the similarity between different types of code features. Token-based methods treat source code as natural language text. Methods such as *BLEU* (Papineni et al., 2002) and *ROUGE* (Lin, 2004) parse the source code into a sequence of tokens and match their similarity using *N-Gram*. In particular, *CtystalBLEU* (Eghbali & Pradel, 2023) and *CodeBLEU* (Ren et al., 2020) improve *N-Gram* by emphasizing code programming language-specific tokens. Structural-based methods calculate the similarity between the *Abstract Syntax Tree* (AST) structures (Gabel et al., 2008; Ren et al., 2020) and *Data Flow Graph* (DFG) (Ren et al., 2020). However, these approaches are technically incapable of capturing algorithmic similarity, since neither textual resemblance nor structural overlap can necessarily reflect the behavior of algorithms. As illustrated in Figure 2, even a minor modification of a function (i.e., `np.argmin()` v.s. `np.argmax()`) leads to substantial changes in their behavior. Nevertheless, both their DFG and AST similarities are measured as 1.0, indicating identical structural features in the implementation. This demonstrates that such metrics fail to capture differences in their behavior.

Other methods aim to capture features at a higher level of abstraction, such as control flow (Zhao & Huang, 2018), program dependency graphs (Liu et al., 2023), and code embeddings (Maveli et al., 2025; Günther et al., 2024). These approaches typically rely on training-based models to learn code similarity or to compare code token embeddings (Maveli et al., 2025; Zhou et al., 2023; Zhang et al., 2020). However, understanding the intended behavior of the code without running it is still challenging, as it requires a deep comprehension of code syntax and static semantic properties (Ding et al., 2024). As shown in Figure 2, the embedding model (Günther et al., 2024) also assigns high similarity scores to algorithms with textually similar code, indicating that they likewise struggle to measure underlying behavior for algorithms.

Another line of metrics considers execution-based metrics, which evaluate the similarity of execution results, such as computational accuracy (Roziere et al., 2020) and unit tests. However, these metrics

only compare the final outputs, which are too coarse-grained. As an example in Figure 1 (b), where both *Bubble Sort* and *Quick Sort* algorithms produce a consistent sorted array, yet their underlying behaviors are fundamentally different.

2.2 Empirical Study on Measuring Behavioral Similarity

While the preceding discussion and examples highlight the limitations of existing metrics in reflecting behavior, these cases are still limited. To this end, we curate a code similarity dataset to systematically evaluate whether existing metrics can measure behavioral similarity.

Code similarity can be examined from three different perspectives: code level, behavior level, and the result level, with a total of eight theoretical combinations as listed in Table 1. Among them, cases where all three levels are either entirely similar or entirely dissimilar (i.e., the first row and the last row) are not of interest. Moreover, the case where algorithms exhibit similar behaviors but produce different results is considered infeasible. Therefore, we focus on the remaining four types (*Type-1* to *Type-4*) for constructing our dataset.

Table 1: Eight theoretical combinations of algorithm similarity. "✓" indicates the algorithm is similar in this level, while "✗" indicates not similar.

	Code	Behavior	Result
	✓	✓	✓
	✓	✓	✗
Type-1	✓	✗	✓
Type-2	✓	✗	✗
Type-3	✗	✓	✓
	✗	✓	✗
Type-4	✗	✗	✓
	✗	✗	✗

Dataset Setup. Our dataset comprises four types of algorithm pairs, constructed to decouple code similarity, behavioral similarity, and results similarity. We detail each type with intuition and typical examples below. Specifications for individual data pairs are demonstrated in Appx. § A.1.

- *Type 1: High Code Similarity, Different Behaviors, Similar Results.* These pairs consist of algorithms that are textually similar but exhibit different behaviors. A typical example includes variations of matrix multiplication where subtle differences in the order of `i, j, k` lead to divergent internal processes, despite a high degree of code-level overlap.
- *Type 2: High Code Similarity, Different Behaviors and Results.* These algorithms are similar in code. But a small and critical change (e.g., a parameter or a conditional statement) not only alters their behavior but also leads to different final outputs.
- *Type 3: Low Code Similarity, Similar Behaviors and Results.* This category contains the same algorithms that are implemented in diverse ways (e.g., an iterative vs. a recursive implementation of Depth-First Search) but follow the exact same behaviors and results.
- *Type 4: Low Code Similarity, Different Behaviors, Similar Results.* Algorithms in this type are distinct not only in code implementation but also in behavior. However, they produce the same final output. For example, both *Quicksort* and *Bubblesort* yield a sorted array, but their behavior to swap each sub-array is completely different.

Compared Code Similarity Metrics. We compare our proposed *BehaveSim* against five categories of existing code similarity metrics: ❶ Token-level metrics, including *ROUGE*, *BLEU*, and *Crystal-BLEU*; ❷ Structure-level metrics, such as *AST* similarity; ❸ Hybrid metrics, such as *CodeBLEU*, which combines token and structural features via a weighted average; ❹ Embedding-based metrics, including *CodeBERTScore* (Zhou et al., 2023) and cosine similarity of learned embeddings (Günther et al., 2024); ❺ Execution-based metrics, which evaluate whether two algorithms yield identical results on the same set of inputs.

Results and Analysis. We report the average similarity value on each type of data in Table 2, with results on individual data pairs within each type listed in Appx. § A.1. The results provide compelling evidence for the necessity of our approach. We can conclude from the results that: Code-level metrics (Token-based, Structure-based, and Embedding-based) consistently yield high similarity scores for *Type-1* and *Type-2* data, as these pairs are designed to be similar in their implementation. However, they fail to measure *Type-3* pairs, which have identical behaviors but different code implementations, revealing that they cannot distinguish between superficial code differences and fundamental algorithmic behavior.

Table 2: Average similarity on four types of data calculated by various metrics.

Method Type	Method Name	Type-1	Type-2	Type-3	Type-4
Based on Token Match	*ROUGE*	0.95	0.96	0.7	0.47
	BLEU	0.83	0.94	0.42	0.16
	CrystalBLEU	0.97	0.99	0.68	0.51
Based on Structure	*AST*	0.96	1.0	0.76	0.57
Combine Token Match and Structure	*CodeBLEU*	0.97	0.94	0.91	0.75
Based on Embedding	*CodeBertScore*	0.84	0.97	0.6	0.38
	CodeEmbedding	0.99	0.99	0.9	0.84
Based on Execution Results	–	1.0	0.0	1.0	1.0
Similarity of their behavior (Ours)	*BehaveSim*	0.56	0.73	1.0	0.46

Although the execution-based metric correctly identifies *Type-3* and *Type-4* pairs as having identical results, this coarse-grained measure is incapable of distinguishing between *Type-1* (different behaviors) and *Type-4* (different behaviors, but same results), suggesting that simply checking for outputs is insufficient for evaluating behavioral similarity.

In contrast, our proposed *BehaveSim* metric demonstrated a correct understanding of behavioral similarity. *BehaveSim* reports a score of 1.0 on *Type-3* data, correctly identifying algorithms with similar behavior despite their textual differences. Simultaneously, *BehaveSim* assigns lower similarity scores to the other types of data, which are less similar in behavior. These results collectively underscore the necessity of *BehaveSim* in identifying behavioral similarity.

Algorithm 1: Generate Problem-solving Trajectory for an Algorithm.

Input: Initial solution s_0; Candidate generation function f_1;
Solution update strategy f_2; Maximum number of steps T
Output: The problem-solving trajectory $\mathcal{T}$

```
// Initialize the current solution and the trajectory
```
$s \leftarrow s_0$
$\mathcal{T} \leftarrow (s_0)$ `// The trajectory starts with the initial solution`
for $t \leftarrow 1$ **to** T **do**
  $s' \leftarrow f_1(s)$; `// f1: Internal logic to generate new candidate(s)`
  $s \leftarrow f_2(s, s')$; `// f2: Strategy to update the current solution`
  Append s to $\mathcal{T}$; `// Record the new solution in the trajectory`
return $\mathcal{T}$

3 Measuring Behavioral Similarity between Algorithms

Scope of Algorithms. This work focuses on a class of algorithms that solve problems iteratively. This scope includes algorithms for sorting, search, and optimization, where the solution is generated step by step rather than obtained in a single computation. We do not consider machine learning models, as their training and inference dynamics fall outside the iterative problem-solving paradigm studied in this work.

Problem-solving Trajectory. Iterative algorithms start from an initial solution and progressively produce a sequence of intermediate or partial solutions at each step. For example, in convex optimization, an intermediate solution may be a complete but suboptimal vector; in binary tree traversal, a partial solution might refer to a subset of nodes that does not yet constitute a full path. We collect these progressively generated solutions into a sequence, which we refer to as the algorithm's problem-solving trajectory. This trajectory reflects the behavioral characteristics of the algorithm. We summarize the general iterative procedure in Algorithm 1, where f_1 represents an internal logic for generating new candidate solution(s), and f_2 denotes a strategy for updating the current solution.

Pairwise Distance Between Two Solutions. We first define a distance measure between individual solutions before measuring the distance between two trajectories. The calculation of pairwise distance

depends on the solution types of the problem. The pairwise distance between different types of solutions is defined as below.

- *Categorical, ordinal, permutation solutions.* For these types, we adopt the edit distance $d_{\text{edit}}(x, y)$ as the pairwise distance. To ensure comparability across problem instances, the distance is normalized as:
$$d(x, y) = \frac{d_{\text{edit}}(x, y)}{d_{\max}},$$
where $d_{\max}$ is the maximum possible edit distance for the given problem.
- *Discrete and continuous solutions.* For these cases, we employ the Euclidean distance $d_{\text{euc}}(x, y) = \|x - y\|_2$, normalized by a problem-specific upper bound of distance D (e.g., the possible maximum distance in the domain):
$$d(x, y) = \frac{\|x - y\|_2}{D}.$$

Distance Between Trajectories. Given the pairwise distances between solutions, we define the distance between two trajectories using the *Dynamic Time Warping* (DTW) metric (Senin, 2008). DTW is particularly suitable because it aligns trajectories of potentially different lengths or with temporal shifts, enabling it to capture local similarities of two trajectories. This property is useful for comparing their trajectories. For example, in the *Traveling Salesman Problem* (TSP) (Matai et al., 2010), two algorithms may produce similar routes with different iteration timings, and DTW can still recognize such local behavioral similarity. Formally, given two trajectories $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$, the DTW distance is defined as:

$$DTW(X, Y) = \min_{\pi \in \mathcal{A}(m,n)} \sum_{(i,j)\in\pi} d(x_i, y_j),$$

where $\mathcal{A}(m, n)$ denotes the set of all possible alignment paths between the two trajectories. We define the *AlgoSim* score by normalizing the DTW distance by the length of the shorter trajectory:

$$\text{BehaveSim}(X, Y) = 1 - \frac{DTW(X, Y)}{|X|},$$

where $|X|$ represents the length of the shorter trajectory. We note that alternative methods for calculating trajectory distance are also feasible in this scenario. For example, we may calculate the *mean pairwise distance* for respective solutions. Further discussion on the choices of trajectory distances is provided in Appx. § A.2. Moreover, we highlight two additional considerations in the practical usage of our method: ❶ Average over multiple starting points. To compare the *BehaveSim* of two algorithms robustly, we evaluate the algorithms on multiple starting points (i.e., multiple problem instances) and compute the average *BehaveSim* as their final similarity. ❷ Truncate trajectories. It is often sufficient to compare only the early stages of trajectories before convergence. This avoids overemphasizing the convergent region, where trajectories of different algorithms may always be similar to each other.

3.1 VISUALIZATION

Optimizing Rosenbrock Function. We consider the continuous optimization problem of the Rosenbrock function, where each solution is represented as a continuous vector. Figure 3 (a) visualizes the problem-solving trajectories of different optimization algorithms (SGD (Fletcher, 1970), BFGS (Shanno, 1970), CG (Hestenes & Stiefel, 1952) on the same problem. For each pair of algorithms, we report the similarity score averaged over two different starting points. As shown in the left two plots (SGD vs. BFGS), the trajectories are less similar, leading to lower similarity scores. In contrast, the right two plots (CG vs. BFGS) show more similar trajectories, which is reflected in higher similarity scores.

TSP Problem. In the TSP, each solution (or partial solution) corresponds to a permutation of cities that defines the visiting order. As shown in Figure 3 (b), we compare the problem-solving behavior of three algorithms and visualize their partial solutions at steps 5, 25, and 50. The starting city is marked with a red star. We find that the similarity between *Algo1* and *Algo3* is higher than that between *Algo1* and *Algo2*, as reflected in both similarity results and intuitive observation, indicating that our approach can handle both continuous and permutation solution types, and it can reflect intuitive behavioral similarity between algorithms.

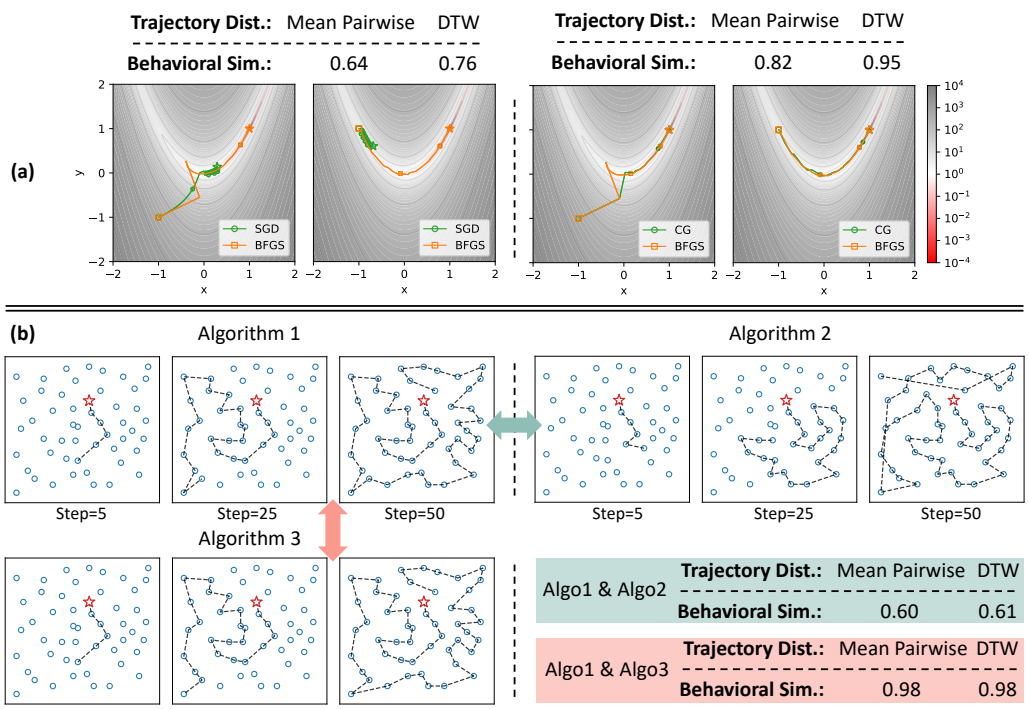


Figure 3: **(a)** Comparison of the problem-solving behavior for optimization algorithms. Each plot shows the trajectory of two algorithms on the *RosenBrock* problem, in which each algorithm is initialized at two different locations. Behavioral similarity is reported using both *mean pairwise distance* and DTW. **(b)** Comparison of TSP algorithms. Each plot shows the partial solution at different steps (5, 25, 50). The red star denotes the starting city.

4 Use Cases of BehaveSim

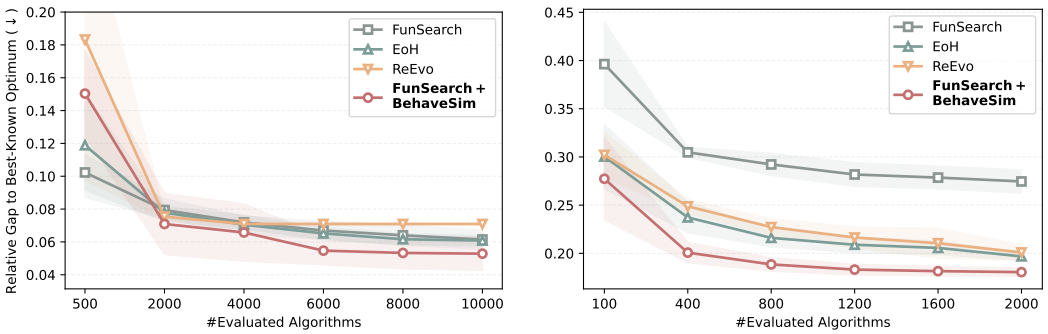
This section demonstrates two direct use cases of *BehaveSim* in LLM-AAD and algorithm analysis. *BehaveSim* enables direct control over the behavioral diversity among generated algorithms, which can potentially enhance search performance. *BehaveSim* also provides a behavioral perspective for algorithm analysis, which may offer new insights beyond traditional code-level analysis.

4.1 Integration in LLM-AAD Method

Method. *BehaveSim* can integrate into existing LLM-AAD methods to enhance the behavior diversity of the candidate algorithm. We demonstrate an example for synergistically integrating *BehaveSim* with *FunSearch* (Romera-Paredes et al., 2024). The key idea is to augment the multi-island database with a behavioral diversity management strategy, leveraging *BehaveSim* to cluster algorithms with similar behaviors within the same island. Due to the space limit, the implementation details are elaborated in Appx. §.A.4.

AAD Tasks and Compared Methods. We evaluate our approach on two AAD tasks, including Admissible Set Problem (ASP) (Romera-Paredes et al., 2024), Traveling Salesman Problem (TSP) (Matai et al., 2010). We compare our search method against three state-of-the-art LLM-AAD methods: *FunSearch* (Romera-Paredes et al., 2024), *EoH* (Liu et al., 2024), and *ReEvo* (Ye et al., 2024). We set the maximum number of evaluations to 10,000 for ASP due to its higher complexity. For TSP, the maximum numbers of evaluations are set to 2,000. Each candidate algorithm is evaluated with a timeout of 50 seconds, which is sufficient to cover nearly all feasible algorithms observed in our study. All methods interact with the recent closed-source *GPT-5-Nano*[1] model via API. More detailed settings on individual AAD tasks as well as compared methods are provided in Appx. § A.5.1.

[1] https://openai.com/index/introducing-gpt-5/



(a) ASP (b) TSP

Figure 4: Convergence of the top-10 algorithm sets produced by different search methods on three AAD tasks. The y-axis reports the relative gap to the best-known optimum (lower is better). Markers denote the mean over three independent runs; shaded regions indicate standard deviation.

Table 3: Performance of the top-1 and top-10 algorithms obtained by each AAD method. Metrics are the relative gap to the best-known optimum (%), reported as mean ± standard deviation over three runs (lower is better). Best results are in **bold**; second-best are underlined.

	ASP Performance		TSP Performance	
	Top-1	Top-10	Top-1	Top-10
FunSearch	$\underline{5.84}_{\pm 0.55}$	$\underline{6.15}_{\pm 0.59}$	$25.22_{\pm 0.56}$	$27.46_{\pm 1.21}$
EoH	$5.99_{\pm 0.20}$	$6.08_{\pm 0.31}$	$\underline{18.86}_{\pm 0.20}$	$\underline{19.68}_{\pm 0.42}$
ReEvo	$7.09_{\pm 0.10}$	$7.09_{\pm 0.10}$	$20.06_{\pm 0.83}$	$20.09_{\pm 0.82}$
Ours	$\mathbf{5.24}_{\pm 1.05}$	$\mathbf{5.28}_{\pm 1.01}$	$\mathbf{17.37}_{\pm 0.23}$	$\mathbf{18.05}_{\pm 0.42}$

Results Figure 4 presents the convergence curves of the performance of top-10 algorithms obtained by each method, with markers indicating mean performance aggregated over three independent runs and shaded regions denoting standard deviations. Complementary results for the top-1 and top-10 performances are reported in Table 3. The performance is calculated by the relative gap to the best-known optimum, with lower values indicating better performance.

We find that *FunSearch+BehaveSim* demonstrates superior performance, achieving the best performance on both top-1 and top-10 algorithms across these tasks. We note that both *EoH* and *ReEvo* are relatively sophisticated approaches compared to *FunSearch*, as they either co-evolve thought with code or incorporate an explicit reflection mechanism. Despite their complexity, our results suggest that simply encouraging behavioral diversity in algorithm search can yield comparable or even greater improvements. These results indicate that coupling search methods with the behavioral similarity improves both sample efficiency and final solution quality in LLM-AAD applications, underscoring the effectiveness of integrating *BehaveSim* into LLM-AAD. Due to the space limit, please refer to Appx. § A.5 for ablation studies and diversity analysis.

4.2 Algorithm Analysis

Our *BehaveSim* offers a novel behavioral perspective for analyzing algorithms. To illustrate its effectiveness, we randomly sample 30 algorithms from the final database checkpoint generated by the *FunSearch+BehaveSim* method in a single AAD run on the TSP problem. Figure 5 presents the clustering results obtained with two similarity measures, *BehaveSim* and *CodeBLEU*, where a higher branching point between two algorithms or clusters indicates a lower similarity.

We observe a significant discrepancy in their clustering results. Notably, we find that *Code6* and *Code8* have consistent behavior (as indicated by the red circle in the left clustering figure, the branch linking *Code6* and *Code8* is at the bottom line), yet they are distinct in code implementation (the branch linking *Code6* and *Code8* in the right clustering figure is very high). By visualizing the code of *Code6* and *Code8*, we realize that the reason they are similar in behavior is that they are inherently

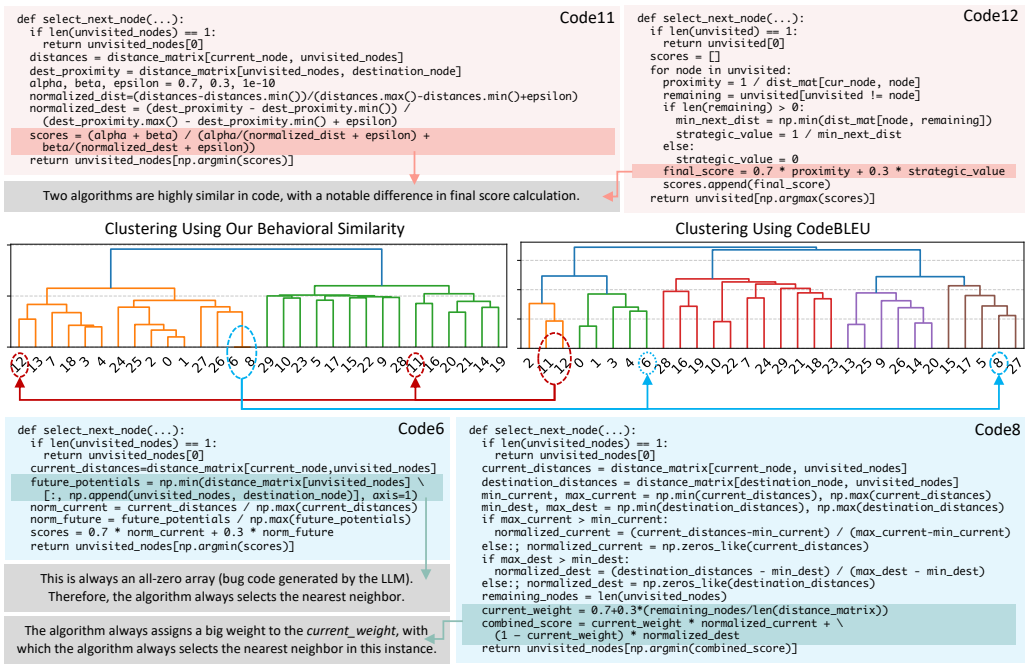


Figure 5: Clustering results based on two similarity measures. (i) *Code11* and *Code12* are clustered together by CodeBLEU, yet they exhibit distinct behaviors due to differences in their logic for computing the final score. (ii) *Code6* and *Code8* differ in code structure but display consistent behaviors. This is because a bug in Code6 sets the "`future_potentials`" field to an all-zero array, making its score solely determined by the distance to neighbors. Similarly, *Code8* assigns a dominant weight to the distance term, which overrides other factors. As a result, both *Code6* and *Code8* essentially follow the strategy of selecting the nearest unvisited neighbor.

selecting the nearest node. We also observe *Code11* and *Code12* share high *CodeBLEU* similarity (shown in the black circle in the right clustering figure), while being different in their behavior. The difference lies in their logic for calculating the final scores, which leads to the difference in behavior.

These observations highlight the contributions of analyzing algorithms from a behavioral perspective: It may help reveal which parts of the code lead to critical changes in their behavior, and discover similar design logic through finding similar behavior.

5 Conclusion

This paper introduces *BehaveSim*, a novel metric measuring algorithm similarity from a behavioral perspective. Through empirical comparisons with existing metrics, we conclude that *BehaveSim* can distinguish algorithms with disparate behaviors, whereas existing methods cannot. Two use cases of *BehaveSim* are demonstrated, showcasing the effectiveness of maintaining behavioral diversity in the algorithm search and its application in analyzing algorithm behaviors.

Limitations: First, our *BehaveSim* is currently designed for iterative algorithms and may not be directly applicable to other types of algorithms, such as machine learning algorithms. Future work could explore extending the concept of behavioral similarity to a wider range of algorithms. Second, the definitions of the problem-solving trajectory and the pairwise distance between solutions need to be customized specifically. Automating this process for arbitrary tasks remains an open challenge. Finally, algorithm similarity and novelty can be assessed along multiple dimensions beyond problem-solving behavior, such as computational complexity (e.g., time complexity, space complexity, etc.) or implementation structure. These perspectives are also crucial for AAD. In this work, we focus solely on behavioral similarity, leaving other dimensions for future exploration.

REFERENCES

Erick Cantú-Paz et al. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning, 2024. URL https://arxiv.org/abs/2406.01006.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution, 2024. URL https://arxiv.org/abs/2301.09043.

Aryaz Eghbali and Michael Pradel. Crystalbleu: Precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. doi: 10.1145/3551349.3556903. URL https://doi.org/10.1145/3551349.3556903.

Roger Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3): 317–322, 1970.

Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 321–330, 2008. doi: 10.1145/1368088.1368132.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents, 2024. URL https://arxiv.org/abs/2310.19923.

Magnus R Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

R. Holmes and G.C. Murphy. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 117–125, 2005. doi: 10.1109/ICSE.2005.1553554.

Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 664–675, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568292. URL https://doi.org/10.1145/2568225.2568292.

Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pp. 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-1013/.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *International Conference on Machine Learning*, pp. 32201–32223. PMLR, 2024.

Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 345–357, 2023. doi: 10.1109/ICSE48619.2023.00040.

Jinzhu Mao, Dongyun Zou, Li Sheng, Siyi Liu, Chen Gao, Yue Wang, and Yong Li. Identify critical nodes in complex network with large language models, 2024. URL https://arxiv.org/abs/2403.03962.

Rajesh Matai, Surya Prakash Singh, and Murari Lal Mittal. Traveling salesman problem: an overview of applications, formulations, and solution approaches. *Traveling salesman problem, theory and applications*, 1(1):1–25, 2010.

Nickil Maveli, Antonio Vergari, and Shay B. Cohen. What can large language models capture about code functional equivalence?, 2025. URL https://arxiv.org/abs/2408.11081.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing, 2015. URL https://arxiv.org/abs/1409.5718.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin (eds.), *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040/.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7): 470–495, 2009. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2009.02.007. URL https://www.sciencedirect.com/science/article/pii/S0167642309000367.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Pavel Senin. Dynamic time warping algorithm review. *Information and Computer Science Department University of Hawaii at Manoa Honolulu, USA*, 855(1-23):40, 2008.

David F Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24(111):647–656, 1970.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. Llm-sr: Scientific equation discovery via programming with large language models, 2025. URL https://arxiv.org/abs/2404.18400.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 476–480, 2014. doi: 10.1109/ICSME.2014.77.

Reiko Tanese. *Distributed genetic algorithms for function optimization*. University of Michigan, 1989.

Niki van Stein and Thomas Bäck. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 2024.

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, 2024. URL https://arxiv.org/abs/2409.03733.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. Multi-objective evolution of heuristic using large language model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 27144–27152, 2025.

Yiming Yao, Fei Liu, Ji Cheng, and Qingfu Zhang. Evolve cost-aware acquisition functions using large language models. In *International Conference on Parallel Problem Solving from Nature*, pp. 374–390. Springer, 2024.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution, 2024. URL https://arxiv.org/abs/2402.01145.

Rui Zhang, Fei Liu, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Understanding the importance of evolutionary search in automated heuristic design with large language models. In *International Conference on Parallel Problem Solving from Nature*, pp. 185–202. Springer, 2024.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert, 2020. URL https://arxiv.org/abs/1904.09675.

Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pp. 141–151, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236068. URL https://doi.org/10.1145/3236024.3236068.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code, 2023. URL https://arxiv.org/abs/2302.05527.

A Appendix

A.1 Detailed Results on Algorithm Similarity Dataset

We provided similarity results on individual data pairs for each type of data. Please refer to Table 4, Table 5, Table 6, Table 7, and Table 8 for detailed data pair specifications and results.

Table 4: Comparison of similarity metrics on **Type 1** cases, where code is syntactically similar, but execution paths are different while yielding similar final results. **Case 1**: Two matrix multiplication algorithms with different loop orders (ijk vs. jik). **Case 2**: Two BFS binary tree traversal algorithms, one expanding the left child first, the other expanding the right. **Case 3**: Two DFS binary tree traversal algorithms, one expanding the left child first, the other expanding the right.

Method	**Case 1**	**Case 2**	**Case 3**
ROUGE	0.89	0.98	0.98
BLEU	0.65	0.93	0.92
CrystalBLEU	0.92	0.98	1.00
AST	0.88	1.00	1.00
CodeBertScore	0.61	0.96	0.96
CodeEmbedding	0.97	0.99	1.00
CodeBLEU	0.96	0.96	0.99
BehaveSim	0.32	0.69	0.68

A.2 Choice of Trajectory Distance Measures

This experiment investigates the effect of different trajectory similarity measures on assessing behavioral similarity. We consider four measures: mean pairwise distance, Dynamic Time Warping (DTW) distance, Edit Distance with Real Penalty (ERP), and the average cosine similarity between

Table 5: Comparison of similarity metrics on **Type 2** cases, where code is syntactically similar, but both execution paths and final results are different. **Case 1**: Two graph traversal algorithms differing only by ‘min()‘ vs. ‘max()‘ to select the next node (nearest vs. farthest). **Case 2**: Two online bin packing algorithms with slight hyperparameter variations. **Case 3**: Another pair of online bin packing algorithms with slight hyperparameter variations.

Method	Case 1	Case 2	Case 3
ROUGE	0.95	0.97	0.97
BLEU	0.92	0.95	0.95
CrystalBLEU	0.99	0.99	0.99
AST	1.00	1.00	1.00
CodeBertScore	0.96	0.98	0.96
CodeEmbedding	0.99	1.00	0.99
CodeBLEU	0.88	0.97	0.97
BehaveSim	0.70	0.90	0.59

Table 6: Comparison of similarity metrics on **Type 3** cases, where code is syntactically dissimilar, but execution paths and final results are similar. **Case 1**: Recursive vs. iterative BFS. **Case 2**: Recursive vs. iterative Bubble Sort. **Case 3**: Recursive vs. iterative Insertion Sort. **Case 4**: Two implementations of Merge Sort. **Case 5**: Two equivalent implementations of First Fit for bin packing. **Case 6**: Two equivalent implementations of Best Fit for bin packing.

Method	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
ROUGE	0.74	0.61	0.57	0.51	0.88	0.91
BLEU	0.55	0.23	0.23	0.14	0.72	0.65
CrystalBLEU	0.75	0.57	0.58	0.62	0.88	0.70
AST	0.80	0.67	0.76	0.55	0.85	0.92
CodeBertScore	0.71	0.43	0.51	0.30	0.79	0.87
CodeEmbedding	0.91	0.88	0.89	0.82	0.96	0.95
CodeBLEU	0.85	0.87	0.88	0.89	0.98	0.99
BehaveSim	1.00	1.00	1.00	1.00	1.00	1.00

Table 7: Comparison of similarity metrics on **Type 4** cases where code and execution paths are dissimilar, but final results are similar (Part 1 of 2, Cases 1-9). Cases 1-14 involve pairwise comparisons of 6 different sorting algorithms.

Method	1	2	3	4	5	6	7	8	9
ROUGE	0.54	0.48	0.56	0.49	0.55	0.48	0.65	0.41	0.54
BLEU	0.12	0.22	0.17	0.20	0.21	0.10	0.31	0.12	0.18
CrystalBLEU	0.38	0.70	0.56	0.68	0.62	0.40	0.68	0.37	0.51
AST	0.64	0.60	0.70	0.57	0.56	0.66	0.64	0.50	0.69
CodeBertScore	0.31	0.52	0.37	0.32	0.35	0.49	0.46	0.31	0.43
CodeEmbedding	0.65	0.83	0.88	0.65	0.86	0.83	0.90	0.81	0.88
CodeBLEU	0.74	0.79	0.88	0.73	0.72	0.76	0.75	0.69	0.71
BehaveSim	0.64	0.49	0.79	0.69	0.44	0.42	0.46	0.45	0.64

trajectory segments. Figure 6 illustrates the similarities among trajectories computed using these methods.

We observe that while mean pairwise distance, DTW, and ERP suggest that CG and *NelderMead-Adaptive* are more similar, cosine similarity instead indicates that *CG* and *Newton-CG* exhibit greater similarity. This implies that different trajectory similarity measures capture different aspects of the trajectories.

Table 8: Comparison of similarity metrics on **Type 4** cases (Part 2 of 2, Cases 10-18, continued). Cases 10-14 involve sorting algorithms. Cases 15-18 involve shortest path algorithms.

Method	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
ROUGE	0.44	0.37	0.38	0.47	0.52	0.42	0.65	0.29	0.25
BLEU	0.07	0.08	0.09	0.14	0.22	0.14	0.38	0.07	0.05
CrystalBLEU	0.44	0.58	0.61	0.46	0.54	0.48	0.53	0.25	0.39
AST	0.45	0.39	0.48	0.65	0.64	0.50	0.73	0.49	0.42
CodeBertScore	0.40	0.39	0.42	0.25	0.44	0.28	0.49	0.30	0.24
CodeEmbedding	0.81	0.80	0.83	0.85	0.83	0.88	0.88	0.78	0.76
CodeBLEU	0.73	0.72	0.79	0.68	0.66	0.75	0.85	0.77	0.77
BehaveSim	0.44	0.51	0.36	0.78	0.29	0.37	0.35	0.00	0.05

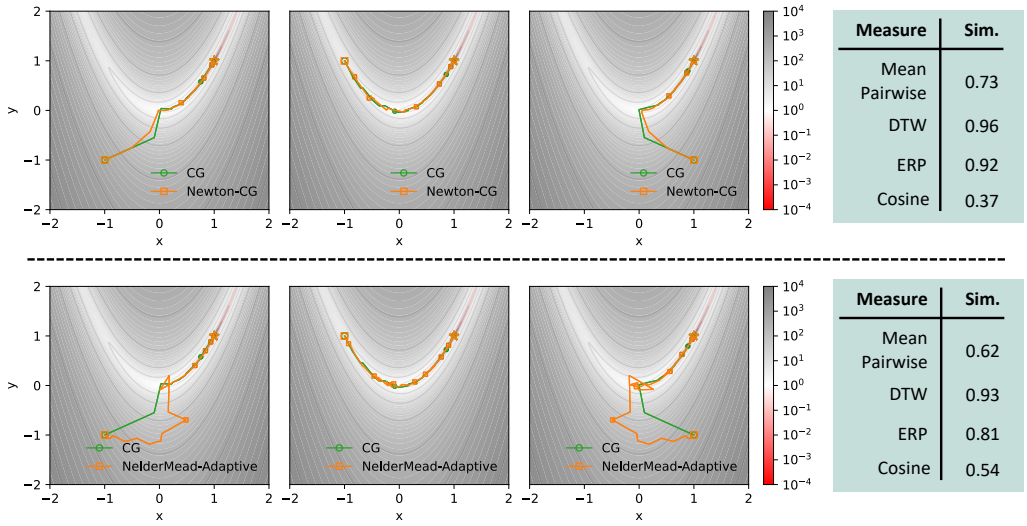


Figure 6: Demonstration of problem-solving trajectories on *RosenBrock* function.

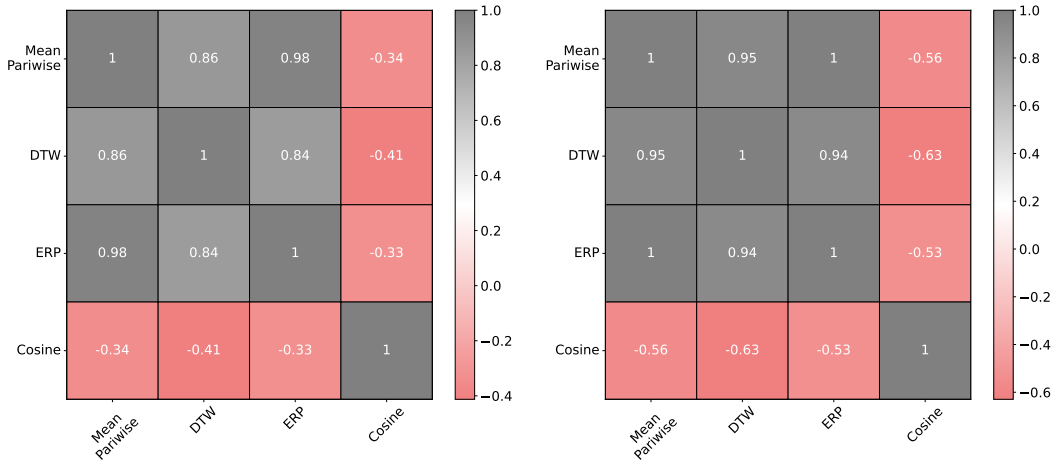
Specifically, we evaluate the problem-solving trajectories of eight algorithms on the Rosenbrock function: *CG*, *Newton-CG*, *SGD*, *Adam*, *BFGS*, *L-BFGS-B*, *NelderMead*, and *NelderMead-Adaptive*. For each pair of algorithms, we compute their trajectory similarity using the four measures, then we report Kendall's Tau (KTau) and Spearman's rank correlation coefficient to quantify their correlations. The correlation matrix is shown in Figure 7.

The results show that mean pairwise distance, DTW, and ERP exhibit high correlations with one another, while their correlations with cosine similarity are comparatively low. This suggests that mean pairwise distance, DTW, and ERP can serve as interchangeable alternatives, whereas cosine similarity primarily captures the directional consistency of trajectory segments and may therefore be particularly relevant in certain application scenarios.

A.3 Related Work on LLM-AAD

The integration between large language models (LLMs) with serach methods has become a pravailing paradimg in automated algorithm design (AAD) (Liu et al., 2024; Zhang et al., 2024), leading to notable advances across a spectrum of AAD applications, including mathematiacal discovery (Romera-Paredes et al., 2024), combinatorial optimization (Liu et al., 2024; Ye et al., 2024), Bayesian optimization (Yao et al., 2024), black-box optimization (van Stein & Bäck, 2024), and science discovery (Shojaee et al., 2025).

Notably, recent advances have emphasized the importance of diversity in algorithm search (Romera-Paredes et al., 2024; Novikov et al., 2025). For example, methods such as *FunSearch* (Romera-Paredes



(a) Kendall's Tau (KTau)

(b) Spearman Rank Correlation

Figure 7: Rank correlations of the similarities calculated by four trajectory similarity measures.

et al., 2024)and *LLM-SR* (Shojaee et al., 2025) adopt a multiple-island-based programs database to enhance diversity. *AlphaEvolve* (Novikov et al., 2025) combines the multiple-islands-based model with MAP elite (Mouret & Clune, 2015) to futhur improve diversity. *PlanSearch* (Wang et al., 2024) performs search in idea and pseudo code spaces to increase diversity. Other methods incorporate certain algorithm similarity metrics in the search process. For instance, *MEoH* (Yao et al., 2025) embeds AST distances into multi-objective algorithm search, Mao et al. (2024) clusters algorithms within sub-populations according to their embedding distance. In this paper, we introduce combining behavioral similarity with a search method to enhance the diversity and search performance.

A.4 Implementation Details of Our Search Method

Our search pipeline begins with a pre-defined "template algorithm" (detailed in Appx. § A.7), which provides task information for LLMs, including input/output for the designed algorithm, task descriptions, and argument formats. To initialize the database, N_{init} randomly generated algorithms are clustered based on behavioral similarity and subsequently allocated to different islands. The search process then proceeds iteratively: algorithms are selected from a database to serve as examples for the LLM to generate new algorithms. These new algorithms are evaluated to determine their fitness score and behavioral trajectory, and are then registered back into the appropriate island in the database. The search terminates after a fixed number of N_{eval} evaluations.

Algorithms Database. The algorithms database preserves a population of diverse algorithms obtained in the search phase. Inspired by prior works such as *FunSearch* (Romera-Paredes et al., 2024) and *AlphaEvolve* (Novikov et al., 2025), our database adopts a multiple-island-based population (Tanese, 1989; Cantú-Paz et al., 1998) to promote diversity. As shown in Figure 8(a), the database consists of a fixed size of N_{isl} islands, each of which groups algorithms with similar problem-solving behaviors as measured by our *BehaveSim*. Within each island, algorithms are further grouped into several clusters, where each cluster contains algorithms with an identical fitness score.

For the initialization of the database, instead of cloning a single seed algorithm to each island (like in *FunSearch*), we first sample a set of $N_{\text{init}} = 100$ algorithms. We then perform clustering based on their *BehaveSim* similarity and register these initial algorithms into the N_{isl} islands.

Algorithm Selection. The algorithm selection phase selects two different algorithms from the algorithm database as few-shot examples. We propose two selection strategies, **S1** and **S2**, specifically:

- **Inter-island Selection (S1):** To ensure the communication of two distinct islands, **S1** strategy choosees two distinct islands in the algorithms database. We subsequently obtain an algorithm from each of them.

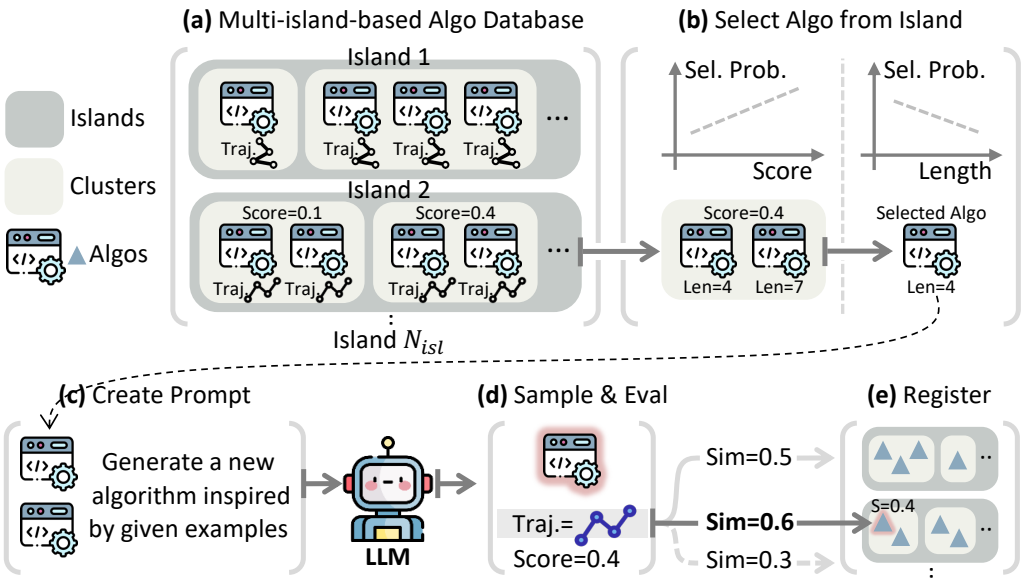


Figure 8: Overview of the Proposed Algorithm Search Method. **(a)** We utilize a multi-island algorithm database, where each island contains clusters of algorithms grouped by identical scores. **(b)** In the selection phase, an island is chosen from the database. Within the selected island, clusters are prioritized by score, with preference given to higher scores. Once a cluster is selected, an algorithm is chosen, favoring shorter implementations to promote conciseness. **(c)** The selected algorithms are used to create a prompt, which guides the LLM to generate a new algorithm. **(d)** The newly generated algorithm is evaluated to determine its fitness score and problem-solving trajectory. **(e)** Finally, the evaluated algorithm is assigned to the island with the highest behavioral similarity and placed into the appropriate cluster.

- **Intra-island Selection (S2):** As adopted in *FunSearch*, we randomly choose one island, and select two distinct algorithms within it.

To balance the utility of two strategies **S1** and **S2**, we set a hyperparameter p_{s1}, which determines the probability of adopting the **S1** strategy in the selection phase.

As shown in Figure 8 (b), the selection of an algorithm from a given island is a two-step process. First, we select a cluster within the island, giving preference to clusters with higher fitness scores. Second, within the chosen cluster, we select an algorithm prioritizing those with a shorter implementation (e.g., fewer lines of code) to promote conciseness and simplicity.

Sampling and Evaluation. As depicted in Figure 8, once two algorithms are selected, they are formatted into a prompt for the LLM. Following a two-shot prompting approach inspired by *FunSearch*, we sort the examples by their fitness scores in ascending order. The prompt demonstrates that the second algorithm performs better than the first, and then instructs the LLM to generate a potentially better algorithm. We perform two queries to the LLM for each prompt to obtain two new candidate algorithms. Due to the constraint of spaces, the elaborated prompt contents are shown in Appx. § A.6.

Algorithms sampled from the LLM are sent to a sandbox (implemented as a separate process isolated from the main process) for secure evaluation, in which we record a fitness score s representing its performance of solving the specific problem, and a trajectory t indicating its problem-solving behavior. We discard infeasible algorithms that either raise errors through evaluation or exceed timeout restrictions during evaluation.

Register Algorithms into Algorithm Database. After evaluation, feasible algorithms are registered in the database. A key distinction from previous methods is that we do not simply register the new algorithm back to its source island. Instead, we use *BehaveSim* to place it on the most similar island

in the database. Given a newly evaluated algorithm with trajectory t and a database of N_{isl} islands, where t_j^i is the trajectory of the j-th algorithm in island I_i, the target island is determined by:

$$\text{target\_island} = \underset{i \in [1, N_{\text{isl}}]}{\arg\min} \left(\frac{1}{|I_i|} \sum_{t_j^i \in I_i} \text{BehaveSim}(t, t_j^i) \right)$$

This equation calculates the average *BehaveSim* between the new algorithm's trajectory and all trajectories within each island and selects the island with the highest average similarity. The new algorithm is then placed in the appropriate cluster within the target island, or a new cluster is initialized if none exists for its fitness score.

Restarting Islands. To prevent the search from getting stuck in local optima and to manage the database size, we periodically restart a portion of the islands. Every two hours, we identify the $N_{\text{isl}}/2$ islands with the lowest-scoring best algorithms. We discard all algorithms within these islands and re-initialize them by randomly importing the best algorithms from the surviving $N_{\text{isl}}/2$ islands. This mechanism ensures a continuous influx of new ideas and prevents the database from becoming over-bloated with suboptimal solutions.

A.5 Experiment Details and Analysis

A.5.1 Settings for AAD Tasks and Methods

Admissible Set Problem (ASP). ASP aims to maximize the size of the set while fulfilling the criteria below: (1) The elements of the set are vectors belonging to $\{0, 1, 2\}^n$. (2) Each vector has the same number w of non-zero elements but a unique support. (3) For any three distinct vectors, there is a coordinate in which their three respective values are $\{0, 1, 2\}$, $\{0, 1, 2\}$, $\{0, 1, 2\}$. Following prior works (Romera-Paredes et al., 2024), we set $n = 15$ and $w = 10$ in this work.

Partial Solutions in ASP. The objective is to design a priority function that scores candidate vectors. At each step, the highest-scoring vector is added to the set, forming a partial solution. The distance between two partial solutions is defined as the edit distance between their current sets.

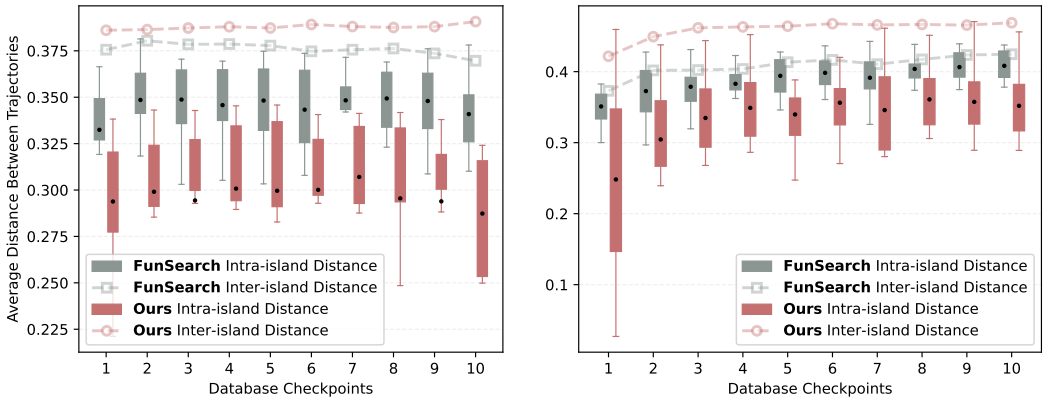
Traveling Salesman Problem (TSP). TSP aims to find a route that minimizes the total traveling distance for a salesman required to visit each city exactly once before returning to the starting point. We investigate the constructive heuristic design for TSP (Matai et al., 2010). Specifically, we adopt an iteratively constructive framework to start from one node and iteratively select the next node until all nodes have been selected, and return to the start node. The task is to design a heuristic for choosing the next node to minimize the route length. We generate five TSP instances, each comprising 50 cities, for training.

Partial Solutions in TSP. At each step, the algorithm selects the next city, yielding a progressively constructed route. A partial solution is defined as the ordered list of visited cities. The distance between two solutions is the edit distance between their routes, with similarity aggregated across five instances.

Settings for LLM-AAD Methods. For *FunSearch*, we use 10 islands for the database. Each prompt presents two reference algorithms, and we sample two algorithms per prompt. For *EoH* and *ReEvo*, the population size is set to 20. To ensure fairness, *EoH* is terminated based on the total evaluation budget instead of the maximum number of generations. All methods are evaluated under identical evaluation budgets and stopping criteria for a fair comparison.

A.5.2 Analysis of Diversity

Motivations. A core challenge in the search process is to balance exploration across diverse solution spaces with exploitation within promising regions. We hypothesize that our method with an optimized algorithms database can achieve a more effective balance compared to *FunSearch*. To verify this, we analyze the behavioral diversity of algorithms within and across islands. We compare our method against *FunSearch* under an identical multi-island database setup, where the only difference is the



(a) ASP (b) TSP

Figure 9: Comparison of intra- and inter-island distances for *FunSearch* and our method. Each box shows the distribution of intra-island distances across 10 islands per checkpoint, while the curves track average inter-island distances. Lower distances indicate similar problem-solving behaviors; higher distances indicate more diverse behaviors.

population management strategy. At each database checkpoint, we sample $N = 50$ algorithms from each of the 10 islands as the prototypes.

Intra-island distance. This metric quantifies the behavioral coherence within an island, reflecting local exploitation. For a set of representative trajectories $P_k = \{t_1, \ldots, t_N\}$ from an island k, the distance is:

$$D_{\text{intra}}(P_k) = \frac{1}{\binom{N}{2}} \sum_{1 \le i < j \le N} \left(1 - \text{BehaveSim}(t_i, t_j)\right), \quad t_i, t_j \in P_k \tag{1}$$

Inter-island distance. This metric measures the behavioral separation between islands, indicating global exploration. For two distinct islands k and l, the distance is:

$$D_{\text{inter}}(P_k, P_l) = \frac{1}{N^2} \sum_{t_i \in P_k} \sum_{t_j \in P_l} \left(1 - \text{BehaveSim}(t_i, t_j)\right) \tag{2}$$

Figure 9 visualizes these metrics, where boxes show the distribution of intra-island distances over all islands, and the curves track the average inter-island distance. We observe that:

- Our method yields consistently lower intra-island distances (red boxes vs. green). This indicates that each island maintains a more behaviorally coherent population, enabling focused **exploitation**.
- Simultaneously, our method achieves higher inter-island distances (red curves vs. green). This demonstrates that the islands are more behaviorally distinct, promoting global **exploration** across diverse problem-solving strategies.

These results reveal that our method demonstrates a superior balance between exploitation and exploration, which is crucial for the robust discovery of novel, high-performing algorithms.

A.5.3 Ablation Study

To better understand the contribution of each component in our framework, we conduct ablation studies on the ASP task. We focus on two factors: (i) the probability p_{s1} of adopting the inter-island selection strategy (**S1**) versus the intra-island strategy (**S2**), and (ii) the effect of clustering within islands during algorithm database construction. The results are shown in Figure 10.

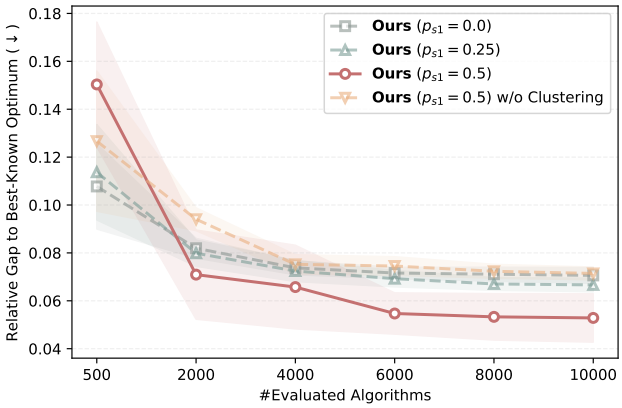


Figure 10: Ablation study on ASP. We evaluate the effect of inter-island selection probability p_{s1} and the clustering operation in the initialization of the algorithm database.

Effect of Inter-Island Selection (p_{s1}). We vary p_{s1} from 0 to 0.5. When $p_{s1} = 0$ (purely intra-island selection), the method degenerates into a variant similar to *FunSearch*, which restricts algorithm mixing to within a single island. Increasing p_{s1} consistently improves performance, with the best results obtained at $p_{s1} = 0.5$. This demonstrates the importance of promoting cross-island communication: algorithms from different islands capture diverse problem-solving behaviors, and combining them enhances exploration of the search space.

Effect of Clustering. We also evaluate a variant without clustering during database initialization. As shown in Figure 10, the absence of clustering significantly degrades performance, even when inter-island selection is enabled ($p_{s1} = 0.5$).

A.6 Prompt Content

The prompt content used in *FunSearch+BehaveSim* is shown below. The following prompt is used when we are initializing the algorithm database, or when there is only one algorithm on a certain island. where the field `template_algorithm` will be replaced by the template algorithm provided in the individual AAD task.

Prompt Content with Single Example Algorithm

```
Please help me design a novel Python algorithm function. Here is
    an example algorithm function implementation:

[Version 1]
{template_algorithm}

Please generate an improved version of the algorithm. Think
    outside the box. Do not modify the function signature (i.e.,
    function name, args, ...). Please generate your algorithm in
    ```python ...``` block. Only output the code and do not give
    additional outputs.
```

The following prompt is used to generate new candidate algorithms. The fields `algorithm_v1` and `algorithm_v2` will be replaced by two algorithms selected from the database. The algorithms are sorted according to their fitness score, with the lower one replacing the `algorithm_v1` and the higher one replacing `algorithm_v2`.
```

Prompt Content with Two Example Algorithms

```
Please help me design an novel Python algorithm function. Here is
    an example algorithm function implementation:

[Version 1]
{algorithm_v1}

We find that the below version outperforms [Version 1].

[Version 2]
{algorithm_v2}

Please generate an improved version of the algorithm. Think
    outside the box. Do not modify the function signature (i.e.,
    function name, args, ...). Please generate your algorithm in
    ```python ...``` block. Only output the code and do not give
    additional outputs.
```

A.7 Template Algorithm

We initialize all methods except for EoH with identical template algorithms to ensure fairness. The template algorithms employed in the ASP and TSP tasks are shown in the following listings.

Template Algorithm for ASP

```python
import math
import numpy as np

def priority(el: tuple[int, ...], n: int, w: int) -> float:
    """Returns the priority with which we want to add `el` to the
        set.
    Args:
        el: the unique vector has the same number w of non-zero
            elements.
        n : length of the vector.
        w : number of non-zero elements.
    """
    return 0.
```

Seed Algorithm for TSP

```python
import numpy as np

def select_next_node(
    current_node: int,
    destination_node: int,
    unvisited_nodes: np.ndarray,
    distance_matrix: np.ndarray
) -> int:
    """Design a novel algorithm to select the next node in each
        step.
    Args:
        current_node: ID of the current node.
        destination_node: ID of the destination node.
        unvisited_nodes: Array of IDs of unvisited nodes.
        distance_matrix: Distance matrix of nodes.
```
```

```
    Return:
        ID of the next node to visit.
    """
    next_node = unvisited_nodes[0]
    return next_node
```

B USE OF LARGE LANGUAGE MODELS

Large Language Models are employed in two ways in this work. (i) We use LLMs to aid or polish writing in this paper. (ii) This work investigates LLM-based automated algorithm design. Therefore, LLMs are used in the experiments.