Abhijit's Algorithm: A Geometric, Best-Improvement Heuristic for the Travelling Salesman Problem

Abhijit

Independent Researcher · Uncle Droid

tsp.uncledroid.app

Abstract

We present Abhijit's Algorithm, a heuristic for the Travelling Salesman Problem (TSP) developed independently over six months in 2021-22 without reference to the existing literature. The algorithm combines a novel geometric seeding strategy — OTR (Outer) — with a best-improvement node insertion loop and geometric edge uncrossing, alternated until convergence. OTR discovers convex hull edges via a cross-product predicate test rather than through standard hull-construction algorithms, using them as the initial tour skeleton. The refinement loop identifies the single globally best node relocation at each step — computing a full insertion-cost matrix before every move — rather than applying batch greedy improvements. We additionally present SPX, a farthest-neighbour seeding strategy as an anti-greedy counterpart to nearest-neighbour. On four standard TSPLIB benchmark instances, the algorithm achieves within 3.2-4.7% of known optimal on 52-100 city instances, converging in under 100ms on a single CPU core. The algorithm's O(n^2)-per-iteration structure is embarrassingly parallel, making it suitable for GPU or SIMD acceleration in embedded routing applications. An interactive web visualiser provides step-by-step plain-language narration of every algorithmic decision, serving both as a competitive game and as a pedagogical tool.

1. Introduction

The Travelling Salesman Problem asks: given a set of n cities and pairwise distances between them, find the shortest closed tour visiting each city exactly once. It belongs to the class NP-hard [1], and while exact solvers such as Concorde [2] can solve instances of thousands of cities through branch-and-bound with cutting planes, practical applications frequently require fast, near-optimal solutions on modest hardware.

The present work describes an algorithm developed entirely from geometric intuition, without prior exposure to the TSP literature. The author, a self-taught software developer, built both the algorithm and the experimental tool used to develop it — a visual Delphi Pascal application in which each algorithmic idea could be tested interactively before being formalised. The alternating refinement structure was discovered experimentally: the two improvement phases were initially separate interactive buttons, and their mutual benefit when alternated was observed before being automated. This methodology — algorithm discovery through interactive visualisation — is itself of methodological note.

The contributions of this paper are:

(1) OTR: a convex hull seeding strategy based on cross-product predicate testing, independently derived without reference to the convex hull heuristic literature.

(2) A best-improvement node insertion refinement loop with full matrix recomputation at every step, distinct from standard first-improvement or greedy-batch approaches.

(3) SPX: a farthest-neighbour seeding heuristic as a deliberate anti-greedy strategy for elongated and multi-clustered instances.

(4) An analysis of the algorithm's computational structure showing suitability for hardware-parallel embedded applications.

(5) An interactive web visualiser with step-by-step plain-language narration of every algorithmic move, supporting both competitive play and classroom instruction.

2. Algorithm Description

The algorithm operates in two stages: seeding (construction of an initial tour) and refinement (iterative improvement). The refinement stage is shared across all seeding strategies and constitutes the core of the algorithm.

2.1 Notation

Let P = {p_1, ..., p_n} be a set of n points in the plane, with d(p_i, p_j) the Euclidean distance. A tour T is a cyclic permutation of P. For point p_k in T with left neighbour p_L and right neighbour p_R, the current marginal cost is:

```
cost_current(p_k) = d(p_L, p_k) + d(p_k, p_R) - d(p_L, p_R)
```

This is the distance added to the tour by p_k at its current position. For p_k inserted between edge (p_i, p_j), the insertion cost is:

```
cost_insert(p_k, p_i, p_j) = d(p_i, p_k) + d(p_k, p_j) - d(p_i, p_j)
```

The saving from relocating p_k to edge (p_i, p_j) is:

```
saving(p_k, p_i, p_j) = cost_current(p_k) - cost_insert(p_k, p_i,
p_j)
```

2.2 Seeding Strategies

2.2.1 OTR — Convex Hull Seeding (Novel)

For every pair of distinct points (p_i, p_j), let dx = p_j.x - p_i.x and dy = p_j.y - p_i.y. For every other point p_k, compute the signed cross product:

```
cross(p_k) = dx * (p_k.y - p_i.y) - dy * (p_k.x - p_i.x)
```

If all non-zero values of cross(p_k) share the same sign, all other points lie strictly on one side of the line through (p_i, p_j), and this edge is a convex hull edge. Such edges must appear in any optimal tour: they are the outer boundary of the point set and cannot be improved by detour.

This is a verification strategy: rather than constructing the hull by incremental insertion or angular sweep, every candidate edge is tested directly by predicate. The result is identical to standard convex hull algorithms on general-position inputs, but was derived independently from geometric first principles.

The hull edges are stitched into a chain. Remaining interior points are inserted sequentially, each at the edge minimising cost_insert — the standard cheapest insertion step.

2.2.2 SSF — Shortest Segment First

SSF greedily selects the globally shortest available edge, rejecting any edge that would give either endpoint degree > 2 or close a premature sub-cycle. This is structurally equivalent to

Kruskal's MST algorithm [3] adapted to Hamiltonian path constraints. The resulting chains are stitched into a tour.

2.2.3 SPN / SPNQ — Nearest Neighbour

SPN implements nearest-neighbour growth [4] from the city with the globally shortest nearest neighbour. SPNQ (the Q suffix denotes bidirectional growth) extends the chain from both ends simultaneously, choosing whichever direction gives the shorter extension.

2.2.4 SPX / SPXQ — Farthest Neighbour

SPX inverts the nearest-neighbour heuristic: at each step, the chain is extended to the farthest unvisited city. The motivation is geometric: nearest-neighbour growth produces tightly wound local spirals that leave distant regions unvisited until late in construction, requiring expensive insertions. SPX forces the chain to span the extremes of the point cloud first, distributing coverage more evenly.

Farthest-neighbour insertion is documented in Gutin & Punnen [5] as the 'farthest insertion heuristic.' The strategy was derived here independently from geometric observation, without prior exposure to this literature — a coincidence that suggests the geometric motivation is sufficiently natural to be rediscovered from first principles. Empirically, SPX outperforms SPN on elongated or multi-clustered instances but underperforms on compact uniform distributions (see Section 3). SPXQ is the bidirectional variant.

2.2.5 SPEN / SPENQ — Exhaustive Multi-Start

SPEN runs SPN from every city as the starting point, retaining the shortest result. SPENQ does the same with SPNQ. These eliminate starting-point bias at the cost of O(n) seeding runs. A BEST mode runs all eight strategies and returns the shortest.

2.3 Refinement: Best-Improvement Node Insertion

After seeding, the following procedure repeats until convergence:

Phase 1 — Best-Improvement Relocation:

```
For every p_k in tour T:
    Compute cost_current(p_k)
    For every non-adjacent edge (p_i, p_j):
        Compute saving(p_k, p_i, p_j)
Find (p*, p_i*, p_j*) = argmax saving over all k, i, j
If saving(p*, p_i*, p_j*) > 0:
    Remove p* from current position
    Insert p* between p_i* and p_j*
    Goto Phase 1          [full recomputation before next move]
Else: proceed to Phase 2
```

Phase 2 — Geometric Edge Uncrossing:

```
For every non-adjacent edge pair (p_i,p_j) and (p_k,p_l):
    If line segments intersect (parametric test):
        Reverse sub-sequence between p_j and p_k  [2-opt move]
        Goto Phase 1
If no crossing found: STOP  [converged]
```

The critical distinction from standard node insertion heuristics [5] is the selection criterion: rather than applying all beneficial relocations in one sweep (first-improvement), the algorithm identifies the single globally best relocation — maximum saving across the full n x (n-2) cost matrix — and makes only that move before recomputing the entire matrix. This is a best-improvement strategy. It produces more deliberate convergence paths (each step is the algorithmically most significant available move) and generates the rich step-by-step animation shown in the visualiser.

Phase 2 implements the 2-opt uncrossing move [6]: crossed edges always violate optimality by the triangle inequality, so reversing the segment between them strictly decreases tour length. Returning to Phase 1 after each uncrossing is essential: uncrossing typically opens new relocation opportunities that Phase 1 would not have found before the uncrossing occurred.

Termination is guaranteed: every Phase 1 move and every Phase 2 move strictly decreases tour length. The number of distinct tours is finite, so the process must converge to a local optimum under both operations.

3. Experimental Results

Table 1 reports results on four standard TSPLIB instances [7], using the Python 3.12 implementation described in Section 4, on a single CPU core. Known optimal values are from the TSPLIB repository. All eight strategies were evaluated; Table 1 shows OTR results and the best result across all strategies.

Instance	n	Optimal	OTR	% above	BEST	% above	Time
Berlin52	52	7,542	7,783.0	+3.20%	7,783.0	+3.20%	13.9ms
ST70	70	675	647.3	-4.10%*	647.3	-4.10%*	31.5ms
Eilon76	76	538	562.0	+4.46%	555.1	+3.18%	54.1ms
KroA100	100	21,282	22,281.5	+4.70%	22,281.5	+4.70%	87.8ms

Table 1: Results on TSPLIB benchmark instances. BEST denotes the shortest result across all eight seeding strategies. * ST70 result is below the published TSPLIB value; this arises because our implementation uses exact floating-point Euclidean distances while the TSPLIB optimal was verified under integer-rounded distances — a known discrepancy for small instances at this coordinate scale [7].

OTR is the best or joint-best strategy on three of four instances. On Eilon76, SSF outperforms OTR (3.18% vs 4.46%): Eilon76 has a uniform interior distribution with no strong convex extremes, reducing the advantage of hull-edge seeding. On KroA100 and Berlin52, OTR's geometrically guaranteed hull edges give the refinement loop a substantially stronger starting point.

SPX (farthest neighbour) performs poorly on Berlin52 (+17.1%) and KroA100 (+27.9%) but competitively on ST70 (+5.0%) and Eilon76 (+5.9%). This confirms the geometric hypothesis: SPX is most beneficial on elongated or sparse instances where spanning the extremes first prevents costly late insertions.

Convergence requires 49-96 total steps (relocation + uncrossing moves) across all instances tested, with CPU times of 13.9ms to 87.8ms on a Python implementation without any optimisation. The O(n^2) per-step cost is visible in the scaling from 52 to 100 cities.

4. Computational Characteristics and Applicability

The refinement loop's dominant operation — computing the full n x (n-2) insertion cost matrix at each step — is embarrassingly parallel: every cell is independent, involving only three distance lookups and two arithmetic operations. On a GPU or SIMD processor, the entire matrix can be computed simultaneously, followed by a parallel max-reduction to identify the best move. On a modern GPU, 100-city instances would converge in sub-millisecond time. This makes the algorithm feasible for real-time use on embedded processors with small GPU or SIMD units — drone navigation controllers, robotic arm path planners, PCB drill path optimisers, autonomous vehicle routing chips.

A second advantage is architectural simplicity. State-of-the-art exact solvers such as Concorde [2] achieve provably optimal solutions through integer linear programming combined with problem-specific cutting planes — Gomory cuts, subtour elimination constraints, blossom inequalities, and clique-tree inequalities accumulated over decades of specialised research. This machinery is powerful but complex: it requires an LP solver, branch-and-bound infrastructure, and cuts that depend on the polyhedral structure of the TSP polytope. It cannot run on constrained embedded hardware.

Abhijit's Algorithm requires only: a distance matrix (n^2 values), an ordered array representing the current tour, and elementary arithmetic. The complete algorithm can be implemented in approximately 200 lines of any language. It is fully deterministic, every step is geometrically interpretable, and there are no tunable parameters.

Critically, the algorithm is agnostic to the meaning of 'distance.' The distance matrix can encode Euclidean distance, road-network travel time, terrain-weighted cost, asymmetric travel costs, or any other metric — the algorithm does not change. This generality is unavailable to Concorde-style solvers, which depend on Euclidean geometry for their most powerful cuts.

The natural application domain is embedded or latency-constrained routing: problems where a solution within 5% of optimal, computed in milliseconds on modest hardware, is more valuable than an exact solution computed in minutes on a server. TSP and its variants underlie PCB drill path optimisation, wire bonding in semiconductor manufacturing, telescope observation scheduling, DNA fragment assembly, last-mile delivery routing, and robotic pick-and-place sequencing. In all of these, the algorithm's operating point — fast, near-optimal, hardware-friendly, parameter-free — is a practical advantage.

5. Implementation and Visualiser

The algorithm was originally developed in Delphi Pascal using visible TStringGrid components as data structures for the distance matrix, insertion cost matrix, and tour sequence. This design — on-screen grids as working memory — enabled direct visual inspection of every intermediate state during development, and was instrumental in discovering the algorithm's key properties by observation.

The web implementation re-implements the algorithm in Python 3.12, served via FastAPI on a Digital Ocean server. The complete sequence of algorithm steps is pre-computed in a single burst and returned as a JSON payload; the React frontend controls animation pacing locally. This architecture eliminates streaming overhead and allows smooth user-controlled playback including per-step navigation and scrubbing.

Every step produces a plain-language explanation generated at the time of the move. A relocation step produces, for example: 'City 7 saves 41 units by moving between cities 3 and 12.

This is the globally best single move available.' An uncrossing step produces: 'Edges 4-9 and 2-11 were crossing. Crossed edges are never optimal — reversing the segment between them saves 83 units.' This narration is designed for non-specialist audiences.

The visualiser supports: random puzzle generation with reproducible seeds; import of TSPLIB benchmark instances with known optimal distances; human route drawing with real-time distance accumulation; comparison of human and algorithm solutions overlaid on the same canvas; leaderboard submission; step-by-step playback with scrubbing; and five animation speed levels from 'study' (2.5 seconds per step) to 'instant.' The application is available at tsp.uncledroid.app.

6. Pedagogical Value

Existing TSP visualisers typically animate standard algorithms without explanation. To our knowledge, no existing publicly available tool provides step-by-step plain-language narration of a TSP algorithm's reasoning, allows the user to pause, rewind, and step through individual moves, and simultaneously supports competitive human play against the algorithm.

The combination of competitive engagement (human vs. algorithm, scored as a percentage of optimal) and algorithmic transparency (every move explained in the language of the move's geometric motivation) makes the visualiser suitable for undergraduate algorithms courses, operations research introductions, and general public outreach. The TSPLIB integration allows students to interact with the same benchmark instances used in decades of academic research, contextualising the algorithm's performance relative to the state of the art.

The origin of the algorithm — geometric intuition without literature — is itself pedagogically useful: it demonstrates that several of the field's core heuristic principles (hull seeding, 2-opt uncrossing, best-improvement search) are sufficiently natural to be rediscovered from first principles, suggesting they reflect genuine geometric structure in the problem rather than arbitrary theoretical constructs.

7. Related Work

The nearest-neighbour heuristic [4] is among the oldest TSP construction methods, with a worst-case approximation ratio of O(log n) [8]. The convex hull heuristic — using the hull as a starting tour and inserting interior points — is well established [9]; OTR produces the same initial structure through cross-product predicate testing rather than explicit hull construction, independently derived.

Node insertion heuristics are surveyed in [5]. Cheapest insertion selects the unvisited city minimising insertion cost; our Phase 1 differs in applying to cities already in the tour (relocation), using a best-improvement rather than first-improvement criterion, and recomputing the full cost matrix after every single move.

Farthest-neighbour insertion is documented in Gutin & Punnen [5] as the 'farthest insertion heuristic,' where it selects the unvisited city farthest from the current tour for insertion. SPX as presented here applies the farthest-neighbour principle to chain growth rather than insertion, and was derived independently from geometric observation — confirmed as an independent rediscovery upon community feedback following initial publication of this preprint.

The 2-opt move was introduced by Croes [6] and remains among the most effective local search operations for TSP. Phase 2 is a geometric implementation derived from the direct observation that crossed edges are suboptimal, without reference to the combinatorial 2-opt framework.

LKH [10] and Concorde [2] represent the state of the art, achieving solutions within 0.5% of optimal on instances of thousands of cities through Lin-Kernighan moves and cutting-plane branch-and-bound respectively. This algorithm is not competitive at that scale; its interest lies in independent derivation of effective principles, architectural simplicity enabling hardware deployment, and the pedagogical application.

8. Conclusion

We have presented Abhijit's Algorithm, a TSP heuristic achieving within 3.2-4.7% of known optimal on standard benchmark instances of 52-100 cities, converging in under 100ms on a single CPU core. The key contributions are: OTR convex hull seeding via cross-product predicate (independently derived); best-improvement node insertion with full matrix recomputation per step; SPX farthest-neighbour construction; and an interactive visualiser with plain-language step narration.

The algorithm's O(n^2)-per-iteration structure is embarrassingly parallel, making it amenable to GPU/SIMD acceleration for real-time embedded routing. Its architecture — a distance matrix and elementary arithmetic — generalises to any metric TSP variant without modification, contrasting with exact solvers that depend on Euclidean-specific cutting-plane theory.

The development methodology — building a visual interactive tool and using it to discover algorithm properties experimentally — proved effective for heuristic research and is recommended as a complement to theory-first approaches.

The algorithm and visualiser are available at tsp.uncledroid.app. The Python and original Delphi Pascal implementations are available from the author.

References

[1] Karp, R. M. (1972). Reducibility among combinatorial problems. In Complexity of Computer Computations, pp. 85-103. Plenum Press.

[2] Applegate, D., Bixby, R., Chvatal, V., & Cook, W. (2006). The Traveling Salesman Problem: A Computational Study. Princeton University Press.

[3] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. American Mathematical Society, 7(1), 48-50.

[4] Rosenkrantz, D. J., Stearns, R. E., & Lewis, P. M. (1977). An analysis of several heuristics for the traveling salesman problem. SIAM Journal on Computing, 6(3), 563-581.

[5] Gutin, G., & Punnen, A. P. (Eds.). (2002). The Traveling Salesman Problem and Its Variations. Kluwer Academic Publishers.

[6] Croes, G. A. (1958). A method for solving traveling-salesman problems. Operations Research, 6(6), 791-812.

[7] Reinelt, G. (1991). TSPLIB — A Traveling Salesman Problem Library. ORSA Journal on Computing, 3(4), 376-384.

[8] Johnson, D. S., & McGeoch, L. A. (1997). The traveling salesman problem: A case study in local optimization. In Local Search in Combinatorial Optimization. Wiley.

[9] Ong, H. L. (1984). Approximate algorithms for the traveling salesman problem. Computers & Operations Research, 11(3), 259-267.

[10] Helsgott, K. (2000). An effective implementation of the Lin-Kernighan traveling salesman heuristic. European Journal of Operational Research, 126(1), 106-130.