ATTRIOPT: ATTRIBUTION-BASED CREDIT ASSIGNMENT FOR EVOLUTIONARY API-CALLING OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly used to solve complex tasks by composing sequences of calls to external APIs, forming executable API workflows. However, optimizing such API workflows remains challenging because existing methods typically rely on a single evaluation signal of each workflow, providing little information about which API calls actually contribute to performance. This coarse feedback limits the effectiveness of subsequent search, especially when workflow evaluation is expensive. To address this problem, we propose AttriOpt, a statistical attribution framework for evolutionary optimization of LLM-generated API workflows. AttriOpt exploits accumulated evaluations across workflows to estimate fine-grained attribution scores for API identities, parameter configurations, and interaction patterns. These attribution signals are then incorporated into evolutionary search to effectively guide the API calling process. Experiments on mixed-integer linear programming, traveling salesman problem, and logic synthesis show that AttriOpt consistently outperforms existing API-calling methods and LLM-driven evolutionary approaches. Further ablation and attribution analyses demonstrate that these attribution signals provide more informative guidance than relying solely on a single evaluation signal for each workflow.

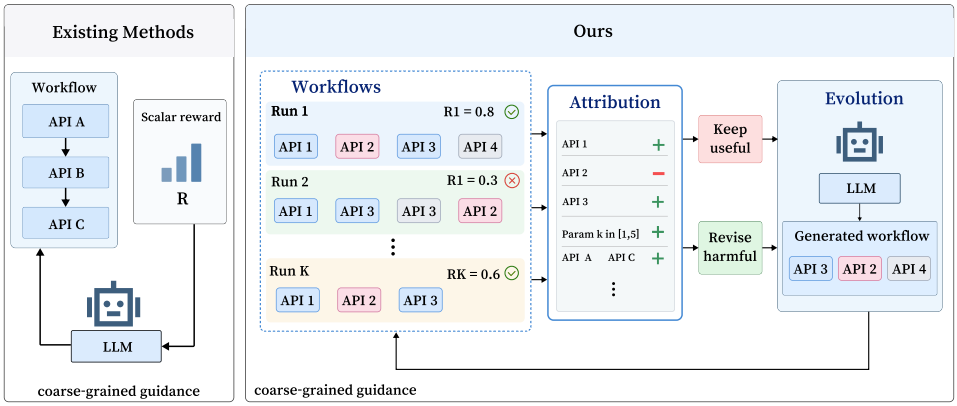

Figure 1: Comparison between coarse-grained workflow feedback and fine-grained component attribution in AttriOpt.

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated strong capabilities in reasoning (OpenAI, 2023; Ye et al., 2025; Wei et al., 2026). However, many real-world tasks require access to information, computation, or actions beyond the models' parametric knowledge. To extend these capabilities, recent work enables LLMs to interact with external tools through application programming interfaces (APIs) (Yao et al., 2023; Schick et al., 2023; Qin et al., 2024; Li et al., 2023; Patil et al., 2023; Polyakov et al., 2025). By interacting with external tools through APIs, LLMs can extend their

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

problem solving capabilities to complex computational tasks, such as using optimization solvers for scientific engineering problems, and electronic design automation tools for chip design.

Existing approaches for enabling LLMs to use external APIs can be broadly divided into two paradigms. One paradigm directly generates tool use trajectories without an explicit optimization loop. For example, ReAct (Yao et al., 2023) and ToolLLM (Qin et al., 2024) perform reasoning, tool selection, and execution to complete a task. The other paradigm optimizes tool use behavior based on execution feedback. In this paradigm, reinforcement learning methods, such as ToolRL (Qian et al., 2025), improve tool use policies through task rewards, whereas evolutionary methods, such as FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), and MCTS-AHD (Zheng et al., 2025), iteratively generate, evaluate, and refine candidate programs, heuristics, or workflows.

Despite the substantial success of these approaches, they still largely rely on coarse-grained feedback from the overall performance of complete workflows, as shown in the left part of Fig. 1. This leads to a key credit assignment problem: the final reward is assigned to the entire workflow while the contribution of each individual API call remains unclear. As a result, contributing API calls in one workflow may not be effectively identified and reused, while ineffective calls can persist when embedded in otherwise strong candidates.

To overcome this bottleneck, we observe that similar API calls recur across different workflows and search contexts while being associated with different outcomes. As shown in the left part of Fig. 1, these repeated evaluations provide reusable experience for identifying how individual API calls, parameter configurations, and interaction patterns affect performance. By continuously accumulating this evidence during search, we converts past API use into fine-grained guidance for subsequent optimization.

To this end, we propose AttriOpt, a statistical attribution-guided evolutionary framework that continuously learns from accumulated experience. AttriOpt consists of three stages. First, candidate API workflows are generated, evaluated, and stored with their outcomes in a historical archive. Second, each workflow is decomposed into fine grained API components, including API identities, parameter configurations, and interaction patterns. AttriOpt estimates attribution scores from accumulated evaluations using statistical attribution with hierarchical shrinkage. Finally, these attribution signals guide evolutionary search, and newly evaluated workflows are added back to the archive for subsequent attribution and evolution.

We evaluate AttriOpt on three representative API based optimization tasks, including Mixed Integer Linear Programming(MILP), Traveling Salesman Problem(TSP), and logic synthesis recipe optimization. Extensive experiments demonstrate consistent improvements over existing methods across different tasks. Further ablation and attribution analyses show that fine grained attribution provides more informative credit assignment than workflow level feedback, enabling more effective identification and reuse of useful API decisions during optimization.

Our contributions are summarized as follows:

- We identify credit assignment as a key challenge in LLM-generated API workflow optimization, where coarse-grained workflow feedback entangles the effects of individual API calls and prevents their contributions from being effectively distinguished.
- We propose AttriOpt, which continuously accumulates experience from evaluated API workflows and converts this execution history into fine-grained statistical attribution signals.
- We evaluate AttriOpt on three representative API-based optimization tasks. It demonstrates consistent improvements over existing methods and validate the effectiveness of attribution-guided optimization for LLM-generated API workflows.

2 Related work

2.1 Tool-Integrated Reasoning of LLMs

Recent advances in LLMs have shifted from pure text generation toward interaction with external tools and execution environments (Polyakov et al., 2025; Schick et al., 2023; Qin et al., 2024; Li et al., 2023). In practical systems,LLMs are increasingly integrated into complex computational

workflows for scientific computing (Novikov et al., 2025) and electronic design automation (Wu et al., 2025). In such settings, a candidate solution can be represented as a structured program or workflow that must be executed by external tools to obtain task feedback.

Early work largely formulated tool use as a generation problem, where an LLM selects a tool, produces a valid invocation, and incorporates the returned result (Karpas et al., 2022; Parisi et al., 2022; Gao et al., 2023). Subsequent studies extended this paradigm to multi-step interaction involving repeated reasoning, tool invocation, observation, and plan revision (Yao et al., 2023; Qin et al., 2024; Du et al., 2024). More recently, LLM-guided evolutionary methods have treated programs, configurations, and workflows as candidate solutions that are iteratively generated, evaluated, and selected (Romera-Paredes et al., 2024; Liu et al., 2024; Ye et al., 2024; van Stein & Bäck, 2024; Zheng et al., 2025). Despite their different mechanisms, these approaches commonly rely on end-to-end task performance as the primary optimization signal.

2.2 LLM FOR EVOLUTIONARY COMPUTATION

The integration of LLMs and Evolutionary Computation (EC) has emerged as an important paradigm for black box optimization. In this paradigm, LLMs generate and revise programs, heuristics, and configurations, while EC iteratively improves candidate solutions through population based search and task specific evaluation. Building on this framework, LLM driven evolutionary search has achieved substantial progress in automated algorithm design, combinatorial optimization, program optimization, and complex system configuration, and has gradually expanded from direct candidate evolution toward more sophisticated search process design.

In automated heuristic and algorithm design, FunSearch (Romera-Paredes et al., 2024), EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), HSEvo (Dat et al., 2025), MCTS-AHD (Zheng et al., 2025), and CSF (Zhang et al., 2026a) use LLMs to generate, refine, or transfer executable heuristics and program structures for complex optimization problems. Beyond direct candidate evolution, recent studies have increasingly focused on improving the evolutionary process itself (Gungordu et al., 2026). ImprovEvolve (Kravatskiy et al., 2026), E2OC (Qiu et al., 2026), and ARES-LSHADE (Naeem et al., 2026) incorporate LLMs into the design, adaptation, or coordination of evolutionary operators, allowing the search strategy to evolve together with candidate solutions. Other approaches, including STABLE (Zhang et al., 2026b), A2DEPT (Chen et al., 2026), and Helix (Su et al., 2026), extend LLM driven evolution toward hierarchical, multicomponent, and multiagent optimization, where multiple interacting parts are optimized in a coordinated manner.

3 METHOD

3.1 PROBLEM FORMULATION

Given a task q and an API library $\mathcal{A} = \{a_1, \ldots, a_M\}$, an LLM generates a structured workflow $W = [w_1, \ldots, w_L]$, where each API call $w_\ell = (a_\ell, \mathbf{p}_\ell)$ consists of a selected API $a_\ell \in \mathcal{A}$ and its parameter configuration $\mathbf{p}_\ell$. Executing W in the target environment yields a scalar reward $R(W; q)$. The optimization objective is to identify a high-performing workflow $W^\star = \arg\max_{W \in \mathcal{W}(q,\mathcal{A})} R(W; q)$, where $\mathcal{W}(q, \mathcal{A})$ denotes the space of valid API workflows for task q.

Existing optimization methods typically use $R(W; q)$ as the primary feedback signal for ranking, selecting, or updating complete workflows. Although this strategy can identify high-performing candidates, the scalar reward provides limited information about the contributions of individual API calls, parameter configurations, and their interactions. Consequently, effective components are difficult to identify and reuse across workflows, whereas ineffective components may be retained within otherwise high-performing candidates.

To address this limitation, we formulate workflow optimization as a component-level attribution problem. During search, AttriOpt maintains an archive of evaluated workflows and their outcomes, $\mathcal{H}_t = \{(W_n, y_n)\}_{n=1}^{N_t}$. Each workflow is decomposed into fine-grained API components. For each component c_j, AttriOpt estimates an attribution score θ_j from the accumulated evaluations in $\mathcal{H}_t$ to characterize its association with workflow performance. These attribution signals are subsequently used to guide workflow evolution.

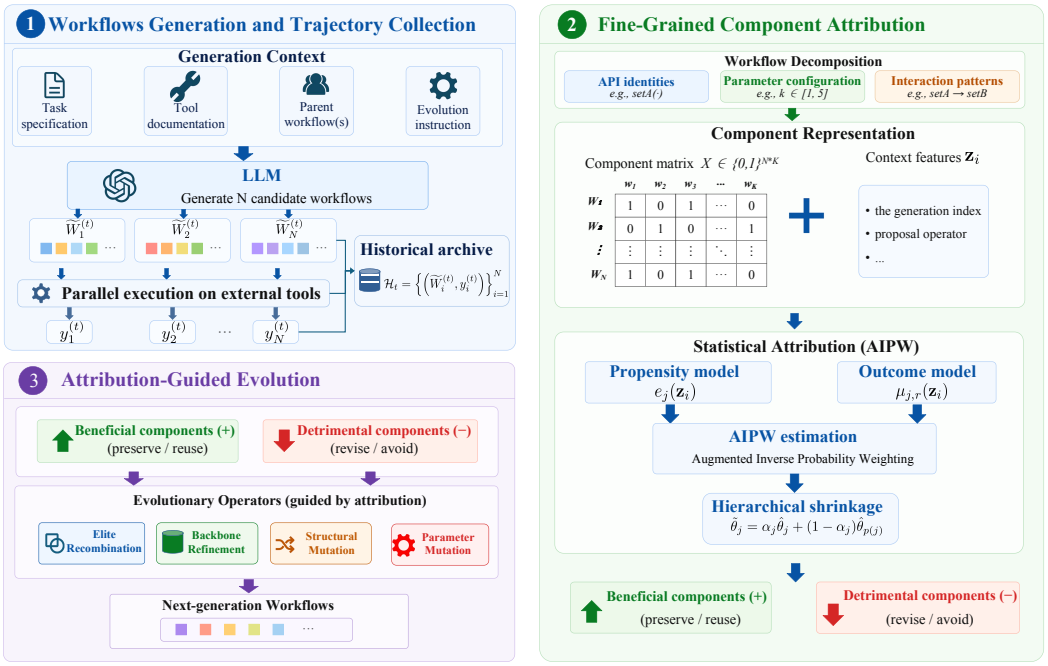

Figure 2: Overview of AttriOpt. We first generate candidate API workflows, evaluate them in the target environment, and store the resulting workflows and outcomes in a historical archive. The evaluated workflows are then decomposed into fine grained API components for statistical attribution, and the resulting attribution scores are used to guide the evolutionary process.

3.2 Workflow Generation and Trajectory Collection

At iteration t, AttriOpt maintains a population of evaluated workflows $\mathcal{P}_{t-1} = \{W_i^{(t-1)}\}_{i=1}^N$. The tool documentation $\mathcal{D}_{\text{tool}}$ describes the functionality, inputs, outputs, and invocation constraints of APIs in $\mathcal{A}$. AttriOpt generates N candidate workflows in parallel. For the i-th proposal at iteration t, the LLM receives

$$\mathcal{G}_i^{(t)} = \left(q, \mathcal{D}_{\text{tool}}, \mathcal{R}, \mathcal{S}_i(\mathcal{P}_{t-1}), \mathcal{M}_i^{(t)}\right),$$

where $\mathcal{R}$ defines the workflow and execution constraints, $\mathcal{S}_i(\mathcal{P}_{t-1})$ contains the selected parent workflow(s), and $\mathcal{M}_i^{(t)}$ specifies the evolutionary instruction. For the initial population, the parent workflows and evolutionary instructions are omitted, and the LLM independently generates diverse candidates from the available API library.

The i-th candidate is generated as $\widetilde{W}_i^{(t)} = \text{LLM}(\mathcal{G}_i^{(t)})$, where $\widetilde{W}_i^{(t)} = [w_{i,1}^{(t)}, \ldots, w_{i,L_i}^{(t)}]$ and each API call is represented as $w_{i,\ell}^{(t)} = (a_{i,\ell}^{(t)}, \mathbf{p}_{i,\ell}^{(t)})$. The candidate set $\mathcal{C}_t = \{\widetilde{W}_1^{(t)}, \ldots, \widetilde{W}_N^{(t)}\}$ is evaluated in parallel. Each candidate produces an outcome $y_i^{(t)} = R(\widetilde{W}_i^{(t)}; q)$, and the corresponding outcome vector is $\mathbf{y}^{(t)} = [y_1^{(t)}, \ldots, y_N^{(t)}]$. All evaluated workflows and their outcomes are retained in the archive, $\mathcal{H}_t = \mathcal{H}_{t-1} \cup \{(\widetilde{W}_i^{(t)}, y_i^{(t)})\}_{i=1}^N$. AttriOpt then uses the accumulated archive for fine-grained component attribution.

3.3 Fine-Grained Component Attribution

Given the accumulated archive $\mathcal{H}_t$, AttriOpt decomposes the evaluated workflows into three types of API components: API identities, parameter configurations, and interaction patterns. Suppose that K unique components are extracted from $N_t = |\mathcal{H}_t|$ workflows. Each workflow W_i is represented by a binary vector $\mathbf{x}_i \in \{0,1\}^K$, where $x_{ij} = \mathbb{I}[c_j \in \phi(W_i)]$ indicates whether component c_j appears in W_i. Here, i indexes evaluated workflows and j indexes extracted API components.

Stacking these representations yields the component matrix $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_{N_t}]^\top \in \{0,1\}^{N_t \times K}$, with corresponding performance outcomes $\mathbf{y} = [y_1, \ldots, y_{N_t}]^\top$.

The goal of attribution is to estimate the contribution of each API component to workflow performance. However, because components are sampled non-uniformly during evolution and their occurrence depends on the search context, directly comparing workflows with and without a component can lead to biased attribution. To account for this bias, AttriOpt represents each workflow W_i with a context vector $\mathbf{z}_i$ containing the generation index, proposal operator, and other frequently observed API components. These vectors form the context matrix $\mathbf{Z} = [\mathbf{z}_1, \ldots, \mathbf{z}_{N_t}]^\top$.

For each component c_j, attribution requires estimating two quantities: the probability that the component appears under a given search context and the expected workflow performance conditioned on its presence or absence. We model the former using a propensity model,

$$e_j(\mathbf{z}_i) = P(x_{ij} = 1 \mid \mathbf{z}_i),$$

where $e_j(\mathbf{z}_i)$ denotes the probability that component c_j appears in workflow W_i under context $\mathbf{z}_i$. We model the latter using an outcome model,

$$\mu_{j,r}(\mathbf{z}_i) = \mathbb{E}\left[y_i \mid x_{ij} = r, \mathbf{z}_i\right], \qquad r \in \{0,1\},$$

where $\mu_{j,r}(\mathbf{z}_i)$ denotes the expected performance of workflow W_i under context $\mathbf{z}_i$ when the presence state of component c_j is r. Specifically, $r = 1$ indicates that component c_j is present, while $r = 0$ indicates that it is absent. In our implementation, $\hat{e}_j$ is fitted using ℓ_2-regularized logistic regression, while $\hat{\mu}_{j,0}$ and $\hat{\mu}_{j,1}$ are fitted separately using ridge regression.

These two models provide complementary information for statistical attribution. AttriOpt integrates their estimates using Augmented Inverse Probability Weighting (AIPW). The propensity estimate accounts for the non-uniform occurrence of component c_j across different search contexts, while the outcome estimates characterize the expected performance when the component is present or absent. The attribution score is computed as

$$\hat{\theta}_j = \frac{1}{N_t} \sum_{i=1}^{N_t} \left[\hat{\mu}_{j,1}(\mathbf{z}_i) - \hat{\mu}_{j,0}(\mathbf{z}_i) + \frac{x_{ij}\left(y_i - \hat{\mu}_{j,1}(\mathbf{z}_i)\right)}{\hat{e}_j^*(\mathbf{z}_i)} - \frac{(1 - x_{ij})\left(y_i - \hat{\mu}_{j,0}(\mathbf{z}_i)\right)}{1 - \hat{e}_j^*(\mathbf{z}_i)} \right].$$

Here, $\hat{e}_j^*(\mathbf{z}_i) = \mathrm{clip}(\hat{e}_j(\mathbf{z}_i), \epsilon, 1 - \epsilon)$ with $\epsilon = 10^{-3}$ limits extreme inverse propensity weights. The resulting $\hat{\theta}_j$ is a signed statistical attribution score for component c_j. Positive values indicate a favorable association with workflow performance after accounting for the observed context, whereas negative values indicate an unfavorable association.

Since low-frequency components may yield high-variance attribution estimates, AttriOpt further applies Empirical Bayes hierarchical shrinkage. Related components are grouped according to their type and associated API structure. Let $g(j)$ denote the group containing component c_j, $\hat{\theta}_{g(j)}$ the corresponding group-level attribution, $\hat{\sigma}_j^2$ the estimated variance of $\hat{\theta}_j$, and $\hat{\tau}_{g(j)}^2$ the empirical variance among components within the group. The regularized attribution is

$$\tilde{\theta}_j = \alpha_j \hat{\theta}_j + (1 - \alpha_j)\hat{\theta}_{g(j)}, \qquad \alpha_j = \frac{\hat{\tau}_{g(j)}^2}{\hat{\tau}_{g(j)}^2 + \hat{\sigma}_j^2}.$$

This regularization preserves well-supported estimates while shrinking uncertain estimates toward their group-level values. The resulting $\tilde{\theta}_j$ is used to guide subsequent workflow evolution.

3.4 Attribution-Guided Evolution

Given the regularized attribution scores $\{\tilde{\theta}_j\}_{j=1}^K$, AttriOpt converts the attribution results into explicit guidance for subsequent workflow evolution. Components with high positive attribution are treated as promising building blocks, whereas strongly negative components are regarded as candidates for revision or removal. To account for the reliability of each attribution estimate, we define the empirical support of component c_j as $s_j = \sum_{i=1}^{N_t} x_{ij}$, which records the number of evaluated workflows containing that component.

Based on the attribution scores and empirical support, AttriOpt constructs two guidance sets,

$$\mathcal{C}_t^+ = \mathrm{Top}_{k_+}\left\{(c_j, \tilde{\theta}_j, s_j) \mid \tilde{\theta}_j > 0\right\}, \qquad \mathcal{C}_t^- = \mathrm{Top}_{k_-}\left\{(c_j, -\tilde{\theta}_j, s_j) \mid \tilde{\theta}_j < 0\right\},$$

where $\mathcal{C}_t^+$ contains the k_+ most positively attributed components and $\mathcal{C}_t^-$ contains the k_- most negatively attributed components. Each retained component is accompanied by its attribution score and empirical support. Components in $\mathcal{C}_t^+$ are preferentially preserved or reused, whereas those in $\mathcal{C}_t^-$ are prioritized for revision, removal, or further exploration.

After updating the archive $\mathcal{H}_t$, high-performing workflows are selected to form the parent pool $\mathcal{P}_t$. Each parent is represented by its API-call sequence together with the attribution scores of its constituent components. For the i-th proposal in the next iteration, the evolutionary instruction is conditioned on the selected operator and the two attribution-guidance sets,

$$\mathcal{M}_i^{(t+1)} = \mathcal{M}\left(o_i^{(t+1)}, \mathcal{C}_t^+, \mathcal{C}_t^-\right),$$

where $o_i^{(t+1)}$ denotes the evolutionary operator assigned to the proposal. Together with the selected parent workflow(s), this instruction is incorporated into the generation context $\mathcal{G}_i^{(t+1)}$, enabling the LLM to explicitly distinguish components that should be preserved, revised, or further explored.

AttriOpt employs several complementary evolutionary operators. *Elite recombination* combines promising components from multiple parent workflows. *Backbone refinement* preserves the main structure of a strong parent while revising weak or negatively attributed components. *Structural mutation* modifies the API-call sequence by adding, removing, or replacing calls, whereas *parameter mutation* preserves the workflow structure and adjusts API configurations.

The attribution signals further guide proposal allocation across these operators. When reliable attribution evidence is limited, more proposals are assigned to exploration-oriented operators. As reliable positive signals accumulate, the search gradually places greater emphasis on recombination, refinement, and parameter modification, allowing promising components to be exploited while uncertain regions remain explored.

4 Experiments

4.1 Experimental settings

To evaluate AttriOpt across different API-based optimization scenarios, we evaluate AttriOpt on three representative API-based optimization tasks, including MILP solver configuration with SCIP, TSP solver optimization with LKH-3, and logic synthesis recipe optimization with Yosys and ABC.

4.2 Mixed-integer linear programming Optimization

Experimental setup. Mixed-integer linear programming is widely used in scheduling, resource allocation, network design (Bengio et al., 2021). Modern MILP solvers integrate components such as presolving, cutting planes, branch-and-bound, and primal heuristics, whose configurations can substantially affect solving performance (Achterberg, 2009; Bestuzheva et al., 2023; Gleixner et al., 2021). We therefore formulate MILP solver configuration as an API-calling task. Given a MILP instance and SCIP, the LLM generates a bounded sequence of PySCIPOpt calls to configure the global heuristic policy and individual heuristic parameters, with the objective of minimizing solving time. We evaluate AttriOpt on four synthetic MILP benchmarks, including Combinatorial Auction, Facility Location, Set Cover, and Maximum Independent Set.

Experimental results. Table 1 summarizes the results on four MILP benchmark classes. Using solving time as the evaluation metric, AttriOpt achieves the best performance across all classes, outperforming both direct API workflow generation and feedback-based optimization methods. Compared with default SCIP, AttriOpt reduces solving time by 63.51%, 54.60%, 40.62%, and 13.53% on Cauctions, Facility, SetCover, and IndSet, respectively. We further observe that the evolutionary baselines achieve competitive performance across the four MILP classes and generally show smaller standard deviations than direct API workflow generation methods. This suggests feedback-based optimization methods can progressively refine promising solver configurations through repeated

Table 1: Comparison of solving time across four MILP problem classes. Results are reported as mean ± standard deviation over three independent runs. Lower values indicate better performance, and boldface highlights the best result in each column.

Method	Cauctions ↓	Facility ↓	SetCover ↓	IndSet ↓
Solver Baseline				
SCIP Default	3.431 ± 0.616	49.876 ± 0.557	9.881 ± 1.985	3.843 ± 0.220
Direct API Workflow Generation Methods				
ReAct	2.20 ± 1.18	22.87 ± 14.16	8.53 ± 6.98	4.25 ± 3.07
PathWise	1.38 ± 0.27	31.02 ± 18.38	11.04 ± 7.97	4.03 ± 2.86
LLMCompiler	2.70 ± 0.86	47.70 ± 7.73	8.10 ± 6.53	5.35 ± 3.86
Feedback-Based Optimization Methods				
EoH	1.461 ± 0.035	39.068 ± 0.678	8.678 ± 0.246	3.413 ± 0.105
ReEvo	2.837 ± 0.043	41.874 ± 2.293	8.488 ± 0.505	3.760 ± 0.115
MCTS-AHD	1.741 ± 0.087	37.819 ± 4.221	6.342 ± 0.843	4.031 ± 0.551
AttriOpt	$\mathbf{1.252 \pm 0.111}$	$\mathbf{22.644 \pm 0.286}$	$\mathbf{5.867 \pm 0.678}$	$\mathbf{3.323 \pm 0.292}$

Table 2: Comparison of optimality gap on TSP100, TSP200, and TSP500 under a 10 s time limit. Results are reported as mean ± standard deviation over three independent runs. Lower values indicate better performance, and boldface highlights the best result in each column.

Method	TSP100 Gap (%) ↓	TSP200 Gap (%) ↓	TSP500 Gap (%) ↓
Direct API Workflow Generation Methods			
ReAct	3.18 ± 5.77	3.91 ± 0.54	3.18 ± 1.09
PathWise	0.55 ± 0.23	0.29 ± 0.13	0.92 ± 0.34
LLMCompiler	3.48 ± 2.27	4.18 ± 0.67	5.09 ± 1.18
Feedback-Based Optimization Methods			
EoH	1.41 ± 4.82	4.95 ± 8.02	12.76 ± 4.41
ReEvo	1.78 ± 5.39	0.70 ± 3.42	4.18 ± 6.72
MCTS-AHD	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$	13.91 ± 2.32
AttriOpt	1.22 ± 4.14	$\mathbf{0.00 \pm 0.00}$	$\mathbf{0.00 \pm 0.00}$

selection and variation. Moreover, compared with MCTS-AHD, the strongest evolutionary baseline overall, AttriOpt reduces solving time by 28.09%, 40.13%, 7.49%, and 17.56% on Cauctions, Facility, SetCover, and IndSet, respectively. These gains indicate that fine-grained attribution can alleviate the credit assignment problem caused by coarse-grained feedback from complete workflows, providing more informative guidance for subsequent search.

4.3 Traveling Salesman Problem Optimization

Experimental setup. The Traveling Salesman Problem (TSP) seeks the shortest tour that visits each city exactly once and returns to the starting city (Karp, 1972; Applegate et al., 2006). We use LKH-3 as the external solver, building on the Lin–Kernighan family of local search heuristics (Helsgaun, 2000), and formulate its configuration as an API workflow optimization task. We evaluate AttriOpt on synthetically generated TSP instances with 100, 200, and 500 nodes. Solving time and optimality gap are used as the evaluation metrics, where lower values indicate better performance.

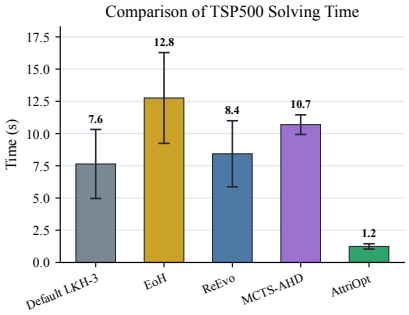

Figure 3: Comparison of solving time across different methods on TSP500.

Experimental results. As shown in Table 2, AttriOpt achieves competitive performance across different TSP sizes and exhibits increasingly clear advantages on larger instances, with gaps of 1.22%, 0.00%, and 0.00% on TSP100, TSP200, and TSP500, respectively. The advan-

Table 3: Comparison on 20 combinational circuits selected from the EPFL benchmark. All methods optimize a single synthesis workflow shared across the 20 circuits. Lower LUT6 count is better.

Method	Total LUT6 ↓	Average LUT6 ↓	Improvement (%) ↑
Initial Mapping	112,050	5,602.50	–
EoH	90,797	4,539.85	18.97
ReEvo	91,158	4,557.90	18.65
MCTS-AHD	90,326	4,516.30	19.39
AttriOpt	**89,486**	**4,474.30**	**20.14**

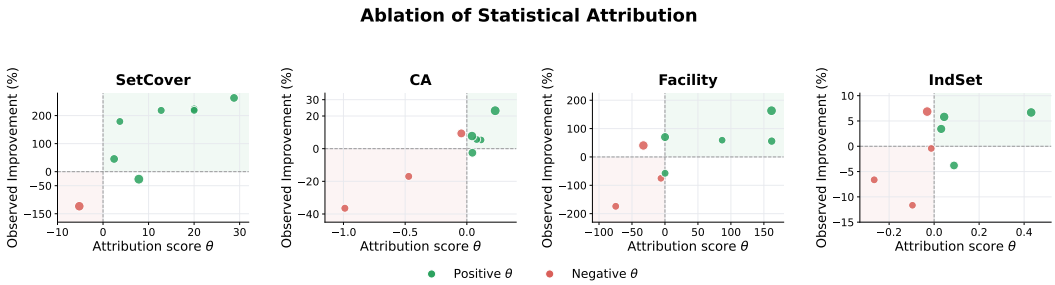

Figure 4: Ablation analysis of statistical attribution across four MILP benchmarks. The attribution scores show clear directional agreement with the observed performance improvement of API components.

tage is particularly pronounced on the more challenging TSP500 instances. Compared with ReEvo, the strongest feedback-based baseline, AttriOpt reduces the gap from 4.18% to 0.00%. Compared with PathWise, the strongest direct API workflow generation method, AttriOpt likewise reduces the gap from 0.92% to 0.00%. As shown in Fig. 3, AttriOpt also reduces the solving time on TSP500 from 7.6 s with the default LKH-3 solver to only 1.2 s, corresponding to an 84.21% reduction. In contrast, the other optimization methods require more solving time than the default LKH-3 solver. This further demonstrates that AttriOpt's fine-grained attribution mechanism can identify beneficial API components from execution feedback and explicitly reinforce them during evolution, alleviating the credit assignment problem of coarse-grained feedback and enabling more effective optimization of both solution quality and solving efficiency.

4.4 Logic Synthesis Recipe Optimization

Experimental setup. Logic synthesis recipe optimization (Yu et al., 2018; Haaswijk et al., 2018; Neto et al., 2023) searches for a sequence of synthesis operations that reduces implementation cost while preserving circuit functionality. We formulate this process as an API-calling task using Yosys (Wolf & Glaser, 2013) and ABC (Brayton & Mishchenko, 2010) as the external synthesis toolchain. Given a combinational Verilog circuit, Yosys first converts the design into a gate-level logic network. The LLM then generates a bounded sequence of ABC synthesis API calls. ABC executes the resulting recipe and maps the optimized network to LUT6s. We evaluate AttriOpt on 20 combinational circuits selected from the EPFL benchmark (Amarú et al., 2015). Fitness is measured by the average LUT6 count, where lower values indicate better synthesis quality. Detailed task descriptions and evaluation settings are provided in the Appendix.

Experimental results. As shown in Table 3, AttriOpt achieves the lowest total and average LUT6 counts on the 20 EPFL combinational circuits. Compared with the initial ABC mapping flow, AttriOpt reduces the total LUT6 count from 112,050 to 89,486, corresponding to an improvement of 20.14%, demonstrating its effectiveness in improving overall synthesis quality. AttriOpt also outperforms existing LLM-driven evolutionary methods. Compared with the strongest baseline, MCTS-AHD, AttriOpt further reduces the average LUT6 count from 4,516.30 to 4,474.30. These results indicate that fine-grained attribution guidance provides more effective optimization signals for searching shared synthesis workflows.

4.5 Ablation Study

Ablation of Each Component. To assess the contribution of each component in AttriOpt, we compare the full framework with two ablated variants on TSP200 under a fixed solver time limit of 10 s. w/o Attribution removes fine-grained attribution and performs evolution using only coarse-grained feedback from complete workflows, while w/o Attri-GE retains the attribution analysis but replaces attribution-guided evolution with conventional selection, crossover, and mutation. As shown in Table 4, AttriOpt achieves the lowest tour length of 10.69. Removing fine-grained attribution increases the tour length to 10.75, showing that coarse-grained feedback alone provides limited guidance for distinguishing beneficial and detrimental components. Replacing attribution-guided evolution further increases the tour length to 10.88, indicating that attribution signals are most effective when explicitly incorporated into the evolutionary process. These results demonstrate the complementary roles of fine-grained attribution and attribution-guided evolution.

Table 4: Ablation study on TSP200 under a 10 s solver time limit. Lower tour length is better.

Method	Tour Length ↓
w/o Attribution	10.75
w/o Attri-GE	10.88
AttriOpt	**10.69**

Attribution Analysis. To assess whether the estimated attribution scores provide meaningful guidance for evolution, we compare the sign of each attribution score with the observed performance change of the corresponding API component. As shown in Fig. 4, positive attribution is expected to correspond to performance improvement, while negative attribution should indicate degradation. The attribution signals show consistent directional agreement across all four MILP benchmarks. Specifically, 8 of 9 components are aligned on SetCover, and 6 of 8 are aligned on each of CA, Facility, and IndSet, yielding an overall agreement of 26 out of 33 components (78.8%). Most positively attributed components lie in the upper-right region, while negatively attributed components are mainly concentrated in the lower-left region. The remaining mismatches are largely associated with attribution scores close to zero. Overall, these results indicate that the estimated scores can effectively distinguish favorable and unfavorable API components, providing useful fine-grained guidance for subsequent attribution-guided evolution.

Effect of Different LLM Backbones. Finally, we evaluate AttriOpt with DeepSeek-Coder, GPT-5, and Qwen on TSP200, using the same 10 s solver time limit for all backbones. As shown in Table 5, AttriOpt achieves tour lengths of 10.42, 10.07, and 10.34, respectively. Although the absolute performance varies across language models, AttriOpt remains effective under all three backbones, indicating that the proposed framework is not dependent on a particular LLM.

Table 5: Performance of AttriOpt with different LLM backbones on TSP200 under a 10 s solver time limit. Lower tour length is better.

LLM Backbone	Tour Length ↓
DeepSeek-Coder	10.42
GPT-5	**10.07**
Qwen	10.34

5 Conclusion

In this work, we identify credit assignment as one of the key challenges in optimizing API workflows generated by LLMs. Existing methods mainly rely on rewards assigned to complete workflows. Although such rewards reflect overall performance, they provide limited information about the contribution of individual API calls and parameter settings, making it difficult to determine which decisions should be retained or modified during optimization. To address this limitation, we propose AttriOpt, which integrates statistical attribution into evolutionary search. AttriOpt uses accumulated evaluations from multiple workflows to estimate the contribution of API calls to task performance. The resulting attribution scores are then used to guide the preservation, modification, and recombination of API decisions during evolution, providing more informative feedback than the overall workflow reward alone. Experiments on MILP solver configuration, traveling salesman problem optimization, and logic synthesis show that AttriOpt consistently outperforms existing methods for API calling and LLM driven evolutionary search. Further ablation studies, attribution analyses, and experiments with different LLM backbones demonstrate the effectiveness and general applicability of the proposed approach.

REFERENCES

Tobias Achterberg. Scip: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *International Workshop on Logic and Synthesis*, 2015.

David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2): 405–421, 2021.

Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, et al. Enabling research through the scip optimization suite 8.0. *ACM Transactions on Mathematical Software*, 49(2):1–21, 2023.

Robert K. Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pp. 24–40. Springer, 2010.

Bin Chen, Shouliang Zhu, Beidan Liu, Yong Zhao, Tianle Pu, Huichun Li, and Zhengqiu Zhu. A2dept: Large language model-driven automated algorithm design via evolutionary program trees. *arXiv preprint arXiv:2604.24043*, 2026.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 2025.

Yu Du, Fangyun Wei, and Hongyang Zhang. Anytool: Self-reflective, hierarchical agents for large-scale api calls. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11812–11829, 2024.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799, 2023.

Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, et al. Miplib 2017: Data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13:443–490, 2021.

Oguzhan Gungordu, Siheng Xiong, and Faramarz Fekri. Pathwise: Planning through world model for automated heuristic design via self-evolving llms. *arXiv preprint arXiv:2601.20539*, 2026.

Winston Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Frédéric Kaplan, Sabine Süsstrunk, and Giovanni De Micheli. Deep learning for logic optimization algorithms. In *IEEE International Symposium on Circuits and Systems*, 2018.

Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pp. 85–103. Springer, 1972.

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, et al. Mrkl systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*, 2022.

Alexey Kravatskiy, Valentin Khrulkov, and Ivan Oseledets. Improvevolve: Ask alphaevolve to improve the input solution and then improvise. *arXiv preprint arXiv:2602.10233*, 2026.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, 2023.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223, 2024.

Abdullah Naeem, Md Wasi Ul Kabir, Manish Bhatt, Ayon Dey, Anav Katwal, and Md Tamjidul Hoque. Ares-lshade: Autoresearch-enhanced lshade with memetic polish for the gnbg benchmark. *arXiv preprint arXiv:2605.13877*, 2026.

Walter Lau Neto, Yingjie Li, Pierre-Emmanuel Gaillardon, and Cunxi Yu. Flowtune: End-to-end automatic logic optimization exploration via domain-specific multiarmed bandit. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(6):1912–1925, 2023.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023.

Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

Gregory Polyakov, Ilseyar Alimova, Dmitry Abulkhanov, Ivan Sedykh, Andrey Bout, Sergey Nikolenko, and Irina Piontkovskaya. Toolreflection: Improving large language models for real-world api calls with self-generated data. In *Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025)*, pp. 184–199, 2025.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. In *Advances in Neural Information Processing Systems*, 2025.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, et al. ToolLLM: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, 2024.

Junhao Qiu, Xin Chen, Liang Ge, Liyong Lin, Zhichao Lu, and Qingfu Zhang. Evolving interdependent operators with large language models for multi-objective combinatorial optimization. *arXiv preprint arXiv:2601.17899*, 2026.

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

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Chang Su, Zhongkai Hao, Zhizhou Zhang, Zeyu Xia, Youjia Wu, Hang Su, and Jun Zhu. Helix: Evolutionary reinforcement learning for open-ended scientific problem solving. In *International Conference on Learning Representations*, 2026.

Niki van Stein and Thomas Bäck. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *arXiv preprint arXiv:2405.20132*, 2024.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2026.

Clifford Wolf and Johann Glaser. Yosys—a free verilog synthesis suite. In *Proceedings of the Austrochip Workshop on Microelectronics*, pp. 47–52, 2013.

Haoyuan Wu, Haisheng Zheng, Zhuolun He, and Bei Yu. Divergent thoughts toward one goal: Llm-based multi-agent collaboration system for electronic design automation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 1710–1721, 2025.

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

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. Limo: Less is more for reasoning. *arXiv preprint arXiv:2502.03387*, 2025.

Cunxi Yu, Houping Xiao, and Giovanni De Micheli. Developing synthesis flows without human knowledge. In *Proceedings of the 55th Annual Design Automation Conference*, pp. 50:1–50:6, 2018.

Zhihao Zhang, Siyuan Li, Xiaoying Chen, Xuening Chen, Tianle Zhang, Jian Zhao, and Peng Liu. Csf: A curriculum-based scaffolding framework for heuristic evolution in combinatorial optimization. *Pattern Recognition*, pp. 114659, 2026a.

Zhiyao Zhang, Shenghao Wu, Xingyu Wu, and Kay Chen Tan. Semantics-aware bilevel co-evolution: Towards automated multicomponent algorithm design. *arXiv preprint arXiv:2606.29953*, 2026b.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 78338–78373, 2025.

A Appendix

A.1 Background

A.1.1 Mixed-Integer Linear Programming

Mixed-Integer Linear Programming (MILP) is a general optimization framework in which the objective and constraints are linear, while a subset of decision variables is restricted to integer values. A MILP can be written as

$$\min_{\mathbf{x}} \quad \mathbf{c}^\top \mathbf{x} \qquad \text{s.t.} \quad \mathbf{A}\mathbf{x} \leq \mathbf{b}, \qquad \mathbf{x}_{\mathcal{I}} \in \mathbb{Z},$$

where $\mathbf{x} \in \mathbb{R}^n$ denotes the decision variables, $\mathbf{c}$ is the objective coefficient vector, $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ defines the linear constraints, and $\mathcal{I}$ specifies the subset of variables subject to integrality constraints. MILP provides a unified formulation for a wide range of combinatorial optimization problems, including scheduling, resource allocation, facility location, set covering, and combinatorial auctions.

Modern MILP solvers typically rely on a branch-and-bound framework augmented by multiple algorithmic components. At each search node, the integrality constraints are relaxed to obtain a linear programming relaxation that provides a bound on the optimal objective. If the relaxation contains fractional integer variables, the solver branches on selected variables and recursively explores the

resulting subproblems. The search terminates when all relevant nodes are pruned or the desired optimality criterion is reached.

The performance of this procedure depends heavily on several interacting solver components. *Presolving* simplifies the original formulation before and during search by removing redundant constraints, tightening variable bounds, and detecting implied relations. *Cutting-plane methods* strengthen the linear relaxation by introducing valid inequalities that remove fractional solutions without excluding feasible integer solutions. *Branching* determines how the search tree is partitioned and therefore directly affects its size and structure. *Primal heuristics* search for feasible integer solutions during optimization, providing incumbent solutions that can improve pruning and accelerate convergence.

These components expose a large number of configurable parameters controlling whether a procedure is activated, how frequently it is executed, and how much computational effort it receives. Their effects are highly instance dependent and strongly coupled. For example, increasing the frequency of a heuristic may improve incumbent discovery on one class of instances while introducing unnecessary computational overhead on another.

In this work, we use SCIP as the target MILP solver and access its configuration interface through PySCIPOpt. Rather than modifying the internal optimization algorithms of SCIP, the LLM constructs executable API workflows that configure solver behavior before optimization. A workflow may control the global heuristic policy as well as parameters of individual heuristics, such as activation status, execution frequency, offset, and search-depth related settings. The resulting configuration is then applied to a set of MILP instances and evaluated according to its solving performance.

A.1.2 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classical combinatorial optimization problem that seeks the shortest tour visiting each city exactly once and returning to the starting city. Given a set of cities $\mathcal{V} = \{1, \dots, n\}$ and pairwise distances d_{ij}, let $x_{ij} \in \{0, 1\}$ indicate whether the edge between cities i and j is included in the tour. The TSP can be formulated as

$$\min_{\mathbf{x}} \quad \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V},\, j \neq i} d_{ij} x_{ij},$$

subject to the constraints that each city is visited exactly once and that the selected edges form a single connected tour.

In this work, we use LKH-3 as the target TSP solver. LKH-3 builds on the Lin–Kernighan heuristic and improves candidate tours through variable-depth edge exchanges and repeated local search. Its performance depends on several interacting components, including candidate-set construction, local search, perturbation, recombination, and search-budget allocation. These components jointly determine the balance between search quality and computational cost.

We expose these solver decisions through a structured API interface. The LLM generates an executable API workflow that configures the corresponding search components and parameters before invoking LKH-3. The resulting workflow is evaluated on a collection of TSP instances using the obtained tour length, where lower values indicate better solution quality.

A.1.3 Logic Synthesis Recipe Problem

Logic synthesis transforms a hardware description into a functionally equivalent circuit with improved implementation quality. In this work, we use Yosys (Wolf & Glaser, 2013) and ABC (Brayton & Mishchenko, 2010) as the synthesis toolchain. Yosys first converts a combinational Verilog design into a logic network, and ABC then applies a sequence of logic optimization operations before mapping the optimized network to 6-input lookup tables (LUT6s).

We expose seven ABC operations to the LLM, including *strash*, *balance*, *rewrite*, *refactor*, *resub*, *dc2*, and *fraig*. A synthesis workflow is an ordered sequence of these operations. Since each operation changes the intermediate logic network, the effect of a subsequent operation depends on the transformations applied before it.

We evaluate each candidate workflow on 20 combinational circuits selected from the EPFL benchmark (Amarú et al., 2015). The same workflow is applied to all circuits. After executing the work-

flow, ABC maps each optimized circuit to LUT6s, and the number of LUT6s is recorded as the synthesis result. The fitness of a workflow W is defined as the average LUT6 count across all circuits,

$$F(W) = \frac{1}{N} \sum_{i=1}^{N} \mathrm{LUT6}(W, D_i),$$

where D_i denotes the i-th circuit. Lower values indicate better synthesis quality.

The optimization objective is therefore to identify a shared synthesis workflow that achieves a low average LUT6 count across the circuit set. Since the final score reflects the joint effect of all operations in the workflow, the contribution of an individual API call is not directly observable, making this task suitable for studying fine-grained attribution.

A.2 Implementation Details

A.2.1 Mixed-Integer Linear Programming

Task Definition. We study API-based solver optimization for mixed-integer linear programming. Each instance is provided as an LP file and solved by SCIP through PySCIPOpt. Given an MILP instance I, a candidate workflow W consists of a bounded sequence of SCIP parameter-setting API calls executed before optimization. The underlying MILP formulation, including its variables, constraints, and objective, remains unchanged. For each candidate workflow, we first load the instance, apply the API calls specified by W, and then invoke SCIP to solve the original problem. The optimization objective is to minimize the solver-reported solving time,

$$T(W, I) = \texttt{model.getSolvingTime()}.$$

For a test set $\mathcal{D}$, performance is measured by the average solving time

$$\bar{T}(W) = \frac{1}{|\mathcal{D}|} \sum_{I \in \mathcal{D}} T(W, I),$$

where lower values indicate better performance.

Datasets. We evaluate AttriOpt on four representative MILP problem classes with distinct combinatorial structures, including Set Covering, Maximum Independent Set, Combinatorial Auction, and Capacitated Facility Location. Following the benchmark configuration, n and m denote the numbers of decision variables and constraints, respectively. The detailed settings are described below.

Set Covering. The Set Covering problem aims to select a minimum-cost subset of available sets such that every element in the universe is covered by at least one selected set. Its MILP formulation contains binary variables indicating whether each candidate set is selected, together with covering constraints for all elements. We use instances with $n = 1000$ decision variables and $m = 500$ constraints. This problem is characterized by a large number of binary decisions and sparse covering constraints, making the solver behavior sensitive to presolving, cutting planes, branching, and primal heuristics.

Maximum Independent Set. The Maximum Independent Set problem seeks the largest subset of vertices in a graph such that no two selected vertices are connected by an edge. Each vertex is associated with a binary decision variable, while each edge induces a constraint preventing both endpoints from being selected simultaneously. We use instances with $n = 500$ decision variables and approximately $m = 1953$ constraints. The resulting MILPs contain strong combinatorial dependencies between binary variables and provide a representative benchmark for evaluating branch-and-bound and heuristic configurations.

Combinatorial Auction. The Combinatorial Auction problem determines a set of non-conflicting bids that maximizes the total accepted bid value. Each bid corresponds to a binary decision variable, while constraints ensure that each item can be allocated to at most one accepted bid. We use instances with $n = 500$ decision variables and $m = 192$ constraints. Although the number of constraints is relatively small, each bid may involve multiple items, resulting in strong coupling among decision variables and a highly combinatorial search space.

Capacitated Facility Location. The Capacitated Facility Location problem jointly determines which facilities to open and how customers are assigned to the opened facilities while respecting facility capacities. Its MILP formulation includes binary facility-opening variables and customer-assignment variables, together with demand, assignment, and capacity constraints. We use instances with $n = 10100$ decision variables and $m = 10203$ constraints. Compared with the other benchmarks, Facility Location contains a substantially larger MILP formulation, making it particularly useful for evaluating whether the optimized API workflows remain effective on large-scale instances.

SCIP Default Baseline. The SCIP baseline uses the default solver configuration without modifying any solver parameter. Each instance is directly loaded and optimized as

$$\texttt{readProblem} \rightarrow \texttt{optimize}.$$

No heuristic, presolving, separating, randomization, or other solver-control parameter is modified.

```
model = Model()
model.readProblem(instance_path)
model.optimize()
solve_time = model.getSolvingTime()
```

Allowed SCIP API Space. All candidate workflows operate within the same fixed solver-control space and are allowed to modify only SCIP primal-heuristic configurations before optimization. Candidates cannot modify the MILP formulation, add or remove constraints, change the objective, access solution values, invoke external computation, or directly control `readProblem` and `optimize`.

At the global level, the workflow may select one of four heuristic settings:

```
model.setHeuristics(SCIP_PARAMSETTING.DEFAULT)
model.setHeuristics(SCIP_PARAMSETTING.FAST)
model.setHeuristics(SCIP_PARAMSETTING.AGGRESSIVE)
model.setHeuristics(SCIP_PARAMSETTING.OFF)
```

For each permitted primal heuristic `<name>`, the workflow may further modify three integer parameters:

```
model.setIntParam("heuristics/<name>/freq", integer)
model.setIntParam("heuristics/<name>/freqofs", integer)
model.setIntParam("heuristics/<name>/maxdepth", integer)
```

The candidate heuristic set is:

```
ofins, reoptsols, trivialnegation, trivial, rounding,
shifting, intshifting, oneopt, twoopt, clique, repair,
rens, rins, dualval, indicator, scheduler, alns,
localbranching, trustregion
```

The parameters follow SCIP's native semantics. In particular, `freq=-1` disables a heuristic, `freq=0` restricts it to the root node, and positive values specify periodic execution. The parameter `freqofs` controls the frequency offset, while `maxdepth` specifies the maximum search depth at which the heuristic can be executed. Setting `maxdepth=-1` imposes no explicit depth limit.

Candidate Workflow Format. A candidate workflow is represented as a short ordered sequence of PySCIPOpt API calls executed after loading the MILP instance and before calling `model.optimize()`. For example,

```
model.setHeuristics(SCIP_PARAMSETTING.FAST)
model.setIntParam("heuristics/rens/freq", 0)
model.setIntParam("heuristics/rins/freq", 15)
model.setIntParam("heuristics/alns/freq", -1)
model.setIntParam("heuristics/rens/maxdepth", 20)
```

Each API call modifies only the solver configuration. Different calls may enable, disable, or alter the execution frequency and depth of individual heuristics, allowing the workflow to directly influence SCIP's search behavior. All compared methods use the same workflow representation and executable API interface.

Evaluation Protocol For each method, the best discovered workflow is evaluated on the same held-out test instances under an identical software and hardware environment. We report the mean and standard deviation of SCIP solving time,

$$\bar{T} \pm \sigma_T.$$

All solving times are obtained from `model.getSolvingTime()` rather than Python wall-clock time, excluding file I/O, Python execution, and logging overhead. A run is considered successful when SCIP returns an optimal solution to the original MILP instance.

A.2.2 Traveling Salesman Problem

Task Definition We study API-based solver optimization for the Traveling Salesman Problem using LKH-3 as the external solver. Each instance consists of n two-dimensional points and is converted into TSPLIB format before execution. A candidate workflow W is represented as a bounded sequence of solver-configuration API calls that control the search behavior of LKH-3. The TSP instance itself, including node coordinates and edge costs, remains unchanged throughout optimization.

Given an instance I, the workflow is first translated into an LKH-3 parameter file and then executed by the solver. LKH-3 returns a feasible tour π, whose length is

$$L(W, I) = \sum_{t=1}^{n} d(\pi_t, \pi_{t+1}), \qquad \pi_{n+1} = \pi_1,$$

where $d(\cdot, \cdot)$ denotes the Euclidean distance computed according to the TSPLIB `EUC_2D` convention. Lower tour length indicates better solution quality.

Default LKH-3 Baseline The default baseline uses the native LKH-3 search configuration without additional search-control modifications. The parameter file contains only the fields required for problem loading, output generation, and reproducibility:

```
PROBLEM_FILE = instance.tsp
OUTPUT_TOUR_FILE = output.tour
SEED = s
TRACE_LEVEL = 1
```

No additional configuration is applied to parameters such as `RUNS`, `MOVE_TYPE`, `PATCHING_C`, `PATCHING_A`, `CANDIDATE_SET_TYPE`, `INITIAL_TOUR_ALGORITHM`, or `MAX_TRIALS`. Thus, the baseline preserves the native search behavior of LKH-3. For the fixed-budget solution-quality evaluation, the same 10 s execution budget is imposed on all methods externally to ensure a fair comparison.

Allowed LKH-3 API Space. All candidate workflows are restricted to the same predefined solver-configuration space. We expose a set of wrapper APIs corresponding to major LKH-3 search components, including repeated search, local search neighborhood, patching, candidate-set construction, initial-tour generation, and search budget. The allowed API calls are:

```
api.set_runs(integer in [1, 5])
api.set_move_type(integer in [2, 5])
api.set_patching(c_integer in [0, 5], a_integer in [0, 2])
api.set_candidate_set(
    "ALPHA" | "NEAREST-NEIGHBOR" | "QUADRANT"
)
api.set_initial_tour(
    "GREEDY" | "NEAREST-NEIGHBOR" |
    "WALK" | "QUICK-BORUVKA"
)
api.set_max_trials(integer in [50, 2000])
```

These APIs are translated into the corresponding LKH-3 parameter-file entries:

```
RUNS
MOVE_TYPE
PATCHING_C
PATCHING_A
CANDIDATE_SET_TYPE
INITIAL_TOUR_ALGORITHM
MAX_TRIALS
```

Here, `RUNS` controls the number of independent search runs, while `MOVE_TYPE` determines the local-search move structure. `PATCHING_C` and `PATCHING_A` control the patching strategy used to combine subtours. `CANDIDATE_SET_TYPE` determines how candidate edges are constructed, and `INITIAL_TOUR_ALGORITHM` specifies the strategy used to generate the initial tour. Finally, `MAX_TRIALS` controls the maximum number of search trials.

Candidate workflows are not allowed to modify node coordinates, edge costs, or the underlying TSP instance, nor can they post-process the returned tour or invoke external solvers. Therefore, all methods operate within the same executable solver-control space.

Candidate Workflow Format. Each candidate workflow is represented as a short ordered sequence of API calls. For example,

```
api.set_runs(2)
api.set_move_type(4)
api.set_patching(0, 0)
api.set_candidate_set("QUADRANT")
api.set_initial_tour("NEAREST-NEIGHBOR")
```

The workflow is converted into an LKH-3 parameter file before execution. Different API calls control different parts of the search procedure, while their combination determines the overall solver behavior. All compared methods use the same API interface and parameter ranges; they differ only in how candidate workflows are generated and optimized.

Evaluation Protocol. We primarily evaluate solution quality using the percentage gap of the best feasible tour found within a fixed 10 s execution budget. For each TSP instance I, all methods are evaluated under the same time limit. Let $L^\star(I)$ denote the reference tour length and $L(W, I)$ denote the best feasible tour length obtained by workflow W within the time budget. The percentage gap is defined as

$$\mathrm{Gap}(W, I) = \frac{L(W, I) - L^\star(I)}{L^\star(I)} \times 100\%.$$

Lower values indicate better solution quality. For a test set $\mathcal{D}$, the average gap is

$$\overline{\mathrm{Gap}}(W) = \frac{1}{|\mathcal{D}|} \sum_{I \in \mathcal{D}} \mathrm{Gap}(W, I).$$

Each method is independently evaluated three times under the same experimental setting, and we report the mean and standard deviation of the resulting performance. This protocol is used for the TSP100, TSP200, and TSP500 results reported in the main experiments.

We additionally evaluate solving efficiency using the natural runtime of LKH-3. In this setting, no explicit LKH-3 `TIME_LIMIT` is specified, and runtime is measured as the wall-clock time from launching the LKH-3 process to its termination:

$$T(W, I) = t_{\mathrm{end}} - t_{\mathrm{start}}.$$

This runtime analysis is reported separately from the fixed budget solution quality evaluation.

Execution and Validation. LKH-3 is invoked through Python's `subprocess` interface. For each candidate workflow, the evaluator generates an independent parameter file, launches the LKH-3 executable, and reads the resulting tour file. The returned tour is checked to ensure that every node appears exactly once before its length is computed.

For Euclidean instances, node coordinates are converted to integer coordinates for TSPLIB compatibility, and LKH-3 uses the standard `EUC_2D` distance rule. The resulting tour length is subsequently rescaled to the original coordinate system for evaluation. If LKH-3 fails to return a valid tour or an external safety timeout is triggered, the run is explicitly recorded as a failure rather than being silently removed from evaluation.

A.2.3 Logic Synthesis Recipe Problem

Task Definition. We study API-based workflow optimization for logic synthesis. Given a combinational circuit, the goal is to discover a sequence of synthesis transformations that reduces the circuit size after technology mapping. Each candidate workflow consists of a bounded sequence of ABC/Yosys API calls applied before final mapping. The optimized circuit is then mapped to 6-input lookup tables (LUT6s), and the resulting LUT6 count is used as the optimization objective.

For a circuit C and workflow W, let $\mathrm{LUT}(W, C)$ denote the number of LUT6s after executing W and performing final mapping. The objective is

$$\min_{W} \ \mathrm{LUT}(W, C).$$

For a benchmark set $\mathcal{D}$, we evaluate a workflow by its aggregate synthesis quality across all circuits.

Default Synthesis Baseline. The default baseline follows a fixed synthesis flow without any learned or searched transformation sequence. Each circuit is directly processed by the synthesis backend and mapped to LUT6s using the same final mapping procedure as all compared methods. The resulting LUT6 count serves as the reference for evaluating API-based workflow optimization.

Allowed API Space. All candidate workflows are restricted to the same predefined ABC/Yosys transformation space. The allowed synthesis APIs are:

```
api.strash()
api.balance()
api.rewrite()
api.rewrite_z()
api.refactor()
api.refactor_z()
api.resub()
api.resub_z()
api.drewrite()
api.drefactor()
api.dc2()
api.fraig()
api.dch()
api.if_map(k=6)
```

These APIs cover structural hashing, logic balancing, rewriting, refactoring, resubstitution, don't-care-based optimization, functional reduction, and technology mapping. The final mapping operation `api.if_map(k=6)` must appear exactly once as the last API call, and the LUT size is fixed to $k = 6$ for all methods.

Candidate workflows cannot modify the original circuit, change the LUT size, invoke external synthesis tools, access files directly, or perform custom post-processing. Thus, all methods operate within the same synthesis-control space.

Candidate Workflow Format. Each candidate workflow is represented as an ordered sequence of synthesis API calls. For example:

```
api.strash()
api.balance()
api.rewrite()
api.resub()
api.dc2()
api.if_map(k=6)
```

The execution order is important because each transformation operates on the intermediate logic network produced by previous operations. Consequently, both the selected APIs and their ordering can affect the final mapped circuit size.

The outer evaluation pipeline is responsible for loading the circuit, executing the workflow, performing the final LUT6 mapping, and extracting the resulting LUT6 count. All compared methods therefore share the same API space, synthesis backend, and evaluation procedure, differing only in how candidate workflows are generated and optimized.

Evaluation Protocol. The primary evaluation metric is the LUT6 count after final mapping. For a workflow W evaluated on a benchmark set $\mathcal{D}$, we report both the total LUT6 count

$$\mathrm{TotalLUT}(W) = \sum_{C \in \mathcal{D}} \mathrm{LUT}(W, C),$$

and the average LUT6 count

$$\mathrm{AvgLUT}(W) = \frac{1}{|\mathcal{D}|} \sum_{C \in \mathcal{D}} \mathrm{LUT}(W, C).$$

Lower values indicate better synthesis quality.

We additionally report the relative improvement over the default synthesis baseline:

$$\mathrm{Improvement}(W) = \frac{\mathrm{TotalLUT}_{\mathrm{base}} - \mathrm{TotalLUT}(W)}{\mathrm{TotalLUT}_{\mathrm{base}}} \times 100\%.$$

A larger improvement indicates a more effective synthesis workflow. All methods are evaluated using the same circuit set, synthesis backend, API space, mapping target, and evaluation budget.

A.2.4 Parameter Settings

Common LLM Settings. Unless otherwise specified, all LLM-based methods use the same external LLM API endpoint and decoding configuration. We use `deepseek-coder` with temperature 0.9 and top-p 0.95. We do not impose an explicit `max_tokens` limit; instead, the prompt constrains the output to a short strategy description followed by executable API calls. Non-LLM baselines, such as default SCIP and default LKH-3, do not involve LLM queries.

Attribution Settings. AttriOpt uses the same attribution module across all tasks. Attribution is activated after at least five workflows have been evaluated. A component is individually estimated only when both its presence and absence are supported by at least two workflows. For interaction motifs, we use a stricter support requirement:

$$\min(s_j, N_t - s_j) \geq \max(6, \lceil 0.1 N_t \rceil).$$

For component c_j, the covariate vector $\mathbf{z}_i$ is constructed only from information available before workflow evaluation, including the generation index, proposal operator, and occurrence indicators of other coarse-grained components. The indicator of c_j itself is excluded. Post-execution quantities, such as solving time, optimality gap, synthesis quality, and runtime statistics, are not included to avoid post-treatment leakage.

The propensity model $e_j(\mathbf{z}_i)$ is implemented as an L_2-regularized logistic regression optimized for 250 gradient-descent steps with learning rate 0.1 and regularization strength 10^{-3}. Estimated propensity scores are clipped to $[0.05, 0.95]$. The conditional outcome models $\mu_{j,0}(\mathbf{z}_i)$ and $\mu_{j,1}(\mathbf{z}_i)$ are implemented as ridge regressions with regularization strength 10^{-3} and are fitted separately on workflows without and with component c_j. The intercept is not regularized.

To stabilize sparse estimates, empirical-Bayes shrinkage is applied with a minimum prior variance of 10^{-6}. Root-level component effects are shrunk toward zero, while fine-grained parameter effects are shrunk toward the attribution score of their corresponding parent API component. For attribution-guided prompting, we set

$$k_+ = k_- = 4,$$

and provide the top four positive and negative components together with their attribution scores and empirical support. If fewer stable components are available, only the available components are included.

Search Budgets. For MILP, the budget of 120 evaluated workflows is applied independently to each of the four benchmarks. Each benchmark contains 50 test instances, and SCIP is given a time limit of 60 s per instance.

For TSP, we use 20 training instances during workflow optimization and 50 held-out instances for final evaluation. The search budget is 256 evaluated workflows. Unless otherwise specified, each candidate is evaluated under a 10 s LKH-3 time limit during search.

For logic synthesis, the search budget is 64 evaluated workflows. Each candidate workflow is evaluated on all 20 EPFL combinational circuits. The execution timeout for Yosys/ABC is 30 s per circuit.

A.2.5 PROMPTS

AttriOpt uses a unified operator-conditioned prompt structure across all tasks. The task description, optimization objective, available API space, and task-specific constraints are instantiated according to the target environment, while the operator instruction is determined by the evolutionary operator. During attribution-guided evolution, the prompt additionally provides the positive and negative component sets $\mathcal{C}_t^+$ and $\mathcal{C}_t^-$ estimated from the accumulated archive.

The general template is:

```
Task:
<TASK_DESCRIPTION>

Objective:
<OBJECTIVE_DESCRIPTION>

Available APIs:
<API_LIST>

Task-specific constraints:
<TASK_CONSTRAINTS>

Operator:
<OPERATOR_NAME>

Operator instruction:
<OPERATOR_INSTRUCTION>

Positive attribution guidance:
<POSITIVE_COMPONENTS>

Negative attribution guidance:
<NEGATIVE_COMPONENTS>

Parent workflow(s):
<PARENT_WORKFLOWS>

Generate a valid API workflow using only the provided APIs.
```

Elite Recombination. For elite recombination, the prompt provides selected parent workflows together with attribution guidance. The LLM is instructed to construct a new workflow by recombining useful structures from the parents, while preferentially preserving positively attributed components and avoiding negatively attributed ones.

Backbone Refinement. For backbone refinement, a high-performing workflow is provided as the backbone. The LLM is instructed to preserve its main structure while selectively revising components indicated as unfavorable by the attribution feedback.

Structural Mutation. Structural mutation modifies the API composition of a parent workflow. The LLM is instructed to insert, remove, or replace API calls while using $\mathcal{C}_t^+$ and $\mathcal{C}_t^-$ as guidance.

Parameter Mutation. Parameter mutation preserves the overall API structure of the parent workflow and modifies its parameter configurations according to the attribution feedback.

Across all operators, the generated workflow is restricted to the predefined API space and must satisfy the execution constraints of the corresponding task.

A.2.6 GENERATED CODE

The LLM-generated output is inserted into a fixed evaluator. The evaluator handles instance loading, solver execution, result parsing, and metric computation. The generated code only specifies the API-call workflow. Below we show representative generated workflows from the three tasks.

MILP solver configuration. For MILP, the generated code directly configures SCIP heuristics through PySCIPOpt before calling `model.optimize()`. One generated workflow used in our experiments is:

```
def customize_solver(model, instance, trace) -> None:
    """Write direct PySCIPOpt parameter-setting code here."""
    model.setHeuristics(SCIP_PARAMSETTING.FAST)
    model.setIntParam("heuristics/rounding/freq", 0)
    model.setIntParam("heuristics/dualval/freq", 0)
    model.setIntParam("heuristics/oneopt/freq", 0)
    model.setIntParam("heuristics/rounding/freqofs", 1)
```

Another generated MILP workflow is:

```
def customize_solver(model, instance, trace) -> None:
    """Write direct PySCIPOpt parameter-setting code here."""
    model.setHeuristics(SCIP_PARAMSETTING.DEFAULT)
    model.setIntParam("heuristics/alns/freq", 3)
    model.setIntParam("heuristics/rens/freq", 0)
```

TSP solving. For TSP, the generated workflow configures LKH-3 through API calls, which are then translated into an LKH parameter file. A generated workflow used for TSP500 is:

```
api.set_runs(2)
api.set_move_type(4)
api.set_patching(0, 0)
api.set_candidate_set("QUADRANT")
api.set_initial_tour("NEAREST-NEIGHBOR")
```

This corresponds to the following LKH-3 parameters:

```
RUNS = 2
MOVE_TYPE = 4
PATCHING_C = 0
PATCHING_A = 0
CANDIDATE_SET_TYPE = QUADRANT
INITIAL_TOUR_ALGORITHM = NEAREST-NEIGHBOR
```

Logic synthesis. For logic synthesis, the generated code is a sequence of ABC/Yosys transformation calls. One generated workflow used in the EPFL experiment is:

```
def customize_solver(api, instance, trace):
    """Write direct ABC/Yosys synthesis API-call code only."""
    api.strash()
    api.drefactor()
    api.rewrite_z()
    api.dc2()
    api.balance()
```

```
    api.fraig()
    api.if_map(k=6)
```

The corresponding ABC command sequence is:

```
strash; drefactor; rewrite -z; dc2; balance; fraig; if; mfs2
```