# SPO: DISCOVERING ADAPTIVE LARGE NEIGHBORHOOD SEARCH OPERATORS VIA STACKELBERG PROGRAM OPTIMIZATION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Large neighborhood search (LNS) relies critically on destroy and repair operators, whose effectiveness depends on both adaptation to the evolving LNS state and interaction between the two roles. We introduce Stackelberg Program Optimization (SPO), an LLM-based framework for discovering adaptive executable destroy–repair programs. SPO conditions operator decisions on a compact LNS state, allowing state-dependent behavior to emerge through program discovery, and organizes destroy–repair discovery as a Stackelberg interaction over program space that reflects their asymmetric dependency. Role-specific credits evaluate destroy programs as leaders and repair programs as conditional follower responses, guiding a coupled optimization process that combines LLM generator learning with population-based evolutionary search over programs. Experiments on the traveling salesperson problem and capacitated vehicle routing problem show that SPO outperforms strong baselines across a broad range of settings and generalizes beyond the discovery scale to larger instances and benchmark sets. Behavioral analyses further demonstrate state-dependent operator behavior and coupled destroy–repair improvement during discovery.

## 1 INTRODUCTION

Combinatorial optimization problems arise widely in routing, scheduling, planning, and resource allocation, yet their rapidly growing search spaces often make exact optimization computationally prohibitive. Large neighborhood search (LNS) has emerged as a powerful metaheuristic framework for obtaining high-quality solutions within practical computational budgets. It repeatedly destroys part of a current solution and repairs the resulting partial structure, enabling broad exploration while preserving useful solution components (Shaw, 1998; Pisinger & Ropke, 2018; Mara et al., 2022). Its effectiveness, however, depends critically on the design of its destroy and repair operators.

Effective LNS operator discovery should account for two intrinsic properties: **adaptation** over the LNS search process and **interaction** between destroy and repair, rather than treating operators as fixed and isolated rules. As LNS progresses, the strategy that is effective at one stage may no longer be appropriate at another. For example, an operator may preserve the current solution more carefully while progress is being made, but change how it modifies the solution once the LNS search stagnates. Meanwhile, destroy and repair are inherently coupled: the neighborhood induced by destruction shapes the reconstruction problem faced by repair, making the effectiveness of either dependent on the other. This raises a central question: *how can we discover LNS operators that adapt to the evolving LNS state while coordinating destroy and repair?*

Designing such adaptation by hand is difficult: beyond specifying what an operator should do, one must determine how its strategy should respond to the evolving LNS state. Recent advances in LLM–based algorithm discovery offer a natural alternative by enabling complex algorithmic behaviors to be discovered directly as executable programs (Romera-Paredes et al., 2024; Liu et al., 2024; Ye et al., 2024). Building on this capability, we represent LNS operators as *adaptive operator programs* whose decisions are conditioned on the evolving LNS state. In this way, adaptation becomes part of the operator program itself, allowing both the operator strategy and its state-dependent changes during an LNS rollout to emerge through discovery rather than be prescribed by hand.



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.

Interaction fundamentally changes how destroy and repair programs should be discovered: their effectiveness is coupled, yet their roles are asymmetric. We therefore introduce *Stackelberg Program Optimization* (SPO), which formulates their discovery as a leader–follower interaction (von Stackelberg, 1934). Rather than optimizing the two roles independently, SPO carries this interaction throughout program generation, evaluation, and optimization. In each program-discovery round, destroy candidates are generated first, followed by conditional repair responses, and the resulting program pairs are evaluated through LNS rollouts, with their performance translated into role-specific credits for the leader and follower programs. These credits drive a coupled generator–population optimization process: we use Group Relative Policy Optimization (GRPO) (Shao et al., 2024) to update the role-specific LLM generators, while evolutionary selection and variation update the program populations. Together, they shape program generation in subsequent rounds.

Experiments on the traveling salesperson problem (TSP) and capacitated vehicle routing problem (CVRP) show that SPO discovers high-quality destroy–repair programs and generalizes from the discovery setting to larger instances and benchmark distributions. Behavioral analyses provide evidence that the discovered operators exhibit adaptive behavior in response to the evolving LNS state, while destroy and repair develop in a coupled manner during discovery. Controlled ablations confirm the contributions of Stackelberg credit assignment and state conditioning. Together, these results show that SPO provides a framework for discovering adaptive and interacting LNS operators.

## 2 Related Work

### 2.1 Large Neighborhood Search

LNS explores large implicit neighborhoods through iterative destruction and reconstruction (Shaw, 1998; Pisinger & Ropke, 2018; Mara et al., 2022). Classical methods use handcrafted destroy and repair heuristics, such as related removal and regret-based insertion (Shaw, 1998; Ropke & Pisinger, 2006; Pisinger & Ropke, 2018; Voigt, 2025). Learning-based LNS methods have introduced parametric models into components of LNS, including reconstruction (Hottung & Tierney, 2020; Falkner et al., 2022), removal (Chen et al., 2020; Hottung et al., 2025b), and neighborhood selection in integer programming (Song et al., 2020; Sonnerat et al., 2021; Wu et al., 2021; Huang et al., 2023). Another line of work focuses on controlling which operators are applied during LNS execution. ALNS selects among predefined operators using performance-based weights (Ropke & Pisinger, 2006), while subsequent methods use bandit or reinforcement-learning approaches for operator selection and LNS search control (Reijnen et al., 2024; Cai et al., 2025). Recent work considers dependencies between destroy and repair in operator selection and coordination (Yu et al., 2026b).

### 2.2 LLM-Based Operator Discovery

LLM-based automatic heuristic design searches directly over executable programs, using task performance as feedback to iteratively improve generated heuristics. EoH evolves heuristic programs through LLM-based generation and evolutionary operators, while FunSearch combines an LLM with an evaluator to iteratively search for high-performing programs (Liu et al., 2024; Romera-Paredes et al., 2024). ReEvo further incorporates reflective feedback from heuristic evaluations to guide subsequent evolution (Ye et al., 2024). Subsequent work has extended this paradigm along several directions, including more efficient heuristic-program discovery (Wu et al., 2025), co-evolution of solver programs and instance generators (Ke et al., 2026), and joint optimization of heuristic programs and the LLM generator (Huang et al., 2026).

Recent work has applied this paradigm to LNS components. LLM-LNS discovers neighborhood-selection strategies for large-scale MILPs, determining which variables are released for reoptimization (Ye et al., 2025). For vehicle routing, VRPAgent discovers problem-specific destroy and reinsertion-ordering operators while retaining a fixed greedy insertion procedure (Hottung et al., 2025a). G-LNS discovers both destroy and repair programs for LNS (Zhao et al., 2026), with SpecAHD further specializing repair heuristics to different local regions in large-scale routing (Lai et al., 2026), while full-component evolution applies program discovery across multiple components of ALNS (Yu et al., 2026a). SPO differs in how destroy and repair are represented and optimized: their programs condition their behavior on the evolving LNS state, while their discovery is organized through an asymmetric leader–follower formulation.

# 3 PRELIMINARIES

## 3.1 LARGE NEIGHBORHOOD SEARCH

Consider a combinatorial optimization instance $I$ in minimization form, with feasible solution set $\mathcal{X}(I)$ and objective $f_I(x)$. At iteration $t$, LNS maintains an incumbent solution $x_t \in \mathcal{X}(I)$. Let $\xi_t$ collect the stochastic choices made by the operators. A destroy operator $D$ removes a subset of the incumbent's components and returns a partial solution $\tilde{x}_t$ together with the removed set $V_t$; a repair operator $R$ then reconstructs a feasible candidate $x'_t$:

$$(\tilde{x}_t, V_t) = D(x_t; \xi_t), \qquad x'_t = R(\tilde{x}_t, V_t; \xi_t). \tag{1}$$

An acceptance rule uses the candidate $x'_t$ to determine the next incumbent $x_{t+1}$. Writing $\xi = (\xi_0, \ldots, \xi_{T-1})$, repeated updates produce the solution trajectory

$$\tau(D, R; I, \xi) = (x_0, x_1, \ldots, x_T). \tag{2}$$

## 3.2 STACKELBERG GAMES

Stackelberg games (von Stackelberg, 1934) model sequential interactions in which two decisions are interdependent: a leader acts first and shapes the decision faced by a follower, while the follower's anticipated response in turn affects the leader's optimal decision, as in supplier–retailer pricing (Liu et al., 2007) or defender–attacker resource allocation (Korzhyk et al., 2010). Formally, let $U(s_L, s_F)$ denote the utility shared by the leader and follower. The follower selects a best response to the leader's strategy, and the leader optimizes while anticipating this response:

$$s_F^\star(s_L) \in \arg\max_{s_F} U(s_L, s_F), \; s_L^\star \quad \in \arg\max_{s_L} U\big(s_L, s_F^\star(s_L)\big). \tag{3}$$

This formulation captures the asymmetric dependence between the two decisions.

# 4 STACKELBERG PROGRAM OPTIMIZATION

We formulate adaptive operator discovery as *Stackelberg Program Optimization* (SPO). SPO maintains role-specific LLM generators $\pi_{\theta_D}$ and $\pi_{\theta_R}$, which generate executable destroy and repair programs whose decisions can depend on the evolving LNS state. Destroy programs act as leaders and repair programs as followers; their joint evaluations yield role-specific credits that drive coupled generator–population optimization. Figure 1 summarizes the framework.

During program discovery, candidate destroy–repair program pairs are evaluated through LNS rollouts. The discovery process searches over executable programs, whereas each rollout performs the underlying solution search for a given instance. Throughout, *program* refers to sampled executable code, while *operator* refers to the destroy or repair function instantiated by a program within LNS. Within each rollout, the programs remain fixed, while their decisions may vary with the current solution and the LNS state. At test time, the selected programs are directly executed within LNS.

## 4.1 ADAPTIVE OPERATOR PROGRAMS

LNS is inherently history dependent: the same incumbent solution may call for different decisions depending on whether it has just been improved or has remained unchanged over many iterations. Let $h_t$ denote the LNS history up to iteration $t$, including previous solutions and improvement outcomes. We introduce a compact *LNS state*

$$z_t = \phi(h_t), \tag{4}$$

and provide this state as an input to both operator programs:

$$(\tilde{x}_t, V_t) = D(x_t, z_t; \xi_t), \qquad x'_t = R(\tilde{x}_t, V_t, z_t; \xi_t). \tag{5}$$

This formulation makes adaptation part of the operator discovery space. Rather than encoding a fixed heuristic, an adaptive program can jointly determine an operator strategy and how its decisions change with the LNS state. Equivalently, a program $O(x, z)$ compactly represents a state-indexed family of operators $\{O_z : x \mapsto O(x, z)\}_{z \in \mathcal{Z}}$, whose effective strategy may vary with the LNS state.

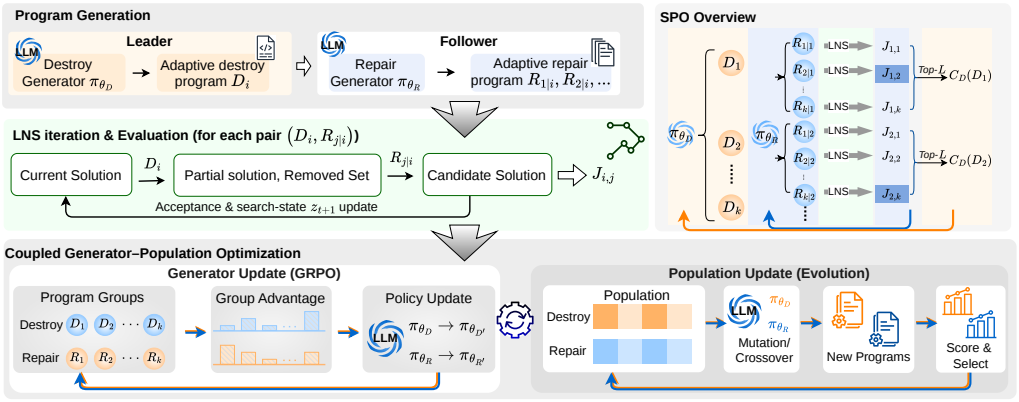



Figure 1: Overview of Stackelberg Program Optimization (SPO). **Program Generation** produces leader destroy programs and conditionally generated follower repair programs. **LNS Iteration & Evaluation** executes each program pair with LNS-state conditioning and evaluates its rollout utility. **Coupled Generator–Population Optimization** uses role-specific credits to update the LLM generators via GRPO and evolve the executable-program populations. **SPO Overview** summarizes the hierarchical generation, evaluation, and optimization process.

The formulation is agnostic to the particular state representation $\phi$. Ideally, the LNS state should provide trajectory information complementary to the current solution while remaining compact and avoiding unnecessary dependence on problem- or instance-specific quantities. We therefore deliberately use a minimal, problem-independent signal,

$$z_t = \text{number of iterations since the last improvement.} \tag{6}$$

It is reset to zero after an improvement and incremented otherwise, providing a simple measure of recent progress versus increasing stagnation. This choice introduces negligible overhead and leaves the state-dependent decision logic itself to be discovered by the program. Richer representations could incorporate additional trajectory information, while their design and transferability across problems and instance distributions remain open directions.

### 4.2 Stackelberg Formulation for Operator Discovery

We formulate destroy–repair discovery as a Stackelberg interaction, with destroy programs acting as leaders and repair programs as followers. Reflecting this ordering, SPO first samples destroy candidates and then generates repair candidates conditionally on each destroy:

$$D_i \sim \pi_{\theta_D}(\cdot \mid \gamma_D), \qquad R_{j|i} \sim \pi_{\theta_R}(\cdot \mid \gamma_R, D_i), \tag{7}$$

where $\gamma_D$ and $\gamma_R$ denote the corresponding generation contexts, specifying the operator role, program interface, generation constraints, and optional program context. Here $R_{j|i}$ denotes the $j$-th repair candidate generated conditionally on destroy program $D_i$.

To evaluate a destroy–repair pair, let $\tau(D, R; I, \xi)$ denote its LNS rollout on instance $I$, where $\xi$ captures rollout stochasticity, and let

$$x^\star_\tau \in \arg\min_{x \in \tau(D,R;I,\xi)} f_I(x) \tag{8}$$

be the best feasible solution encountered. We define the pair utility as

$$J(D, R) = \mathbb{E}_{I,\xi}\left[\frac{f_I(x_0) - f_I(x^\star_\tau)}{\max|f_I(x_0)|, \epsilon_{\text{den}}}\right], \qquad \epsilon_{\text{den}} > 0, \tag{9}$$

where the expectation is over training instances and stochastic rollouts, with higher values indicating greater improvement. The small constant $\epsilon_{\text{den}}$ stabilizes the denominator.

The follower best response to a destroy program is

$$R^\star(D) \in \arg\max_{R \in \mathcal{R}} J(D, R), \tag{10}$$

and the leader anticipates this response:

$$D^\star \in \arg\max_{D \in \mathcal{D}} J\big(D, R^\star(D)\big). \tag{11}$$

Since exact best responses cannot be computed over the space of executable programs, SPO realizes these objectives using sampled candidates and role-specific credit assignment.

### 4.3 Stackelberg Credit Assignment

The Stackelberg formulation induces distinct optimization signals for the two operator roles. A repair candidate is evaluated by how well it responds to the destroy program on which it is conditioned, whereas a destroy candidate is evaluated through the strong repair responses it elicits.

**Follower credit.** For a destroy candidate $D_i$, let $\mathcal{R}_i = \{R_{1|i}, \ldots, R_{N|i}\}$ denote its conditionally generated repair candidates. Following Equation 10, each repair candidate receives the credit

$$C_R(R_{j|i}; D_i) = J(D_i, R_{j|i}). \tag{12}$$

**Leader credit.** Since the exact best response in Equation 10 is unavailable, we approximate it using the strongest conditionally sampled repair candidates. Let $\mathrm{Top}_L(\mathcal{R}_i)$ contain the $L$ candidates with the highest pair utilities. The leader credit is

$$C_D(D_i) = \frac{1}{L} \sum_{R_{j|i} \in \mathrm{Top}_L(\mathcal{R}_i)} J(D_i, R_{j|i}). \tag{13}$$

Averaging the top-$L$ responses rather than taking a single empirical maximum reduces sensitivity to rollout noise while retaining emphasis on strong follower responses.

### 4.4 Coupled Generator–Population Optimization

SPO jointly improves the two components governing program generation in Equation 7: the role-specific generators $\pi_{\theta_D}$ and $\pi_{\theta_R}$, and the program populations that provide context through $\gamma_D$ and $\gamma_R$. GRPO optimizes the generators from role-specific credits, while evolutionary optimization updates the populations used as program context. These two processes are coupled across discovery rounds, as each generation is produced by the current generators under the current population contexts and subsequently used to update both. Algorithm 1 (Appendix A.1) summarizes SPO training.

**GRPO-based generator update.** We use Group Relative Policy Optimization (GRPO) (Shao et al., 2024) to optimize the role-specific generators based on the credits defined above. Specifically, $C_D(D_i)$ is used for destroy programs and $C_R(R_{j|i}; D_i)$ for repair programs. For either role, let $\{y_i\}_{i=1}^B$ denote a group of programs sampled from $\pi_{\theta_{\text{old}}}(\cdot \mid \gamma)$ under the same generation context $\gamma$. Given their corresponding credits $\{C_i\}_{i=1}^B$, we compute the group-centered advantage

$$A_i = C_i - \frac{1}{B} \sum_{j=1}^{B} C_j. \tag{14}$$

The probability ratio between the updated and sampling policies is

$$\rho_i(\theta) = \frac{\pi_\theta(y_i \mid \gamma)}{\pi_{\theta_{\text{old}}}(y_i \mid \gamma)}. \tag{15}$$

The corresponding generator is updated by maximizing the clipped objective

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}\left[\frac{1}{B} \sum_{i=1}^{B} \min(\rho_i(\theta) A_i, \mathrm{clip}(\rho_i(\theta), 1-\epsilon, 1+\epsilon)\, A_i)\right], \tag{16}$$

where $\epsilon > 0$ controls clipping. This update is applied separately to the destroy and repair generators, $\pi_{\theta_D}$ and $\pi_{\theta_R}$. For repair updates, the sampling and probability-ratio expressions are conditioned on the same destroy program $D_i$ within each group; this dependence is omitted for brevity.

**Evolutionary population update.** We use evolutionary optimization to update the role-specific program populations that provide context for subsequent generation. At discovery round $k$, let $\mathcal{P}_D^{(k)}$ and $\mathcal{P}_R^{(k)}$ denote the current destroy and repair populations. To generate new candidates, programs from these populations are incorporated into $\gamma_D$ and $\gamma_R$ as program context. The corresponding role-specific generators then generate mutations of individual programs or crossovers that combine multiple programs, producing candidate sets $\mathcal{G}_D^{(k)}$ and $\mathcal{G}_R^{(k)}$.

Destroy programs can be compared using $C_D$. Repair candidates, however, may originate from different conditioning destroy programs, making their conditional credits dependent on different leader contexts. To obtain a common basis for repair population selection, we therefore construct a diverse leader panel $\mathcal{E}_D = \{D_1, \ldots, D_M\}$ and define the cross-play score

$$C_R^{\mathrm{pop}}(R) = \frac{1}{M} \sum_{m=1}^{M} J(D_m, R). \tag{17}$$

The populations are selected from $\mathcal{P}_D^{(k)} \cup \mathcal{G}_D^{(k)}$ and $\mathcal{P}_R^{(k)} \cup \mathcal{G}_R^{(k)}$, using $C_D$ and $C_R^{\mathrm{pop}}$, respectively.

# 5 Experiments

We evaluate SPO on the traveling salesperson problem (TSP) and capacitated vehicle routing problem (CVRP) to answer three questions: (i) whether SPO discovers adaptive operators that improve solution quality; (ii) whether the discovered operators generalize across instance sizes and distributions; (iii) how Stackelberg credit assignment and LNS-state conditioning contribute to performance.

## 5.1 Experimental Setup

**Common settings.** We use `Qwen3.5-9B` (Qwen Team, 2026) as a shared backbone with role-specific LoRA adapters (Hu et al., 2021) for the destroy and repair generators. SPO runs for 30 program-discovery rounds and maintains at most 10 executable programs per role. Discovery evaluates candidate pairs using 100-step LNS rollouts with non-worsening acceptance and retains the pair with the highest discovery utility $J$ found so far. Final evaluation runs the retained pair for 500 LNS iterations. See Appendices A, B, and C for implementation and evaluation details.

**Evaluation metric.** Discovery optimizes the relative-improvement utility $J$ in Equation 9. Final performance is measured by the reference gap $\mathrm{Gap}(x) = 100(f(x) - f^{\mathrm{ref}})/f^{\mathrm{ref}}$, using benchmark optimal or best-known values when available and high-quality solver solutions otherwise. Results are averaged over test instances; lower is better.

### 5.1.1 Traveling Salesperson Problem

**Setting and benchmarks.** SPO is trained on uniformly sampled 50-node Euclidean TSP instances. We evaluate the discovered operators on held-out uniform instances of increasing size and on TSPLIB (Reinelt, 1991), which provides benchmark instances with more heterogeneous geometric structure than the synthetic training distribution. Reference values for synthetic instances are obtained with Concorde (Applegate et al., 2011), while TSPLIB uses the optimal tour lengths.

**Baselines.** We consider two baseline groups. Handcrafted methods include the construction heuristics Nearest Neighbor (NN) and Farthest Insertion (FI), the local-search methods 2-opt and 3-opt (Rosenkrantz et al., 1977; Croes, 1958; Lin, 1965), and ALNS (Ropke & Pisinger, 2006), which adaptively selects from a fixed operator portfolio. Program-discovery methods include the general LLM-based frameworks EoH (Liu et al., 2024), FunSearch (Romera-Paredes et al., 2024), and ReEvo (Ye et al., 2024), together with G-LNS (Zhao et al., 2026), which specifically discovers destroy and repair programs for LNS. All program-discovery baselines use the same final LNS iteration budget as SPO; EoH, FunSearch, and ReEvo also use SPO's destroy–repair interface (Appendix C).

### 5.1.2 Capacitated Vehicle Routing Problem

**Setting and benchmarks.** SPO is trained on synthetic 50-customer CVRP instances with uniformly sampled coordinates and demands. We evaluate the discovered operators on held-out uni-

Table 1: Mean final reference gaps (%) on TSP and CVRP benchmarks. Bold indicates the best reported result in each column; lower is better.

**(a) Traveling Salesperson Problem**

| Method | Uniform instances (number of nodes $n$) | | | | TSPLIB (instance-size range) | | |
|---|---|---|---|---|---|---|---|
| | 50 | 100 | 200 | 500 | 50–100 | 100–200 | $> 200$ |
| NN | 21.295 | 23.807 | 24.884 | 25.999 | 26.039 | 23.143 | 25.710 |
| FI | 6.261 | 7.711 | 8.909 | 10.535 | 7.336 | 7.490 | 11.662 |
| 2-opt | 3.796 | 4.197 | 5.078 | 5.179 | 4.120 | 3.707 | 5.600 |
| 3-opt | 2.232 | 2.905 | 3.900 | 4.578 | 2.859 | 3.041 | 4.764 |
| ALNS | 1.720 | 2.370 | 3.106 | 4.103 | 1.865 | 2.414 | 4.106 |
| EoH | 1.222 | 4.582 | 8.843 | 15.708 | 2.305 | 3.859 | 11.627 |
| FunSearch | 1.201 | 2.372 | 5.717 | 18.016 | 1.670 | 2.045 | 6.445 |
| ReEvo | 1.252 | 3.707 | 6.642 | 11.577 | 2.210 | 3.869 | 11.166 |
| G-LNS | 0.483 | 1.385 | 2.589 | 6.305 | 0.597 | 1.196 | 4.145 |
| **SPO** | **0.212** | **0.642** | **1.274** | **3.139** | **0.408** | **0.574** | **4.040** |

**(b) Capacitated Vehicle Routing Problem**

| Method | Uniform instances (number of customers $n$) | | | | CVRPLIB (instance family) | | | |
|---|---|---|---|---|---|---|---|---|
| | 50 | 100 | 150 | 200 | A | B | P | X |
| CW | 5.929 | 6.418 | 6.271 | 6.056 | 5.089 | 4.416 | 8.102 | 6.013 |
| OR-Tools | 6.036 | 7.245 | 9.811 | 10.902 | 4.626 | 4.183 | 4.401 | 7.755 |
| LS | 5.450 | 5.590 | 5.752 | 5.218 | 5.121 | 5.793 | 7.706 | 5.529 |
| ALNS | 5.892 | 6.000 | 6.956 | 7.165 | 3.979 | 3.486 | 3.910 | 9.680 |
| EoH | 3.842 | 7.497 | 12.252 | 15.788 | 4.451 | 3.811 | 3.607 | 16.199 |
| FunSearch | 3.855 | 6.092 | 13.142 | 16.251 | 3.992 | 4.444 | 2.853 | 16.279 |
| ReEvo | 5.737 | 8.531 | 10.616 | 14.870 | 5.088 | 5.388 | 3.587 | 15.747 |
| G-LNS | 7.704 | 8.206 | 7.357 | 7.535 | 3.634 | 2.542 | 4.030 | 5.266 |
| **SPO** | **3.190** | **3.221** | **3.273** | **4.093** | **2.445** | **1.918** | **2.351** | **5.008** |

form instances and on the A, B, P, and X families from CVRPLIB (Augerat et al., 1995; Uchoa et al., 2017), which cover benchmark instances with varied spatial, demand, and capacity characteristics.

**Baselines.** The handcrafted and solver baselines comprise Clarke–Wright savings (CW) (Clarke & Wright, 1964), Google OR-Tools (Furnon & Perron, 2024), local search (LS) using relocate, swap, and 2-opt neighborhoods (Vidal et al., 2014), and ALNS (Ropke & Pisinger, 2006). We use the same program-discovery group where applicable: EoH, FunSearch, ReEvo, and G-LNS.

### 5.2 Main Results

Table 1 reports mean final reference gaps on TSP and CVRP. All LNS-based methods are evaluated with the same budget of 500 destroy–repair iterations.

**TSP.** SPO achieves the lowest reference gap in all seven TSP groups. Its advantage generally widens beyond the $n = 50$ discovery setting as several program-discovery baselines degrade more strongly with instance size, and remains strongest up to $n = 500$. It also leads across all three TSPLIB ranges, showing that the discovered operators transfer beyond the synthetic discovery distribution.

**CVRP.** CVRP provides a complementary test under capacity constraints and multi-route solution structure. SPO achieves the lowest gap in every group despite substantial variation in the strongest competing method across synthetic sizes and CVRPLIB families. On synthetic instances, its performance degrades much more slowly with scale than several program-discovery baselines, and the advantage also transfers across all four CVRPLIB families.

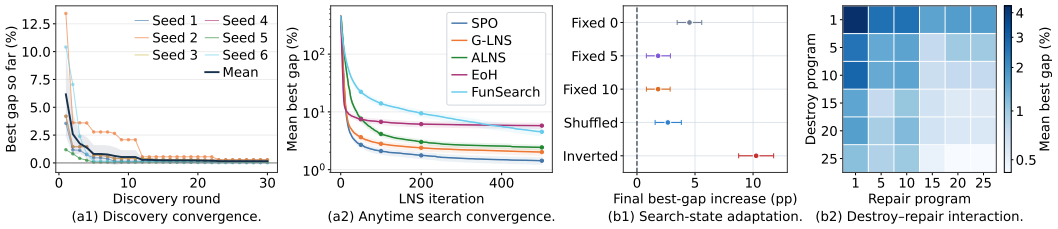



Figure 2: **Optimization dynamics and behavioral analysis of SPO.** (a1) Best-so-far discovery gap across six TSP runs; thick line: mean, shading: across-seed variation. (a2) Anytime best-gap convergence during TSP LNS execution. (b1) State interventions on one discovered TSP pair: *Fixed* $k$ sets $z_t = k$; *Shuffled* permutes the observed state sequence; *Inverted* maps $z_t < 10$ to 30 and otherwise to 0. Shown are mean final-gap increases over unmodified execution with 95% bootstrap CIs from 2,000 instance–seed resamples. (b2) Pairwise combinations of destroy and repair programs from the indicated rounds of one CVRP discovery run; colors show mean best gap (power norm $\gamma = 0.5$). Lower is better; pp = percentage points.

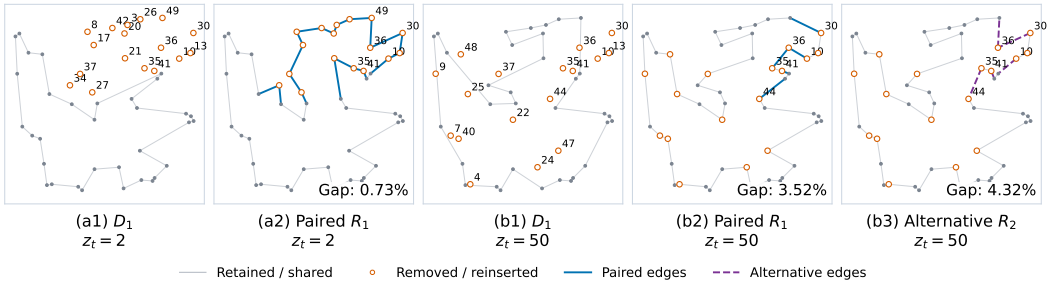



Figure 3: **Illustrative case study of state adaptation and destroy–repair interaction.** The same TSP instance and incumbent are used throughout. (a1–a2) Destroy and subsequent repair by a discovered pair $(D_1, R_1)$ under the true state $z_t = 2$. (b1–b2) The same pair under the counterfactual state $z_t = 50$. (b3) Starting from the same destruction as in (b1), the paired repair $R_1$ is replaced by an alternative repair $R_2$, yielding a different reconstruction.

### 5.3 Optimization Dynamics

Figure 2(a1) tracks the best-so-far discovery gap across six independent runs with different random seeds. The runs consistently improve over discovery rounds and converge to similarly low gaps, indicating stable discovery performance across seeds. (a2) compares LNS convergence using SPO's discovered operators with that of the baselines. SPO attains lower gaps early and continues improving after several baselines have largely plateaued, showing that its advantage persists throughout LNS execution rather than emerging only at the final evaluation point.

### 5.4 Behavioral Analysis

**LNS-state adaptation.** Figure 2(b1) shows that disrupting the correspondence between the LNS state and search progress consistently degrades performance, with inversion producing the largest effect. This indicates that the tested pair makes meaningful use of the state signal rather than treating it as an incidental input. Figure 3 provides a qualitative TSP case study of state adaptation. At a trajectory point with true state $z_t = 2$, a discovered pair produces the destroy and repair decisions in (a1–a2). Keeping the instance and incumbent fixed, we counterfactually set $z_t = 50$ and rerun the same pair in (b1–b2). The destroy pattern shifts from a relatively localized region of the tour to a more dispersed one, leading to a different reconstruction and illustrating a state-dependent transition from local refinement to broader exploration.

**Destroy–repair interaction.** Figure 2(b2) shows coupled improvement across discovery. While performance improves along both role axes, programs developed at similar stages also tend to form locally strong combinations, suggesting that destroy and repair are progressively optimized in re-

Table 2: Controlled TSP ablations of Stackelberg credit assignment and LNS-state adaptation. Values are mean final reference gaps (%); lower is better.

| Variant | Uniform instances (number of nodes $n$) | | | | TSPLIB (instance-size range) | | |
|---|---|---|---|---|---|---|---|
| | 50 | 100 | 200 | 500 | 50–100 | 100–200 | $> 200$ |
| SPO | **0.212** | **0.642** | **1.274** | **3.139** | **0.408** | **0.574** | **4.040** |
| w/o Stackelberg Credit | 1.398 | 1.908 | 3.089 | 5.995 | 1.309 | 2.054 | 6.647 |
| w/o State Conditioning | 0.273 | 0.816 | 2.686 | 6.484 | 0.592 | 0.962 | 6.850 |

lation to one another. Figure 3(b1–b3) provides an operator-level view of this interaction. The destruction in (b1) fixes the partial tour and removed nodes, thereby defining the subproblem faced by repair. Keeping this destroy output unchanged, replacing the paired $R_1$ with $R_2$ leads to a different reconstruction and a higher gap in this instance. This illustrates the conditional nature of destroy–repair interaction: the effect of a repair program depends on the particular partial structure produced by destroy, rather than being independent of its partner.

### 5.5 Ablation Study

**Stackelberg credit assignment.** We isolate the contribution of Stackelberg credit by replacing the role-specific leader–follower credits with a shared pair-level credit, where both programs in a destroy–repair pair receive the same rollout score. The **w/o Stackelberg Credit** variant consistently degrades across uniform and TSPLIB instances (Table 2), showing that coupling the two programs through joint evaluation alone is not sufficient. Explicitly distinguishing their leader and follower roles provides a more effective credit signal for optimizing interacting destroy and repair programs.

**LNS-state adaptation.** We then remove the LNS-state input while preserving the Stackelberg credit structure. As shown in Table 2, the resulting **w/o State Conditioning** variant remains close to SPO around the discovery scale, but the performance gap widens as the evaluation scale grows. The same pattern extends to larger TSPLIB instances. This suggests that state conditioning is less important near the discovery setting itself, but becomes increasingly important for maintaining effective operator behavior as the discovered programs are transferred beyond it.

## 6 Conclusion

We introduced Stackelberg Program Optimization (SPO) for discovering adaptive operators in large neighborhood search. SPO represents destroy and repair operators as executable programs whose behavior can adapt over the course of LNS execution, and organizes their discovery through a leader–follower formulation that captures the asymmetric dependency between the two roles. Role-specific credit assignment, together with coupled generator–population optimization, guides the discovery of interacting operator programs. Experiments on TSP and CVRP show that the discovered operators achieve strong solution quality across instance sizes and benchmark distributions, while the ablation and behavioral analyses provide evidence for adaptive operator behavior and coupled destroy–repair improvement during discovery. Together, these results demonstrate the value of accounting for both search dynamics and inter-program dependencies in automated operator discovery.

Although we instantiate SPO for destroy–repair discovery, the formulation provides a general optimization framework for other systems with leader–follower dependencies. Extending SPO beyond two roles to multi-component or multi-stage systems is a natural direction toward more general structured program optimization. More broadly, our adaptive formulation suggests expanding program discovery from optimizing isolated algorithmic components toward optimizing more of the algorithmic system itself. Mechanisms that are traditionally specified outside an individual program—such as when and how its strategy should change during execution—can instead be incorporated into the program search space and discovered jointly with the underlying algorithmic logic. This opens a direction toward system-level algorithm discovery, where both component behavior and the mechanisms governing its evolution during execution are optimized within a unified search space.

## AI USE STATEMENT

In this work, we used a large language model to generate destroy and repair programs as part of the proposed algorithm discovery method. Generative AI tools also assisted with experimental design, code development, and interpretation of results. Additionally, we used generative AI tools to iteratively revise and polish manuscript text with the authors. The authors reviewed all AI-assisted work and checked the generated programs and reported analyses. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REPRODUCIBILITY STATEMENT

The source code is available in an anonymous repository at `https://anonymous.4open.science/r/SPO-2A12/`. Section 4 describes SPO and its evaluation, while Appendix A provides program-discovery and training details. Appendix B specifies the problem settings, synthetic instance generation, and operator interfaces; Appendix C describes the evaluation protocols and baselines. Appendix D documents the program-generation contexts, and Appendix E summarizes the discovered operators. The benchmark instances are publicly available from TSPLIB (`https://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html`) and CVRPLIB (`https://galgos.inf.puc-rio.br/cvrplib/en/instances`).

## REFERENCES

David L Applegate, Robert E Bixby, Vašek Chvátal, and William J Cook. The traveling salesman problem: a computational study. In *The traveling salesman problem*. Princeton university press, 2011.

Philippe Augerat, Jose Manuel Belenguer, Enrique Benavent, Angel Corberán, Denis Naddef, and Giovanni Rinaldi. Computational results with a branch and cut code for the capacitated vehicle routing problem. *Rapport de recherche- IMAG*, 1995.

Junyang Cai, Serdar Kadioğlu, and Bistra Dilkina. BALANS: Multi-armed bandits-based adaptive large neighborhood search for mixed-integer programming problems. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pp. 2566–2574, 2025.

Mingxiang Chen, Lei Gao, Qichang Chen, and Zhixin Liu. Dynamic partial removal: A neural network heuristic for large neighborhood search. *arXiv preprint arXiv:2005.09330*, 2020.

G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.

Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6): 791–812, 1958.

Jonas K Falkner, Daniela Thyssens, and Lars Schmidt-Thieme. Large neighborhood search based on neural construction heuristics. *arXiv preprint arXiv:2205.00772*, 2022.

Vincent Furnon and Laurent Perron. OR-Tools Routing Library. Google, 2024. URL `https://developers.google.com/optimization/routing/`. Version 9.10.

André Hottung and Kevin Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. In *ECAI 2020: 24th European Conference on Artificial Intelligence*, pp. 443–450, 2020.

André Hottung, Federico Berto, Chuanbo Hua, Nayeli Gast Zepeda, Daniel Wetzel, Michael Römer, Haoran Ye, Davide Zago, Michael Poli, Stefano Massaroli, et al. VRPAgent: LLM-driven discovery of heuristic operators for vehicle routing problems. *arXiv preprint arXiv:2510.07073*, 2025a.

André Hottung, Paula Wong-Chung, and Kevin Tierney. Neural deconstruction search for vehicle routing problems. *Transactions on Machine Learning Research*, 2025b. ISSN 2835-8856.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Taoan Huang, Aaron M Ferber, Yuandong Tian, Bistra Dilkina, and Benoit Steiner. Searching large neighborhoods for integer linear programs with contrastive learning. In *International conference on machine learning*, pp. 13869–13890. PMLR, 2023.

Ziyao Huang, Weiwei Wu, Kui Wu, Wei-Bin Lee, and Jianping Wang. Calm: Co-evolution of algorithms and language model for automatic heuristic design. In *International Conference on Learning Representations*, 2026.

Xinyi Ke, Kai Li, Junliang Xing, Yifan Zhang, and Jian Cheng. Game-theoretic co-evolution for LLM-based heuristic discovery. In *Forty-third International Conference on Machine Learning*, 2026.

Dmytro Korzhyk, Vincent Conitzer, and Ronald Parr. Complexity of computing optimal stackelberg strategies in security resource allocation games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pp. 805–810, 2010.

Kezhao Lai, Yutao Lai, and Hai-Lin Liu. SpecAHD: Localize to specialize for automated heuristic design in large-scale routing problems. *arXiv preprint arXiv:2607.23676*, 2026.

Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.

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, pp. 32201–32223. PMLR, 2024.

Liming Liu, Mahmut Parlar, and Stuart X Zhu. Pricing and lead time decisions in decentralized supply chains. *Management science*, 53(5):713–725, 2007.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Setyo Tri Windras Mara, Rachmadi Norcahyo, Panca Jodiawan, Luluk Lusiantoro, and Achmad Pratama Rifai. A survey of adaptive large neighborhood search algorithms and applications. *Computers & Operations Research*, 146:105903, 2022.

David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pp. 99–127. Springer, 2018.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

Robbert Reijnen, Yingqian Zhang, Hoong Chuin Lau, and Zaharah Bukhsh. Online control of adaptive large neighborhood search using deep reinforcement learning. In *Proceedings of the international conference on automated planning and scheduling*, volume 34, pp. 475–483, 2024.

Gerhard Reinelt. TSPLIB-A traveling salesman problem library. *ORSA journal on computing*, 3(4): 376–384, 1991.

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

Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pp. 417–431. Springer, 1998.

Jialin Song, Ravi Lanka, Yisong Yue, and Bistra Dilkina. A general large neighborhood search framework for solving integer linear programs. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2020.

Nicolas Sonnerat, Pengming Wang, Ira Ktena, Sergey Bartunov, and Vinod Nair. Learning a large neighborhood search algorithm for mixed integer programs. *arXiv preprint arXiv:2107.10201*, 2021.

Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858, 2017.

Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. A unified solution framework for multi-attribute vehicle routing problems. *European Journal of Operational Research*, 234(3):658–673, 2014.

Stefan Voigt. A review and ranking of operators in adaptive large neighborhood search for vehicle routing problems. *European Journal of Operational Research*, 322(2):357–375, 2025.

Heinrich von Stackelberg. *Marktform und Gleichgewicht*. J. Springer, 1934.

Niels A Wouda, Leon Lan, and Wouter Kool. PyVRP: A high-performance VRP solver package. *INFORMS Journal on Computing*, 36(4):943–955, 2024.

Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. Efficient heuristics generation for solving combinatorial optimization problems using large language models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 3228–3239, 2025.

Yaoxin Wu, Wen Song, Zhiguang Cao, and Jie Zhang. Learning large neighborhood search policy for integer programming. *Advances in Neural Information Processing Systems*, 34:30075–30087, 2021.

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. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Huigen Ye, Hua Xu, An Yan, and Yaoyang Cheng. Large language model-driven large neighborhood search for large-scale MILP problems. In *ICML*, 2025.

Shaohua Yu, Tianyu Chen, Linyan Liu, and Jakob Puchinger. Large language model-driven full-component evolution of adaptive large neighborhood search. *arXiv preprint arXiv:2603.06996*, 2026a.

Shaohua Yu, Wenhao Mao, Zigao Wu, and Jakob Puchinger. New adaptive mechanism for large neighborhood search using dual actor-critic. *arXiv preprint arXiv:2601.11414*, 2026b.

Baoyun Zhao, He Wang, and Liang Zeng. G-LNS: Generative large neighborhood search for LLM-based automatic heuristic design. *arXiv preprint arXiv:2602.08253*, 2026.

## A Program Discovery and Training Details

This section provides the training procedure and implementation details of SPO; problem-specific settings and evaluation protocols are described in Appendices B and C, respectively.

### A.1 Overall Training Algorithm

Algorithm 1 summarizes the complete SPO training procedure. In each discovery round, SPO conditionally generates and evaluates destroy–repair programs, updates the role-specific generators with GRPO, and evolves the two program populations. Throughout discovery, SPO retains the evaluated destroy–repair pair with the highest utility $J$ found so far and returns it after $K$ rounds.

**Algorithm 1** Stackelberg Program Optimization

**Require:** Role-specific LLM generators $\pi_{\theta_D}, \pi_{\theta_R}$

Initialize program populations $\mathcal{P}_D^{(1)}, \mathcal{P}_R^{(1)}$
**for** program-discovery round $k = 1, \ldots, K$ **do**
  Construct destroy generation contexts from $\mathcal{P}_D^{(k)}$
  Generate destroy candidates $\mathcal{G}_D^{(k)}$ using $\pi_{\theta_D}$
  **for all** $D_i \in \mathcal{G}_D^{(k)}$ **do**
    Construct repair generation contexts from $\mathcal{P}_R^{(k)}$, conditioned on $D_i$
    Generate repair candidates $\mathcal{R}_i$ using $\pi_{\theta_R}$
    Evaluate $J(D_i, R_{j|i})$ for all $R_{j|i} \in \mathcal{R}_i$ and compute $C_D(D_i)$ using Equation 13
  **end for**
  $\mathcal{G}_R^{(k)} \leftarrow \bigcup_{D_i \in \mathcal{G}_D^{(k)}} \mathcal{R}_i$
  Update the best-so-far pair using the pairs evaluated in round $k$
  Center $C_D(D_i)$ and $J(D_i, R_{j|i})$ within their generation-context groups
  Update $\pi_{\theta_D}$ and $\pi_{\theta_R}$ with GRPO
  Construct a diverse leader panel $\mathcal{E}_D^{(k)}$
  Evaluate cross-play score $C_R^{\mathrm{pop}}(R)$ for repair population candidates
  Select $\mathcal{P}_D^{(k+1)}$ from $\mathcal{P}_D^{(k)} \cup \mathcal{G}_D^{(k)}$ using $C_D$ with diversity-aware survivor selection
  Select $\mathcal{P}_R^{(k+1)}$ from $\mathcal{P}_R^{(k)} \cup \mathcal{G}_R^{(k)}$ using $C_R^{\mathrm{pop}}$ with diversity-aware survivor selection
**end for**
Select the retained best-so-far pair $(D_{\mathrm{sel}}, R_{\mathrm{sel}})$
**return** selected pair $(D_{\mathrm{sel}}, R_{\mathrm{sel}})$ and populations $\mathcal{P}_D^{(K+1)}, \mathcal{P}_R^{(K+1)}$

### A.2 Model, Adapters, and Sampling

SPO uses `Qwen3.5-9B` as a frozen shared backbone with separate trainable LoRA adapters for the destroy and repair roles. Program generation uses vLLM with the corresponding role-specific adapter, while adapter updates are performed separately in PyTorch. Generation and training run sequentially on a single NVIDIA A100-SXM4 GPU with 40 GB of memory.

The maximum context length is 8,192 tokens. Programs are sampled with temperature $0.8$ and top-$p$ $0.95$, with maximum generation lengths of 1,200 tokens for TSP and 2,048 tokens for CVRP. All adapters use rank 16, scaling factor 32, zero dropout, and bfloat16 computation. The TSP adapters target the query, key, value, and output projections, while the CVRP adapters additionally target the gate, up, and down projections.

### A.3 Discovery Budget and Runtime

Each discovery round generates 30 destroy candidates from three mutation and two crossover groups, with six responses per group. For each valid destroy, SPO conditionally generates up to 30 repair candidates, yielding up to 900 destroy–repair pairs per round. Over 30 rounds, the nominal upper budget is 900 destroy and 27,000 repair candidates. Candidate pairs are evaluated using 100-step LNS rollouts; validation and final evaluation use 500-step rollouts.

A representative TSP discovery run generated 900 destroy and 17,850 repair programs and evaluated approximately 14,500 program pairs. It took approximately 17.3 hours, with rollout evaluation parallelized across 160 workers on Intel Xeon Platinum 8358 CPUs.

### A.4 Role-Specific Credits and Generator Updates

The destroy and repair adapters use independent AdamW optimizers (Loshchilov & Hutter, 2017), each with learning rate $5 \times 10^{-6}$, clipping radius $\epsilon = 0.2$, and maximum gradient norm $1.0$. The optimization micro-batch sizes are 3 for TSP and 2 for CVRP; gradients are accumulated over all usable samples for each role before clipping and a single optimizer step.

The follower best response is approximated using conditionally generated repair programs. For each destroy program $D_i$, the leader credit $C_D(D_i)$ averages the utilities of its top $L = 2$ sampled repair responses, while each repair program receives the conditional follower credit $C_R(R_{j|i}; D_i) = J(D_i, R_{j|i})$.

For both roles, advantages are centered within each group of programs sampled from the same generation context. Samples without a finite credit or usable policy log-probability are omitted from the update. The resulting advantages are optimized using Equation 16.

### A.5 Population Evolution

**Population.** SPO maintains role-specific populations as the memory of executable programs discovered so far. In each round, these populations provide parent context for mutation and crossover, and are then updated by selecting survivors from the union of existing programs and newly generated executable candidates. Keeping a bounded population, rather than only the single best program, preserves alternative executable designs that can seed later generations. We run 30 program-discovery rounds and keep at most 10 executable programs per role.

**Candidate generation and survivor selection.** In the first round, the destroy and repair populations are initialized without parent programs. In subsequent rounds, candidates are generated primarily through mutation and crossover, using the strategies and source code of programs in the current populations as context. If insufficient eligible parents are available, the corresponding generation slots fall back to parent-free generation.

Programs that fail the executable-program gates are excluded before population selection. Destroy programs are deduplicated by whitespace-normalized source code. Repair programs are deduplicated by both normalized source code and the destroy-strategy context used for their generation. Survivor selection first retains the candidate with the highest population-selection score and then greedily selects subsequent candidates according to

$$s(p) + 0.1 \min_{q \in S} d(p, q),$$

where $s(p)$ is the population-selection score, $S$ is the set of already selected survivors, and $d(p, q)$ is the Jaccard distance between the token sets of their strategy descriptions.

**Repair cross-play selection.** Repair population selection follows the cross-play evaluation described in the main text. Destroy survivors are selected first and used to construct a common leader panel. For TSP, this panel contains up to five evolved destroy programs; for CVRP, it uses a hybrid panel with up to two evolved destroy programs and five fixed classical destroy operators. Each repair candidate is evaluated against the same panel, and the resulting common-panel score $C_R^{\text{pop}}$ is used for repair survivor selection. This population score is separate from the destroy-conditional repair credit used for GRPO updates.

## B Problem Settings and Operator Interfaces

This section specifies the problem instances, reference solutions, initial solutions, operator interfaces, and validity requirements used in our experiments.

### B.1 Traveling Salesperson Problem

**Problem definition.** For a Euclidean TSP instance with coordinates $\{c_i\}_{i=1}^n \subset \mathbb{R}^2$, a solution is a permutation $\sigma$ of the $n$ nodes. Defining $\sigma_{n+1} = \sigma_1$, its objective value is

$$f(\sigma) = \sum_{i=1}^{n} \left\| c_{\sigma_i} - c_{\sigma_{i+1}} \right\|_2 . \tag{18}$$

**Data and reference solutions.** Synthetic coordinates are sampled independently and uniformly from $[0, 1)^2$. Discovery uses six 50-node instances, with two stochastic LNS rollouts per instance for candidate evaluation. The held-out synthetic test groups reported in Table 1 contain 32 instances per group. Evaluation additionally uses TSPLIB instances, grouped by instance size as in Table 1. Synthetic reference tour lengths are obtained with Concorde (Applegate et al., 2011), whereas TSPLIB uses the reported optimal tour lengths (Reinelt, 1991). Benchmark distances and reference values are computed using matching edge-weight conventions.

**Initialization and operator interfaces.** Each LNS rollout starts from a seeded random tour. The destroy program has the interface

```
destroy(dist, current_tour, steps_since_improvement, rng)
    -> (partial_tour, removed_nodes)
```

where `dist` is the distance matrix, `current_tour` is the current complete tour, and `rng` is the supplied random-number generator. The returned destruction may remove at most $35\%$ of the nodes.

The repair program has the interface

```
repair(dist, partial_tour, removed_nodes,
       steps_since_improvement, rng)
    -> complete_tour
```

and must return a complete valid tour.

### B.2 Capacitated Vehicle Routing Problem

**Problem definition.** A CVRP instance contains a depot 0, customers $\{1, \ldots, n\}$ with demands $q_i$, and vehicle capacity $Q$. A feasible solution consists of depot-returning routes such that every customer appears in exactly one route and the total demand of every route does not exceed $Q$. The objective is the sum of route lengths, including travel from and back to the depot.

**Data and reference solutions.** For synthetic instances, the depot is located at $(0.5, 0.5)$, customer coordinates are sampled independently and uniformly from $[0, 1)^2$, demands are sampled uniformly from the integers $\{1, \ldots, 9\}$, and $Q = 50$. Discovery uses six 50-customer instances. The synthetic test groups reported in Table 1 use the corresponding held-out instances at each evaluation size.

Synthetic reference costs are produced by one PyVRP run per instance with a fixed random seed and a size-dependent time limit: 10 seconds for $n = 50$, 20 seconds for $n = 100$, 30 seconds for $n = 150$, and 60 seconds for $n = 200$ (Wouda et al., 2024). These values are heuristic references rather than certified optima. Evaluation additionally uses the A, B, P, and X families of CVRPLIB (Augerat et al., 1995; Uchoa et al., 2017), with their corresponding benchmark reference costs. We therefore report reference gaps rather than optimality gaps for all groups.

**Initialization and operator interfaces.** Each LNS rollout starts from a seeded random capacity-feasible solution. Routes contain customer indices only, with depot 0 implicit at both ends. The destroy program has the interface

```
destroy(dist, demands, capacity, current_routes,
        steps_since_improvement, rng)
    -> (partial_routes, removed_customers)
```

and may remove at most $35\%$ of the customers. The repair program has the interface

```
repair(dist, demands, capacity, partial_routes,
```

```
    removed_customers, steps_since_improvement, rng)
-> complete_routes
```

and must return a complete capacity-feasible solution.

## C Baseline Implementations and Evaluation Budgets

We group baselines into handcrafted methods, test-time search methods, and program-discovery methods. For each family, we specify both its test-time procedure and, where applicable, its offline discovery budget.

**Handcrafted baselines.** Nearest Neighbor (NN) and Farthest Insertion (FI) for TSP, and Clarke–Wright savings (CW) for CVRP, each perform a single construction without restarts.

For TSP, 2-opt starts from the NN solution and terminates when no improving move remains, with at most $n^2$ accepted improving moves. The 3-opt baseline starts from the NN+2-opt solution and performs at most 500 additional improving moves. For CVRP, local search starts from the CW solution and applies intra-route 2-opt, relocate, and swap moves for at most 500 improving moves.

**Test-time search baselines.** ALNS (Ropke & Pisinger, 2006) directly searches on each test instance using its own adaptive operator-selection procedure. It is evaluated for 500 destroy–repair iterations per instance and evaluation seed, and we report the best solution encountered during the rollout.

For CVRP, OR-Tools is run directly on each test instance with size-dependent solver search limits selected to approximately match the wall-clock time of a 500-iteration SPO LNS run at the corresponding scale:

| Instance size | Limit | Instance size | Limit |
|---|---|---|---|
| $n \leq 20$ | 1 s | $20 < n \leq 50$ | 2 s |
| $50 < n \leq 100$ | 5 s | $100 < n \leq 150$ | 8 s |
| $150 < n \leq 200$ | 10 s | $200 < n \leq 500$ | 30 s |
| $500 < n \leq 750$ | 60 s | $n > 750$ | 120 s |

The limit applies to solver search rather than total end-to-end execution time.

**Program-discovery baselines.** These methods first obtain destroy–repair programs offline and then evaluate the selected programs for 500 destroy–repair iterations per instance and evaluation seed, reporting the best solution encountered during each rollout.

For G-LNS (Zhao et al., 2026), we directly use the operator programs and implementation released by its authors; we do not retrain the method or assign it a new discovery budget.

EoH (Liu et al., 2024), FunSearch (Romera-Paredes et al., 2024), and ReEvo (Ye et al., 2024) are general automatic program-discovery frameworks. We implement them using the same problem-specific destroy–repair interfaces and LNS runner as SPO. Each method is assigned a nominal budget of 13,950 jointly generated destroy–repair candidates, corresponding to 27,900 operator programs and matching SPO's nominal total of 900 destroy and 27,000 repair generations. Each candidate is evaluated using two 100-step LNS rollouts with different random seeds. Runs are stopped once their discovery trajectories plateau, so this budget is an upper bound rather than the realized candidate count. Final candidates are selected through periodic 500-step validation evaluations, with time limits of 10 seconds for TSP and 30 seconds for CVRP.

## D Program Generation Contexts and Prompts

This appendix describes the serialized generation contexts used by the role-specific LLM generators introduced in Section 4.4. A generation context is not a single fixed prompt. It is assembled from a shared task contract and, depending on the discovery stage, additional population-derived context.

The shared contract specifies the optimization problem, the destroy or repair operator interface, feasibility requirements, and the required output format: a concise strategy statement followed by one executable Python function. We refer to this shared contract as the basic generation prompt. During initialization, the same-role parent context is empty, so the generator samples a new program directly from this basic contract. During evolutionary discovery, the same contract is retained, but the context is augmented with parent strategies and source programs sampled from the current population.

For destroy generation, the context corresponds to $\gamma_D$. For repair generation, the context corresponds to $\gamma_R$ and additionally includes destroy-specific information for the paired destroy operator $D_i$. In the initialization templates below, this destroy-conditioning information is exposed through the placeholder `destroy_strategy`. In evolutionary rounds, the serialized context records whether the paired destroy information and the same-role parent information are supplied as strategy text, source code, or both.

### D.1 Basic Initialization Prompts

The following prompt boxes reproduce the basic initialization templates used for TSP and CVRP destroy and repair generation. These templates contain the complete callable contract and requested program behavior for each operator role. Later mutation and crossover prompts reuse the same role-specific contract, but append the evolutionary context described in Section D.2.

**TSP Destroy Initialization Prompt**

You are an expert in designing adaptive destroy operators for TSP Large Neighborhood Search. Discover a reusable adaptive destroy operator for Large Neighborhood Search on the Traveling Salesman Problem. The operator should create meaningful perturbations that enable effective reconstruction.

Inputs: distance_matrix (distance_matrix[i][j]: the distance from node i to node j), current_solution, steps_since_improvement, and rng.
Output: partial_solution and removed_nodes.

Design one adaptive removal policy that identifies meaningful removal regions from the current tour and uses steps_since_improvement to change concrete removal decisions rather than applying a fixed policy.

Destroy requirements:
– removed_nodes must contain node ids from current_solution
– construct removed_nodes before partial_solution
– partial_solution must preserve the relative order of current_solution
– every node must appear exactly once
– never remove more than 35% of the tour

Scale–aware guidance: derive removal size and candidate scope from the tour size, favor local structural evaluation over exhaustive search, and avoid fixed thresholds tied to a particular problem scale.

Output exactly two parts.
STRATEGY: one concise sentence naming the exact decision the code will implement, not a broader idea.
CODE: exactly one Python function. The function must implement the main mechanism named in STRATEGY.

**TSP Repair Initialization Prompt**

You are an expert in designing adaptive repair operators for TSP Large Neighborhood Search. Discover a reusable adaptive repair operator for Large Neighborhood Search on the Traveling Salesman Problem. The operator should reconstruct a complete tour from the destroy output.

Inputs: distance_matrix (distance_matrix[i][j]: the distance from node i to node j), partial_solution, removed_nodes, steps_since_improvement, and rng.
– partial_solution is shorter than the complete tour; use len(partial_solution) for current insertion positions and len(distance_matrix) only as the total node count.
Output: repaired_tour (the complete tour after reinserting all removed nodes).

Use this paired destroy strategy as context when designing the repair:
{destroy_strategy}

Design one reconstruction policy that complements the destroy result by using the partial solution and removed nodes as reconstruction context. steps_since_improvement may adjust insertion order, candidate scope, or tie handling when useful.

Repair requirements:
– reinsert every removed node exactly once
– preserve the relative order of retained nodes
– use the retained tour only as insertion context
– do not perform additional tour optimization

Scale–aware guidance: derive insertion candidate scope from the current tour size and the number of removed nodes, favor local insertion evaluation over exhaustive search, and avoid fixed thresholds tied to a particular problem scale.

Output exactly two parts.
STRATEGY: one concise sentence naming the exact decision the code will implement, not a broader idea.
CODE: exactly one Python function. The function must implement the main mechanism named in STRATEGY.

**CVRP Destroy Initialization Prompt**

You are an expert in designing adaptive destroy operators for CVRP Large Neighborhood Search. Discover a reusable adaptive destroy operator for Large Neighborhood Search on the Capacitated Vehicle Routing Problem. The operator should remove customers while preserving route order and capacity feasibility of the retained partial routes.

Inputs: distance_matrix, demands, capacity, current_routes, steps_since_improvement, and rng.
– Node 0 is the depot and never appears inside current_routes.
– demands[0] is zero. Every customer node has positive demand.
Output: partial_routes and removed_customers.

Design one adaptive removal policy that selects meaningful customers or route regions and uses steps_since_improvement to change concrete removal decisions rather than applying a fixed policy.

Destroy requirements:
– removed_customers must contain customer ids from current_routes
– construct removed_customers before partial_routes
– partial_routes must preserve the relative order of retained customers within each route
– every customer must appear exactly once across partial_routes plus removed_customers
– never remove depot 0
– never remove more than 35% of customers
– retained partial routes must remain capacity–feasible

Scale–aware guidance: derive removal size and candidate route scope from the route count, customer count, and capacity pressure; favor local route/customer evaluation over exhaustive search; avoid fixed thresholds tied to a particular CVRP size or vehicle count.

Output exactly two parts.
STRATEGY: one concise sentence naming the exact decision the code will implement, not a broader idea.
CODE: exactly one Python function. The function must implement the main mechanism named in STRATEGY.

**CVRP Repair Initialization Prompt**

You are an expert in designing adaptive repair operators for CVRP Large Neighborhood Search. Discover a reusable adaptive repair operator for Large Neighborhood Search on the Capacitated Vehicle Routing Problem. The operator should reconstruct complete feasible routes from the destroy output.

Inputs: distance_matrix, demands, capacity, partial_routes, removed_customers, steps_since_improvement, and rng.
– Node 0 is the depot and never appears inside routes.
– Each returned route must have total demand <= capacity.
Output: repaired_routes.

Use this paired destroy strategy as context when designing the repair:
{destroy_strategy}

Design one reconstruction policy that complements the destroy result by reinserting all removed customers into feasible routes. steps_since_improvement may adjust insertion order, candidate scope, or tie handling when useful.

Repair requirements:
– reinsert every removed customer exactly once
– every original retained customer must remain exactly once
– never include depot 0 inside returned routes
– every returned route load must be <= capacity
– preserve retained route order whenever feasible
– do not perform additional global route optimization after insertion

Scale–aware guidance: derive feasible insertion candidate scope from the current route count, customer count, removed–customer count, and residual capacities; favor bounded capacity–feasible insertion evaluation over exhaustive search; avoid fixed thresholds tied to a particular CVRP size or vehicle count.

Output exactly two parts.
STRATEGY: one concise sentence naming the exact decision the code will implement, not a broader idea.
CODE: exactly one Python function. The function must implement the main mechanism named in STRATEGY.

### D.2 Evolutionary Prompt Forms

During program discovery, SPO maintains separate destroy and repair populations. The first round initializes these populations using prompts without same-role parents. In later rounds, new candidates are generated from the current population through mutation or crossover. These evolutionary prompts are constructed by augmenting the basic destroy or repair prompt with population-derived parent context, while keeping the same operator interface, feasibility requirements, and output format.

For mutation, the prompt includes one parent strategy and source program, together with an instruction describing the intended type of modification. For crossover, the prompt includes two parent strategies and source programs, and asks the generator to produce a new program using the parents as design context. The resulting candidates are then evaluated and used by the training and population-update procedure described in Appendix A.

Table 3 summarizes the evolutionary prompt forms. The table describes the instruction type rather than reproducing the full serialized prompt text.

Table 3: Evolutionary prompt forms used during program discovery.

| Prompt form | Parents | Instruction |
|---|---|---|
| Mechanism replacement | One | Replace one major decision mechanism while preserving the remainder of the parent design. |
| State-dependent control redesign | One | Preserve the main mechanism but alter thresholds, budgets, candidate scopes, or adaptation schedules. |
| Simplification | One | Remove redundant or unnecessarily complex components without introducing a new major mechanism. |
| Divergent crossover | Two | Use the parents as records of explored ideas and generate a substantially different heuristic. |
| Shared-principle crossover | Two | Extract a shared high-level principle and implement it through a new program structure. |

## E Discovered Operator Programs

We summarize two destroy–repair pairs discovered for each problem. These examples illustrate different ways of adapting the destruction scale or sampling strategy and prioritizing difficult reconstruction decisions. TSP-1 and CVRP-1 are used for the main evaluation, while TSP-2 and CVRP-2 were obtained from separate discovery runs. Their complete executable programs and identifiers are available in the anonymous repository's `discovered_programs/` directory.

**TSP-1: adaptive removal size and regret-based repair.** The destroy program increases its removal size with stagnation, selects a random anchor, and ranks the remaining positions using a local edge-disagreement score. The repair program recomputes the two best insertion costs for each removed node, samples from the highest-regret candidates, and adjusts the selected insertion position using local distances.

**TSP-2: stagnation-triggered sampling.** The destroy program constructs a candidate window around a selected tour edge and changes its sampling rule after 20 steps without improvement, removing at most $35\%$ of the tour. The repair program uses a density-dependent local window and insertion-cost margins to order removed nodes, followed by lowest-cost insertion.

**CVRP-1: local-to-global removal and capacity-aware repair.** The destroy program switches from route-segment removal to global customer sampling after five steps without improvement, subject to a $35\%$ removal cap. The repair program orders customers by opportunity cost and selects feasible insertions using a capacity-pressure penalty whose weight increases with stagnation.

**CVRP-2: randomized removal and adaptive regret repair.** The destroy program forms a customer candidate list from route residual capacities, shuffles the candidates, and removes up to $35\%$ of them. The repair program orders customers by insertion regret, adds the range of feasible route costs when the stagnation count is positive, and applies cheapest feasible insertion.

## F LNS-State Intervention Details

Figure 2(b1) evaluates a fixed discovered TSP operator pair while modifying only the LNS-state signal supplied during execution. The interventions are defined as follows:

- **Fixed $k$:** set $z_t = k$ throughout the rollout, with $k \in \{0, 5, 10\}$;
- **Shuffled:** randomly permute the state sequence observed in the corresponding unmodified rollout;
- **Inverted:** set $z_t = 30$ when the observed state is below 10, and set $z_t = 0$ otherwise.

For each instance–seed pair $u$ and intervention $a$, we compute the paired change in the final best reference gap,

$$\Delta^u(a) = g^u_{(a)} - g^u_{(\text{base})}, \tag{19}$$

where $g^u_{(\text{base})}$ denotes the result obtained using the unmodified state signal. Figure 2(b1) reports the mean paired difference and its $95\%$ confidence interval, computed from 2,000 bootstrap resamples of the instance–seed pairs.