

Preserving Optimization Algorithm Expertise by means of Executable Algorithm Knowledge Graphs: A Worked Example on the TSP

Camilo Chacón Sartori, José H. García, Andrei V. Tomut
0000-0002-8543-9893, 0000-0002-5752-4759, 0000-0001-8288-4054
Catalan Institute of Nanoscience and Nanotechnology (ICN2),
CSIC and BIST
Bellaterra, Barcelona, Spain
Email: camilo.chacon@icn2.cat

Christian Blum
0000-0002-1736-3559
Artificial Intelligence Research Institute (IIIA-CSIC)
Bellaterra, Spain
Email: christian.blum@iiia.csic.es

Abstract—Procedural knowledge and expertise in algorithm design are usually hidden in source code and reproduced for each new optimization problem. In this work, we deal with the important question of how to store and encode this expertise in a reusable way. This is done by so-called Generative Executable Algorithm Knowledge Graphs (GEAKGs), which store expertise as a typed, executable graph: nodes are algorithmic roles that hold validated operators, edges encode admissible compositions, learned pheromone weights record effective sequences, and a deterministic Symbolic Executor produces solutions by traversal with no runtime language-model calls. This paper is a worked example. We construct a GEAKG for the Traveling Salesman Problem step by step—deriving an eleven-role schema, generating and validating executable operators, assembling an ontology-constrained topology, and learning edge weights with the MAX-MIN Ant System—and then run it. The frozen snapshot returns feasible solutions across three model scales, including a fully local model, and reaches single-digit-percent optimality gaps at zero deployment tokens. A controlled check confirms that the ontology constraint lowers the search gap by concentrating it on valid transitions. The cross-domain transfer study is deferred to the companion full version; the goal here is a reproducible recipe for building and executing this representation on a single domain.

Index Terms—executable knowledge graphs, procedural knowledge, ant colony optimization, automatic algorithm design, traveling salesman problem

I. INTRODUCTION

PROCEDURAL knowledge in algorithm design—which operator to apply, in which order, under which conditions—tends to live in two places: inside source code, and inside the transient search traces of a solver. Both are hard to reuse. When a new problem arrives, the solution construction, local search, and perturbation logic is typically rewritten, and the record of which sequences worked is discarded with the run.

Generative Executable Algorithm Knowledge Graphs (GEAKGs) [1] represent this knowledge as a graph that can be stored, inspected, and executed. Typed nodes are algorithmic roles that hold validated executable operators; edges encode the compositions an ontology admits; a weight function

records, per edge, how effective a transition has been; and a set of symbolic rules conditions execution on runtime state. A deterministic engine, the Symbolic Executor, turns the graph into solutions by traversing it—selecting the next role from the learned transition weights, applying a bound operator, and accepting the result on improvement—without any language-model call at deployment.

a) Scope of this paper: The first paper in which GEAKGs were introduced (see [1]) studies in detail *how and if* GEAKGs transfer across domains and *when* reuse pays off. Here we do something narrower and complementary: we give a single-domain, end-to-end *worked example*. Using the Traveling Salesman Problem (TSP) as the running case, we show how a GEAKG is built and run, with enough detail to reproduce the pipeline. Cross-domain transfer is out of scope by design, and we point to [1] for this issue.

b) Contributions:

- A step-by-step construction of a GEAKG for the TSP: deriving the role schema from problem structure, generating and validating executable operators, assembling the ontology-constrained topology, and learning edge weights with ant colony optimization (ACO); see Section IV.
- A description of the Symbolic Executor and the lifecycle separation—offline acquisition and refinement, online execution at zero language-model tokens; see Section V.
- A single-domain empirical account: feasibility across three model scales, optimality gaps on TSPLIB instances, and a controlled ontology-constraint ablation; see Section VI.

The paper unfolds as follows. Section II positions the paper within the scientific literature; Section III states the model; Section IV builds the four layers for the TSP in turn; Section V describes the executor; and Section VI reports the single-domain results together with the inspectable snapshot.

II. BACKGROUND

Three topics place GEAKG in context: knowledge graphs as a representation, ant colony optimization as its offline

learning mechanism, and the recent use of language models as algorithmic operators. We take each in turn and then explain the relation of GEAKGs with each of them.

a) *Executable procedural knowledge graphs*: Classical knowledge graphs store *facts*: entities and the relations among them, queried for retrieval and inference [2]. Instead, a GEAKG stores *procedure*. Its nodes are not static entities but roles bound to code, and traversing the graph runs that code to produce a solution. The graph is therefore generative—it yields objects when executed—while keeping the inspectability of a graph representation: roles, admissible edges, weights, and rules can each be examined on their own.

b) *Ant colony optimization*: Edge weights are learned with the MAX-MIN Ant System (MMAS), a variant of ant colony optimization in which pheromone values are bounded to a fixed interval $[\tau_{\min}, \tau_{\max}]$ and only high-quality solutions reinforce edges [3], [4]. The bounded dynamics keep the learned distribution from collapsing onto a single path, which suits a representation meant to be reused. In a GEAKG, MMAS runs offline over a fixed topology and produces the weight function that the executor later reads.

c) *Language-model operators and automatic algorithm design*: A line of recent systems uses large language models to generate algorithmic components: LLaMEA evolves metaheuristics as code [5], FunSearch searches over programs against a scorer [6], and EoH and ReEvo evolve heuristics with language-model variation [7], [8]. In these systems, the procedural knowledge is held implicitly in the generated programs. A GEAKG keeps the generated or hand-written operators as components inside a persistent graph, so the composition structure and the effectiveness weights become explicit, stored artifacts. The language model is used once, offline, to populate the graph; it is not in the loop at execution time.

d) *Position*: GEAKG sits between two longer traditions. Machine learning has been combined with metaheuristics for over a decade—as fitness surrogates, operator selectors, or configuration policies—where a learned model assists a hand-coded search. Hyper-heuristics similarly select or generate heuristics during the run. The language-model-operator systems above move generation into the operator itself. GEAKG keeps those generated operators but wraps a persistent, typed graph around them, so the composition structure (L0), the operator content (L1), and the effectiveness weights (L2) become separate, inspectable, storable artifacts rather than transient search state. We make no priority claim over these lines; the contribution of this paper is a concrete construction-and-execution recipe on a single domain.

III. THE GEAKG MODEL

With GEAKGs placed into context, we now state it precisely. A GEAKG separates three concerns into three layers (Fig. 1): the ontology-constrained *topology* (L0), the *executable operators* bound to roles (L1), and the *learned edge weights* (L2). The formal object ties them together.

Definition 1 (GEAKG). A GEAKG is a tuple $\mathcal{G} = (\mathcal{S}, V, E, \Lambda, \Phi, \Sigma)$ where the RoleSchema $\mathcal{S} = (\mathcal{R}, \mathcal{K}, \kappa, T, K_0, \text{Re})$ is the ontology: $\mathcal{R}$ is a set of roles, $\mathcal{K}$ a set of categories, $\kappa : \mathcal{R} \rightarrow \mathcal{K}$ a category assignment, $T \subseteq \mathcal{K} \times \mathcal{K}$ the admissible category transitions, K_0 the entry categories, and Re the revisitable categories. The nodes are $V = \mathcal{R}$; the edges satisfy $(v_i, v_j) \in E \Rightarrow (\kappa(v_i), \kappa(v_j)) \in T$. The map $\Lambda : V \rightarrow 2^{\mathcal{O}}$ binds executable operators to roles, $\Phi : E \rightarrow \mathbb{R}^+$ stores learned pheromone weights, and Σ is a set of condition-to-action rules over execution state.

The schema makes the structure well-formed: every edge respects an admissible category transition, and every required category is reachable from an entry node. L1 makes it executable: every operator in $\Lambda(v)$ can be invoked at runtime and applied to a candidate solution. L2 makes it tunable: Φ is fitted offline and then frozen.

Separating the three layers is what makes each one inspectable and replaceable on its own: the topology can be checked for well-formedness independently of any operator, an operator can be swapped without touching the weights, and the weights can be relearned without regenerating code. Section VI shows that each layer is also load-bearing for the result.

a) *Cost*: The three stages carry distinct costs. Offline learning is dominated by MMAS: each of N iterations samples m trajectories and evaluates each on a problem instance set, resulting in $O(Nm|E|c_{\text{eval}})$ for an evaluation cost c_{eval} . Operator generation incurs a one-time language-model cost per role. Execution costs $O(Bc_{\text{op}})$ for a step budget B and per-operator cost c_{op} , with no language-model term. The persisted snapshot stores topology, weights, and rules in $O(|E|)$ space, which for these schemas is in kilobyte-scale.

IV. BUILDING A GEAKG FOR THE TSP

We now construct a GEAKG for the TSP in four steps. The construction is offline; the result is a compact snapshot that the executor later reads.

A. Step 1 — The RoleSchema

The following role schema is derived from the structure of permutation-based search following the principles of Iterated Local Search (ILS) [9]. Three phases recur in ILS-based metaheuristics for routing: build a starting tour, improve it locally, and escape local optima by perturbation. These become the three categories $\mathcal{K} = \{\text{CONSTRUCTION}, \text{INTENSIFICATION}, \text{PERTURBATION}\}$. Within them we place eleven roles ($|\mathcal{R}| = 11$)—for example, a nearest-neighbor constructor, a 2-opt intensifier, and a double-bridge perturbation—each a typed slot to be filled by one or more operators.

The admissible transitions T encode the control flow of the phase structure: construction precedes intensification, intensification and perturbation alternate, and construction is not revisited mid-run. This yields $|T| = 7$ admissible category transitions with a single entry category and two of three categories revisitable. Instantiated over the eleven roles, the

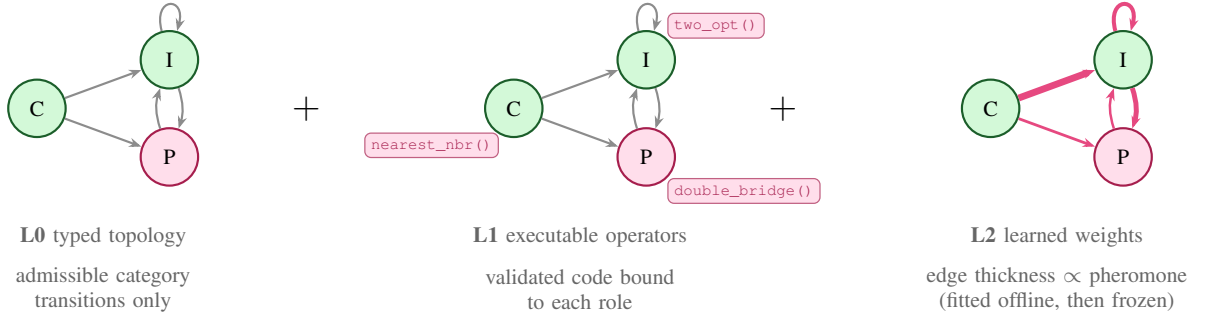


Fig. 1. Anatomy of a GEAKG, shown on a three-category skeleton (C: solution construction, I: intensification, P: perturbation). **L0** fixes which role-to-role transitions the ontology admits. **L1** binds a validated executable operator to each role. **L2** assigns each edge a learned pheromone weight (drawn as edge thickness). The Symbolic Executor reads the assembled snapshot and produces solutions by traversal, with no language-model call at run time.

resulting topology features fourteen representative directed edges, corresponding to a density of 0.13 (Table I). Figure 2 shows the schema and one execution trajectory through it.¹

The derivation of the RoleSchema follows the following steps, reusable for other problem families:

- 1) Identify the recurring phases of the chosen metaheuristic family. Here we considered ILS: construct a tour, intensify it locally, and perturb to escape. Then make them the categories $\mathcal{K}$.
- 2) Enumerate role slots inside each phase at the granularity of each operator’s intended effect (for example, mild versus strong perturbation), giving $\mathcal{R}$.
- 3) Read the admissible category transitions T off the control flow: construction precedes intensification; intensification and perturbation alternate.
- 4) Mark the entry category K_0 and which categories are revisitable.
- 5) Express conditional transitions—when to leave intensification for perturbation—as rules Σ (Section IV-E).

The eleven roles obtained for ILS-like permutation optimization are listed in Table II.

TABLE I
THE OPTIMIZATION ROLESHEMA USED FOR THE TSP WORKED EXAMPLE.

Component	Value
Roles $ \mathcal{R} $	11
Categories $ \mathcal{K} $	3
Admissible category transitions $ T $	7
Entry categories $ \bar{K}_0 $	1
Revisitable categories	2 / 3
Representative L0 edges	14
Directed density	0.13

B. Step 2 — Generating and validating operators (L1)

Each role must now be filled with executable operators. Operators may be hand-written or generated by a language

¹Note that the same eleven-role schema applies, unchanged, to other permutation problems. However, this reuse is the subject of [1]; we do not pursue it here.

TABLE II
THE ELEVEN ROLES OF THE PERMUTATION-OPTIMIZATION SCHEMA, GROUPED BY CATEGORY, WITH A REPRESENTATIVE OPERATOR FOR EACH. THE MIDDLE CATEGORY (LOCAL SEARCH) REALIZES INTENSIFICATION.

Category	Role	Representative operator
Construction	const_greedy	nearest neighbor
	const_insertion	cheapest insertion
	const_savings	Clarke–Wright savings
	const_random	randomized build
Local search (intensify)	ls_intensify_small	2-opt, swap
	ls_intensify_medium	3-opt, or-opt
	ls_intensify_large	Lin–Kernighan
	ls_chain	chained / VND
Perturbation	pert_escape_small	double bridge
	pert_escape_large	ruin-and-recreate
	pert_adaptive	history-based escape

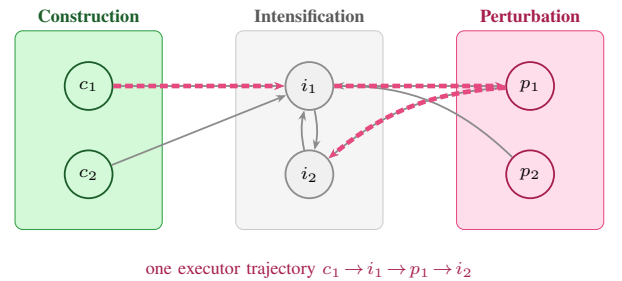


Fig. 2. The TSP RoleSchema split into three categories containing roles. Gray arrows are ontology-admissible transitions (L0). The thick pink dashed path is one trajectory produced by the Symbolic Executor, which selects the respectively next role from the learned weights (L2). Solution construction is entered once; intensification and perturbation alternate.

model from a structural description of the role—its category, its expected input and output (a distance matrix D storing the distances between cities, and a tour), and a short specification of its intended effect. Generation is offline and one-time.

However, a generated candidate operator is admitted only after a validation step that executes it on probe problem instances and checks that it runs without error and returns a feasible tour (a permutation of all cities); candidates that fail are discarded. This makes L1 content behavioral: a role

is populated by code whose effect on probe inputs has been observed, not by a description of an effect. Listing 1 shows a representative construction operator.

```
def greedy_nearest_neighbor(
    D, start=None, **kw):
    n = len(D)
    if start is None:
        start = random.randint(0, n-1)
    tour, seen = [start], {start}
    while len(tour) < n:
        cur = tour[-1]
        best, bd = None, float("inf")
        for city in range(n):
            if city in seen:
                continue
            if D[cur][city] < bd:
                bd = D[cur][city]
                best = city
        tour.append(best)
        seen.add(best)
    return tour
```

Listing 1. A representative construction operator bound to a construction role (D is the matrix of distances between cities): build a tour by repeatedly appending the nearest unvisited city. A candidate is kept only if validation executes it and returns a feasible permutation.

An intensification role is filled the same way. Listing 2 shows a 2-opt operator bound to `ls_intensify_small`: it removes two tour edges and reconnects them whenever the exchange shortens the tour. The same validation gate applies—the operator is admitted only after it runs and returns a feasible permutation.

```
def two_opt(tour, D, max_iter=1000,
    **kw):
    tour = tour.copy(); n = len(tour)
    improved = True
    while improved:
        improved = False
        for i in range(n - 1):
            for j in range(i + 2, n):
                a = tour[i]
                b = tour[i+1]
                c = tour[j]
                d = tour[(j+1) % n]
                g = D[a][c] + D[b][d]
                g -= D[a][b] + D[c][d]
                if g < 0:
                    s = tour[i+1:j+1]
                    s.reverse()
                    tour[i+1:j+1] = s
                    improved = True
    return tour
```

Listing 2. A representative intensification operator: 2-opt local search bound to `ls_intensify_small` (D is the distance matrix). It reverses a segment when swapping the edges (a, b) , (c, d) for (a, c) , (b, d) lowers the tour length.

C. Step 3 — Assembling the topology (L_0)

With roles populated, the topology is assembled by instantiating only the edges the schema admits: an edge (v_i, v_j) is created when $(\kappa(v_i), \kappa(v_j)) \in T$. Edges that would violate the phase structure—for instance, a perturbation feeding directly back into construction—are never created. The resulting graph for the TSP features the fourteen representative edges of Table I. Section VI shows that this constraint concentrates the search budget on valid transitions and measurably lowers the gap.

Next, the initial edge weights of the topology are generated. The template in Listing 3 supplies the role catalog and weight

guidelines and returns a candidate MetaGraph—a set of roles and weighted edges—constrained to a valid JSON schema through structured-output decoding. The returned edges are then filtered against the admissible category transitions T : any edge that violates the ontology is dropped, and the retained edges form L_0 , with initial weights to be refined by MMAS. The template also requires the escape transitions (local search to perturbation) to carry a condition, which seeds the rule set Σ of Section IV-E.

```
You are an expert in combinatorial optimization
metaheuristics.
Return a meta-algorithm for TSP/JSSP/VRP.

AVAILABLE ROLES
CONSTRUCTION: const_greedy, const_insertion,
               const_savings, const_random
LOCAL SEARCH: ls_intensify_{small,medium,large},
               ls_chain
PERTURBATION: pert_escape_{small,large},
               pert_adaptive

TASK
1. Select 8-12 roles.
2. Define 12-18 weighted edges (0.0-1.0).
3. Escape edges (ls -> perturbation) MUST
   carry a condition.

REQUIRED CONDITIONAL EDGE
ls_intensify_small -> pert_escape_small
{when: stagnation, threshold: 3, boost: 2.0}

WEIGHT GUIDELINES
construction -> ls : 0.80-0.90
ls -> ls (escalate): 0.65-0.80
ls -> perturbation : 0.30-0.50 (conditional)
perturbation -> ls : 0.85-0.95
Design seed: <<SEED>>
```

Listing 3. Abbreviated MetaGraph-generation template (the full catalog also exposes a meta-heuristic control category not used in this eleven-role schema). Structured-output decoding constrains the response to a valid JSON schema.

D. Step 4 — Learning edge weights (L_2)

The weight function Φ is adjusted by MMAS over the fixed topology, offline and with no language-model call. Each iteration samples trajectories passing through the admissible edges, evaluates the resulting tours in the TSP context, and reinforces the edges used by the best trajectories; pheromones evaporate at each iteration and are clamped to $[\tau_{\min}, \tau_{\max}]$ (Algorithm 1). The bounded update keeps the learned distribution diffuse enough to retain several viable role sequences. The output is the frozen weight function that the executor reads.

E. Step 5 — Conditional rules (Σ)

The rule set Σ conditions the topology traversal at runtime. A conditional edge carries a trigger and a multiplier: when the trigger holds at the current state, the edge's selection probability is scaled by a boost factor; otherwise, the edge keeps its base weight. The triggers are read from the search trace—stagnation (iterations without improvement), gap to the incumbent, or recent acceptance rate. The escape edges use the stagnation trigger: after a fixed number of non-improving steps, the local-search-to-perturbation edges are boosted, which steers traversal toward perturbation when intensification stalls. Rules operate in this boost mode rather than a hard gate, so a conditioned edge is made more likely but

Algorithm 1 Offline L2 weight learning (MMAS)

Require: topology (V, E) , operators Λ , problem instances $\mathcal{I}$, maximum number of iterations N

- 1: initialize $\Phi(e) \leftarrow \tau_{\max}$ for all $e \in E$
- 2: **for** $t = 1$ **to** N **do**
- 3: sample trajectories $\{\tau^{(k)}\}$ by weighted role selection over E
- 4: evaluate each $\tau^{(k)}$ on $\mathcal{I}$, record tour cost
- 5: $\Phi \leftarrow (1 - \rho) \Phi$ // evaporation
- 6: reinforce edges of the best trajectory: $\Phi(e) += \Delta$
- 7: clamp $\Phi(e) \leftarrow \min(\max(\Phi(e), \tau_{\min}), \tau_{\max})$
- 8: **end for**
- 9: **return** frozen Φ

never forced or forbidden; the admissible-transition envelope of L0 is unchanged. This keeps Σ a soft, state-dependent overlay on the learned weights.

V. RUNNING IT: THE SYMBOLIC EXECUTOR

The four steps described in the previous section leave a single artifact—the snapshot—and execution is performed via graph traversal. For its deployment, the symbolic executor reads the snapshot $(\mathcal{S}, V, E, \Lambda, \Phi, \Sigma)$ and produces solutions/tours (Algorithm 2). From the current state, the executor selects the next role by sampling the learned transition weights restricted to admissible edges, applies one operator bound to that role, evaluates the resulting candidate solution/tour in the TSP context, and accepts it on improvement. No language-model call occurs at any point in this loop. The cost of the loop is the cost of running operators and evaluating tours: a frozen snapshot completes hundreds of evaluations in seconds.

a) A walkthrough: The following fictitious execution illustrates the loop. Traversal starts at a construction role—say `const_greedy`, which builds a nearest-neighbor tour—and then moves to a local-search role such as `ls_intensify_small`, where the 2-opt operator is applied, and the resulting solution is accepted if it improves the previous tour. While improvements continue, traversal stays in the local-search category, escalating to `ls_intensify_medium` along the higher-weighted edges. When the stagnation trigger fires (Section IV-E), the boosted escape edge raises the probability of `pert_escape_small`, whose double-bridge move perturbs the incumbent before traversal returns to local search. Because each candidate is accepted only on improvement, the incumbent cost is non-increasing across the run; the learned weights set how long the search intensifies before perturbation. The pink trajectory in Fig. 2 traces one such sequence.

b) Lifecycle separation: Building and running a GEAKG splits cleanly into stages (Fig. 3): acquisition (operator generation and validation) and refinement (ACO weight learning) are offline; persistence is a compact snapshot; application is symbolic execution. The language-model cost is confined to acquisition and is incurred once. This separation also yields

Algorithm 2 Symbolic Executor (zero runtime model tokens)

Require: snapshot $(\mathcal{S}, V, E, \Lambda, \Phi, \Sigma)$, instance x , budget B

- 1: $s \leftarrow$ entry role in K_0 ; $\text{best} \leftarrow$ initial tour
- 2: **for** $b = 1$ **to** B **do**
- 3: $A \leftarrow \{v \mid (s, v) \in E\}$ // L0-admissible successors
- 4: sample $v \in A$ with prob. $\propto \Phi(s, v) \cdot \beta_{\Sigma}(s, v)$ // β_{Σ} : rule boost
- 5: $o \leftarrow$ operator from $\Lambda(v)$; $\text{cand} \leftarrow o(\text{best}, x)$
- 6: **if** $\text{cost}(\text{cand}) < \text{cost}(\text{best})$ **then**
- 7: $\text{best} \leftarrow \text{cand}$
- 8: **end if**
- 9: $s \leftarrow v$
- 10: **end for**
- 11: **return** best

an audit trail—roles, edges, operators, weights, and rules are stored and can be examined separately.

VI. SINGLE-DOMAIN RESULTS

Next, we report on the results obtained with the ILS-inspired GEAKG for the TSP. All numbers are taken from [1]; no new experiment is introduced here. Instances were taken from the TSPLIB [10]; gaps are to the known optimum.

a) Feasibility across language-model scales: Because operators are validated before admission, the executor is not dependent on a single high-capacity language model as a generator. In fact, the constructed GEAKG returns feasible tours for all seven considered TSP instances when its operators are synthesized with each of three language model scales—`gpt-5.2`², `gpt-4o-mini`, and a fully local `Qwen2.5-14B`—whereas a standalone code-evolution baseline (LLaMEA) returns a valid tour only in one of the seven cases (in the local 10k-token configuration). Feasibility here is a property of the validated pool and the bounded traversal, and it is retained at the smallest scale tested (Table III).

TABLE III
SINGLE-DOMAIN TSP SUMMARY. FEASIBLE TOURS ARE OUT OF SEVEN INSTANCES; DEPLOYMENT USES NO LANGUAGE-MODEL TOKEN REGARDLESS OF WHICH MODEL SCALE SYNTHESIZED THE OPERATORS OFFLINE.

Configuration	Feasible	Deploy. tokens
GEAKG (gpt-5.2 operators)	7 / 7	0
GEAKG (gpt-4o-mini operators)	7 / 7	0
GEAKG (Qwen2.5-14B, local)	7 / 7	0
LLaMEA standalone (local, 10k)	1 / 7	—

b) Quality at zero deployment tokens: Executing the frozen snapshot produces solutions with optimality gaps of 3.4% to 8.0% across four instances over eight seeds, at no language-model cost during execution. Embedding one strong operator evolved with LLaMEA (50k-token budget) into the

²Exact model versions: `gpt-5.2-0111`, `gpt-4o-mini`, and `Qwen2.5-14B`, at generation and repair temperatures 0.7 and 0.2–0.3, respectively.

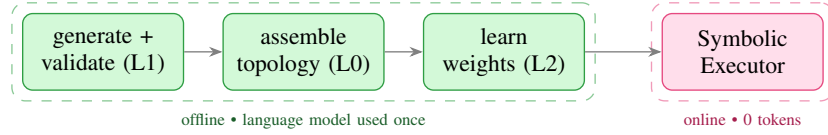


Fig. 3. Lifecycle of a GEAKG. Acquisition and refinement are offline (green); the language model is used once, during acquisition. The persisted snapshot is then executed online (pink) with no language-model call.

pool improves the source solver further: a hybrid configuration wins 5/7 TSP instances under a matched 50k-token budget, and the single embedded operator accounts for gap reductions of 65% on TSPLIB instance `pr226` and 54% on `pcb442` relative to the standalone baseline. The point of interest for this paper is the deployment profile: competitive tours at zero runtime tokens, with the generation cost paid once, offline.

c) *Does the ontology constraint help or only restrict?*:

A natural concern is that the admissible-transition constraint simply removes options. We test this hypothesis directly by applying MMAS to both the schema-constrained topology and an unrestricted topology containing every role-to-role edge. The constrained topology consistently achieves a lower mean TSP gap on every considered problem instance: for example, for `berlin52`, 1.37 versus 3.19, and for `kroA100`, 3.47 versus 5.23 percentage points, averaged over eight random seeds; see also Fig. 4. The unrestricted graph wastes search effort on semantically invalid transitions, whereas the ontology concentrates exploration on valid ones, improving the search.

d) *Do the learned weights help?*: Using the learned weights vs. uniform weights, holding operators and topology fixed, lowers the gap on three of four TSP instances, with the largest improvement on the biggest instance (`pr226`, a reduction of 1.58 percentage points) and differences within run-to-run noise on mid-sized instances. Whether the same weights remain useful after a domain change (moving from the TSP to a different combinatorial optimization problem) is a transfer question, deferred to [1].

e) *Is the operator content load-bearing?*: As a third check, holding the topology and weights fixed and replacing every operator body with a no-op inflates/degrades the gap to 298%–1978%: with no transformation left, the executor is reduced to its initial multistart tours. The constructed operators carry the result; they are not decoration. A full per-layer decomposition of operator quality, learned weights, and their interaction is reported in [1].

A. Inspecting the snapshot

Because the result is a stored graph, it can be read directly. Listing 4 shows a fragment of the deployable core learned on the source instances. The seed weights from the generation template are near-uniform guidelines; after offline learning, some edges are reinforced toward $\tau_{\max}$ —the `const_greedy`→`ls_intensify_small` entry and the `ls_intensify_small`→`pert_escape_small` escape both reach 1.0—while an edge absent from the best trajectories decays toward $\tau_{\min}$, here `const_insertion`→`ls_intensify_medium` falling

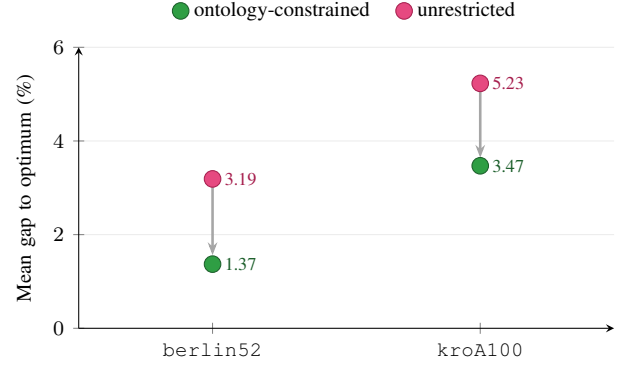


Fig. 4. The ontology constraint lowers the search gap rather than only pruning transitions. Each instance shows the mean TSP gap to optimum under the schema-constrained versus unrestricted topology (ant colony trained from scratch, eight seeds); the arrow marks the reduction. Lower is better.

from 0.90 to 0.35. The whole deployable core is fourteen typed edges over twenty-three operators, serialized to kilobyte-scale JSON. The best traversal found during offline refinement begins as follows: `const_random` → `ls_chain` → `ls_intensify_small` → `ls_intensify_medium` → `ls_intensify_large`. Then, it interleaves further `ls_chain` and `ls_intensify_large` steps with `pert_adaptive`, alternating intensification and perturbation as the schema admits and reaching a mean 6.0% gap over all considered instances. The point is not the particular numbers but that they are legible: the learned policy is a small table one can read, diff across runs, and reuse. Figure 5 draws the same snapshot in full, with each edge's thickness set to its learned weight.

```
{
  "metagraph": "TSP_ils", // 14 typed edges
  "seed_edges": {
    "const_greedy -> ls_intensify_small": 0.90,
    "const_random -> ls_chain": 0.90,
    "ls_intensify_small -> pert_escape_small": {
      "w": 0.45, "when": "stagnation", "thr": 3
    }
  },
  "learned_pheromones": { // after offline ACO
    "const_greedy -> ls_intensify_small": 1.00,
    "ls_intensify_small -> ls_intensify_medium": 0.90,
    "ls_intensify_small -> pert_escape_small": 1.00,
    "ls_intensify_large -> pert_escape_large": 0.53,
    "const_insertion -> ls_intensify_medium": 0.35
  },
  "best_gap": 0.060, "operators": 23
}
```

Listing 4. A fragment of the persisted TSP snapshot (`TSP_ils`): seed edge weights from the template, the learned pheromones after ACO, and summary fields. Learned weights diverge from the near-uniform seeds—reinforced edges rise to 1.0, avoided edges decay.

VII. IMPLEMENTATION AND REPRODUCIBILITY

The results above depend on a few fixed settings, recorded here for reproducibility. The pipeline is parameterized by the MMAS settings in Table IV. Pheromones are bounded to $[\tau_{\min}, \tau_{\max}] = [0.01, 10.0]$, evaporated at rate $\rho = 0.1$, and updated by the best ant of each iteration over 15 ants; selection mixes pheromone and edge-weight terms with exponents $\alpha = 1.0$ and $\beta = 2.0$; a satisfied rule multiplies the corresponding edge probability by a boost value from $[1.5, 2.5]$. Operators were generated with the gpt-5.2, gpt-4o-mini, and Qwen2.5-14B models, at generation and repair temperatures 0.7 and 0.2–0.3, with structured-output decoding to a fixed JSON schema; a candidate is admitted only after it executes on probe instances and returns a feasible permutation. In the run that produced the snapshot of Listing 4, all thirteen generated candidates passed validation—no syntax, runtime, timeout, or invalid-result failures—over four refinement rounds, yielding a pool of twenty-three operators across the eleven roles. All reported gaps are means over eight seeds, with the TSP source instances taken from the TSPLIB. The deployable snapshot—typed topology, learned weights, and rules—serializes to kilobyte-scale JSON and is read by the executor without modification.

TABLE IV
MMAS CONFIGURATION USED FOR OFFLINE L2 LEARNING.

Symbol	Meaning	Value
α	pheromone exponent	1.0
β	edge-weight exponent	2.0
ρ	evaporation rate	0.1
m	ants per iteration	15
—	max roles per sequence	6–8
$\tau_{\min}$	lower pheromone bound	0.01
$\tau_{\max}$	upper pheromone bound	10.0
β_{Σ}	rule boost (when triggered)	1.5–2.5

VIII. DISCUSSION

The worked example presented in this paper shows that the representation can be built and run end-to-end on a single domain. Three properties stand out. First, the lifecycle is separable: acquisition and refinement are offline, and application uses no language-model call, which keeps deployment cost low and leaves an inspectable snapshot. Second, the artifact is compact—the deployable core serializes to kilobyte-scale JSON—so it is cheap to store and to examine. Third, validation makes the operator pool behavioral, which is what lets the same construction run at the smallest model scale tested.

The construction also has limitations. The RoleSchema requires human modeling; for the TSP example from this paper, schema design took hours rather than days, and the cost is amortized across a problem family. The executor performs no online weight update, so behavior at deployment is fixed by the offline snapshot. And the evidence reported here is single-domain by design: feasibility, gaps, and one structural ablation on TSP. Whether the ontology-constrained structure and the learned weights carry across domains—and under

which target-side search budgets reuse pays—is exactly the question taken up in [1].

a) Design choices that favor reuse: Two further choices are made with reuse in mind. Bounding the pheromones to $[\tau_{\min}, \tau_{\max}]$ keeps the learned distribution from collapsing onto a single role sequence, so a snapshot retains several viable orderings rather than narrowing to a single trajectory—which matters when the same graph is run again. Accept-if-improving traversal makes the executor’s output monotone in cost and tolerant of a fraction of unhelpful operators: a no-op or a role-agnostic random move is simply not accepted, which is the mechanism behind the graceful degradation observed when operators are corrupted (Section VI). Diversity and robustness matter for a stored, reused artifact, and both follow from the construction without online tuning.

b) Outlook: Treating the snapshot as a first-class artifact suggests a lifecycle around it. Once procedural knowledge is a stored graph, operations familiar from knowledge graphs apply: a snapshot can be versioned as operators are added, queried for the role sequences it favors, and merged with another. The worked example fixes only the construction-and-execution core; the surrounding tooling—query languages for procedural graphs, snapshot diffing across runs, and provenance for each operator and weight—is where the representation would earn its keep beyond a single solver. The single-domain core makes those next steps well-posed, because each layer is already a separate, serialized, inspectable object.

IX. CONCLUSION

We gave a worked example of constructing and executing a Generative Executable Algorithm Knowledge Graph for the preservation and use of algorithm knowledge on the Traveling Salesman Problem: deriving an eleven-role schema, generating and validating operators, assembling an ontology-constrained topology, learning edge weights with MMAS, and running the result with a deterministic, zero-token executor. On the TSP, the constructed graph returns feasible tours across three model scales and reaches single-digit-percent gaps at no deployment-token cost, and a controlled check shows the ontology constraint improves the search. Throughout, the four layers stay separate and inspectable: the schema, the validated operators, the learned weights, and the conditional rules can each be read, drawn, and serialized on their own. The recipe is meant to be reproducible, and the worked artifact—construction template, operator listings, and the learned snapshot of Fig. 5—is small enough to study end to end. The cross-domain transfer story is the subject of [1].

ACKNOWLEDGMENTS

Camilo Chacón Sartori acknowledges funding from R&D Project PID2022-138283NBI00, funded by MICIU/AEI/10.13039/501100011033 and “FEDER – A way of making Europe”.

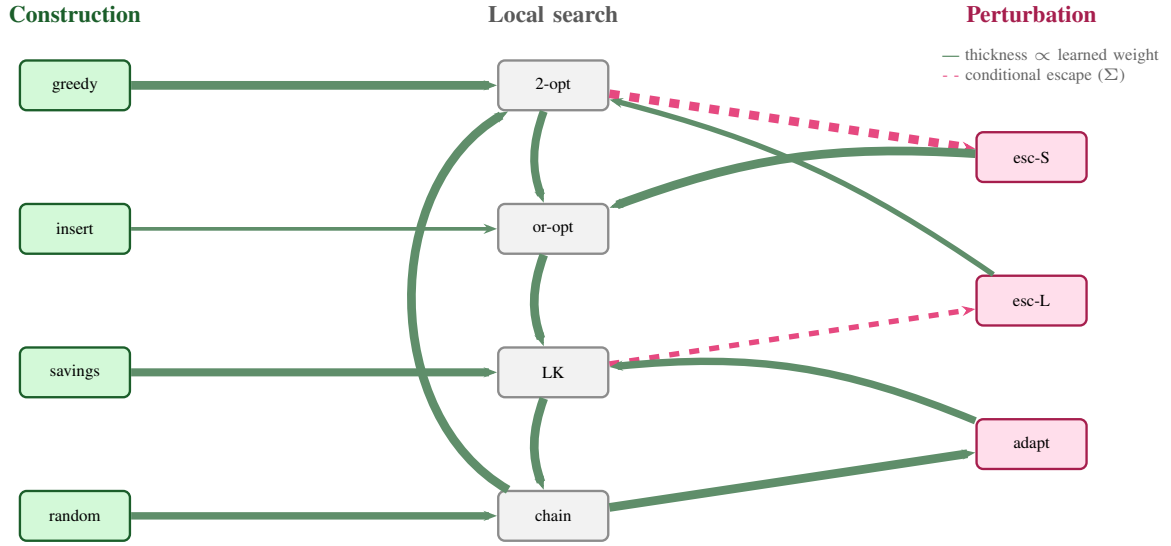


Fig. 5. The learned TSP snapshot (TSP_{ils}) drawn in full: eleven roles in three categories, fourteen edges, each edge's thickness set to its learned pheromone weight. Reinforced transitions—the $greedy \rightarrow 2-opt$ entry, the intensification chain, the $2-opt \rightarrow esc-S$ escape—are thick; the $insert \rightarrow or-opt$ edge has decayed toward τ_{min} and is thin. Dashed pink edges are conditional escapes whose probability is boosted on stagnation. This is the same artifact serialized in Listing 4.

REFERENCES

- [1] C. Chacón Sartori, J. H. García, A. V. Tomut, and C. Blum, “GEAKG: Generative executable algorithm knowledge graphs,” arXiv preprint arXiv:2603.27922, 2026.
- [2] A. Hogan, E. Blomqvist, M. Cochez *et al.*, “Knowledge graphs,” *ACM Computing Surveys*, vol. 54, no. 4, pp. 1–37, 2021.
- [3] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.
- [4] T. Stützle and H. H. Hoos, “MAX-MIN ant system,” *Future Generation Computer Systems*, vol. 16, no. 8, pp. 889–914, 2000.
- [5] N. van Stein and T. Bäck, “LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics,” *IEEE Transactions on Evolutionary Computation*, vol. 29, no. 2, pp. 331–345, 2025.
- [6] B. Romera-Paredes, M. Barekatin, A. Novikov *et al.*, “Mathematical discoveries from program search with large language models,” *Nature*, vol. 625, pp. 468–475, 2024.
- [7] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang, “Evolution of heuristics: Towards efficient automatic algorithm design using large language model,” in *Proc. Int. Conf. on Machine Learning (ICML)*, 2024, pp. 32 201–32 223.
- [8] H. Ye, J. Wang, Z. Cao, F. Berto, C. Hua, H. Kim, J. Park, and G. Song, “ReEvo: Large language models as hyper-heuristics with reflective evolution,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
- [9] H. R. Lourenço, O. C. Martin, and T. Stützle, “Iterated local search: Framework and applications,” in *Handbook of metaheuristics*. Springer, 2019, pp. 129–168.
- [10] G. Reinelt, “TSPLIB—a traveling salesman problem library,” *ORSA Journal on Computing*, vol. 3, no. 4, pp. 376–384, 1991.