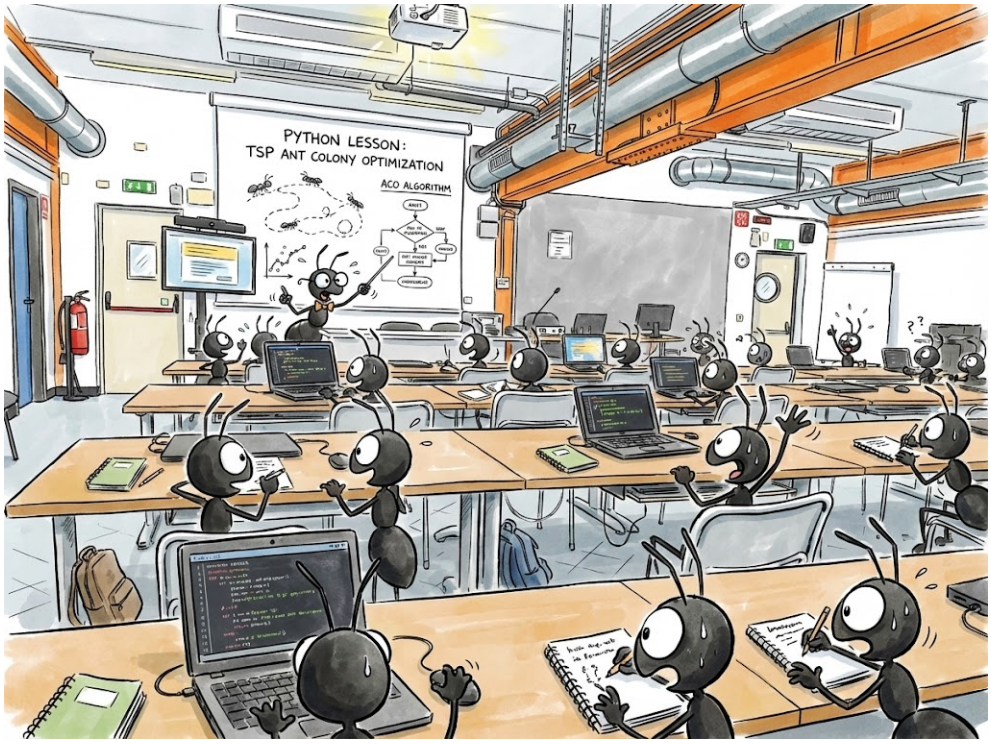
SOLVING THE TSP

WITH MMAS & PYTHON



EDOARDO PASIO

17TH December 2025

ABSTRACT

The *Travelling Salesman Problem* (TSP) is a cornerstone of combinatorial optimization. This paper explores the application of a *Max-Min Ant System* (MMAS) meta-heuristic to solve TSP instances using a high-performance Python implementation based on `NumPy` and `Numba` for Just-In-Time (JIT) compilation and parallel execution. The algorithm constructs tours in parallel at the ant level, leveraging nearest-neighbor candidate lists with a robust fallback mechanism that guarantees feasibility. To improve solution quality beyond purely constructive search, we apply a lightweight local search operator to the iteration-best tour: a pairwise vertex-swap neighborhood evaluated by incremental (delta) cost updates and bounded by a small number of passes. Pheromone updates follow a best-so-far elitist MMAS strategy, where evaporation is implemented via a persistence factor and pheromone values are clamped within dynamically updated bounds derived from the current global best solution. We analyze the algorithmic design choices and discuss empirical performance on benchmark instances and a real-world campus routing case study.

CONTENTS

1 DESCRIPTION OF THE REAL PROBLEM AND DATA

This project addresses a realistic pedestrian routing and location optimization problem on the University of Genoa campus. The campus environment is characterized by complex topography, with steep elevation changes, stairways, and fragmented pedestrian paths distributed across a hilly urban area.

The objective is to compute an optimal closed walking tour (e.g., for patrols, logistics, or maintenance) that visits a predefined set of university buildings exactly once and returns to the starting point. Unlike classical Euclidean *Travelling Sales-*

man Problem (TSP) instances, walking costs on campus are inherently *direction-dependent*: moving uphill generally requires more time and effort than moving downhill, leading to asymmetric travel costs.

1.1 Dataset and Data Acquisition

To obtain realistic walking distances, we rely on the Open Source Routing Machine (OSRM) [5]. OSRM processes OpenStreetMap (OSM) data [4] and computes shortest pedestrian paths while accounting for the actual street network, stairways, one-way paths, and elevation-related constraints.

In practice, OSRM is executed locally inside a Docker container. A Python [8] preprocessing script sends HTTP requests to the local OSRM server and retrieves a distance (travel-time) matrix D, where each entry d_{ij} represents the walking time from building i to building j.

This preprocessing step is external to the optimization code presented in this work; the solver assumes that the distance matrix is already available as a NumPy [1] array.

2 GRAPH CONSTRUCTION AND PROBLEM DEFINITION

The routing problem is modeled as a complete directed graph

$$G = (V, E),$$

where:

- **Nodes (V)**: Each node corresponds to a university building or point of interest.
- **Edges (E)**: A directed edge (i, j) exists for every ordered pair of distinct nodes.
- **Weights**: Each edge (i, j) is associated with a weight $w_{ij} = d_{ij}$, representing the walking time obtained from OSRM.

Because the distance matrix may satisfy $d_{ij} \neq d_{ji}$, the problem is an *Asymmetric TSP* (ATSP).

3 MAX-MIN ANT SYSTEM (MMAS)

To solve the ATSP, we adopt the *Max-Min Ant System* (MMAS) metaheuristic, originally proposed by Stützle and Hoos [6]. MMAS is a variant of *Ant Colony Optimisation* (ACO) designed to improve convergence stability through strong elitism and explicit pheromone bounds.

3.1 Heuristic Information and Transition Rule

The algorithm maintains two matrices:

- **Pheromone matrix** τ_{ij}, representing learned desirability.

- **Visibility matrix** $\eta_{ij} = \frac{1}{d_{ij}}$, encoding greedy distance information.

At each construction step, an ant located at node i chooses the next node j among the unvisited candidates with probability:

$$p_{ij} = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{l \in N(i)} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta}}, \tag{1}$$

where α and β control the relative importance of pheromone and heuristic information.

3.2 Candidate Lists

To reduce computational complexity, each node is associated with a candidate list containing its k nearest neighbors (default $k = 20$). During tour construction, the next node is sampled via roulette-wheel selection restricted to this list, ignoring already visited nodes. If the candidate list is exhausted (i.e., all candidates are visited), the selection falls back to roulette-wheel sampling over the set of all nodes, again excluding visited nodes. As a final robustness measure, if no probabilistic move is possible due to degenerate weights, the algorithm deterministically selects the first available unvisited node to guarantee completion of a feasible tour.

3.3 Pheromone Update Rule

The implementation follows a strict *best-so-far elitist* MMAS update strategy. After each iteration, pheromone trails are updated exclusively using the current global best tour found so far T^{bs} with length L_{bs}.

Evaporation is implemented through a *persistence* factor $\rho \in [0, 1)$ by scaling all pheromone values:

$$\tau_{ij} \leftarrow \rho \cdot \tau_{ij}. \tag{2}$$

Immediately after evaporation, all pheromone values are clamped to the admissible interval:

$$\tau_{\min} \leq \tau_{ij} \leq \tau_{\max}. \tag{3}$$

Pheromone is then deposited along the edges of the global best tour T^{bs}:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{bs}, \tag{4}$$

where

$$\Delta\tau_{ij}^{bs} = \begin{cases} \dfrac{1}{L_{bs}} & \text{if } (i,j) \in T^{bs}, \\ 0 & \text{otherwise.} \end{cases}$$

After deposition, pheromone values are clamped to the upper bound $\tau_{\max}$ only; the lower bound is guaranteed by the prior evaporation-and-clamping step.

The upper pheromone bound is dynamically updated whenever a new global best tour is found:

$$\tau_{\max} = \frac{1}{(1-\rho)\, L_{bs}}, \tag{5}$$

and the lower bound is set as:

$$\tau_{\min} = \max\left(\frac{\tau_{\max}}{1000}, 10^{-10}\right). \tag{6}$$

This bounding mechanism prevents stagnation while maintaining sufficient exploration throughout the search.

3.4 Symmetry Considerations

Although the distance matrix is asymmetric, the implementation optionally enforces *symmetric pheromone updates*. When enabled, pheromone deposited on edge (i, j) is mirrored onto (j, i). This design choice stabilizes learning while still allowing asymmetric costs to influence tour construction through the heuristic matrix.

4 IMPLEMENTATION DETAILS

The solver is implemented in Python 3.11 with a strong emphasis on performance.

4.1 Libraries and Optimization

- **NumPy** [1]: Dense matrix storage and numerical operations.
- **Numba** [3]: Just-In-Time compilation of computational kernels using `@njit`, enabling parallel execution without the Python GIL.

The most computationally intensive component—tour construction for multiple ants—is parallelized at the ant level.

4.2 Local Search Integration

To enhance solution quality beyond pure constructive heuristics, we integrate a local search phase applied to the iteration-best tour before pheromone reinforcement. In the provided implementation, this operator is *not* a classical 2-opt segment reversal. Instead, it explores a *pairwise vertex-swap* neighborhood: two positions in the tour are swapped, and the move is accepted if it yields a strictly improving change in total tour length.

To keep the local search computationally lightweight, the move evaluation is performed using an incremental (delta) cost computation that only inspects the affected edges. Moreover, the procedure is bounded by a small number of full passes over the tour (a safety valve) to prevent excessive runtime on poor initial tours. By applying this refinement only to the best tour of each iteration, we balance exploitation and computational cost, ensuring that pheromone updates reinforce a locally improved structure and accelerating convergence in practice.

4.3 Precomputation Strategy

To reduce runtime overhead:

- The distance-dependent heuristic η_{ij} is combined with pheromones into a *choice information matrix* $\tau_{ij}^{\alpha}\eta_{ij}^{\beta}$.
- This matrix is recomputed once per iteration, avoiding repeated exponentiation inside inner loops.

Algorithm 1 Parallel MMAS Iteration with Bounded Vertex-Swap Local Search

Require: Distance matrix $D \in \mathbb{R}^{N\times N}$, pheromone matrix $\tau \in \mathbb{R}^{N\times N}$, candidate lists $\mathrm{NN} \in \mathbb{N}^{N\times k}$, parameters α, β, ρ, number of ants m
Ensure: Updated global best tour T^{bs} and length L_{bs}

Precompute choice information: $H_{ij} \leftarrow (\tau_{ij})^{\alpha} \cdot \left(\frac{1}{D_{ij}+\epsilon}\right)^{\beta}$
for ant $a = 1 \dots m$ **in parallel do**
    Choose random start node s_a
    $T^{(a)} \leftarrow \textsc{ConstructTour}(s_a, \mathrm{NN}, H)$ ▷ *Roulette selection on NN list; global fallback if needed*
    $L_a \leftarrow \textsc{Evaluate}(T^{(a)}, D)$
$a^* \leftarrow \arg\min_a \{L_a\}$
$T^{ib} \leftarrow T^{(a^*)}$
$L_{ib} \leftarrow L_{a^*}$
▷ *Improve iteration-best tour using bounded vertex-swap local search*
$(T^{ib}, L_{ib}) \leftarrow \textsc{LocalSearchVertexSwap}(T^{ib}, D)$
if $L_{ib} < L_{bs}$ **then**
    $T^{bs} \leftarrow T^{ib}$
    $L_{bs} \leftarrow L_{ib}$
    $\textsc{UpdatePheromoneBounds}(L_{bs})$
▷ *Best-so-far MMAS pheromone update*
$\tau \leftarrow \rho \cdot \tau$
$\tau \leftarrow \textsc{Clamp}(\tau, \tau_{\min}, \tau_{\max})$
$\textsc{DepositBestSoFar}(\tau, T^{bs}, 1/L_{bs})$
return T^{bs}, L_{bs}

5 COMPUTATIONAL RESULTS

We evaluated the proposed MMAS implementation on a subset of standard benchmark instances from TSPLIB, covering both *Symmetric TSP* (STSP) and ATSP topologies. The results are summarized in Table 1.

Table 1: Comparison of results with optimal values for TSP and ATSP instances

Instance	Type	Cost	Optimal[†]	Gap (%)	Time (s)
berlin52	symmetric	7544.37	7542	0.03	1.31
eil101	symmetric	667.67	629	6.15	1.73
kroA100	symmetric	21 474.70	21 282	0.91	2.12
pr124	symmetric	59 385.67	59 030	0.60	1.97
pcb442	symmetric	54 121.26	50 778	6.58	14.22
ftv33	asymmetric	1580.00	1286	22.86	1.24
ftv44	asymmetric	1957.00	1613	21.33	1.21
ftv64	asymmetric	2574.00	1839	39.97	1.54
berlin52	symmetric	7544.37	7542	0.03	1.33
eil101	symmetric	650.38	629	3.40	1.76

Continued on next page

(Continued)

kroA100	symmetric	21 471.96	21 282	0.89	1.77
pr124	symmetric	59 086.81	59 030	0.10	2.03
pcb442	symmetric	55 165.65	50 778	8.64	11.41
ftv33	asymmetric	1681.00	1286	30.72	1.25
ftv44	asymmetric	2077.00	1613	28.77	1.37
ftv64	asymmetric	2678.00	1839	45.62	1.54

† The *Optimal* column indicates the best known solution for the instance.
Note: All instances were executed with the following parameter configuration: $n_{ants} = 100$, $max_{iterations} = 1000$, $\alpha = 1.0$, $\beta = 2.25$, $\rho = 0.98$.

The performance on symmetric instances (e.g., `berlin52`, `pr124`) is highly effective, with optimality gaps frequently falling below 1%. Even on larger instances like `pcb442`, the algorithm maintains a competitive gap. Conversely, the asymmetric `ftv` instances proved more challenging, yielding larger gaps, suggesting that the landscape of these specific problems may require specialized tuning.

5.1 Case Study: University of Genoa Campus

We applied the algorithm to the specific instance of the University of Genoa campus. The problem consists of $N = 47$ locations (buildings), the details of which can be retrieved from the university's EasyAcademy portal [7]. The edge weights represent walking times in seconds derived from OSRM.

Using the configuration ($\alpha = 1.0, \beta = 2.25, \rho = 0.98, n_{ants} = 100$), we executed the algorithm three times. The best tour length achieved was **14794.80 s**, with an average runtime of **1.06 s** per execution.

For validation, we compared this against the state-of-the-art LKH-2 solver [2], which found the best known solution of **14789.2 s**. Although our solution is not strictly optimal, the gap is negligible ($\approx$ 0.04%), demonstrating that the Python-based MMAS can effectively find near-optimal routes for this practical instance.

REFERENCES

[1] Charles R Harris, K Jarrod Millman, St éfan J van der Walt et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (2020), pp. 357–362.

[2] Keld Helsgaun. *An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic*. Report. DAT/RUC Report. See also http://akira.ruc.dk/~keld/research/LKH/. Roskilde University, 2000.

[3] Siu Kwan Lam, Antoine Pitrou and Stanley Seibert. 'Numba: A LLVM-based Python JIT compiler'. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[4] OpenStreetMap contributors. *OpenStreetMap*. 2024. URL: https://www.openstreetmap.org/.

[5] Project OSRM. *Open Source Routing Machine*. 2024. URL: http://project-osrm.org/.

[6] Thomas Stützle and Holger H Hoos. 'MAX-MIN ant system'. In: *Future generation computer systems* 16.8 (2000), pp. 889–914.

[7] University of Genoa. *EasyAcademy: University Spaces and Classrooms*. 2025. URL: https://easyacademy.unige.it/spazi/index.php?_lang=en (visited on 17/12/2025).

[8] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.