A Fast Quasi-Linear Heuristic for the Close-Enough Traveling Salesman Problem

1 Introduction

The *Close-Enough Traveling Salesman Problem* (CETSP) is a continuous generalization of the classical Traveling Salesman Problem (TSP), where each target is associated with a circular neighborhood that the salesman must intersect rather than a precise point. The CETSP thus combines the discrete optimization of the visiting sequence with the continuous optimization of visiting positions, and it naturally models path-planning tasks such as radio meter reading, drone-based inspection, and robotic welding (Lei and Hao 2024; Di Placido, Archetti, and Cerrone 2022).

Despite recent progress, most high-performing CETSP algorithms rely on population-based or local-improvement metaheuristics that achieve strong solution quality at the expense of high computational cost. In this work, we present a new heuristic for the CETSP that is inspired by the quasi-linear *pair-center algorithm* for the Euclidean TSP proposed by Formella (2024). While originally designed for point-based TSP instances, the pair-center concept extends naturally to the CETSP once disk geometry and intersection logic are handled properly. Our resulting method runs in expected $O(n \operatorname{polylog} n)$ time, making it markedly faster and more scalable than current state-of-the-art CETSP solvers. Although it does not seek to outperform metaheuristic approaches in final tour length, its runtime efficiency allows application to extremely large or real-time instances that remain intractable for existing techniques.

2 Related Work

3 Algorithm

Our approach follows the overall structure of the quasi-linear *pair-center algorithm* introduced by Formella (2024) for the Euclidean TSP, but extends it to handle circular neighborhoods as in the Close-Enough Traveling Salesman Problem. Conceptually, the method retains the same two-phase organization: a *clustering phase* that builds a hierarchical representation of the instance, and a *construction phase* that incrementally expands a feasible tour from this hierarchy.

In the clustering phase, the algorithm recursively merges the closest pair of geometric objects into a new composite "proxy" object until a single hierarchy remains. In the CETSP setting, these objects are circles rather than points as in Formella (2024). The result is a binary clustering tree whose internal nodes represent proxy circles.

The construction phase then traverses this tree to build a closed tour. As in the original pair-center approach, the tour is dynamically maintained so that it remains feasible at every step. However, the continuous nature of the CETSP introduces two additional challenges: first, multiple circles may correspond to a single effective tour point when their feasible regions overlap; second, each tour point admits continuous local optimization within its circle. To address these, the algorithm performs lightweight dynamic updates—reinserting or locally re-optimizing tour points—while preserving near-linear expected runtime.

Overall, the method retains the speed and structural simplicity of the pair-center algorithm while incorporating the geometric flexibility required for close-enough constraints. Detailed definitions, data structures, and optimization steps are presented in the following sections.

3.1 Data Structures

We make extensive use of geometric search structures to maintain and query spatial relationships efficiently. In particular, we employ an R*-tree to store bounding boxes of the current set of circles, enabling logarithmic-time expected queries for nearest neighbors, insertions, and deletions. The resulting operations have expected $O(n \operatorname{polylog} n)$ runtime; see Formella (2024) for a detailed review and analysis.

3.2 Clustering Phase

The clustering phase closely follows the structure of the pair-center algorithm described by Formella (2024), with appropriate modifications to handle circular regions instead of point targets. As in the original approach, the goal is to construct a hierarchical binary tree over all targets by iteratively merging the closest pair of geometric objects, replacing them with a single representative (a *proxy circle*). The leaves of the resulting tree correspond to the original input disks, and each internal node stores its proxy circle and the associated merge distance.

Preprocessing. Before clustering, we remove redundant circles—specifically, any circle that is completely contained within another. This is performed by approximating each circle with its axis-aligned bounding box, sorting all boxes by decreasing radius, and using the R*-tree to query for boxes that fully cover the current one. To mitigate alignment bias from axis-aligned boxes, we randomize the instance by applying a random global rotation to all circle centers. This randomization slightly improves expected merging quality while maintaining near-linear preprocessing cost.

Merge Step. Let c_1 and c_2 denote the two circles chosen for merging. Their *effective distance* is defined as

$$d(c_1, c_2) = \|p_1 - p_2\| - r_1 - r_2,$$

that is, the Euclidean gap between their boundaries. Intuitively, smaller values indicate strong overlap or proximity, which makes the pair more representative for early merging.

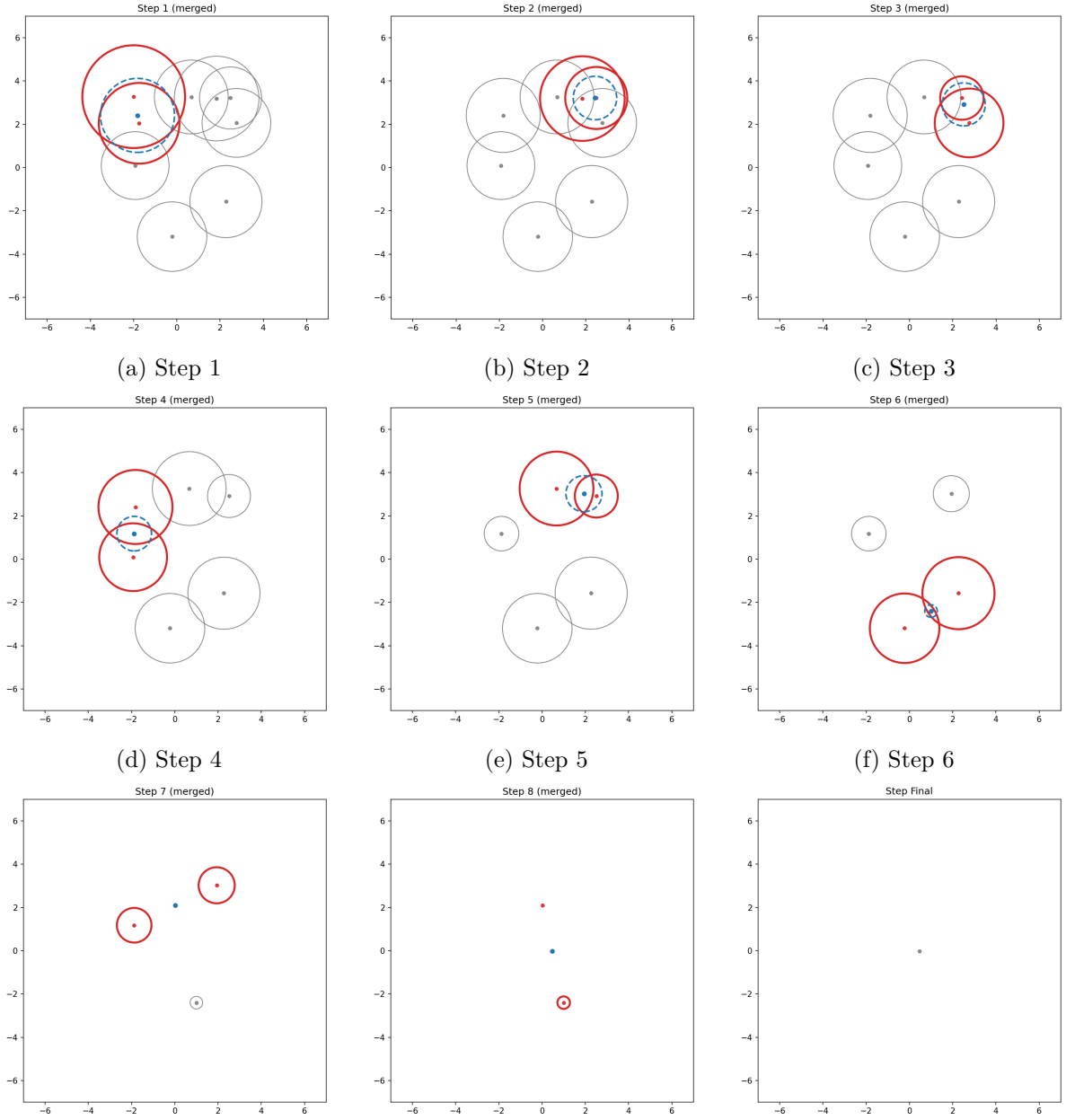
The merge procedure operates as follows:

1. Select the pair (c_1, c_2) with the minimal $d(c_1, c_2)$ value.
2. Remove c_1 and c_2 from the R*-tree and from the corresponding neighbor structures.
3. Compute their proxy circle c'.
4. Insert c' into the R*-tree and update all relevant neighbor lists.
5. Insert c' as an internal node in the binary tree, with c_1 and c_2 as its children.

To maintain efficiency, we adopt the same general mechanism as Formella (2024). Each active circle keeps a list of its current nearest neighbors, and a global min-heap stores tuples of the form $(d(c_i, c_j), i, j)$. When a pair is merged, only the affected neighbors of c_i and c_j are updated—an operation bounded by a constant factor related to the planar kissing number (six in $\mathbb{R}^2$).

Approximate Neighbor Search. Because the effective distance $d(c_1, c_2) = \|p_1 - p_2\| - r_1 - r_2$ depends on both position and radius, exact nearest-neighbor search under this metric is impractical. Instead, we perform approximate search by querying the k (small constant) closest bounding boxes (from a certain center p_1) and evaluating $d(c_1, c_2)$ explicitly for those candidates. This is inexact in two ways: boxes do not perfectly encapsulate the circles, and the distance

any value of $\|p_1 - p_2\| - r_2$ not greater than 0 are treated the same. However, in practice, small constant values of k suffice to preserve both runtime efficiency and clustering quality.



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6

(g) Step 7 (h) Step 8 (i) Final

Figure 1: Visualization of the clustering phase for a sample instance with $n = 9$ disks. Each subfigure shows one merge step, where the two closest circles (highlighted) are replaced by their proxy circle (dashed blue outline). The final frame (bottom right) shows the single remaining cluster.

Proxy Circle Computation. When merging two circles $c_1 = (p_1, r_1)$ and $c_2 = (p_2, r_2)$, we define their *proxy circle* $c' = (p', r')$ as a compact approximation of the lens-shaped region covered by their union. In contrast to simply using the midpoint between centers, our implementation constructs c' geometrically along the line connecting p_1 and p_2.

Specifically, let $d = \|p_1 - p_2\|$ denote the center distance. If a circle covers another circle, we simply take the smaller circle. If the circles do not intersect at all ($d \geq r_1 + r_2$), we take the

midpoint between the circle boundaries and radius 0 (i.e. a single point).

Otherwise, we construct a proxy circle. We determine the two points where the line connecting p_1 and p_2 intersects the boundaries of the two circles, and place the new center p' at the midpoint between these boundary points. The resulting circle approximates the geometric "lens" formed by the intersection. We compute both the overlap depth

$$\delta = \tfrac{1}{2}(r_1 + r_2 - d)$$

and the half-chord length

$$h = \sqrt{r_1^2 - a^2}, \quad a = \frac{r_1^2 - r_2^2 + d^2}{2d},$$

which jointly define a plausible range for the new radius. To introduce mild stochastic diversity, the final radius r' is drawn uniformly from the interval $[\delta, h]$. This randomization helps avoid degenerate clustering patterns and allows multiple independent runs to explore slightly different hierarchical structures.

Although this proxy construction is heuristic, it consistently yields stable and geometrically balanced merges in practice, while maintaining constant-time computation and preserving the algorithm's overall quasi-linear runtime.

3.3 Construction Phase

The construction phase of our algorithm follows the overall strategy of the pair-center algorithm described by Formella (2024), with modifications to handle circular regions instead of point targets. The core objective is to insert each original circle into the tour by iteratively decomposing the hierarchical structure, maintaining a closed tour where a single tour point can represent multiple covered circles.

Tour Initialization and Expansion. The process begins by initializing the tour with a single tour point corresponding to the center point of the root of the binary tree. The root proxy circle itself is inserted into a global max-heap, which is ordered by the merge distance stored at the center node. The algorithm proceeds iteratively, expanding the subtrees until the max-heap is empty.

Circle Decomposition and Removal. In each step, the maximal entry in the heap—a circle $c = (O, r)$—is popped from the heap. This circle c is then removed from the tour structure. This removal is handled conditionally: if the tour point that contained c represented only this single circle, the tour point is entirely removed from the sequence. If the tour point represented a list of multiple circles (including c), only c is removed from that list, and the tour point remains active.

Insertion of Generating Circles. The process then focuses on inserting the two generating circles, c_i and c_j, that formed the proxy circle c. Insertion is performed by checking two possibilities:

- **Check existing points:** The first priority is to assign the circle to an existing tour point to incur zero additional travel cost. This is possible if the existing tour point is not greater than the circle's radius. An R-tree of tour points is used for efficient querying of suitable existing points.
- **Create new tour point:** If no existing point can cover the circle, a new tour point P must be created inside the circle (O, r) and inserted into an existing tour segment AB. The optimal placement of P must minimize the added tour length, $\Delta L = |AC| + |BC| - |AB|$.

Approximate Tour Placement. Finding the globally optimal insertion point is computationally intensive (this is Alhazen's problem). Therefore, we use an approximation: we first select the k nearest tour segments AB using an R-tree of tour segments. For each segment, we put a provisional new point P at the bisection of the angle $\angle AOB$, where O is the circle center. We then compare all points P to find the best-fitting segment. This method is a fast and effective approximation of the optimal point (Lei and Hao 2024). Optionally, we can also apply a single Newton-Raphson iteration to refine this approximate placement. If the generating circles c_i or c_j are themselves proxy circles (internal nodes), they are inserted into the max-heap for further hierarchical expansion. The algorithm terminates once the max-heap is empty, indicating that all original leaf-node circles have been assigned to tour points, resulting in a complete tour.

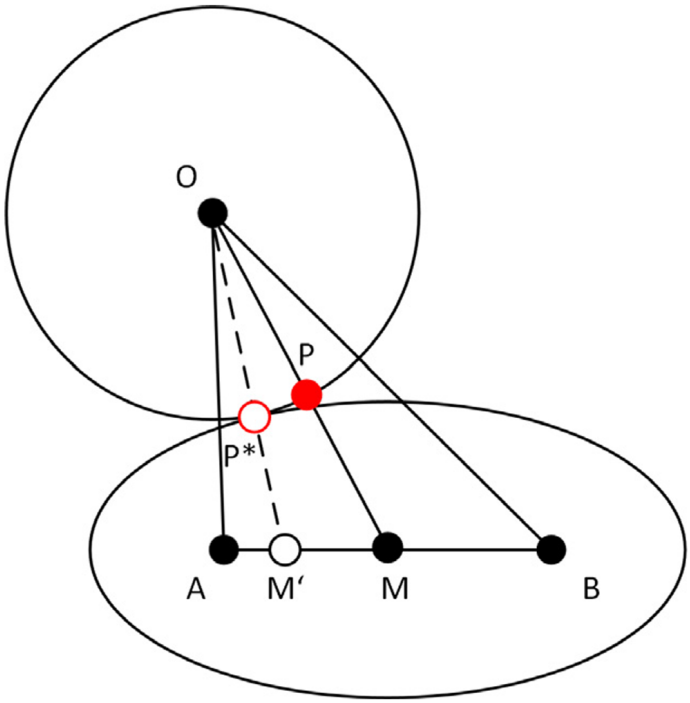


Figure 2: Approximate solution to the Alhazen problem for placing a new tour point within a circle. The bisection of angle $\angle AOB$ is used as a fast approximation of the optimal insertion point (Lei and Hao 2024).

3.4 Point Optimization

During our construction, some tour points may be geometrically and structurally suboptimal. To further improve solution quality without sacrificing runtime scalability, we introduce two lightweight local optimization mechanisms: *reinsertion* and *point reoptimization*. Both are designed to be incremental and self-regulating, preserving the algorithm's overall quasi-linear behavior while opportunistically improving the tour.

Reinsertion. During construction, tour points are created incrementally and may become suboptimal as the tour evolves. To correct this, we periodically remove a tour point and reinsert all circles associated with it as if they were new circles. This procedure allows local restructuring of the tour without recomputing the full solution.

Because global reinsertion is computationally expensive, we perform it selectively based on a heuristic "energy" model. Each tour point maintains an energy value E that quantifies its local instability. Whenever a new circle is inserted into a tour point, we add $+3$ energy to that point and -1 from each of its immediate neighbors. When the energy of a tour point reaches zero, it

triggers a reinsertion event: the point is removed from the tour, and all circles it represented are reinserted individually following the same rules described in the construction phase.

This mechanism adaptively prioritizes reinsertion in regions of the tour where structural changes are concentrated, while naturally suppressing redundant updates in stable areas. Our policy also limits the number of reinsertion events so that we can preserve the near-linear scaling of the algorithm.

Point Reoptimization. A second optimization addresses the geometric placement of existing tour points. Each point is originally positioned based on the local approximation used at its first insertion, which may no longer be optimal once additional circles are added or neighboring segments shift.

The reoptimization problem is as follows. Given a tour segment (A, B) and a tour point P representing a set of circles $\{(O_i, r_i)\}$, we wish to find a new position $P^\star$ within the feasible intersection region of these circles that minimizes

$$\Delta L = |AP^\star| + |BP^\star| - |AB|.$$

This optimization is nontrivial even for a single circle, and becomes analytically intractable for multiple overlapping disks. We therefore adopt a fast geometric approximation that suffices for local improvement.

1. If the segment AB intersects the common feasible region of all circles, the optimal solution incurs zero additional cost; in this case, we simply take a point in the intersection.
2. Otherwise, the optimal $P^\star$ must lie on the boundary of the intersection region. We approximate this by performing a single gradient-descent step: we compute the gradient of ΔL with respect to P and move P maximally along that direction until it would leave any of the circles. This yields a fast, stable adjustment that effectively reduces local tour length.

To control runtime, reoptimization is triggered according to a simple exponential schedule. Specifically, a tour point is reoptimized whenever the number of insertions of circles corresponding to that point reaches 2^n for some integer n. This ensures that frequently updated regions receive more attention while overall computational effort remains bounded.

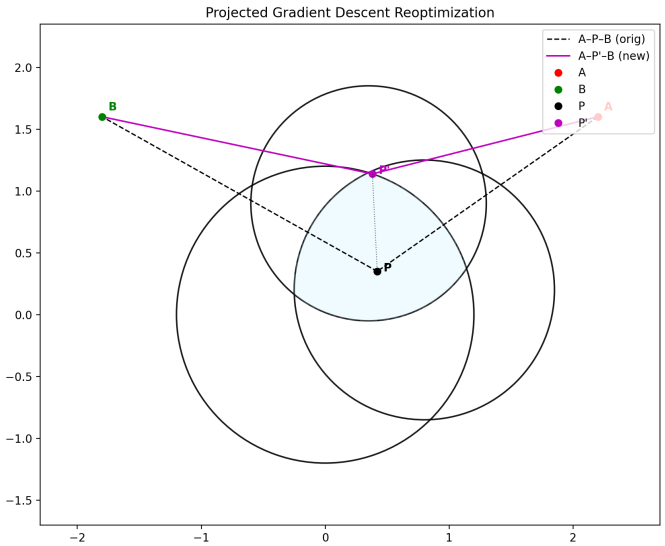


Figure 3: Illustration of the point reoptimization step. The new position $P^\star$ is obtained by projecting the current point P toward the gradient direction of decreasing tour cost, constrained to remain within all corresponding circles.

Together, these two mechanisms maintain geometric consistency of the tour and correct local inefficiencies introduced during hierarchical construction. Both procedures are incremental, spatially localized, and amortized over the construction process, contributing negligibly to total runtime while providing measurable quality improvements.

3.5 Runtime Analysis

We analyze the expected running time of the algorithm. As in the clustering phase, we measure time in terms of basic geometric operations (R-tree queries, insertions, deletions, and heap operations). We assume the standard randomized/average-case performance bounds for R-trees (or similar balanced spatial search structures): each query, insertion or deletion takes expected $O(\log n)$ time. All claims below are in expectation with respect to the randomization used in preprocessing and the probabilistic behavior of the spatial index.

Single-query cost. Retrieving the k nearest bounding boxes (or segments or points) from the R-tree costs expected $O(\log n + k)$. We repeatedly use this fact to bound the cost of approximate nearest-neighbor and segment queries.

3.5.1 Preprocessing phase

We first sort the n axis-aligned bounding boxes by decreasing radius (or equivalently by the contained circle radius). This sorting costs $O(n \log n)$.

After sorting, we process each box once: for the current box we query the R-tree for the k nearest boxes (constant k returned by the spatial index) and test those candidates for full containment; any box found to be fully contained is removed. Let k_i denote the number of candidates examined for the i-th query. The total query cost is

$$\sum_{i=1}^{n} \left(O(\log n) + k_i\right) \;=\; O(n \log n) + \sum_{i=1}^{n} k_i.$$

It remains to bound $\sum_i k_i$ in expectation. By the random global rotation applied during preprocessing, an axis-aligned covering box has a constant (geometric) probability that its inscribed circle also covers the target circle; a simple planar-angle argument gives this probability as a positive constant $= \pi/4$. Hence each successful removal corresponds in expectation to only a constant number of candidate examinations. Since at most n boxes can be removed overall, we obtain $E\left[\sum_i k_i\right] = O(n)$.

Combining terms yields an expected preprocessing cost of

$$T_{\text{preproc}} = O(n \log n) + O(n) = O(n \log n).$$

3.5.2 Clustering phase

The clustering phase performs $n - 1$ merge iterations. In each iteration we:

- extract the current closest pair from a min-heap (expected $O(\log n)$),
- remove two entries from the R-tree (expected $O(\log n)$ each),
- compute a proxy circle (constant-time geometric work),
- insert the proxy into the R-tree (expected $O(\log n)$), and
- update a constant number of neighbor lists (each update involves at most a constant number of R-tree / heap operations and therefore costs expected $O(\log n)$).

Summing the dominant $O(\log n)$ costs over the $O(n)$ merges yields

$$T_{\text{cluster}} = O(n \log n).$$

3.5.3 Construction phase (excluding point-optimizations)

During construction we perform $O(n)$ heap pops and insertions (each costing expected $O(\log n)$), and for each insertion we perform a bounded number of R-tree queries or updates (nearest-point checks, segment queries, and structure updates), each costing expected $O(\log n)$. Thus the dominant cost of construction (excluding point-optimizations) is

$$T_{\text{construct}} = O(n \log n).$$

3.5.4 Cost of reinserts

During the construction phase, we also use the point reinsertion mechanism described above. Each circle reinserted costs $O(\log n)$ expected time, as described in the construction phase. To bound the runtime of our reinsertion policy, we must bound the total number of reinsertions performed. Note that this bound depends on the number of *circles* that are reinserted, not the number of *tour points* that are reinserted.

We prove this using a *gadget problem*. In this gadget problem, there are no circle deletions and energy subtractions can go into any node, not just neighbors. This also allows us to discount the tour structure entirely, and allow our process to work on a infinite set of abstract nodes. Note that these changes can only increase the number of reinsertions performed, so a bound on the gadget problem also bounds our actual reinsertion cost.

Gadget problem description. We consider a collection of nodes (finite or infinite in number), each of which is associated with two integer-valued parameters:

$$E(v) \in \mathbb{Z}_{\geq 0} \quad \text{(energy)}, \qquad W(v) \in \mathbb{Z}_{\geq 0} \quad \text{(weight)}.$$

Initially, all nodes satisfy $E(v) = W(v) = 0$.

We define an *operation* as the following sequence of updates:

1. *Insertion.* Select a node v_+ and perform the update

$$E(v_+) \mathrel{+}= 3, \qquad W(v_+) \mathrel{+}= 1.$$

2. *Energy reduction.* Then, twice, select (possibly different) nodes v_- and perform the update

$$E(v_-) \mathrel{-}= 1.$$

Whenever a node v reaches energy $E(v) = 0$, we say that v *resets*. Let $w = W(v)$ be its weight at the moment of reset. Upon reset, we perform:

$$W(v) \leftarrow 0,$$

and immediately execute w new operations as defined above. These induced operations are called *extra operations*.

More concretely, imagine a stack of updates. Initially, we push $3n$ updates onto the stack, alternating between two energy reductions and one insertion. We then repeatedly pop updates from the stack and execute them. Whenever an energy reduction causes a node to reset, we push $3w$ new operations onto the stack in the same manner, where w is the weight of the node at reset time. The total number of operations is equal to the number of insertions performed.

Suppose we begin with n *original operations* (executed according to the above rule, including all recursively spawned extra operations). Let X denote the total number of extra operations generated in the entire process. Show that the process always terminates (i.e. X is finite) and prove that $X \in O(n)$.

Proof of gadget problem bound. Start with n initial operations. Let X be the total number of extra operations ever spawned. Then the process terminates and

$$X \leq 2n.$$

We proceed by induction on the number of initial operations performed.

Base case: $k = 1$. Execute the first initial operation: it does one $(+3)$ (on some node) and two (-1) updates. No node can reach energy 0 as a result of this single operation, because the only possible decrements are the two (-1)s produced by this operation (they can reduce the node that received $(+3)$ by at most 2, leaving energy ≥ 1). Hence no extra operation can be spawned. So after the first initial op finishes (including any extras it might spawn—there are none) we have a finite prefix, and the number of extra ops produced so far $X_1 = 0 \leq 2 \cdot 1$.

Inductive hypothesis. Fix $k \geq 1$. Assume that after performing the first k initial operations (together with all extra ops they spawn) the process finishes a finite prefix and the total number of extras so far satisfies

$$X_k \leq 2k.$$

Inductive step. Suppose for contradiction that after performing the first $k + 1$ initial operations (and following all cascading extra ops) either (a) the process does not terminate, or (b) it terminates but the total number of extra ops X_{k+1} produced by the first $k + 1$ initial ops satisfies $X_{k+1} > 2(k + 1)$.

If (a) holds, then the number of extra ops produced is infinite; in particular there is a finite time at which the count of extra ops exceeds $2(k + 1)$. Thus in both (a) and (b) we can find a finite prefix of the run that contains the first $k + 1$ initial operations and whose number of extra ops X' satisfies

$$X' > 2(k + 1).$$

Choose the earliest (i.e., shortest) such finite prefix—earliest in the sense of the first time during the run the extra-op count becomes $> 2(k+1)$. By construction this prefix is finite and contains exactly $(k + 1) + X'$ executed operations (initial + extra).

Now count updates in that prefix:

- Every executed operation contributes one $(+3)$ and two (-1) updates. So the prefix contains total $(+3)$-events equal to $(k + 1) + X'$ and total (-1)-events equal to $2((k + 1) + X')$.
- Each extra operation in the prefix corresponds to a reset that consumed a weight unit which in turn came from a prior $(+3)$. To produce one extra operation (i.e. to consume one weight-unit) the corresponding $(+3)$ must have been fully drained by three (-1) decrements on that node. Thus the decrements that were used to create these X' extra ops are at least $3X'$ in number.

But these decrements must be among the $2\big((k + 1) + X'\big)$ total decrements that actually occurred in the prefix. Therefore

$$3X' \leq 2\big((k + 1) + X'\big).$$

Rearranging:

$$3X' \leq 2k + 2 + 2X' \quad \Longrightarrow \quad X' \leq 2(k + 1).$$

This contradicts $X' > 2(k + 1)$, so our assumption was false.

Therefore neither (a) nor (b) can happen: the run started by the first $k + 1$ initial operations (together with all extras they spawn) must terminate and its number of extra ops satisfies $X_{k+1} \leq 2(k + 1)$. This completes the inductive step.

By induction, for every $k \leq n$ the prefix that contains the first k initial operations (and all extras they spawn) is finite and the number of extra operations produced satisfies $X_k \leq 2k$. Applying this to $k = n$ gives termination for the whole process and the desired bound

$$X \leq 2n,$$

so $X = O(n)$.

3.5.5 Cost of point reoptimizations

A reoptimization for a tour point operates only on the (usually small) list of circles associated to that point. By our exponential scheduling (reoptimize after $1, 2, 4, \ldots$ insertions), the total amount of work spent on reoptimizing a single point over the entire run is $O(s)$ where s is the total number of insertions into that point; summing over all points yields an overall reoptimization cost that is proportional to the total number of insertions, i.e. $O(I) = O(n)$. Thus

$$T_{\text{reopt}} = O(n).$$

3.5.6 Total expected time

Combining the dominant terms,

$$T_{\text{total}} \;=\; O(n \log n).$$

Remarks and caveats.

- The $O(n \log n)$ bound hides constant factors coming from the k-nearest queries, box approximations, random rotations, and neighbor-list management; these are implementation-dependent but do not change the asymptotic claim.
- This means that in practice, the constant log term may have a larger exponent.

References

Di Placido, Andrea, Claudia Archetti, and Carmine Cerrone (Sept. 2022). "A genetic algorithm for the close-enough traveling salesman problem with application to solar panels diagnostic reconnaissance". In: *Computers & Operations Research* 145, p. 105831. ISSN: 03050548. DOI: 10.1016/j.cor.2022.105831. URL: https://linkinghub.elsevier.com/retrieve/pii/S0305054822001095 (visited on 06/05/2025).

Formella, Arno (Oct. 2024). "Quasi-linear time heuristic to solve the Euclidean traveling salesman problem with low gap". In: *Journal of Computational Science* 82, p. 102424. ISSN: 18777503. DOI: 10.1016/j.jocs.2024.102424. URL: https://linkinghub.elsevier.com/retrieve/pii/S1877750324002175 (visited on 10/13/2025).

Lei, Zhenyu and Jin-Kao Hao (Mar. 2024). "An effective memetic algorithm for the close-enough traveling salesman problem". In: *Applied Soft Computing* 153, p. 111266. ISSN: 15684946. DOI: 10.1016/j.asoc.2024.111266. URL: https://linkinghub.elsevier.com/retrieve/pii/S1568494624000401 (visited on 06/05/2025).