# GEPA-TSP: Specializing Lin–Kernighan Heuristics to Target Instance Distributions

**Reuben Narad** [1] **Natasha Jaques** [2]

## Abstract

Large language models (LLMs) with reflective evolution have recently been used as search procedures over programs and prompts, offering a new way to adapt existing code to specific workloads. We study this idea in an operations research setting, tuning Concorde (a state-of-the-art exact TSP solver) by evolving the details of its pre-packaged Lin-Kernighan (LK) heuristic. Using GEPA, a reflective prompt evolution algorithm with separate actor and reflector LLMs, we build a sandbox that regenerates and benchmarks candidate heuristics. We target three 400-node TSP distributions of increasing structural complexity: uniform Euclidean, clustered Euclidean with in-cluster discounts, and a road-network derived from Seattle's map. Across many runs, we observe a single robust win on Seattle instances, where GEPA found a change that improves solver time by 5%. On other distributions, the same modification consistently regresses performance. Our results frame GEPA-style LLM search as a distribution-specific tuning method for existing heuristics, and highlight both the possibility and difficulty of beating carefully engineered baselines.

## 1. Introduction

Automated scientific discovery is an emerging area of interest, aiming to accelerate scientific research by automating parts of the work of an investigator. Instead of testing single experiments or code variations at a time, LLMs can run large batches of these tasks: testing many hypotheses, trying many variants of a codebase, or exploring many proof attempts in a loop. Many of these problems can be viewed as search over a large space of hypotheses, programs, or algorithmic components.

In such settings, the searching agent must keep past rollouts in context. If the agent does not know which experiments or variants worked previously, it will search in an inefficient, unguided way that ignores available feedback, making progress in a huge space effectively intractable. At the same time, fully remembering every experiment is impossible: the search space is too large to keep all history in context at once. This tension naturally motivates mechanisms that maintain a compact but useful long-term memory of past attempts.

Prompt learning and evolution provide one such mechanism. In approaches such as the DSPy framework (**?**), the context of an agent as a major component of its performance, treating it as a learned, mutable object. Using a second "reflector" LLM, that context can be optimized to a given reward by being evolved over time based on the results of its earlier attempts. The Genetic Pareto (GEPA) (**?**) algorithm decides which examples to keep in the reflector's context by maintaining a Pareto frontier of past candidates. Similar prompt-evolution ideas have begun to appear in automated discovery systems more broadly, including work like AlphaEvolve, where long-lived records of past trials guide future exploration.

Heuristics in operations research (OR) are a natural testbed for this paradigm. We study the traveling salesman problem (TSP) by taking Concorde and focusing on its Lin–Kernighan (LK) heuristic, which we allow an LLM to modify. We view the search process as testing different configurations or versions of this LK component. Concretely, we frame this as a task within the GEPA framework: a student model proposes rewrites of the LK heuristic, we recompile Concorde, run it on a benchmark of TSP instances solving to optimality, and use its runtime (relative to vanilla Concorde) and number of branch-and-bound search nodes as a reward signal to decide which examples to keep in the reflector model's context. The reflector model then uses its in-context examples (actor prompts, the resulting heuristic code, and their measured performance) to propose changes to the actor's prompt.

[1]University of Washington, Foster School of Business [2]University of Washington, Paul G Allen School of Computer Science and Engineering. Correspondence to: Reuben Narad <rnarad@uw.edu>.

*Proceedings of the $42^{nd}$ International Conference on Machine Learning*, Vancouver, Canada. PMLR 267, 2025. 

Crucially, GEPA optimizes the LK heuristic only with respect to a fixed benchmark of validation instances, rather than the space of all TSPs. Viewed this way, the reflective loop acts as a distribution-specific tuning mechanism: the LLM adapts the heuristic to the particular instance distribution on which its reward is evaluated, instead of searching for a universally superior variant. This framing makes it natural to compare how tuned heuristics behave across contrasting TSP distributions and to study when distribution-specific gains fail to transfer.

Our contributions are threefold. First, we present a reproducible sandbox for LK block injection in Concorde with full artifact logging, designed to provide the reflector with rich feedback for rewriting the actor's prompt. Second, we curate a benchmark of non-Euclidean and Euclidean TSP instance distributions that expose differences between highly tuned baselines and under-optimized regimes. Third, we provide empirical evidence that a reflective LLM loop can discover a small improvement to Concorde on a structured travel-time distribution, while failing to improve and often degrading performance on classical Euclidean benchmarks. We release code, data, and all candidate LK variants to support further work on OR-driven automated science.

## 2. Related Work

### 2.1. Learning to Optimize Solvers

A growing line of research uses machine learning to learn heuristics or policies for combinatorial optimization problems. Neural combinatorial optimization approaches train sequence models with reinforcement learning to construct tours or routes directly (Bello et al., 2017; Kool et al., 2019), while other work learns to imitate or improve classical heuristics such as 2-opt via policy gradients or deep RL (Deudon et al., 2018; d. O. da Costa et al., 2020). Complementary work studies solver portfolios for the TSP, using machine learning to exploit complementarities between different solvers or configurations (Kerschke et al., 2018). Our setting follows the same spirit of learning to optimize solvers, but instead of training a new construction heuristic, we use an LLM to mutate the code of an existing, hand-engineered heuristic (Concorde's LK) and specialize it to particular instance distributions.

### 2.2. LLM-Guided Code Evolution and Algorithm Configuration

Recent work has begun to treat LLMs as hyper-heuristics that propose and refine algorithmic code. ReEvo frames LLMs as reflective hyper-heuristics that iteratively edit and evaluate algorithm implementations, using feedback to guide further changes (Ye et al., 2024). GEPA shows that reflective prompt evolution, driven by a Pareto frontier over past candidates, can match or outperform reinforcement learning on several LM-controlled tasks (Agrawal et al., 2025), while DSPy treats prompt and pipeline configurations as learned objects that can be optimized given a reward signal (Khattab et al., 2023). These ideas connect naturally to the longer-standing literature on automatic algorithm configuration and selection, where systems such as ParamILS tune discrete algorithm parameters (Hutter et al., 2009) and algorithm selection methods choose among a portfolio based on instance features (Kerschke et al., 2019). Our GEPA loop fits within this broader landscape: it mutates and tests LK code inside a sandboxed, deterministic TSP pipeline, effectively performing distribution-specific algorithm configuration at the level of heuristic implementation rather than scalar parameters.

## 3. Method: GEPA for Lin–Kernighan

### 3.1. Background: How Concorde Works

Concorde is an exact branch-and-bound solver that is heavily optimized for the TSP. It starts from the standard integer programming formulation of the TSP and relaxes it to a linear program. Because the LP is convex, it can be solved quickly. If the optimal LP solution happens to be integral, it is also an optimal solution to the original integer program.

When the LP solution is fractional, Concorde selects an edge whose LP value lies strictly between 0 and 1 and branches on it. One child node adds a constraint forcing this edge to be in the tour, and the other child node adds a constraint forcing it to be excluded. Recursively solving these subproblems yields a branch-and-bound tree. Alongside this tree search, Concorde runs a heuristic to maintain a best-known tour. Any branch whose LP relaxation cannot beat the heuristic tour length is pruned. The stronger this heuristic baseline, the more aggressively Concorde can prune, and the faster it converges to the optimal tour. Concorde ships with a carefully engineered Lin–Kernighan (LK) heuristic to provide this baseline.

### 3.2. Turning the Heuristic into a GEPA Task

In Concorde's codebase, the LK heuristic is implemented as a well-defined C file, which we treat as a modular component. We build a workflow that replaces this LK code with a new heuristic block, recompiles Concorde, and then solves a fixed validation set of TSP instances. For each run, we log Concorde's textual output, the total wall-clock time to solve the validation set, and the number of branch-and-bound nodes explored during the search. In GEPA, we use the total time and the node count as rewards.

For the purposes of GEPA, we view this reward as a function of the actor's prompt. Let $\mathcal{P}$ denote the space of actor

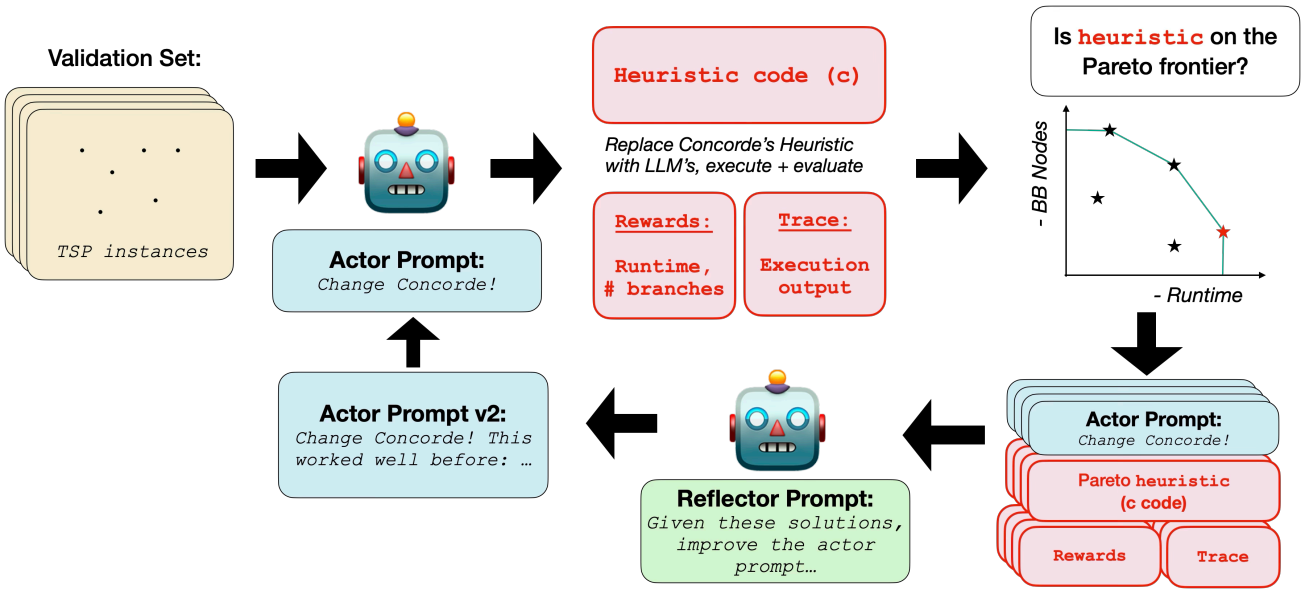


*Figure 1.* GEPA pipeline for modifying Concorde's LK heuristic.

prompts and $\mathcal{H}$ the space of heuristic code blocks. The actor LLM is a (stochastic) mapping

$$A : \mathcal{P} \to \mathcal{H}, \qquad h = A(p),$$

which takes a prompt $p \in \mathcal{P}$ and produces a candidate heuristic $h \in \mathcal{H}$ (the LK replacement). Given a fixed validation set $D_{\text{val}}$ of TSP instances, the Concorde pipeline defines an evaluation map

$$C : \mathcal{H} \times \mathcal{D} \to \mathbb{R}^2 \times \mathcal{T}, \qquad C(h, D_{\text{val}}) = (t, n, \tau),$$

where $t$ is the total runtime on $D_{\text{val}}$, $n$ is the total number of branch-and-bound nodes, and $\tau \in \mathcal{T}$ denotes the emitted trace (logs and solver output). The overall black-box objective that GEPA interacts with is the composition

$$F : \mathcal{P} \to \mathbb{R}^2 \times \mathcal{T}, \qquad F(p) = C(A(p), D_{\text{val}}) = (t, n, \tau).$$

We are interested in minimizing both $t$ and $n$ jointly: a candidate $(t', n')$ is preferred to $(t, n)$ when it Pareto-dominates it, i.e., $t' \leq t$ and $n' \leq n$ with at least one strict inequality. In practice, GEPA maintains a set of non-dominated examples in this two-dimensional objective space and uses them as in-context training data for the reflector. The actor LLM, code generation, compilation, and Concorde evaluation are internal details of this composite map; GEPA only sees prompts, their resulting $(t, n)$ pairs, and the logged artifacts used for reflection.

### 3.3. GEPA Loop Mechanics

We now summarize how GEPA operates in this setting. The state of the system at iteration $k$ consists of the current actor prompt $p_k \in \mathcal{P}$ and a set of evaluated candidates

$$\mathcal{S}_k = \{(p_i, h_i, t_i, n_i, \tau_i)\}_{i=1}^{m_k},$$

where $h_i = A(p_i)$ is the heuristic produced from prompt $p_i$, and $(t_i, n_i, \tau_i) = C(h_i, D_{\text{val}})$ are its measured runtime, branch-and-bound nodes, and trace on the validation set.

At each iteration, the student (actor) model generates one or more candidate heuristics from the current prompt $p_k$. Concretely, we sample a heuristic $h_k = A(p_k)$, inject it in place of Concorde's LK block, recompile the solver, and run it on the 20-instance validation set with three independent repeats. This yields averaged metrics $(t_k, n_k)$ and a trace $\tau_k$, which we add to the candidate set $\mathcal{S}_k$.

GEPA then updates the Pareto frontier in the two-dimensional objective space of (mean time, mean branch-and-bound nodes). Any candidate in $\mathcal{S}_k$ that is Pareto-dominated by another is discarded from the frontier, and candidates that newly lie on the frontier are marked as elites. The reflector model receives a batch of 2–3 examples drawn from these elites, including their prompts, heuristic code, and measured performance, and proposes 2–3 alternative rewrites of the actor prompt. One of these reflector proposals is selected (e.g., uniformly at random) to form the next actor prompt $p_{k+1}$, and the loop continues until the fixed budget of 60 validation evaluations is exhausted.

Over the course of this process, the Pareto frontier maintains the current non-dominated submissions between runtime and search effort, while the actor prompt evolves to

better explain and extend the behaviors of successful candidates. When training finishes, we select a single final candidate from $\mathcal{S}_k$ on the validation set and evaluate its heuristic $h$ on the 200-instance test set.

## 4. Experimental Setup

### 4.1. Benchmarks and Data

Concorde is very fast on small instances. For problems up to about 200 nodes, solution times were typically less than one second and often involved exploring a single branch-and-bound node, so measurement noise dominated any differences between heuristics. To obtain more informative signals, we focused on instances with $n = 400$ nodes, verifying that Concorde typically explored 4–6 branch-and-bound nodes during the solve.

For our experiments, we defined three TSP instance families of increasing structural complexity, each with a generator for sampling new instances:

- **Uniform.** $n = 400$ i.i.d. points in the unit square, with distances given by 2D Euclidian distance.
- **Clustered.** $k = 4$ anisotropic Gaussian clusters in the unit square, with cluster centers sampled uniformly and covariance eigenvalues drawn from $\mathrm{U}[0.001, 0.02]$ with random rotations. Distances are 2D Euclidian, with a $0.5\times$ intra-cluster discount when using explicit distance matrices.
- **Seattle.** A 400-node road-network instance derived from the largest connected component of the Seattle OSM driving graph, restricted to latitude $47.58$–$47.64$ and longitude $-122.36$–$-122.30$. Nodes are sampled uniformly from this component, and edge weights are shortest-path travel times obtained by running Dijkstra's algorithm between all pairs of the 400 nodes.

For each instance type, we generated a validation set of 20 instances and a test set of 200 instances. Because each 400-node instance takes several seconds for Concorde to solve, using a larger validation set would have made each GEPA iteration prohibitively slow.

### 4.2. GEPA Details

For a given GEPA candidate heuristic, we evaluate performance on the 20 validation instances and compute the mean solve time, averaging over three independent reruns to reduce noise. In addition to solve time, we record the total number of branch-and-bound nodes explored. A prompt is added to the candidate set if the heuristic produced from that prompt lies on the Pareto frontier in the two-dimensional space of (mean time, branch-and-bound nodes).

In all experiments, the student model is `gpt-5-nano` and the reflector is `gpt-5-mini`. At each GEPA iteration, the reflector generates 2–3 alternative rewrites of the actor prompt (reflection batch size 2–3), and each candidate heuristic is evaluated with 20 metric calls. As a baseline, we rebuild Concorde with its default LK heuristic in the same sandbox and evaluate it under the same protocol, using a higher repeat count (5 instead of 3) to obtain a more stable reference estimate.

## 5. Results: TSP Adaptation

We ran GEPA on each of the three TSP distributions with a fixed validation budget of 60 evaluations. While each experiment has the same total budget (counting validations of the baseline, new candidates, and re-evaluations of existing Pareto-frontier candidates), the number of GEPA iterations differs across distributions. This is because the size of the Pareto frontier varies, given the stochastic prompt-to-heuristic mapping, different subsets of frontier candidates are re-evaluated at each step. Figure 3 shows the resulting validation performance of the proposed candidates over the course of GEPA.

After running GEPA, we select the overall best candidate heuristic produced on the validation set, and evaluate it on the 200-instance test sets for each distribution. For each family, we report mean runtime and mean branch-and-bound (BB) nodes for vanilla Concorde and for the best GEPA candidate. Results are summarized in Table 1.

On the uniform and clustered Euclidean benchmarks, the best GEPA heuristics are clearly slower than Concorde's default LK: runtimes increase by 7.1% and 3.1%, and BB nodes by 3.1% and 13.0%, respectively on their held-out test sets. In contrast, on the Seattle travel-time distribution, GEPA found a heuristic in iteration 30 that reduces mean runtime by 4.8% and BB nodes by 3.5% on the held-out test set. While not definitive, this pattern matches the intuition that Concorde's built-in LK is already well tuned for standard Euclidean TSPs, while there is more headroom for distribution-specific tuning on the structure of the non-Euclidean road-network instances.

## 6. Discussion

### 6.1. GEPA Overfitting

Across runs, GEPA consistently produced candidate heuristics that outperformed the baseline on the validation set, but we only observe a clear improvement on the test set for the Seattle distribution. Inspecting the reflector's edits to the actor's prompt, most suggestions take the form

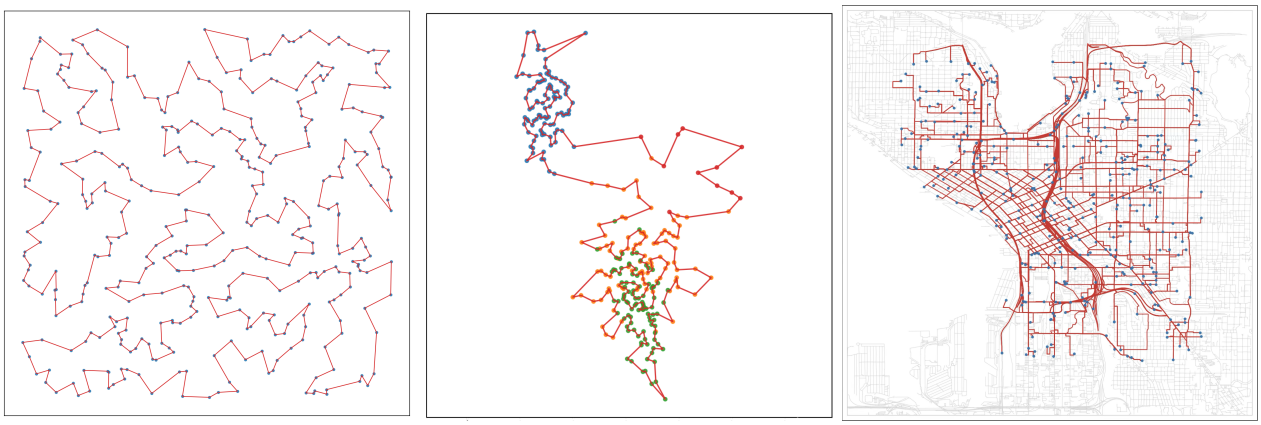

*Figure 2.* Example TSP instances for the three benchmarks: uniform Euclidean (left), clustered Euclidean (middle), and Seattle road network with travel-time distances (right).

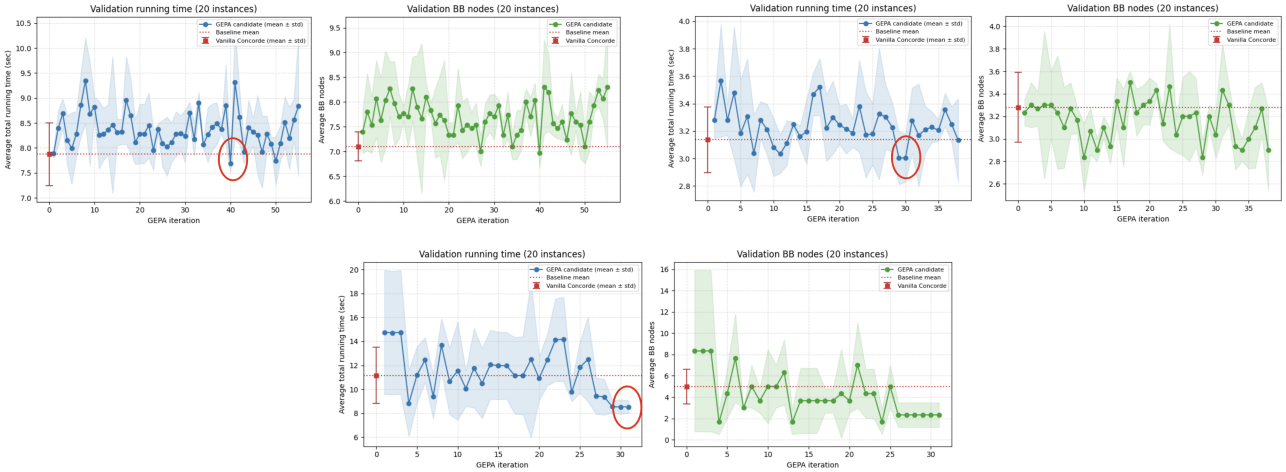


*Figure 3.* GEPA training performance over validation set on the three TSP benchmarks, under a fixed budget of 60 validation evaluations per distribution. Each panel shows average runtime (left/blue) and average branch-and-bound (BB) nodes (right/green) on 20 validation instances, with bands indicating standard deviation and the vanilla Concorde baseline shown in red. Top left: uniform Euclidean; top right: clustered Euclidean; bottom: Seattle travel-time. The GEPA iteration circled in red in each panel is the final candidate we select and evaluate on the 200-instance test set in Table 1.

of small, specific tweaks to the heuristic implementation (for example, changing how flushing is performed or how tours are flipped; see Appendix B for concrete traces and the Seattle-winning heuristic). The natural interpretation is that these modifications overfit to the validation set: they are not general enough to improve performance overall and tend to fail to transfer beyond the specific validation instances.

We also acknowledge that this may reflect limitations of our current GEPA setup. Constructing a GEPA pipeline involves many design choices: the instructions and context given to the reflector, the choice of LLMs, the rewards used to define the Pareto frontier, the size and management of the candidate set, and so on. With more attention to overfitting, for example through larger validation sets or more specific reflection instructions, it is plausible that these effects could be reduced. More broadly, this highlights that prompt evolution frameworks inherit many of the same basic issues as traditional machine learning, including sensitivity to validation protocols and the risk of over-optimistic estimates when tuning against small validation sets.

### 6.2. Why not monotonically increasing rewards?

In many evolutionary and genetic algorithm settings, progress curves are monotone by construction: new candidates that are worse than the current best are simply rejected, and plots typically show the best-so-far reward. Even if many offspring are worse than their parents, the reported curve never decreases.

*Table 1.* Average runtime and branch-and-bound (BB) nodes on the 200-instance test sets. GEPA best denotes the single best heuristic found by GEPA for each distribution. Relative differences are computed as (GEPA − baseline) / baseline.

| Distribution | Vanilla Concorde (time / BB) | GEPA best (time / BB) | $\Delta t$ | $\Delta$ BB |
|---|---|---|---|---|
| Uniform | 4.223 s / 3.84 | 4.521 s / 3.96 | +7.1% | +3.1% |
| Clustered | 4.405 s / 4.60 | 4.543 s / 5.20 | +3.1% | +13.0% |
| Seattle | 3.958 s / 3.68 | 3.768 s / 3.55 | −4.8% | −3.5% |

In our case, rewards are not guaranteed to increase because the mapping from a prompt $p$ to a heuristic $h = A(p)$ is stochastic, due to randomness in the LLM. Re-running the same prompt resamples $A(p)$ and thus its evaluation $F(p)$, so when we periodically reevaluate the candidate set, some prompts can obtain lower rewards than before and the plotted frontier reward can go down. Importantly, when we report test performance we evaluate the final winning *heuristic*, not the prompt, using a fixed sampled implementation.

### 6.3. Practical difficulty with Speed-Based Heuristics as the Reward

Because our objective is based on speed to solve, we are naturally constrained to instances that are small enough for Concorde to solve thousands of times during training. This pushes us toward relatively modest problem sizes, even though the most impactful use cases for improved heuristics are precisely larger instances where solving is harder and heuristics matter more. A similar issue appears in settings like ReEvo (Ye et al., 2024), where the reward is defined in terms of suboptimality: computing suboptimality still requires access to the optimal solution, which again becomes expensive on harder instances.

This creates a basic tension in heuristic discovery. On the one hand, we want artifacts that are useful on hard instances, where runtime and search effort dominate. On the other hand, training and evaluation are easiest on smaller instances, where repeated solves are tractable and noise can be averaged out. Navigating this trade-off is central to making speed-based heuristic learning practical for large-scale problems.

### 6.4. Future Work

The current GEPA setup only allows the LLM to modify a single LK heuristic code block, which defines a relatively narrow action space and naturally limits the scale of possible improvements to small operational speedups. A natural next step is to widen this action space across several files, giving the model the ability to make more substantial changes to Concorde's algorithmic structure rather than just local tweaks; in that regime, we might expect to see more dramatic wins.

To sharpen the view of GEPA as a distribution-specific tuning mechanism, it would also be interesting to deliberately engineer TSP instance families that are adversarial to Concorde's default LK heuristic and test whether GEPA can adapt the heuristic to these harder, targeted distributions. This would more directly probe how far the approach can stretch when the baseline is misaligned with the instance geometry.

Another direction is to try more advanced LLM setups. In particular, reasoning-oriented models may be better at turning time, branch-and-bound, and trace feedback into actionable code changes, whereas a small model like `gpt-5-nano` is unlikely to discover major algorithmic improvements from such signals.

Finally, the initial actor prompt is itself a major design choice, and its effect on the outcome of the search remains underexplored. Further analysis of different initializations, especially prompts that make the actor more or less rigid to reflector edits, could clarify how much of GEPA's behavior is driven by the starting prompt versus the subsequent reflective loop.

## 7. Conclusion

We studied how reflective prompt evolution can tune a classical operations research solver by using GEPA to modify Concorde's embedded Lin–Kernighan heuristic. Our setup treats the actor prompt as the object of optimization, with a sandboxed Concorde pipeline providing two-dimensional feedback in terms of runtime and branch-and-bound nodes on a fixed validation set. Empirically, we find that this reflective loop fails to improve Concorde on standard Euclidean benchmarks, where the built-in LK implementation appears already highly tuned, but discovers a modest speedup on a structured Seattle travel-time distribution. These results support the view of GEPA-style LLM search as a distribution-specific autotuning mechanism for existing heuristics, while also highlighting practical challenges around overfitting and the cost of using speed-based rewards. We release code, data, and all candidate LK variants to facilitate further work on applying prompt-evolution methods to OR-style automated science.

# A. Full Prompts

## A.1. Intial Student Prompt (code-authoring model, evolved artifact)

```
You are rewriting the Lin-Kernighan heuristic block
in Concorde's linkern.c.

Scope
- Restructure only between:
      /* BEGIN LLM HEURISTIC BLOCK */
          ... your code here ...
      /* END LLM HEURISTIC BLOCK */
- You may reorder control flow, change how win/fstack/
  win_cycle are managed, or adjust improve_tour calls.
  Do not touch code outside the markers or change the
  function signature.
- First line inside the block must be a brief C comment
  stating the heuristic idea (e.g., /* plan: batch flips
  before flushing */). This plan is required.

Contract (must preserve)
- Core loop: pop a start, call improve_tour, accumulate
  totalwin, update win/fstack/win_cycle, then subtract
  totalwin from *val.
- Keep helper usage legal: pop_from_active_queue,
  improve_tour, CClinkern_flipper_cycle, MARK, etc.
  No new headers or globals.
- Keep the tour valid; do not skip feasibility checks.
  Always do (*val) -= totalwin; keep structures consistent.
- ANSI C89 only. No dynamic allocation or I/O.

Goals (optimization target)
- Primary: reduce average wall-clock time on
  structured_seattle_time (travel-time weights, n~400)
  with zero failures/timeouts. BB nodes are helpful but
  secondary.
- Prefer robust, repeatable speedups over brittle spikes.
- Make substantive changes (queue policy, batching/flush,
  cycle handling, acceptance rules, flip storage). Cosmetic
  or baseline-equivalent edits are unacceptable.

Safe innovation hints
- Batch or reorder flushes of fstack into win; keep correct.
- Avoid redundant improve_tour calls; skip passes with no gain.
- Adjust pop/processing rules (limited extra work per start,
  early exit on small cumulative gain) without breaking
  correctness.
- Avoid risky hacks: no file names, I/O, randomness, node
  counts; do not remove validity checks.

Output
Return only the replacement block (markers included).
First non-marker line must be the plan comment. No extra
explanations outside the block.
```

## A.2. Reflector Prompt (diagnostics + guidance model)

```
You are the reflection model for Concorde's GEPA loop.
Each round you see:
- The candidate code block (BEGIN/END markers) from the
  student.
- Evaluation feedback: build logs, metrics (wall time,
  branch-and-bound nodes), stderr tail.
```

```
Your job: give actionable guidance so the next candidate
is faster and remains correct.

Priorities
- Target split: structured_seattle_time (travel-time
  weights, n~400). Minimize wall time with zero failures/
  timeouts. BB nodes are secondary.
- Reject flaky wins: any failures/timeouts are a major
  issue. Prefer consistent speedups.
- Encourage meaningful changes (queue policy, batching/
  flush rules, cycle handling, acceptance rules). Call
  out cosmetic or baseline-equivalent submissions.

How to respond
1) Diagnose issues
   - Note crashes/timeouts/missing metrics; penalize them.
   - Call out regressions with specifics (e.g., wall time
     from X->Y, BB nodes jumps).
   - Flag when the code is effectively baseline/cosmetic.
2) Propose improvements
   - Structural levers: pop/queue tweaks, cycle handling,
     early exits on tiny gains, limit work per start, guard
     expensive work when no gain is likely.
   - Avoid repeating recent batching/flush-only tweaks
     (e.g., simple fstack->win flushing); propose a
     different lever if batching was just tried.
   - Aim for reproducible speedups on this split without
     breaking correctness.
3) Preserve the contract
   - Keep win, fstack, win_cycle consistent; maintain
     (*val) -= totalwin; stay within markers; ANSI C rules.

Output format
- Issues: bullets with evidence (metrics/logs), including
  "no substantive change" if applicable.
- Recommendations: bullets with concrete code changes to
  try next.
- Stretch ideas (optional): further experiments.

Be concise but specific. Focus on safe, structural speedups.
```

## B. Reflection Traces and Seattle-Winning Heuristic

We will include the reflection logs, sampled actor/reflector exchanges, and the full LK block for the Seattle-winning heuristic. These artifacts live alongside the evaluation runs in the project repository and will be inserted here in the next revision once the synced run directory is available.

## References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025. URL https://arxiv.org/abs/2507.19457.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. In *International Conference on Learning Representations*, 2017. URL https://arxiv.org/abs/1611.09940.

d. O. da Costa, P. R., Rhuggenaath, J., Zhang, Y., and Akcay, A. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. In *Proceedings of The 12th Asian Conference on Machine Learning*, volume 129 of *Proceedings of Machine Learning Research*, pp. 465–480. PMLR, 2020. URL https://proceedings.mlr.press/v129/costa20a.html.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., and Rousseau, L. Learning heuristics for the TSP by policy gradient. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2018)*, volume 10848 of *Lecture Notes in Computer Science*, pp. 170–181. Springer, 2018. doi: 10.1007/978-3-319-93031-2_12.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009. doi: 10.1613/jair.2861.

Kerschke, P., Kotthoff, L., Bossek, J., Hoos, H. H., and Trautmann, H. Leveraging TSP solver complementarity through machine learning. *Evolutionary Computation*, 26(4):597–620, 2018. doi: 10.1162/evco_a_00215.

Kerschke, P., Hoos, H. H., Neumann, F., and Trautmann, H. Automated algorithm selection: Survey and perspectives. *Evolutionary Computation*, 27(1):3–45, 2019. doi: 10.1162/evco_a_00242.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. DSPy: Compiling declarative language model calls into self-improving pipelines. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2310.03714.

Kool, W., van Hoof, H., and Welling, M. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1803.08475.

Ye, H., Wang, J., Cao, Z., Berto, F., Hua, C., Kim, H., Park, J., and Song, G. Reevo: Large language models as hyper-heuristics with reflective evolution. *arXiv preprint arXiv:2402.01145*, feb 2024. URL https://arxiv.org/abs/2402.01145. NeurIPS 2024.