Lyria: A General LLM-Driven Genetic Algorithm Framework for Problem Solving

Anonymous ACL submission

Abstract

While Large Language Models (LLMs) have demonstrated impressive abilities across various domains, they still struggle with complex problems characterized by multi-objective optimization, precise constraint satisfaction, immense solution spaces, etc. To address the limitation, drawing on the superior semantic understanding ability of LLMs and also the outstanding global search and optimization capability of genetic algorithms, we propose to capitalize on their respective strengths and introduce Lyria, a general LLM-driven genetic algorithm framework, comprising 7 essential components. Through conducting extensive experiments with 4 LLMs across 3 types of problems, we demonstrated the efficacy of Lyria. Additionally, with 7 additional ablation experiments, we further systematically analyzed and elucidated the factors that affect its performance[1].

1 Introduction

Large Language Models (LLMs) have demonstrated versatile abilities across various domains and tasks, benefiting from the large-scale corpora they are trained on (Jiang et al., 2024; Valmeekam et al., 2023; Pan et al., 2023; Tang and Belle, 2024). Nevertheless, their performance remains inferior, especially when faced with complex problems, characterized by their immense solution spaces, precise constraint satisfaction, multi-objective optimization, and domain-specific prior knowledge, such as reasoning (Mittal et al., 2025), planning (Valmeekam et al., 2023), theorem proving (Song et al., 2025), code generation (Jiang et al., 2024), and etc.

Genetic algorithms, a subset of evolutionary algorithms inspired by natural selection, powered by their essential operators such as selection, crossover, and mutation, are commonly used to approach optimal solution by iteratively optimizing the population through generations (Katoch et al., 2021; Gen, 2019; Koza, 1994). They have been studied and applied across diverse fields, such as reasoning (Hameed et al., 2023; Schäfer and Schulz, 2015; Tamaddoni-Nezhad and Muggleton, 2001), planning (Burns et al., 2024; Elshamli et al., 2004), combinatorial optimization (Shao et al., 2023; Kobler, 2009), and symbolic regression (Bertschinger et al., 2024; Ashok et al., 2020), primarily due to their ability to escape local optima, conduct systematical searches, and flexibly integrate domain-specific knowledge, thereby enabling them to approach global optimal solutions (Katoch et al., 2021).

Leveraging the semantic understanding and extensive prior knowledge acquired by LLMs from large-scale corpora (Minaee et al., 2025; Tang et al., 2023), as well as the capacity of genetic algorithm

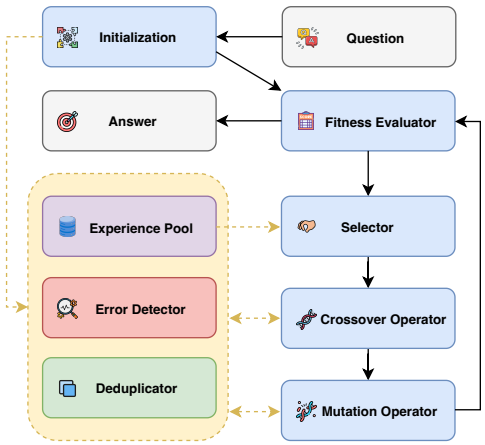

Figure 1: The Lyria framework, consisting of 7 essential components, i.e., *Error Detector*, *Experience Pool*, *Deduplicator*, *Fitness Evaluator*, *Selector*, *Crossover Operator*, and *Mutation Operator*, enables evolving candidate solutions through generations to obtain superior solution.

[1]Our code is available at https://anonymous.4open.science/r/Lyria.

to continuously optimize solutions within immense search spaces (Katoch et al., 2021; Gen, 2019), we propose to integrate them to capitalize on their respective strengths. Therefore, we introduce a general framework called Lyria, consisting of 7 essential components as illustrated in Figure 1, aiming to enhance the ability of LLMs to tackle complex problems. To evaluate its effectiveness, we conducted experiments using 4 LLMs on 3 NP problems against 2 baselines, demonstrating its significant performance improvements. We also executed 7 additional ablation experiments to analyze the impact of various factors on its performance.

We summarize our contributions as follows:

1. We proposed Lyria, a general LLM-driven genetic algorithm framework for problem solving, and demonstrated its effectiveness by evaluating with 4 LLMs and 3 types of NP problems;

2. We constructed dedicated prompts and domain-specific operators tailored to each type of problem, ensuring that Lyria is aligned with their unique requirements;

3. We conducted 7 additional ablation experiments to comprehensively analyze the impact of various factors on the performance of Lyria.

2 Related Work

Recently, research on the integration of LLMs and genetic algorithms has begun to emerge, demonstrating promising results across a variety of tasks.

Through synergistically combining LLMs with evolutionary algorithms, EvoPrompt (Guo et al., 2024) shows its efficacy to optimize discrete prompt generation, outperforming existing automatic prompt generation methods across various LLMs and tasks. In addition, Morris et al. (2024) proposed a novel framework which leverages LLMs to autonomously evolve neural network architectures through feedback-driven code modifications via Evolution of Thought and Character Role Play, while Nasir et al. (2024) proposed a method LLMatic that integrates the code-generation capabilities of LLMs and Quality Diversity algorithms which are a subset of evolutionary algorithms (Cully and Demiris, 2017; Pugh et al., 2016) to efficiently discover network architectures. Furthermore, Hemberg et al. (2024) proposed and demonstrated the way to replace traditional genetic programming operators with LLM-based operators. Moreover, Pinna et al. (2024) proposed an approach based on LLMs and Genetic Improvement to improve code generation.

Nevertheless, a general LLM-driven genetic algorithm framework along with a comprehensive analysis, has yet to be proposed, resulting in an incomplete and unclear understanding of it. Differing from prior work, this paper introduces a general LLM-driven framework comprising 7 principal components and offers a thorough, in-depth analysis of the various factors that may affect its performance.

3 Benchmarks

To evaluate Lyria, we selected 3 NP problems, i.e. Sudoku (Yato and Seta, 2003), Graph Coloring (Gent et al., 2017), and Traveling Salesman Problem (Papadimitriou and Steiglitz, 1976), characterized by their vast solution spaces and stringent constraints satisfaction requirements, thereby supposed to pose significant challenges to LLMs. It is worth noting that Lyria is not restricted to these problems, and the selection of them is only motivated by their complexity and the convenience of generating uncontaminated data. We describe each problem and their metrics in the following sections.

3.1 Sudoku

Sudoku (SK) is a number-placement puzzle played on a 9×9 grid, where each cell must be assigned a digit from 1 to 9. The grid is partitioned into nine 3×3 subgrids. A correct Sudoku solution requires that each row, column, and subgrid contains all digits from 1 to 9 exactly once. The puzzle is typically presented with some cells pre-filled, and a given solution must respect the constraints.

Data Generation We fixed the number of unfilled cells to 40 in each puzzle, generating a total of 50 distinct 9×9 SK instances.

Metrics We evaluate SK solutions using 3 metrics, namely *Correctness*, *Score*, and *Penalized Score*. *Correctness* checks whether a solution satisfies all row, column, and sub-grid constraints, with the correct percentage across all solutions reported as SK_{CR}. *Score* calculates the proportion of valid rows, columns, and subgrids in a solution and averages these three values, with higher scores indicating fewer errors. Since *Score* can be skewed by extremes, *Penalized Score* takes the geometric mean of those three proportions, dampening

solutions that over-optimize one constraint at the expense of others, with the average across all solutions reported as SK_{PS} in a range from 0 to 100. We formally define each metrics and detail them in Appendix A.1.

3.2 Graph Coloring

Graph Coloring (GC) is the task of assigning colors to the vertices of a given graph such that no two adjacent vertices share the same color. Formally, given a graph $G = (V, E)$ and a set of k distinct colors, the goal is to find a function $f : V \to \{0, 1, \ldots, k-1\}$ such that for any edge $(u, v) \in E$, $f(u) \neq f(v)$.

Data Generation We fixed the number of vertices $|V|$ to 9, the size of the color set k to 3 and the edge connection probability to 0.5. This process yields 50 distinct GC instances.

Metrics We evaluate GC solutions using 5 metrics, namely *Excess Color Usage*, *Conflict Ratio*, *Correctness*, *Score*, and *Penalized Score*. *Correctness* verifies if a solution uses exactly k colors and no adjacent vertex is conflicted, with the correct percentage across all solutions reported as GC_{CR}. *Penalized Score* is a modified score that penalizes solutions that use too many distinct colors, e.g., coloring each vertices with a distinct color, and it ranges from 0 to 100, where higher means better, with the average across all solutions reported as GC_{PS}. We formally defined the 5 metrics and detailed them in Appendix A.2.

3.3 Traveling Salesman Problem

Traveling Salesman Problem (TSP) is a route-finding task defined on a set of cities and pairwise distances between them. Formally, let $G = (V, E)$ be a complete undirected graph in which $V = \{v_1, \ldots, v_n\}$ is a set of vertices of the graph and $E = \{(u, v) : u, v \in V, u \neq v\}$ is a set of edges. A distance function $d : V \times V \to \mathbb{R}_{\geq 0}$ assigns each edge (u, v) a nonnegative distance $d(u, v)$. The goal in the TSP is to find a Hamiltonian cycle in G whose total distance is minimized. Formally, a route r is any permutation π of V, with the first element also appearing at the end, forming a cycle, defined as a sequence $r = [v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(1)}]$. The total distance of this route is given as: $\mathrm{D}(r) = d(v_{\pi(n)}, v_{\pi(1)}) + \sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)})$. Thus, the goal of TSP is to determine the route r^* in all possible routes R such that $\forall r \in R$, $\mathrm{D}(r) \geq \mathrm{D}(r^*)$, i.e., $r^* = \min_{\mathrm{r} \in R} \mathrm{D}(r)$.

Dataset Generation For each TSP problem, we fixed the number of cities $|V|$ to 10. The coordinate (x_v, y_v) of city v is sampled as $x, y \overset{i.i.d}{\sim} \mathcal{U}[0, 100]$ and the distance between cities are calculated by Euclidean distance. A start and end city is fixed and noted as v_1. An optimal reference route is then derived by exhaustively enumerating all Hamiltonian cycles beginning and ending at v_1. This procedure is repeated to produce 50 distinct TSP instances.

Metrics We evaluate TSP solutions using 4 metrics, namely *Excess Distance Multiplier* (EDM), *Missing Cities* (MC), *Correctness*, and *Penalized Score*. *Correctness* verifies if a route is the shortest cycle visiting all cities exactly once, with the correct percentage across all solutions reported as TSP_{CR}. *Penalized Score* considers both EDM and MC as penalties and is given in a range of 0 and 100 where higher is better, with the average across solutions reported as TSP_{PS}. We formally defined each of 4 metrics and detailed them in Appendix A.3.

4 Methodology

Lyria comprises 7 primary components: *Error Detector*, *Experience Pool*, *Deduplicator*, *Fitness Evaluator*, *Selector*, *Crossover Operator*, and *Mutation Operator*. We begin with a high-level overview of the framework, followed by a detailed elucidation of each component.

Initially, an LLM generates a population of candidate solutions. Every candidate is scored by the *Fitness Evaluator* and analyzed by the *Error Detector*. Evolution then proceeds in generations: a fraction of the lowest fitness individuals, determined by the *replay rate*, is replaced by the highest fitness candidates drawn from the *Experience Pool*; the *Selector* chooses appropriate parents; the *Crossover Operator*, guided by parental errors, generates offspring until the population size is restored; and the *Mutation Operator* modifies each candidate according to its own errors. After initialization and every crossover and mutation operations, the *Deduplicator* removes duplicates to maintain diversity. The updated population is re-evaluated and advanced in the next generation, until reaching the predetermined maximum number of generations. We demonstrate the pseudo code in Algorithm 1.

4.1 Error Detector

Inspired by Reflexion (Shinn et al., 2023) and Self-Refinement (Madaan et al., 2023), whenever a new

candidate is generated, the error detector (ED) identifies its errors up to a predefined *maximum detected errors*, enabling crossover and mutation operators to learn from past mistakes, thereby promoting the generation of improved candidates.

In Lyria, we proposed two types of EDs. A *Verifier-based* ED invokes external instruments, e.g., parsers, compilers, test suites, model checkers, etc., to examine a candidate against formal criteria and to emit deterministic and unbiased diagnoses. By contrast, an *LLM-based* ED prompts an LLM to introspectively evaluate the candidate, harnessing its knowledge and reasoning abilities trained on corpora. While the latter may hallucinate on identifying errors, it still remains indispensable when external verifiers are unavailable.

Different problem types typically have distinct error spaces. For each type of problem, we designed dedicated EDs and implemented them in both the Verifier-based and LLM-based approaches. For the Verifier-based EDs, we realized them programmatically. For the LLM-based EDs, we crafted dedicated prompt templates and demonstrated them in Prompt Template 1, 2, and 3. Due to space constraints, we detailed all the EDs in Appendix D.

4.2 Deduplicator

During initialization, crossover, and mutation, the deduplicator (DD) discards any individual that duplicates an existing one, requesting replacements until a preset *maximum deduplication attempts* is reached. This prevents identical candidates from dominating and preserves diversity.

Formally, given a sequence of candidates $C = [c_i]_{i=1}^{k}$ where k is the number of candidates generated, a newly generated candidate c_{k+1}, and a predefined *maximum deduplication attempts* τ, the deduplicator $\mathrm{DD}(C, c_{k+1}, \tau)$ operates as follows: if $c_{k+1} \notin C$, it returns c_{k+1}; if $\exists i \in \{1, \ldots, \tau\}$, $c_{k+1}^{(i)} \notin C$ and $\forall j < i$, $c_{k+1}^{(j)} \in C$, it returns $c_{k+1}^{(i)}$; otherwise, it returns $c_{k+1}^{(\tau)}$. Here, $c_{k+1}^{(i)}$ denotes a regenerated candidate. Hence, unique candidates are accepted immediately; duplicates trigger up to τ regenerations, with the first non-duplicate retained or the final candidate accepted if all fail.

4.3 Experience Pool

During initialization and after each generation, candidate solutions with their fitness scores and errors are recorded in the experience pool (EP). Before selection, the lowest fitness individuals in the population are systematically replaced with the highest fitness candidates in EP, with the number of replacements determined by a predefined *replay rate*. The EP preserves high-quality solutions and prevents inferior candidates from dominating the population, averting convergence toward suboptimal regions.

For the EP updating, formally, let EP_t denote EP at generation t, which is initialized as: $\mathrm{EP}_0 = \{(c_i, s_i, e_i) \,|\, c_i \in C_0,\, s_i \in S_0,\, e_i \in E_0\}$, where $C_0 = [c_i]_{i=1}^{n}$ is a sequence of candidates representing the initial population, $S_0 = [s_i]_{i=1}^{n}$ is a sequence of fitness score corresponding to each candidate in C_0, and $E_0 = [e_1]_{i=1}^{n}$ is a sequence of error information of each candidate in C_0. After each generation t, the experience pool is updated as: $\mathrm{EP}_{t+1} = \mathrm{EP}_t \cup \{(c_i, s_i, e_i) \,|\, c_i \in C_t,\, s_i \in S_t,\, e_i \in E_t\}$.

For the candidates replacement before selection, let $C_{t-1} = \{c_1, \ldots, c_n\}$ be the previous population and $\mathrm{C}_t^{EP} = \{c_1^\star, \ldots, c_m^\star\}$ be the candidates from EP at t. For the *replay rate* ρ, we define replacement count $k = \lfloor \rho \cdot n \rfloor$. Let permutation $\sigma^\uparrow$ sort C_{t-1} such that $s_{\sigma^\uparrow(i)} \leq s_{\sigma^\uparrow(j)}$ for all $i < j$, and permutation $\sigma^\downarrow$ sort C_t^{EP} such that $s^\star_{\sigma^\downarrow(i)} \geq s^\star_{\sigma^\downarrow(j)}$ for all $i < j$. We construct the new population $C' = [c'_1, c'_2, \ldots, c'_n]$, in which $c'_i = c^\star_{\sigma^\downarrow(i)}$ if $1 \leq i \leq k$ and $s^\star_{\sigma^\downarrow(i)} > s_{\sigma^\uparrow(i)}$, otherwise $c'_i = c_{\sigma^\uparrow(i)}$.

4.4 Fitness Evaluator

The fitness evaluator (FE) assigns each candidate a score, determining selection probability during evolution and influencing their crossover and mutation opportunities. Higher scores indicate the proximity of solutions to the optimum, increasing the possibility of selection to generate offspring. Lower scores reduce their selection probability. The FE critically guides the evolution and different FEs can exert distinct evolutionary processes. In Lyria, we propose two types of FEs, i.e., Oracle-based FE and LLM-based FE.

Oracle-Based FE The Oracle-based FE leverages an external verifier that deterministically returns a score for a candidate solution. Given a candidate and the associated scoring criteria, the verifier strictly adheres to the criteria, producing a precise score. For all 3 problem types, we implement their *penalized score* metric as the FE criteria in their verifiers to compute their fitness scores.

LLM-Based FE The LLM-based FE eliminates the need for external or handcrafted verifiers by prompting an LLM to generate scores. Although an LLM can occasionally deviate from the ground-truth score, it is still worthy and indispensable, when an external verifier is unavailable or costly to obtain. For each type of problem, we instruct the LLM to compute the fitness score based on their *penalized score* metric. We demonstrate all prompt templates in Prompt Template 4, 5, and 6.

4.5 Selector

Given a sequence of candidate solutions, the selector elects appropriate candidates and prepares a mating pool for subsequent crossover and mutation. While prioritizing individuals with high fitness may promote the generation of superior offspring, exclusively retaining them may impede population diversity and risk premature convergence to local optima. To balance exploration and exploitation, we implement a hybrid selection strategy for the selector that combines truncation selection with tournament selection. Nevertheless, we note that Lyria is not confined to a specific selection strategy and it can be substituted by any alternatives.

Let $C = [c_i]_{i=1}^n$ be a sequence of candidates with fitness scores $S = [s_i]_{i=1}^n$. Let $k_e \in [0, n]$ denote the number of fittest candidates that are directly carried forward in truncation selection. Let $k_r = n - k_e$ denote the number of candidates for tournament selection. The selector sorts C in descending order based on their fitness, giving $C_{\sigma\downarrow} = [c_{\sigma\downarrow(1)}, c_{\sigma\downarrow(2)}, \ldots, c_{\sigma\downarrow(n)}]$. Then, it selects a subsequence of k_e fittest candidates from $C_{\sigma\downarrow}$, corresponding to the truncation selection, as $C_{\text{trunc}} = [c_{\sigma\downarrow(1)}, c_{\sigma\downarrow(2)}, \ldots, c_{\sigma\downarrow(k_e)}]$. Then, let $I_{\text{tour}} = [(x_i, y_i)]_{i=1}^{k_r}$ be a sequence of index-pairs and $x_i, y_i \overset{\text{i.i.d.}}{\sim} \mathcal{U}[1, n]$. The candidates selected by tournament selection is given as $C_{\text{tour}} = [c_1^{\text{tour}}, c_2^{\text{tour}}, \ldots, c_{k_r}^{\text{tour}}]$, where $c_i^{\text{tour}} = c_{x_i}$ if $s_{x_i} > s_{y_i}$, otherwise $c_i^{\text{tour}} = c_{y_i}$. Thus, combining the truncation selection and tournament selection, the selector is defined as: $\text{Select}(C, S, k_e, k_r) = C_{\text{trunc}} \cup C_{\text{tour}}$. Hence, the candidates selected as the mating pool is $C' = \text{Select}(C, S, k_e, k_r)$.

4.6 Crossover Operator

The crossover operator (CO) selects parent pairs from the mating pool, governed by a predefined *crossover rate*. If crossover is skipped, a parent is randomly returned; otherwise, offspring are generated. This iterates until the offspring population reaches the predefined population size. The objective of CO is to combine advantageous traits of parents while suppressing detrimental ones to produce improved offspring with higher fitness. In Lyria, we propose two COs, i.e. LLM-based CO (LCO) and External CO (ECO), which are alternated in evolution based on an *external crossover rate*, for which lower prioritizes LCO while higher prioritizes ECO.

LCO In LCO, an LLM is prompted to merge two parent candidates by integrating their advantageous attributes and excluding their deficiencies, drawing on the experience and prior knowledge of the LLM and their understanding of the error information of the candidates provided by the ED, to produce an improved child. This approach eliminates the need to manually specify any domain-specific strategy, instead fully delegating it to the LLM. We demonstrated the prompts we designed for each of the 3 problem types in Prompt Template 7, 8, and 9.

ECO In ECO, two parent candidates are combined via external procedures or tools, based on domain-specific strategies and also the error information of the given parents. Varying from domains, distinct strategies can be employed, e.g., leveraging external heuristics provided by domain experts, formal logical constraints, etc. We designed specific ECOs for each problem type and detailed all of them in Appendix E.

4.7 Mutation Operator

The mutation operator (MO) applies mutations to each candidate based on a predefined *mutation rate*, returning either the original or mutated candidate. This preserves population diversity and prevents premature convergence. In Lyria, we propose two MOs, i.e., LLM-based MO (LMO) and External MO (EMO), which are alternated in evolution via an *external mutation rate*, for which lower prioritizes LMO while higher prioritizes EMO.

LMO In LMO, an LLM is instructed to mutate a given candidate, by identifying and improving its inferior parts, based on the knowledge of LLMs and their understanding of error information. Similar to LCO, this approach eliminates the necessity to manually construct domain-specific strategies and enables the LLM to autonomously design strategies to modify the given candidate. We designed the prompt for each problem type and demonstrated

Model	Method	SK_{CR}	SK_{PS}	GC_{CR}	GC_{PS}	TSP_{CR}	TSP_{PS}
GPT-4o-Mini	DP	0	39	0	73	0	79
	BoN	6 ↑6	73 ↑34	0	86 ↑13	4 ↑4	94 ↑15
	Lyria	**8** ↑8 ↑2	**73** ↑34 –	0	**97** ↑24 ↑11	**6** ↑6 ↑2	**96** ↑17 ↑2
Qwen2.5:32B-Instruct	DP	0	31	0	74	0	81
	BoN	8 ↑8	76 ↑45	0	87 ↑13	8 ↑8	97 ↑16
	Lyria	**32** ↑32 ↑24	**87** ↑56 ↑11	0	**96** ↑22 ↑9	**30** ↑30 ↑22	**99** ↑18 ↑2
Mistral:7B-Instruct	DP	0	0	0	0	0	60
	BoN	0	5 ↑5	0	84 ↑84	0	80 ↑20
	Lyria	0	**12** ↑12 ↑7	0	**92** ↑92 ↑8	0	**89** ↑29 ↑9
Qwen2.5:7B-Instruct	DP	0	26	0	73	0	34
	BoN	0	55 ↑29	0	84 ↑11	0	88 ↑54
	Lyria	0	**61** ↑35 ↑6	0	**95** ↑22 ↑11	**4** ↑4 ↑4	**95** ↑61 ↑7

Table 1: The results of Correctness and Penalized Score for SK, GC, TSP. For each LLM, across the three methods, the best correctness and penalized score are highlighted with a bold font. Blue arrows ↑↓ indicate performance differences relative to the DP baseline, while red arrows ↑↓ denote differences relative to the BoN baseline.

them in Prompt Template 10, 11, and 12.

EMO In EMO, the mutation is handled by external procedures or tools, guided by domain-specific mutation strategies and also the error information of the candidate, to modify the candidate. We designed specific EMOs for each problem type and elucidated them in detail in Appendix F.

5 Main Experiment

5.1 Baselines

We adapted two baselines, i.e., Direct Prompting (DP) and Best-of-N Direct Prompting (BoN), for comparative evaluation.

For DP, an LLM is invoked once and prompted to generate a solution directly in a zero-shot manner. We designed the prompt templates for each problem type and demonstrated them in Prompt Template 13, 14, and 15.

Since Lyria may sample an LLM up to L times for a single problem, a naive comparison against DP could favor Lyria merely by virtue of increased sampling[2]. To ensure a fair comparison, we adopt the BoN approach. For each problem, BoN draws $N = L$ independent responses from the LLM using the identical prompt template employed by DP and preserves the one with the best metrics as the answer. Additionally, same as Lyria, a deduplicator is introduced to remove redundant answers. This approach equalizes the number of sampling and ensures any observed performance improvements are attributable to the innovations of Lyria.

5.2 Experiment Settings

For a comprehensive evaluation, we selected 4 LLMs: *GPT-4o-Mini*, *Qwen2.5:32B-Instruct*, *Qwen2.5:7B-Instruct*, and *Mistral:7B-Instruct*.

For DP, we set the *temperature* to 0 for greedy decoding and the *maximum generated tokens* to 4096.

For BoN, we set the *temperature* at 0.7 to enable diverse generated answers, the *maximum generated tokens* at 4096, the *sampling times* N at 345 to align the number of queries with Lyria, and the *maximum deduplication attempts* at 3.

For Lyria, we switch on the Oracle-based FE. We set the *temperature* of LLM at 0.7, the *maximum generated tokens* at 4096, the *population size* at 30, the *generations* at 15, the *maximum detected errors* at 3, the *maximum deduplication attempts* at 3, the *replay rate* at 0.6, the *crossover rate* at 0.7, the *external crossover rate* at 0.3, the *mutation rate* at 0.3, and the *external mutation rate* at 0.3.

5.3 Results & Analysis

As shown in Table 1, LLMs struggle across problems in DP[3]. While BoN greatly improves the performance across the problems, Lyria demonstrates its ability to further consistently contribute significant improvement across various LLMs and

[2]The calculation of L is demonstrated in Appendix B.

[3]Results of all metrics for each problem type is shown in Appendix G.

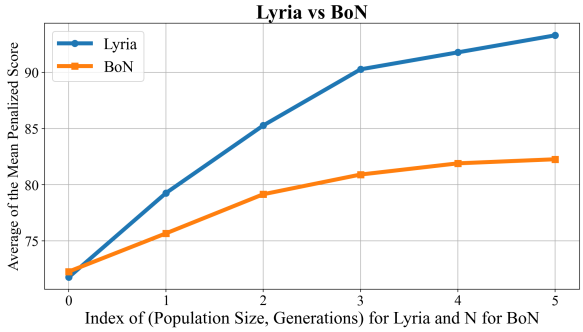

Figure 2: The figure shows the performance comparison between Lyria and BoN, in which the x-axis indexes each parameter set, e.g., index 0 means the pair of ($n_p = 5$, $n_g = 5$) for Lyria and $N = 23$ for BoN, and the y-axis shows the corresponding score averaging across SK_{PS}, GC_{PS}, and TSP_{PS}.

problems. For example, Lyria improves GC_{PS} for *GPT-4o-Mini* by 24% over DP and 11% over BoN, while enhancing SK_{CR} by 32% and 24% and also SK_{PS} by 56% and 11%, for *Qwen2.5:32B-Instruct*, compared to DP and BoN, respectively. In addition, for relatively small LLMs like *Qwen2.5:7B-Instruct*, Lyria also shows its efficacy, by 22% and 11% GC_{PS} increases, and 61% and 7% TSP_{PS} improved, compared to DP and BoN, respectively.

Furthermore, across all LLMs, for SK, Lyria shows an average 10% and 7% increases on SK_{CR} with 34% and 6% increases on SK_{PS}, compared to DP and BoN. For GC, Lyria shows 40% and 10% improvement on GC_{PS}. For TSP, Lyria shows 10% and 7% increases on TSP_{CR} with 32% and 5% improvement on TSP_{PS}. Therefore, across all LLMs and problems, Lyria demonstrates 7% and 5% increases on the correctness and 35% and 7% improvements on the penalized score, compared to DP and BoN, respectively, demonstrating the consistent performance contribution offered by Lyria.

6 Ablation Experiments

To further investigate the impact of various factors that influence the performance of Lyria, we conducted 7 additional experiments. To avoid prohibitive costs, we selected *Qwen2.5:7B-Instruct* and limited the number of problems to 10. Unless otherwise specified, we adhere to the same parameter settings as in the main experiment and refer their performance to the *penalized score* metric.

6.1 Scaling Population Size and Generations

This experiment investigates the impact of scaling *population size* n_p and *generations* n_g on the performance of Lyria. We executed 6 experiment settings, each pairing a n_p and n_g: $(5, 5)$, $(10, 10)$, $(20, 20)$, $(30, 30)$, $(40, 40)$ and $(50, 50)$. For each setting, we applied the BoN baseline for comparison, with the corresponding values of N equal to 23, 80, 300, 660, 1160, and 1800. As demonstrated in Figure 2, averaged across problems, while BoN exhibits diminishing marginal gains as parameters scaled, Lyria demonstrated consistent improvements and increasingly larger performance gaps compared to BoN. We attribute the limitations of BoN to LLMs getting trapped in local optima without effective capacities to extricate themselves from it, resulting in even sampling an arbitrarily large number of answers yet still failing to yield further performance improvements. However, Lyria inherently possesses the capacity to escape local optima, driving substantial performance improvements while increasing n_p and n_g.

In addition, to disentangle the individual contribution of n_p and n_g, we conducted 6 additional experiments settings. We fixed the n_p at 10 while varying n_g at values of 10, 30, and 50, and conversely fixed n_g at 10 while adjusting n_p across values of 10, 30 and 50. For the former, averaged across problems, the penalized scores increase by 4%, while the latter one yields 7% gains. The modest 3% difference between them could result from the limited diversity in smaller populations, causing offspring becoming homologous to their parents, thereby suppressing evolutionary efficacy. However, given this minor gap, we cannot exclude the possibility that it arises from stochastic variation.

6.2 Oracle-Based FE VS LLM-Based FE

This experiment seeks to explore how the performance of Lyria varies when using an Oracle-based FE versus an LLM-based FE. We compared an Oracle-based FE with two LLM-based FEs, one built on *Qwen2.5:7B-Instruct* and the other on *GPT-4o-Mini*.

We observed that, averaged across problems, the Oracle-based FE achieved a penalized score of 84, whereas *Qwen2.5:7B-Instruct* and *GPT-4o-Mini* scored only 51 and 50, respectively. The superior result of Oracle-based FE, as expected, shows that a stronger evaluator markedly boosts the performance of Lyria. Additionally, it is also worth noting that the nearly identical scores of *GPT-4o-Mini* and *Qwen2.5:7B-Instruct* indicate no significant difference in their evaluative capacity, although *GPT-4o-Mini* demonstrates a consistent better prob-

lem solving ability than *Qwen2.5:7B-instruct* as shown in Table 1. We expect the future work to seek to improve the ability of LLMs as evaluator and approach it to oracle level. Thus, the Oracle-based FE in Lyria can be fully replaced by LLMs, which is advantageous when an Oracle-based FE is unavailable or difficult to secure.

6.3 Impact of ED, EP, DD

This experiment aims to investigate the impact of the Error Detector, Experience Pool, and Deduplicator on Lyria.

For ED, we vary the *maximum detected errors* ϵ at values of 0, 3, 6, and 9. For TSP, we do not observe a significant impact when increasing ϵ. In contrast, for SK, as ϵ rises, the SK_{PS} also increased, yielding 4% gains. For GC, increasing ϵ produced a significant 7% improvements. We attribute the performance differences across problems to the varying efficacy of their dedicated design of ECO and EMO.

For EP, we vary the *replay rate* ρ at values of 0, 0.3, and 0.6. We observed that varying ρ did not produce significant changes in GC_{PS} and TSP_{PS}. However, for SK, while setting ρ to 0 and 0.6 yielded scores of 59 and 62, setting $\rho = 0.3$ produces a score of 73, bringing up a significant improvement of 14% and 11% compared to the scores when $\rho = 0$ and $\rho = 0.6$. We attribute this discrepancy to the trade-offs of ρ. When ρ is too low or EP is dropped, since the population of each generation evolves solely by referring to its immediate predecessors, the lack of retained historical best solutions may bias the evolutionary direction. Conversely, when ρ is too high, overreliance on historical best solutions which may themselves be local optima, can homogenize the evolved population and lead to premature convergence on suboptimal solutions.

For DD, we vary the *maximum deduplication attempts* τ at the values of 0, 3, and 6. Averaged across problems, increasing τ does not bring up a significant improvement, which, nevertheless, is as expected and does not mean that the deduplicator is dispensable. Since the solution spaces of all the given problems are considerably immense, and when the population size remains much smaller than the solution space, it results in a low incidence of duplicate individuals, DD therefore may not be invoked. Thus, when encountering problems with comparatively smaller solution spaces, DD could effectively eliminate duplicates and thereby enhance population diversity.

6.4 Impact of ECO and EMO

This experiment aims to investigate the impact of External Crossover Operator and External Mutation Operator on Lyria. Given the close interdependence between these two operators, rather than evaluating their efficacy in isolation, we simultaneously vary both the *external crossover rate* ξ and the *external mutation rate* μ to investigate the efficacy of them. Thus, we construct 3 experiment settings, each pairing a ξ and μ: $(0, 0)$, $(0.3, 0.3)$, and $(0.6, 0.6)$.

For GC, raising ξ and μ induces a 6% performance gain by improving GC_{PS} from 91 to 97. However, for TSP, we did not observe a significant performance gain after increasing the rates. Furthermore, for SK, we observed a 6% performance drop after raising rates. The disparity of the results illustrates that the quality of ECO and EMO designs tailored to specific problems can markedly influence performance. High-quality ECO and EMO enable Lyria to evolve populations more effectively, leading to better performance. We consider that a high-quality ECO and EMO may contain, but are not limited to, extra or superior heuristics beyond what an LLM alone can provide, structural or precise constraints, or expert domain knowledge. Conversely, poor designed of them may trigger performance declines. We consider that inferior operators can synthesize solutions worse than their predecessors, especially when they are frequently used in the case that ξ and μ are elevated, which can introduce low-quality individuals into each generation, thereby degrading performance. Therefore, a meticulous and superior design of ECO and EMO is essential for Lyria.

7 Conclusion

In this work, we introduced Lyria, a general LLM-driven genetic algorithm framework, which integrates the semantic understanding and reasoning abilities of LLMs with the global and systematic search capacity of genetic algorithms, comprising 7 essential components, to solve complex problems. We conducted extensive experiments with 4 LLMs across 3 types of problems to show the superior ability of Lyria, and also conducted 7 additional ablation experiments to demonstrate how various factors affect its performance. We hope this work offers valuable insights into the integration of LLMs with genetic algorithms and sparks further exploration in this field.

8 Limitations

Although our evaluation of Lyria focuses on SK, GC, and TSP problems, the framework is not restricted to these domains. We believe Lyria can be applied in a broader range of domains, especially for those problems characterized by multi-objective or discrete optimization, precise constraints, immense solution spaces, while also needing semantic understanding, such as planning in a dynamic environment, code synthesis, music generation, and other real-world applications. We encourage and expect the integration of Lyria into these domains in future works.

While Lyria exerts significant performance improvements, especially when the population size and generations are increased, it necessarily induces more LLM queries, leading to longer response time and higher costs. At present, it may not therefore be suited to applications that demand immediate responses or have a low budget. Thereby, reducing this overhead is an important goal for subsequent work.

In addition, in Section 6.2, we observed a large performance gap between the Oracle-based FE and the LLM-based FE. Since we do not expect, in practice, an Oracle-based FE is always available, we believe replacing the Oracle-based FE with an LLM-based FE can greatly increase the applicability and convenience of Lyria in real-world applications. Thereby, we expect future work to improve the LLM-based FE to approach the Oracle-based FE.

Finally, to our best knowledge, no existing similar framework is directly comparable to Lyria. Consequently, this paper concentrates more on the internal analysis to ensure that the observed performance improvements arise from the virtue of the framework itself rather than from other factors, e.g., increasing samplings, and examining the contribution and necessity of each component.

References

2019. *Handbook of Metaheuristics*, volume 272 of *International Series in Operations Research & Management Science*. Springer International Publishing, Cham.

Dhananjay Ashok, Joseph Scott, Sebastian Johann Wetzel, Maysum Panju, and Vijay Ganesh. 2020. Logic guided genetic algorithms. *ArXiv*, abs/2010.11328.

Amanda Bertschinger, James Bagrow, and Joshua Bongard. 2024. Evolving form and function: Dual-objective optimization in neural symbolic regression networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '24, page 277–285, New York, NY, USA. Association for Computing Machinery.

Owen Burns, Dana Hughes, and Katia Sycara. 2024. Plancritic: Formal planning with human feedback. *Preprint*, arXiv:2412.00300.

Antoine Cully and Yiannis Demiris. 2017. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation*, 22(2):245–259.

A. Elshamli, H.A. Abdullah, and S. Areibi. 2004. Genetic algorithm for dynamic path planning. In *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)*, volume 2, pages 677–680 Vol.2.

Ian P Gent, Christopher Jefferson, and Peter Nightingale. 2017. Complexity of n-queens completion. *Journal of Artificial Intelligence Research*, 59:815–848.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. *Preprint*, arXiv:2309.08532.

Shaima Hameed, Yousef Elsheikh, and Mohammad Azzeh. 2023. An optimized case-based software project effort estimation using genetic algorithm. *Information and Software Technology*, 153:107088.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. 2024. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. 2021. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, 80(5):8091–8126.

Daniel Kobler. 2009. *Evolutionary algorithms in combinatorial optimizationEvolutionary Algorithms in Combinatorial Optimization*, pages 950–959. Springer US, Boston, MA.

JohnR. Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2).

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh*

Conference on Neural Information Processing Systems.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2025. Large language models: A survey. *Preprint*, arXiv:2402.06196.

Chinmay Mittal, Krishna Kartik, Mausam, and Parag Singla. 2025. Fcorebench: Can large language models solve challenging first-order combinatorial reasoning problems? *Preprint*, arXiv:2402.02611.

Clint Morris, Michael Jurado, and Jason Zutty. 2024. Llm guided evolution - the automation of models advancing models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '24, page 377–384, New York, NY, USA. Association for Computing Machinery.

Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. 2024. Llmatic: Neural architecture search via large language models and quality diversity optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '24, page 1110–1118, New York, NY, USA. Association for Computing Machinery.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore. Association for Computational Linguistics.

Christos H. Papadimitriou and Kenneth Steiglitz. 1976. Some complexity results for the traveling salesman problem. In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, page 1–9, New York, NY, USA. Association for Computing Machinery.

Giovanni Pinna, Damiano Ravalico, Luigi Rovito, Luca Manzoni, and Andrea De Lorenzo. 2024. Enhancing large language models-based code generation by leveraging genetic improvement. In *Genetic Programming*, pages 108–124, Cham. Springer Nature Switzerland.

Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. 2016. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40.

Simon Schäfer and Stephan Schulz. 2015. Breeding theorem proving heuristics with genetic algorithms. In *GCAI*, pages 263–274. Citeseer.

Yinan Shao, Jerry Chun-Wei Lin, Gautam Srivastava, Dongdong Guo, Hongchun Zhang, Hu Yi, and Alireza Jolfaei. 2023. Multi-objective neural evolutionary algorithm for combinatorial optimization problems. *IEEE Transactions on Neural Networks and Learning Systems*, 34(4):2133–2143.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2025. Lean copilot: Large language models as copilots for theorem proving in lean. *Preprint*, arXiv:2404.12534.

Alireza Tamaddoni-Nezhad and Stephen Muggleton. 2001. Using genetic algorithms for learning clauses in first-order logic. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 639–646.

Weizhi Tang and Vaishak Belle. 2024. Tom-lm: Delegating theory of mind reasoning to external symbolic executors in large language models. In *Neural-Symbolic Learning and Reasoning: 18th International Conference, NeSy 2024, Barcelona, Spain, September 9–12, 2024, Proceedings, Part II*, page 245–257, Berlin, Heidelberg. Springer-Verlag.

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

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023. On the planning abilities of large language models: a critical investigation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Takayuki Yato and Takahiro Seta. 2003. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060.

A Metrics

A.1 Sudoku

Given a SK solution s' as a 9×9 matrix with each cell filled, formally we have:

$$s' = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,9} \\ u_{2,1} & u_{2,2} & u_{2,3} & \cdots & u_{2,9} \\ u_{3,1} & u_{3,2} & u_{3,3} & \cdots & u_{3,9} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{9,1} & u_{9,2} & u_{9,3} & \cdots & u_{9,9} \end{bmatrix}.$$

Let $\mathbb{S}$ denote the solutions to all SK problems. We mainly employ 3 metrics to assess the quality of the generated solutions:

Correctness The correctness of s' is defined as $\mathrm{CR(s')}$, which returns 1 if s' satisfies all the row, column, and subgrids constraints, otherwise 0. Formally, let $R(s', i)$ denote the values at row i, $C(s', j)$ denote the values at column j, and $B(s', k)$ denote the k-th 3×3 subgrids. The $\mathrm{CR}(s')$ is given as:

$$\mathrm{CR}(s') = \begin{cases} 1, & \text{if } (\forall i,\ |R(s', i)| = 9) \\ & \quad \wedge (\forall j,\ |C(s', j)| = 9) \\ & \quad \wedge (\forall k,\ |B(s', k)| = 9) \\ & \quad \wedge (\forall i, j,\ s'[i, j] \in \{1, \ldots, 9\}) \\ 0, & \text{otherwise} \end{cases}$$

We report the correctness percentage of $\mathbb{S}$ as Sudoku_{CR}.

Score Evaluating a Sudoku solution solely based on its correctness provides an overly narrow perspective, since when an LLM fails to provide a fully correct solution, it becomes challenging to observe and analyze whether it yields a high-quality albeit suboptimal solution and how closely it approximates the optimal solution. Thus, we define the score of a given s' as the average of percentages of constraints sanctification of rows, columns, and subgrids. Higher score means s' approaches more to the optimal solution. Formally, let $\mathbb{I}(X)$ be an indicator function that returns 1 if $X = \{1, \ldots, 9\}$, otherwise 0. Let $\mathbb{I}^R_{s'} = \sum_{i=1}^{9} \mathbb{I}(R(s', i))$, $\mathbb{I}^C_{s'} = \sum_{j=1}^{9} \mathbb{I}(C(s', j))$, and $\mathbb{I}^B_{s'} = \sum_{k=1}^{9} \mathbb{I}(B(s', k))$. We have $\mathrm{SC}(s')$ as:

$$\mathrm{SC}(s') = 100 \cdot \frac{1}{27}\left(\mathbb{I}^R_{s'} + \mathbb{I}^C_{s'} + \mathbb{I}^B_{s'}\right),$$

We report the average of $\mathbb{S}$ as Sudoku_{SC}

Penalized Score To prevent a solution from resorting to extreme strategies to improve correctness, such as boosting row and column correctness while sacrificing subgrid correctness, and thus ending up far from the optimal solution despite a seemingly high score, we adopt the geometric mean to mitigate the impact of these extreme values on the overall score. Formally, we have $\mathrm{PS}(s')$ as:

$$\mathrm{PS}(s') = 100 \cdot \sqrt[3]{\frac{\mathbb{I}^R_{s'}}{9} \cdot \frac{\mathbb{I}^C_{s'}}{9} \cdot \frac{\mathbb{I}^B_{s'}}{9}},$$

We report the average penalized score of $\mathbb{S}$ as Sudoku_{PS}.

A.2 Graph Coloring

Given a GC solution f', let $\mathbb{F}$ denote the solutions to all GC problems, we employ 5 metrics to more precisely evaluate the solution quality:

Correctness The correctness of f' indicates whether it assigns colors to each vertex such that no adjacent vertex has the same color. Formally:

$$\mathrm{CR}(f') = \begin{cases} 1, & \text{if } \left(\forall v \in V, f'(v) \in \{i\}_{i=0}^{k-1}\right) \wedge \\ & \quad \left(\forall (u, v) \in E, f'(u) \neq f'(v)\right), \\ 0, & \text{otherwise.} \end{cases}$$

We report the correctness percentage for $\mathbb{F}$ as GC_{CR}.

Excess Color Usage We define the excess color usage of a given f' as the number of distinct colors used that exceeds the allowed distinct colors k:

$$\mathrm{ECU}(f') = \left|\{\, f'(v) \mid v \in V \}\right| - k.$$

We report the average of $\mathbb{F}$ as GC_{ECU}.

Conflict Ratio We define the conflict ratio as the ratio of edges whose endpoints share the same color. Define an indicator function:

$$\mathbb{I}_{CF}(f', u, v) = \begin{cases} 1, & \text{if } f'(u) = f'(v), \\ 0, & \text{otherwise.} \end{cases}$$

The conflict ratio CF for a given f' is given as:

$$\mathrm{CF}(f') = \frac{\sum_{(u,v) \in E} \mathbb{I}_{CF}(f', u, v)}{|E|}.$$

We report the average of $\mathbb{F}$ as GC_{CF}.

Score We define the score of a given f' as the percentage of edges colored properly as:

$$\text{Score}(f') = \big(1 - \text{CF}(f')\big) \cdot 100.$$

We report the average of $\mathbb{F}$ as GC_{SC}.

Penalized Score The penalized score is a modified score that penalizes solutions using too many colors, e.g., coloring each vertices with a distinct color. Let $k' = |\{f'(v) : v \in V\}|$ be the total number of distinct colors used. Formally, we have PS as:

$$\text{PS}(f') = \begin{cases} 0, & \text{if } k' \geq |V|, \\ \text{Score}(f'), & \text{if } k' \leq k, \\ \text{Score}(f') \cdot R, & \text{otherwise.} \end{cases}$$

where $R = (1 - \dfrac{k' - k}{|V| - k})$ is the penalty ratio for the score. We report the average of $\mathbb{F}$ as GC_{PS}.

A.3 Traveling Salesman Problem

Given a TSP solution $r' \in R$ and $r' = [v'_1, v'_2, \ldots, v'_{|V|+1}]$, let $\mathbb{T}$ denote the solutions to all problems, we employ 4 metrics to analyze the solution quality:

Correctness The correctness of r' indicates whether it is the shortest route while starting and ending at v'_1. Formally:

$$\text{CR}(r') = \begin{cases} 1, & \text{if } (r' = \min\limits_{r \in R} \text{D(r)}) \\ & \wedge\ (v'_1 = v'_{|v|+1}) \\ & \wedge\ (|r'| = |V| + 1) \\ 0, & \text{otherwise.} \end{cases}$$

We report the average of $\mathbb{T}$ as TSP_{CR}.

Excess Distance Multiplier The excess distance multiplier of r' quantifies the factor by which a solution's distance exceeds the optimal distance $D(r^*)$. Formally:

$$\text{EDM}(r') = \min\Big(3, \frac{D(r') - D(r^*)}{D(r^*)}\Big)$$

The value of 0 indicates the solution's distance matches the optimal distance, while a value of 1 means the solution is twice as long as the option distance and so on. We make the value saturates at 3 as a default and maximum. We report the average of $\mathbb{T}$ as TSP_{EDM}.

Missing Cities A given route r' may skip some cities or revisit others. To measure this, we count the number of distinct cities that are missed from the solution. Formally:

$$\text{MC}(r') = |V| - |\{v \mid v \in r'\}|.$$

We report the average of $\mathbb{T}$ as TSP_{MC}.

Penalized Score The penalized score for r' measures how closely it approximates r^*, while applying penalties to the exceeded distance and also the omission of cities. Let $\text{Dist} = 1 - \frac{\text{EDM}(r')}{3}$ and $\text{Miss} = 1 - \frac{\text{MC(r')}}{|V|}$. Formally:

$$\text{PS}(r') = IS \cdot \min(\text{Dist}, \text{Miss})$$

where $IS = 100$ is the initialized score. We report the average of $\mathbb{T}$ as TSP_{PS}.

B L Calculation of Lyria

Lyria could sample an LLM up to L times for a single problem. Let n_p be the population size, n_g be the generations, η be the crossover rate, ξ be the external crossover rate, κ be the mutation rate, and μ be the external mutation rate. The sample times L are given as:

$$L = n_p + \big(n_p \cdot \eta \cdot (1 - \xi) + n_p \cdot \kappa \cdot (1 - \mu)\big) \cdot n_g$$

Thus, for example, given $n_p = 30, n_g = 15, \eta = 0.7, \xi = 0.3, \kappa = 0.3, \mu = 0.3$, we have $L = 30 + (30 \cdot 0.7 \cdot (1 - 0.3) + 30 \cdot 0.3 \cdot (1 - 0.3)) \cdot 15 = 345$.

C Lyria Pseudo Code

We demonstrate the pseudo code of Lyria in Algorithm 1, in which $\mathcal{P}$ means the problem, n_p means *population size*, n_g means *generations*, k_e means the number of fittest candidates that are directly carried forward in truncation selection, ϵ means the *maximum detected errors*, ρ means *replay rate*, τ means the *maximum deduplication attempts*, η means *crossover rate*, ξ means *external crossover rate*, κ means *mutation rate*, μ means *external mutation rate*, FE means the fitness evaluator which can either be Oracle-based or LLM-based, ED means the error detector which can either be Verifier-based or LLM-based, LCO means the LLM-based crossover operator, ECO means the external crossover operator, LMO means the LLM-based mutation operator, EMO means the external mutation operator, and llm indicates the LLM which accepts a prompt and returns a response.

Algorithm 1 Lyria

procedure LYRIA($\mathcal{P}, n_p, n_g, k_e, \epsilon, \rho, \tau, \eta, \xi, \kappa, \mu$, FE, ED, LCO, ECO, LMO, EMO, llm)
  $population, fitness, best\_fitness, best\_solution, errors$, EP, $\tau' \leftarrow [], [], -\infty, \emptyset, [], \emptyset, 0$
  while $|population| < n_p$ **do** ▷ Initialization
    $prompt \leftarrow$ DPPROMPT($\mathcal{P}$)
    $candidate \leftarrow llm(prompt)$
    if $child \in population$ and $\tau' < \tau$ **then** $\tau' \leftarrow \tau' + 1$; Continue **else** $\tau' \leftarrow 0$ ▷ Deduplicator
    $population, fitness \leftarrow population \parallel [candidate], fitness \parallel [\text{FE}(candidate)]$
    $errors \leftarrow errors \parallel [\text{ED}(candidate, \epsilon)]$
    if $fitness[-1] > best\_fitness$ **then**
      $best\_solution, best\_fitness \leftarrow candidate, fitness[-1]$
    end if
  end while
  for $n'_g \leftarrow 1$ **to** n_g **do** ▷ Start Evolution
    $population, fitness, errors \leftarrow$ Replacing $\lfloor n_p \cdot \rho \rfloor$ candidates with EP ▷ EP Replay
    $selected \leftarrow$ SELECT($population, fitness, k_e, |population| - k_e$) ▷ Selection Phase
    Update $fitness, errors$ to align with $selected$
    $offspring, os\_fitness, os\_errors, \tau' \leftarrow [], [], [], 0$
    while $|offspring| < n_p$ **do** ▷ Crossover Phase
      $c_1, c_2, s_1, s_2, e_1, e_2 \leftarrow$ RANDOMCHOICE($selected, fitness, errors$)
      if RD() $< \eta$ **then** ▷ Apply ECO or LCO , and RD() means $x \overset{i.i.d.}{\sim} \mathcal{U}[0, 1]$
        $child \leftarrow$ ECO($\mathcal{P}, c_1, c_2, e_1, e_2$) **if** RD() $< \xi$ **else** LCO($\mathcal{P}, c_1, c_2, s_1, s_2, e_1, e_2, llm$)
      else
        $child \leftarrow$ RANDOMCHOICE($[c_1, c_2]$)
      end if
      if $child \in offspring$ and $\tau' < \tau$ **then** $\tau' \leftarrow \tau' + 1$; Continue **else** $\tau' \leftarrow 0$
      $offspring, os\_fitness \leftarrow offspring \parallel [child], os\_fitness \parallel [FE(child)]$
      $os\_errors \leftarrow os\_errors \parallel [ED(child, \epsilon)]$
    end while
    $mutated, mt\_fitness, mt\_errors, \tau' \leftarrow [], [], [], 0$
    while $|mutated| < n_p$ **do** ▷ Mutation Phase
      $i \leftarrow |mutated|$
      $c, e, s \leftarrow offspring[i], errors[i], fitness[i]$
      if RD() $< \kappa$ **then** ▷ Apply EMO or LMO
        $child \leftarrow$ EMO($\mathcal{P}, c, e$) **if** RD() $< \mu$ **else** LMO($\mathcal{P}, c, s, e, llm$)
      else
        $child \leftarrow c$
      end if
      if $child \in mutated$ and $\tau' < \tau$ **then** $\tau' \leftarrow \tau' + 1$; Continue **else** $\tau' \leftarrow 0$
      $mutated, mt\_fitness \leftarrow mutated \parallel [child], mt\_fitness \parallel [FE(child)]$
      $mt\_errors \leftarrow mt\_errors \parallel [ED(child, \epsilon)]$
    end while
    $population, fitness \leftarrow mutated, mt\_fitness$ ▷ Update Population
    $errors \leftarrow mt\_errors$
    EP $\leftarrow$ EP $\cup \{(c, s, e) \mid c \in population, s \in fitness, e \in errors\}$
    Update $best\_solution$ and $best\_fitness$
  end for
  return $best\_solution$
end procedure

At Line 2, we initialize essential parameters. From Line 3 to 12, we use DPPROMPT to construct the prompt. For example, we use prompt templates shown in Prompt Template 13, 14, and 15 to construct the prompt for each problem type. Then, llm uses this prompt to generate new candidate. Deduplicator removes redundant candidates, FE assigns fitness score to the candidate, ED detect its errors up to ϵ for which we show the prompt templates we used in Prompt Templates 1, 2, and 3. For Line 13-46, Lyria starts evolution. At Line 14, EP intervenes to replace the lowest fitness candidates from the previous population with the highest fitness candidates from EP along with their fitness scores and errors. At Line 15-16, the selector selects appropriate candidates along with their fitness scores and errors. For Line 17-28, the crossover rate η determines whether to apply the crossover operation, while the external crossover rate ξ determines whether ECO is prioritized. For LCO, we demonstrate the prompt templates we used for each problem type in Prompt Template 7, 8, and 9. For Line 29-41, the mutation rate κ determines whether to apply the mutation operation, while the external mutation rate μ determines whether EMO is prioritized. For LMO, we demonstrate the prompt templates we used for each problem type in Prompt Template 10, 11, and 12. For Line 42-45, $population$, $fitness$, $errors$, and EP are updated, and the evolution advances into the next generation.

In addition, in practice, to prevent unnecessary additional overhead, especially when $best\_fitenss$ attains its optima, we also introduce a fitness threshold. During both initialization and the end of each generation, we check whether $best\_fitness$ meets or exceeds the fitness threshold. The $best\_solution$ is returned earlier if it meets the threshold. For every type of problem, the fitness threshold is set to 100.

D Error Detectors

D.1 SK ED

Given a SK solution s', we define 2 error types as follows:

Syntax Error (XE) For any generated SK solution by LLMs, we require it to be in the format of a 9×9 matrix, with each cell separated by a space, as described in Prompt Template 13. XE indicates whether s' is in a valid format. If the format is correct, $XE(s') = 0$, otherwise $XE(s') = 1$.

Semantic Error (SE) It consists of indices of cells that do not satisfy its row, column, and subgrid constraints. We define it as: $\text{SE}(c) = \{(i, j, t) \mid i, j \in \{1, \ldots, 9\} \wedge t \in \{0, 1, 2\}\}$, where i, j are row and column index respectively and t indicates the unsatisfied constraint, in which 0 indicates row, 1 means column, and 2 refers to subgrid.

D.2 GC ED

Given a GC solution f', represented as a sequence $\mathcal{F}' = [o'_1, o'_2, \ldots, o'_{|V|}]$ where $o_i = f'(v_i)$ is a color assignment for $v_i \in V$. We define two error types:

Syntax Error (SX) For any generated GC solution by LLMs, we require it to be in the format of a list of digits separated by commas, as described in Prompt Template 14. SX verifies whether $\mathcal{F}'$ is in a valid format. If the format is valid, $SX(\mathcal{F}') = 0$, otherwise 1.

Semantic Error (SE) It comprises two components:

- **Conflict Edges (CE)**: It consists of a set of edges where adjacent nodes share the same color, violating coloring constraints: $\text{CE}(f') = \{(u, v) \mid (u, v) \in E \wedge f'(u) = f'(v)\}$
- **Excess Colors Count (ECC)**: It indicates the number of distinct colors used that exceeds the specified color count k: $\text{ECC}(f') = |\mathcal{F}'| - k$

Thus, SE for f' is given as: $\text{SE}(f') = (\text{CE}(f'), \text{ECC}(f'))$.

D.3 TSP ED

Given a TSP solution r', we define two error types:

Syntax Error (XE) For any generated TSP solution by LLMs, we require it to be in the format of a list of digits separated by commas, with the first and last city being the same and equal to 0 and each city index in the range from 0 to $number\_of\_cities - 1$, as described in Prompt Template 15. XE indicates whether the solution is a route in a valid format. If it is valid, $XE(r') = 0$, otherwise 1.

Semantic Error (SE) It comprises two components:

- **Missing Cities (MC)**: It consists of the cities omitted in r': $\text{MC}(r') = \{v \mid v \in V \wedge v \notin r'\}$.

- **Excess Distance (ED)**: It indicates the difference between the distance used in r' and r^*: $\mathrm{ED}(r') = D(r') - D(r^*)$.

Thus, SE is given as: $\mathrm{SE}(r') = (\mathrm{MC}(r'), \mathrm{ED}(r'))$.

E External Crossover Operators

E.1 SK ECO

Let c_1, c_2 be parent candidates as two 9×9 matrix. Let P_{removed} denote the initially removed positions of the puzzle and $\{(i,j) \mid (i,j,t) \in \mathrm{SE}(c)\} \subseteq P_{\text{removed}}$. The crossover operator $\mathrm{CO}_{\text{Sudoku}}(c_1, c_2)$ produces the child c_{child} as:

$$c_{\text{child}} = \begin{cases} c_1 & \text{if } \mathrm{SE}(c_1) = \emptyset, \\ c_2 & \text{if } \mathrm{SE}(c_2) = \emptyset, \\ \Phi(c_2, P_{\text{corr}}(c_1)) & \text{otherwise,} \end{cases}$$

where $P_{\text{corr}}(c_1) = P_{\text{removed}} \setminus \{(i,j) \mid (i,j,t) \in \mathrm{SE}(c_1)\}$ are c_1's corrected positions, and $\Phi(c, P_{\text{corr}})$ updates c by replacing c's values at $\{(i,j) \mid (i,j,t) \in \mathrm{SE}(c)\} \cap P_{\text{corr}}$.

E.2 GC ECO

Let c_1, c_2 be two parent candidates, which are two GC solutions of which each represented as a sequence of color assignments, i.e., $c = [o_1, o_2, \ldots, o_{|V|}]$ where o_i is a color assignment for $v_i \in V$. Let $\mathrm{CV}(c) = \{v \mid \exists (v,u) \in \mathrm{CE}(c)\}$ denote conflict vertices in candidate c. The external crossover operator ECO_{GC} generates the child c_{child} as:

$$\forall i, c_{\text{child}}[i] = \begin{cases} c_2[i] & \text{if } i \in \mathrm{CV}(c_1) \setminus \mathrm{CV}(c_2), \\ c_1[i] & \text{if } i \in \mathrm{CV}(c_2) \setminus \mathrm{CV}(c_1), \\ \mathrm{R}(c_1[i], c_2[i]) & \text{otherwise,} \end{cases}$$

where $i \in \{1, \ldots, |V|\}$ indicates both an index in a color assignment sequence and also a city, and $\mathrm{R}(x,y)$ denotes uniform random selection between x and y. Thus, CO_{GC} transfers non-conflicting colors between parents while preserving valid vertex assignments.

E.3 TSP ECO

Let c_1, c_2 be two parent candidates, which are two routines, for which each routine is a sequence of visited cities. The external crossover operator $\mathrm{ECO}_{\text{TSP}}$ generates the child c_{child} as:

$$c_{\text{child}} = \begin{cases} c_1 & \text{if } \mathrm{SE}(c_1) = (\emptyset, 0), \\ c_2 & \text{if } \mathrm{SE}(c_2) = (\emptyset, 0), \\ \Psi(c_1, c_2, k) & \text{otherwise,} \end{cases}$$

where $k \overset{\text{i.i.d.}}{\sim} \mathcal{U}[1, n-1]$ is a uniformly random crossover point, and

$$\forall i, \Psi(c_1, c_2, k)[i] = \begin{cases} c_1[i] & \text{for } i \le k, \\ c_2[i] & \text{for } i > k, \end{cases}$$

where $i \in \{1, \ldots, |V|\}$ is an index of a seqeunce. Thus, CO_{TSP} merges partial routes from both parents while preserving order.

F External Mutation Operators

F.1 SK EMO

Given a candidate c as a 9×9 matrix, the mutation operator $\mathrm{MO}_{\text{Sudoku}}$ is given as:

$$\mathrm{EMO}_{\text{Sudoku}}(c) = \begin{cases} c & \text{if } \mathrm{SE}(c) = \emptyset, \\ \Theta(c, p, v) & \text{otherwise,} \end{cases}$$

where $p \overset{i.i.d.}{\sim} \mathcal{U}(\{(i,j) \mid (i,j,t) \in \mathrm{SE}(c)\})$ is an error position, $v \overset{i.i.d.}{\sim} \mathcal{U}[1,9]$ is a random value, and $\Theta(c,p,v)$ denotes replacing c's value at the position p with the value v.

F.2 GC EMO

Given a candidate c as a sequence of color assignments, the mutation operator MO_{GC} proceeds as:

$$\mathrm{EMO}_{\text{GC}}(c) = \Gamma(\Lambda(c)).$$

It is composed of two main components, i.e., Conflict Edges Resolution and Excess Colors Correction, denoted as Λ and Γ respectively.

Conflict Edges Resolution is given as:

$$\Lambda(c)[i] = \begin{cases} y & \text{if } c[i] \in \mathrm{CV}(c) \\ c[i] & \text{otherwise,} \end{cases}$$

where $i \overset{i.i.d.}{\sim} \mathcal{U}[1, |c|]$ is a randomly selected index, and $y \overset{i.i.d.}{\sim} \mathcal{U}(\{0, \ldots, k-1\} \setminus \{c[i]\})$ is a new color value. Hence, it reassigns a new color to a vertex if it is conflicted such that $y \neq c[i]$.

Excess Colors Correction is given as:

$$\forall i, \Gamma(c)[i] = \begin{cases} z & \text{if } c[i] \in O, \\ c[i] & \text{otherwise.} \end{cases}$$

Model	Method	SK_{CR}	SK_{SC}	SK_{PS}
GPT-4o-Mini	DP	0	43	39
	BoN	6	74	73
	Lyria	8	74	73
Qwen2.5:32B-Instruct	DP	0	35	31
	BoN	8	77	76
	Lyria	32	87	87
Mistral:7B-Instruct	DP	0	1	0
	BoN	0	11	5
	Lyria	0	16	12
Qwen2.5:7B-Instruct	DP	0	32	26
	BoN	0	57	55
	Lyria	0	62	61

Table 2: The results of all metrics for SK.

where $i \in \{1, \ldots, |c|\}$, $z \overset{i.i.d.}{\sim} \mathcal{U}(\{0, \ldots, k-1\})$ and $O = \{k, \ldots, k + \text{ECC}(c)\}$. Hence, it replaces all color values $\geq k$ with random valid colors $z \in \{0, \ldots, k-1\}$.

F.3 TSP EMO

Give a candidate c as a route, which is a sequence of visited cities, let Y be a set of duplicated cities in c, and let $M = \text{MC}(c)$ be the missing cities, the mutation operator EMO_{TSP} is given as:

$$\text{EMO}_{\text{TSP}}(c) = \begin{cases} c & \text{if } M = \emptyset, \\ \Omega(c) & \text{otherwise,} \end{cases}$$

where Ω is defined as:

$$\forall i, \Omega(c)[i] = \begin{cases} \varphi(i) & \text{if } c[i] \in Y \\ c[i] & \text{otherwise,} \end{cases}$$

in which $i \in \{1, \ldots, |c|\}$, φ is an injection from the first $r = \min(|Y|, |M|)$ elements of Y into M. Hence, it resolves errors by substituting duplicates with missing cities, ensuring the mutated child becomes a route without missing cities.

G Additional Results

We demonstrate the results of all metrics for each problem type in Table 2, 3, and 4.

H Prompts

Prompt Template 1: Sudoku LLM-Based Error Detector

===Instructions===
1. You are a Sudoku expert who can find the errors in a sudoku candidate solution;
2. Given this Sudoku puzzle and its candidate solution, you should find the errors in the candidate solution;
3. The correctness of the solution depends on:
(1) Correct Syntax: it has a correct format, meaning each row, column, and {subgrid_size}x{subgrid_size} square exactly contain {puzzle_grid_size} number, and each cell is separated by space. The solution format must be in the same format as the given puzzle but there is no unfilled dot;
(2) Correct Semantics: for each row, column, and {subgrid_size}x{subgrid_size} square, the numbers 1 to {puzzle_grid_size} should appear exactly once;
4. If the syntax is incorrect, the errors should be "Syntax is wrong" (Noted as Type 1 Error Msg);
5. If the syntax is correct, the errors should be the positions of the wrong numbers in the candidate solution with its conflict type, "row", "col", or "subgrid" (Noted as Type 2 Error Msg);

Model	Method	GC_{CR}	GC_{SC}	GC_{PS}	GC_{ECU}	GC_{CF}
GPT-4o-Mini	DP	0	73	73	0	27
	BoN	0	86	86	0	14
	Lyria	0	97	97	0	4
Qwen2.5:32B-Instruct	DP	0	74	74	0	26
	BoN	0	87	87	0	13
	Lyria	0	96	96	0	4
Mistral:7B-Instruct	DP	0	100	0	12	0
	BoN	0	86	84	0	15
	Lyria	0	93	92	0	7
Qwen2.5:7B-Instruct	DP	0	73	73	0	27
	BoN	0	84	84	0	16
	Lyria	0	95	95	0	5

Table 3: The results of all metrics for GC.

Model	Method	TSP_{CR}	TSP_{PS}	TSP_{EDM}	TSP_{MC}
GPT-4o-Mini	DP	0	79	0.64	0
	BoN	4	94	0.18	0
	Lyria	6	96	0.13	0
Qwen2.5:32B-Instruct	DP	0	81	0.58	0
	BoN	8	97	0.09	0
	Lyria	30	99	0.04	0
Mistral:7B-Instruct	DP	0	60	1.20	2
	BoN	0	80	0.61	0
	Lyria	0	89	0.31	0
Qwen2.5:7B-Instruct	DP	0	34	1.97	5
	BoN	0	88	0.36	0
	Lyria	4	95	0.15	0

Table 4: The results of all metrics for TSP.

6. You should find all the errors in the candidate solution;
7. If there are no errors, the errors should be "No errors" (Noted as Type 3 Error Msg);
8. You can think it thoroughly in any way you want, but You MUST give the errors in the end of your thinking in the format as:
(1) For Type 1 Error Msg: "Syntax is wrong" wrapped in triple backticks as a code block;
(2) For Type 2 Error Msg:
a. Each error is in the format as "i,j,type", where i is the row number and j is the column number starting from 0 and type is the conflict type, "row", "col", or "subgrid";
b. Each error is separated by a newline;
c. All errors should be wrapped in triple backticks as a code block;
(3) For Type 3 Error Msg: "No errors" wrapped in triple backticks as a code block;
(4) You can give comments or explanations before or after the code block but you MUST NOT give any comments or explanations in the code block;
===Type 1 Error Example===
```
Syntax is wrong
```

===Type 2 Error Example===
```
0,0,row
1,0,subgrid
2,1,col
```
===Type 3 Error Example===
```
No errors
```
===Sudoku Puzzle===
```
{puzzle}
```
===Candidate Solution===
```
{candidate}
```

Prompt Template 2: Graph Coloring LLM-Based Error Detector

===Instructions===
1. You are a Graph Coloring expert who can find the errors in a graph coloring candidate solution;
2. Given this Graph Coloring puzzle:
(1) The graph is represented by the adjacency matrix with {n_vertices} vertices, in which "y" means the two vertices are adjacent and "n" means the two vertices are not adjacent;
(2) The goal is to color the vertices with {color_count} colors such that no two adjacent vertices have the same color;
3. Given its candidate solution, you should find all the errors in the candidate solution;
4. The correctness of the solution depends on:
(1) Correct Syntax: the solution should be a list of {n_vertices} integers separated by comma such as "0,1,2", each integer represents the color of the corresponding vertex, and the colors should be integers from 0 to {color_count - 1};
(2) Correct Semantics: for each pair of adjacent vertices, the colors of the two vertices should be different;
4. If the syntax is incorrect, the errors should be "Syntax is wrong" (Noted as Type 1 Error Msg);
5. If the syntax is correct, the errors messages should be in two types:
(1) Type 2.1 Error Msg: the error msg are in the format as "i,j,color", where i is the vertex number, j is the vertex number, and color is the conflict color, and all of them are integers and separated by comma;
(2) Type 2.2 Error Msg: the error msg are in a number which indicates the number of exceeded colors, such as "0" means no exceeded colors, "1" means one exceeded color, "-1" means the colors used are less than the allowed color count, and so on;
6. You should find all the errors in the candidate solution;
7. If there are no errors, the errors should be "No errors" (Noted as Type 3 Error Msg);
8. You can think it thoroughly in any way you want, but You MUST give the errors in the end of your thinking in the format as:
(1) For Type 1 Error Msg: "Syntax is wrong" wrapped in triple backticks as a code block with the language indicator as "t1";
(2) For Type 2.1 Error Msg:
a. Each error is in the format as "i,j,color", where i is the vertex number, j is the vertex number, and color is the conflict color, and all of them are integers and separated by comma;
b. Each error is separated by a newline;
c. All errors should be wrapped in triple backticks as a code block with the language indicator as "t2.1";
(3) For Type 2.2 Error Msg: the number of exceeded colors wrapped in triple backticks as a code block with the language indicator as "t2.2";
(3) For Type 3 Error Msg: "No errors" wrapped in triple backticks as a code block with the language indicator as "t3";
(4) You can give comments or explanations before or after the code block but you MUST NOT give any comments or explanations in the code block;
===Type 1 Error Msg Example===
```t1
Syntax is wrong
```
===Type 2.1 Error Msg Example===

```t2.1
0,1,2
1,2,0
‘‘‘
===Type 2.2 Error Msg Example===
```t2.2
1
```
===Type 3 Error Msg Example===
```t3
No errors
```
===Graph Adjacency Matrix===
```
{adjacency_matrix}
```
===Candidate Solution===
```
{candidate}
```

Prompt Template 3: Travel Salesman Problem Error Detector

===Instructions===
1. You are a Travel Salesman Problem expert who can find all the errors in a TSP candidate solution;
2. Given this Traveling Salesman Problem puzzle:
(1) The distance matrix is a 2D matrix with {n_cities} rows and {n_cities} columns, in which each element represents the distance of traveling from the city in the row to the city in the column;
(2) The goal is to find the shortest path that visits each city exactly once and returns to the origin city;
3. Given its candidate solution, you should find all the errors in the candidate solution;
4. The correctness of the solution depends on:
(1) Correct Syntax:
a. the solution should be a list of {n_cities} integers separated by comma such as "0,1,2", each integer represents the index of the city in the path, and the indexes should be integers from 0 to {n_cities - 1};
b. the first and last city should be the same and should be 0, which means the path should return to the origin city which is 0;
c. the index of city should be in the range from 0 to {n_cities - 1};
(2) Correct Semantics:
a. No missing city: the path should visit each city exactly once and return to the origin city;
b. Optimal path: the path should be the shortest path;
5. If the syntax is incorrect, the errors should be "Syntax is wrong" (Noted as Type 1 Error Msg);
6. If the syntax is correct, the errors messages should be in two types:
(1) Type 2.1 Error Msg: the error msg are in the format of list separated by comma, where each element is a missing city in the path, such as "0,1,2", where 0, 1, 2 are the missing cities, and all of them are integers and separated by comma;
(2) Type 2.2 Error Msg: the error msg are in a number which indicates the exceeded distance, such as "0" means the distance is the optimal distance, "10.5" means the distance exceeds the optimal distance by 10.5, and it should be a float;
6. You should find all the errors in the candidate solution;
7. If there are no errors, the errors should be "No errors" (Noted as Type 3 Error Msg);
8. You can think it thoroughly in any way you want, but You MUST give the errors in the end of your thinking in the format as:
(1) For Type 1 Error Msg: "Syntax is wrong" wrapped in triple backticks as a code block with the language indicator as "t1";
(2) For Type 2.1 Error Msg:
a. the error msg are in the format of list separated by comma, where each element is a missing city in the path, such as "0,1,2", where 0, 1, 2 are the missing cities, and all of them are integers and separated by comma;
c. the error should be wrapped in triple backticks as a code block with the language indicator as "t2.1";
(3) For Type 2.2 Error Msg: the exceeded distance wrapped in triple backticks as a code block with the language indicator as "t2.2";
```

(3) For Type 3 Error Msg: "No errors" wrapped in triple backticks as a code block with the language indicator as "t3";
(4) You can give comments or explanations before or after the code block but you MUST NOT give any comments or explanations in the code block;
===Type 1 Error Msg Example===
```t1
Syntax is wrong
```
===Type 2.1 Error Msg Example===
```t2.1
0,1,2
```
===Type 2.2 Error Msg Example===
```t2.2
10.5
```
===Type 3 Error Example===
```t3
No errors
```
===Distance Matrix===
```
{distance_matrix}
```
===Candidate Solution===
```
{candidate}
```

Prompt Template 4: Sudoku LLM-based Fitness Evaluator

===Instructions===
1. You are a Sudoku expert who can evaluate whether a sudoku candidate solution is correct or not, or how close it is to the correct solution;
2. Given this Sudoku puzzle and its candidate solution, you should evaluate its score. The score is to measure how close the candidate is to the solution;
3. The correctness of the solution depends on:
(1) Correct Syntax: it has a correct format, meaning each row, column, and {subgrid_size}x{subgrid_size} square exactly contain {puzzle_grid_size} number, and each cell is separated by space. The solution format must be in the same format as the given puzzle but there is no unfilled dot;
(2) Correct Semantics: for each row, column, and {subgrid_size}x{subgrid_size} square, the numbers 1 to {puzzle_grid_size} should appear exactly once;
4. If the syntax is incorrect, the fitness score should be 0.0;
5. If the syntax is correct, the score is calculated based on the number of correct numbers in rows, columns, and subgrids, and shown in percentage. R = number of correct rows / {puzzle_grid_size} + {delta}, C = number of correct columns / {puzzle_grid_size} + {delta}, and S = number of correct subgrids / {puzzle_grid_size} + {delta}. The fitness score is calculated based on geometric mean as (R x C x S) ** (1/3) * 100.0, in which higher is better and 0.0 means the candidate is wrong at all while 100.0 means the candidate is correct;
6. In most of time, you should NOT give a score of 0.0 unless <4> are satisfied; You should give a score between 0.0 and 100.0 to indicate how close the candidate is to the correct solution;
7. Think it carefully and do NOT randomly guess the score;
8. You can think it thoroughly in any way you want, but You MUST give the score as a float number in the end of your thinking.
===Sudoku Puzzle===
```
{puzzle}
```
===Candidate Solution===
```
{candidate}
```

Prompt Template 5: Graph Coloring LLM-based Fitness Evaluator

===Instructions===
1. You are a Graph Coloring expert who can evaluate whether a graph coloring candidate solution is correct or not, or how close it is to the correct solution;
2. Given this Graph Coloring puzzle:

(1) The graph is represented by the adjacency matrix with {n_vertices} vertices, in which "y" means the two vertices are adjacent and "n" means the two vertices are not adjacent;
(2) The goal is to color the vertices with {color_count} colors such that no two adjacent vertices have the same color;
3. Given its candidate solution, you should evaluate its score. The score is to measure how close the candidate is to the solution;
4. The correctness of the solution depends on:
(1) Correct Syntax: the solution should be a list of {n_vertices} integers separated by comma such as "0,1,2", each integer represents the color of the corresponding vertex, and the colors should be integers from 0 to {color_count - 1};
(2) Correct Semantics: for each pair of adjacent vertices, the colors of the two vertices should be different;
4. If the syntax is incorrect, the fitness score should be 0.0;
5. If the syntax is correct, the score is calculated based on:
(1) Number of Conflicted Edges (noted as CE): the number of edges that two adjacent vertices have the same color;
(2) Number of Exceeded Colors (noted as EC): the number of colors exceeded the allowed color count;
(3) The score is now calculated as: Max(0, (1 - (CE/{n_edges})) * (1 - EC/({n_vertices - color_count}))) * 100, which means the score does not only depend on the number of conflicted edges but also the number of exceeded colors and ranges from 0.0 to 100.0;
6. In most of time, you should NOT give a score of 0.0 unless your are very sure; You should give a score between 0.0 and 100.0 to indicate how close the candidate is to the correct solution;
7. Think it carefully and do NOT randomly guess the score;
8. You can think it thoroughly in any way you want, but You MUST give the score as a float number in the end of your thinking.
===Graph Adjacency Matrix===
```
adjacency_matrix_str
```
===Candidate Solution===
```
{candidate}
```

Prompt Template 6: Travel Salesman Problem LLM-based Fitness Evaluator

===Instructions===
1. You are a Travel Salesman Problem expert who can evaluate whether a TSP candidate solution is correct or not, or how close it is to the correct solution;
2. Given this Traveling Salesman Problem puzzle:
(1) The distance matrix is a 2D matrix with {n_cities} rows and {n_cities} columns, in which each element represents the distance of traveling from the city in the row to the city in the column;
(2) The goal is to find the shortest path that visits each city exactly once and returns to the origin city;
3. Given its candidate solution, you should evaluate its score. The score is to measure how close the candidate is to the solution;
4. The correctness of the solution depends on:
(1) Correct Syntax:
a. the solution should be a list of {n_cities} integers separated by comma such as "0,1,2", each integer represents the index of the city in the path, and the indexes should be integers from 0 to {n_cities - 1};
b. the first and last city should be the same and should be 0, which means the path should return to the origin city which is 0;
c. the index of city should be in the range from 0 to {n_cities - 1};
(2) Correct Semantics:
a. No missing city: the path should visit each city exactly once and return to the origin city;
b. Optimal path: the path should be the shortest path;
4. If the syntax is incorrect, the fitness score should be 0.0;

5. If the syntax is correct, the score is calculated based on:
(1) Number of Missing Cities (noted as MC): the number of missing cities in the path;
(2) Used Distance (noted as UD): the total distance of the path;
(3) The score is computed as follows (in range of [0...100]):
1) Let base_score = 100;
2) Let OD = the sum of the shortest distances of the path; (You should try to the best to think about its optimal distance)
3) Let ED = UD - OD;
4) Let ED_Multiplier = ED / OD; (calculate how much the distance exceeds the optimal distance, it MUST be in range of [0...{DEFAULT_EDM}])
5) distance_excess_ratio = ED_Multiplier / {DEFAULT_EDM}; (in range of [0...1])
6) distance_correctness = base_score - (base_score * distance_excess_ratio); (in range of [0...100])
7) missing_ratio = MC / (the length of the path - 1); (in range of [0...1])
8) missing_correctness = base_score - (base_score * missing_ratio); (in range of [0...100])
9) The final score is min(distance_correctness, missing_correctness), then clamped so it never goes below 0 or above 100.
6. In most of time, you should NOT give a score of 0.0 unless your are very sure; You should give a score between 0.0 and 100.0 to indicate how close the candidate is to the correct solution;
7. Think it carefully and do NOT randomly guess the score;
8. You can think it thoroughly in any way you want, but You MUST give the score as a float number in the end of your thinking.
===Distance Matrix===
```
{distance_matrix}
```
===Candidate Solution===
```
{candidate}
```

Prompt Template 7: Sudoku LCO

===Instructions===
1. Given this Sudoku puzzle and these two Sudoku candidate solutions, you should thoroughly think both good and bad parts of each candidate and whether they are correct solutions to the puzzle;
2. If you think one of them are already correct, you can give the correct solution directly;
3. If you think the two candidates have good parts or bad parts, you can combine the good parts of both candidates, exclude the bad parts of both candidates, or do both of them simultaneously, aiming at creating a new candidate solution which can be better than the original candidates and approach more to the correct solution;
4. If you think it is not necessary to combine the two candidates, you can also give a new candidate solution which is totally different from the original candidates, aiming at approaching more to the correct solution;
5. After crossover, the solution should approach or become correct, which means:
(1) Correct Syntax: it has a correct format, meaning each row, column, and {subgrid_size}x{subgrid_size} square exactly contain {puzzle_grid_size} number, and each cell is separated by space. The solution format must be in the same format as the given puzzle but there is no unfilled dot;
(2) Correct Semantics: for each row, column, and {subgrid_size}x{subgrid_size} square, the numbers 1 to {puzzle_grid_size} should appear exactly once;
6. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
7. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
8. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in the same format as the puzzle wrapped in triple

backticks as a code block;
9. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Sudoku Puzzle===
```
{puzzle}
```
===Candidate Solution 1===
```
{c1}
```
Score of Candidate Solution 1: {s1} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution 1:
{c1_error}
===Candidate Solution 2===
```
{c2}
```
Score of Candidate Solution 2: {s2} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution 2:
{c2_error}

Now, keep the scores and errors in mind and think about how to combine the two candidates to create a new candidate solution that is better than the original candidates.
You can think in any way but you must finally give a candidate solution wrapped in triple backticks as a code block in the same format as the puzzle:

Prompt Template 8: Graph Coloring LCO

===Instructions===
1. Given this Graph Coloring puzzle:
(1) The graph is represented by the adjacency matrix with {n_vertices} vertices, in which "y" means the two vertices are adjacent and "n" means the two vertices are not adjacent;
(2) The goal is to color the vertices with {color_count} colors such that no two adjacent vertices have the same color;
2. Given these two candidate solutions, you should thoroughly think both good and bad parts of each candidate and whether they are correct solutions to the puzzle;
3. If you think one of them are already correct, you can give the correct solution directly;
4. If you think the two candidates have good parts or bad parts, you can combine the good parts of both candidates, exclude the bad parts of both candidates, or do both of them simultaneously, aiming at creating a new candidate solution which can be better than the original candidates and approach more to the correct solution;
5. If you think it is not necessary to combine the two candidates, you can also give a new candidate solution which is totally different from the original candidates, aiming at approaching more to the correct solution;
6. After crossover, the solution should approach or become correct, which means:
(1) Correct Syntax: the solution should be a list of {n_vertices} integers separated by comma such as "0,1,2", each integer represents the color of the corresponding vertex, and the colors should be integers from 0 to {color_count - 1};
(2) Correct Semantics: for each pair of adjacent vertices, the colors of the two vertices should be different;
7. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
8. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
9. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in a list of integers separated by comma wrapped in triple backticks as a code block;
10. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Graph Adjacency Matrix===

```
{adjacency_matrix}
```
===Candidate Solution 1===
```
{c1}
```
Score of Candidate Solution 1: {s1} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution 1:
{c1_error}
===Candidate Solution 2===
```
{c2}
```
Score of Candidate Solution 2: {s2} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution 2:
{c2_error}

Now, keep the scores and errors in mind and think about how to combine the two candidates to create a new candidate solution that is better than the original candidates.
You can think in any way but you must finally give a candidate solution as a list of integers separated by comma wrapped in triple backticks as a code block:

Prompt Template 9: Travel Salesman Problem LCO

===Instructions===
1. Given this Traveling Salesman Problem puzzle:
(1) The distance matrix is a 2D matrix with {n_cities} rows and {n_cities} columns, in which each element represents the distance of traveling from the city in the row to the city in the column;
(2) The goal is to find the shortest path that visits each city exactly once and returns to the origin city;
2. Given these two candidate solutions, you should thoroughly think both good and bad parts of each candidate and whether they are correct solutions to the puzzle;
3. If you think one of them are already correct, you can give the correct solution directly;
4. If you think the two candidates have good parts or bad parts, you can combine the good parts of both candidates, exclude the bad parts of both candidates, or do both of them simultaneously, aiming at creating a new candidate solution which can be better than the original candidates and approach more to the correct solution;
5. If you think it is not necessary to combine the two candidates, you can also give a new candidate solution which is totally different from the original candidates, aiming at approaching more to the correct solution;
6. After crossover, the solution should approach or become correct, which means:
(1) Correct Syntax:
a. the solution should be a list of {n_cities} integers separated by comma such as "0,1,2", each integer represents the index of the city in the path, and the indexes should be integers from 0 to {n_cities - 1};
b. the first and last city should be the same and should be 0, which means the path should return to the origin city which is 0;
c. the index of city should be in the range from 0 to {n_cities - 1};
(2) Correct Semantics:
a. No missing city: the path should visit each city exactly once and return to the origin city;
b. Optimal path: the path should be the shortest path;
7. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
8. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
9. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in a list of integers separated by comma wrapped in triple backticks as a code block;

10. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Distance Matrix===
```
{distance_matrix}
```
===Candidate Solution 1===
```
{c1}
```
Score of Candidate Solution 1: {s1} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution 1:
{c1_error}
===Candidate Solution 2===
```
{c2}
```
Score of Candidate Solution 2: {s2} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution 2:
{c2_error}

Now, keep the scores and errors in mind and think about how to combine the two candidates to create a new candidate solution that is better than the original candidates.
You can think in any way but you must finally give a candidate solution as a list of integers separated by comma wrapped in triple backticks as a code block:

Prompt Template 10: Sudoku LMO

===Instructions===
1. Given this Sudoku puzzle and this Sudoku candidate solution, you should thoroughly think about the good and bad parts of the candidate and whether it is a correct solution to the puzzle;
2. If you think the candidate is already correct, you can give the correct solution directly;
3. If you think the candidate has bad parts, you can change or improve the bad parts to make it good, aiming at creating a new candidate solution which can be better than the original candidate and approach more to the correct solution;
4. If you think it is not necessary to change the candidate, you can also give a new candidate solution which is totally different from the original candidate, aiming at approaching more to the correct solution;
5. After mutation, the solution should approach or become correct, which means:
(1) Correct Syntax: it has a correct format, meaning each row, column, and {subgrid_size}x{subgrid_size} square exactly contain {puzzle_grid_size} number, and each cell is separated by space. The solution format must be in the same format as the given puzzle but there is no unfilled dot;
(2) Correct Semantics: for each row, column, and {subgrid_size}x{subgrid_size} square, the numbers 1 to {puzzle_grid_size} should appear exactly once;
6. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
7. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
8. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in the same format as the puzzle wrapped in triple backticks as a code block;
9. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Sudoku Puzzle===
```
{puzzle}
```
===Candidate Solution===
```
{candidate}
```
Score of Candidate Solution: {score} (0.0 means the candidate is wrong at all while

100.0 means the candidate is correct)
Errors of Candidate Solution:
{error}

Now, keep the score and errors in mind and think about how to change the candidate to create a new candidate solution that is better than the original candidate.
You can think in any way but you must finally give a candidate solution wrapped in triple backticks as a code block in the same format as the puzzle:

Prompt Template 11: Graph Coloring LMO

===Instructions===
1. Given this Graph Coloring puzzle:
(1) The graph is represented by the adjacency matrix with {n_vertices} vertices, in which "y" means the two vertices are adjacent and "n" means the two vertices are not adjacent;
(2) The goal is to color the vertices with {color_count} colors such that no two adjacent vertices have the same color;
2. Given this candidate solution, you should thoroughly think about the good and bad parts of the candidate and whether it is a correct solution to the puzzle;
3. If you think the candidate is already correct, you can give the correct solution directly;
4. If you think the candidate has bad parts, you can change or improve the bad parts to make it good, aiming at creating a new candidate solution which can be better than the original candidate and approach more to the correct solution;
5. If you think it is not necessary to change the candidate, you can also give a new candidate solution which is totally different from the original candidate, aiming at approaching more to the correct solution;
6. After mutation, the solution should approach or become correct, which means:
(1) Correct Syntax: the solution should be a list of {n_vertices} integers separated by comma such as "0,1,2", each integer represents the color of the corresponding vertex, and the colors should be integers from 0 to {color_count - 1};
(2) Correct Semantics: for each pair of adjacent vertices, the colors of the two vertices should be different;
7. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
8. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
9. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in a list of integers separated by comma wrapped in triple backticks as a code block;
10. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Graph Adjacency Matrix===
```
{adjacency_matrix}
```
===Candidate Solution===
```
{candidate}
```
Score of Candidate Solution: {score} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution:
{error}

Now, keep the score and errors in mind and think about how to change the candidate to create a new candidate solution that is better than the original candidate.
You can think in any way but you must finally give a candidate solution as a list of integers separated by comma wrapped in triple backticks as a code block:

Prompt Template 12: Travel Salesman Problem LMO

===Instructions===
1. Given this Traveling Salesman Problem puzzle:

(1) The distance matrix is a 2D matrix with {n_cities} rows and {n_cities} columns, in which each element represents the distance of traveling from the city in the row to the city in the column;
(2) The goal is to find the shortest path that visits each city exactly once and returns to the origin city;
2. Given this candidate solution, you should thoroughly think about the good and bad parts of the candidate and whether it is a correct solution to the puzzle;
3. If you think the candidate is already correct, you can give the correct solution directly;
4. If you think the candidate has bad parts, you can change or improve the bad parts to make it good, aiming at creating a new candidate solution which can be better than the original candidate and approach more to the correct solution;
5. If you think it is not necessary to change the candidate, you can also give a new candidate solution which is totally different from the original candidate, aiming at approaching more to the correct solution;
6. After mutation, the solution should approach or become correct, which means:
(1) Correct Syntax:
a. the solution should be a list of {n_cities} integers separated by comma such as "0,1,2", each integer represents the index of the city in the path, and the indexes should be integers from 0 to {n_cities - 1};
b. the first and last city should be the same and should be 0, which means the path should return to the origin city which is 0;
c. the index of city should be in the range from 0 to {n_cities - 1};
(2) Correct Semantics:
a. No missing city: the path should visit each city exactly once and return to the origin city;
b. Optimal path: the path should be the shortest path;
7. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
8. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
9. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in a list of integers separated by comma wrapped in triple backticks as a code block;
10. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Distance Matrix===
```
{distance_matrix}
```
===Candidate Solution===
```
{candidate}
```
Score of Candidate Solution: {score} (0.0 means the candidate is wrong at all while 100.0 means the candidate is correct)
Errors of Candidate Solution:
{error}

Now, keep the score and errors in mind and think about how to change the candidate to create a new candidate solution that is better than the original candidate.
You can think in any way but you must finally give a candidate solution as a list of integers separated by comma wrapped in triple backticks as a code block:

Prompt Template 13: Sudoku Direct Prompting

===Instructions===
1. Given this Sudoku puzzle, you should fill in the missing numbers represented by dots;
2. The solution should be correct, which means:
(1) Correct Syntax: it has a correct format, meaning each row, column, and {subgrid_size}x{subgrid_size} square exactly contain {puzzle_grid_size} number, and each cell is separated by space. The solution format must be in the same format

as the given puzzle but there is no unfilled dot;
(2) Correct Semantics: for each row, column, and {subgrid_size}x{subgrid_size} square, the numbers 1 to {puzzle_grid_size} should appear exactly once;
3. The puzzle is guaranteed to have a unique solution;
4. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
5. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
6. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in the same format as the puzzle wrapped in triple backticks as a code block;
7. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Sudoku Puzzle===
```
{puzzle}
```

Prompt Template 14: Graph Coloring Direct Prompting

===Instructions===
1. Given this Graph Coloring puzzle:
(1) The graph is represented by the adjacency matrix with {n_vertices} vertices, in which "y" means the two vertices are adjacent and "n" means the two vertices are not adjacent;
(2) The goal is to color the vertices with {color_count} colors such that no two adjacent vertices have the same color;
2. The solution should be correct, which means:
(1) Correct Syntax: the solution should be a list of {n_vertices} integers separated by comma such as "0,1,2", each integer represents the color of the corresponding vertex, and the colors should be integers from 0 to {color_count - 1};
(2) Correct Semantics: for each pair of adjacent vertices, the colors of the two vertices should be different;
3. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
4. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
5. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in a list of integers separated by comma wrapped in triple backticks as a code block;
6. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Graph Adjacency Matrix===
```
{adjacency_matrix}
```

Prompt Template 15: Travel Salesman Problem Direct Prompting

===Instructions===
1. Given this Traveling Salesman Problem puzzle:
(1) The distance matrix is a 2D matrix with {n_cities} rows and {n_cities} columns, in which each element represents the distance of traveling from the city in the row to the city in the column;
(2) The goal is to find the shortest path that visits each city exactly once and returns to the origin city;
2. The solution should be correct, which means:
(1) Correct Syntax:
a. the solution should be a list of {n_cities} integers separated by comma such as "0,1,2", each integer represents the index of the city in the path, and the indexes should be integers from 0 to {n_cities - 1};
b. the first and last city should be the same and should be 0, which means the path should return to the origin city which is 0;

c. the index of city should be in the range from 0 to {n_cities - 1};
(2) Correct Semantics:
a. No missing city: the path should visit each city exactly once and return to the origin city;
b. Optimal path: the path should be the shortest path;
3. You should check the syntax carefully. If the syntax is incorrect, you should give a new solution which obey the rule "correct syntax";
4. You should check the semantics carefully. If the semantics is incorrect, you should give a new solution which obey the rule "correct semantics";
5. You can think whatever way you want, but at the end of thinking, the final solution should be given and written in a list of integers separated by comma wrapped in triple backticks as a code block;
6. You can give thinking steps or explanation before or after code block but you MUST NOT give any comments or explanations in the code block;
===Distance Matrix===
```
{distance_matrix}
```