

# 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  $O(n \text{ polylog } n)$  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  $O(n \text{ polylog } n)$  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  $c'$ , defined as the minimal enclosing circle covering both  $c_1$  and  $c_2$ . This can be computed in constant time using geometric primitives.
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.

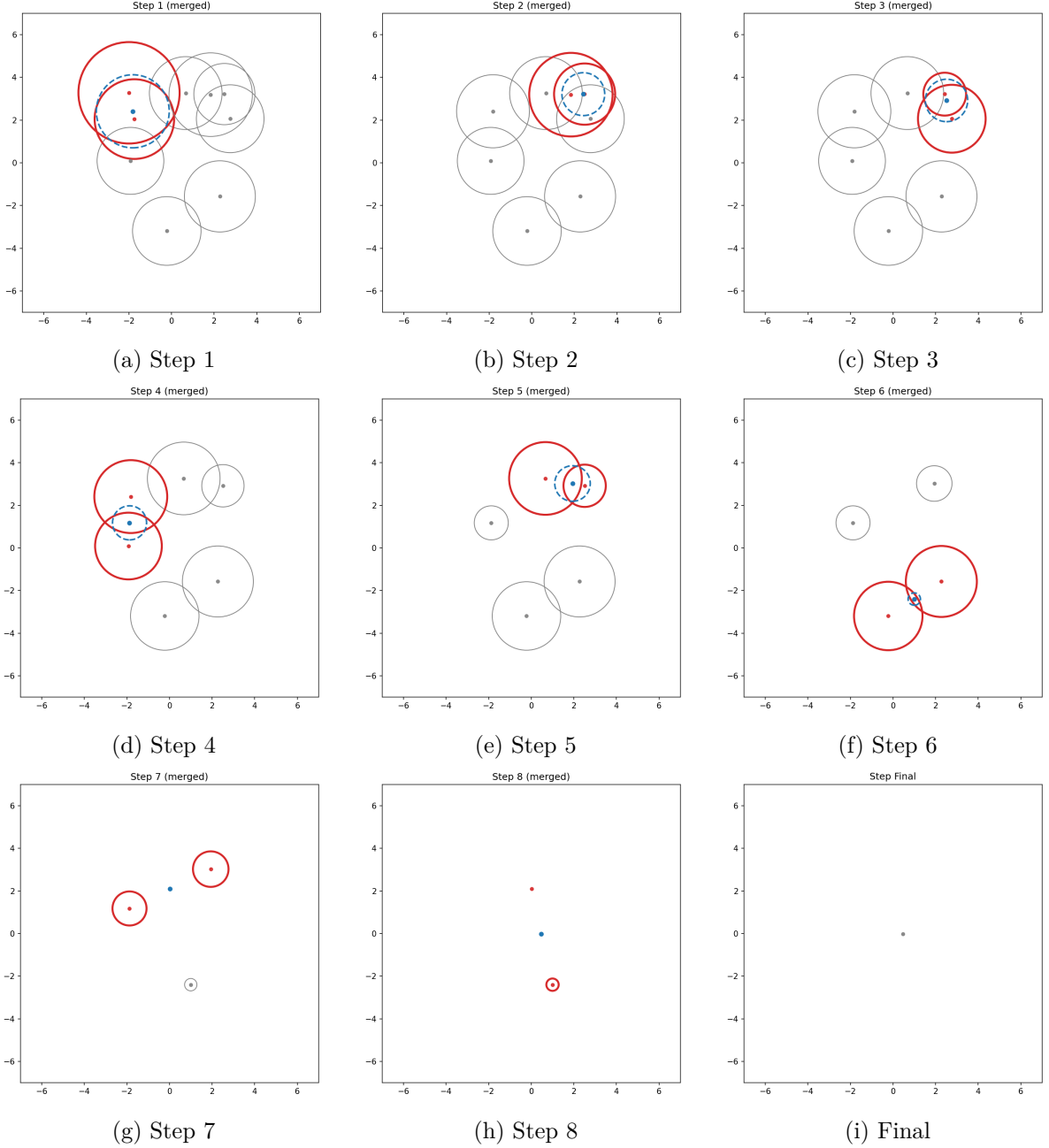


Figure 1: Visualization of the clustering phase for a sample instance with  $n = 9$  disks. Each subfigure shows one merge step, where the two closest circles (highlighted) are replaced by their proxy circle (dashed blue outline). The final frame (bottom right) shows the single remaining cluster.

**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

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

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.

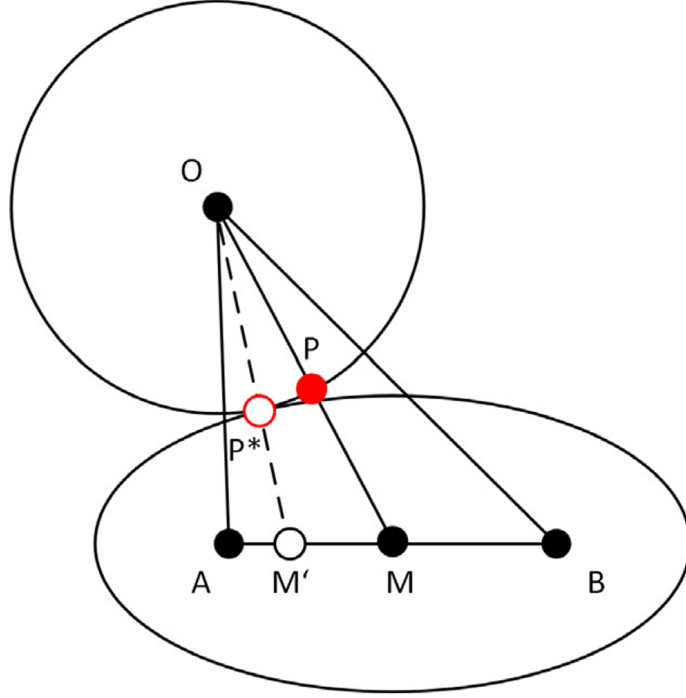


Figure 2: Approximate solution to the Alhazen problem for placing a new tour point within a circle. The bisection of angle  $\angle AOB$  is used as a fast approximation of the optimal insertion point (Lei and Hao 2024).

### 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 near 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  $P^*$  within the feasible intersection region of these circles that minimizes

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

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  $P^*$  must lie on the boundary of the intersection region. We approximate this by performing a single gradient-descent step: we compute the gradient of  $\Delta L$  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.

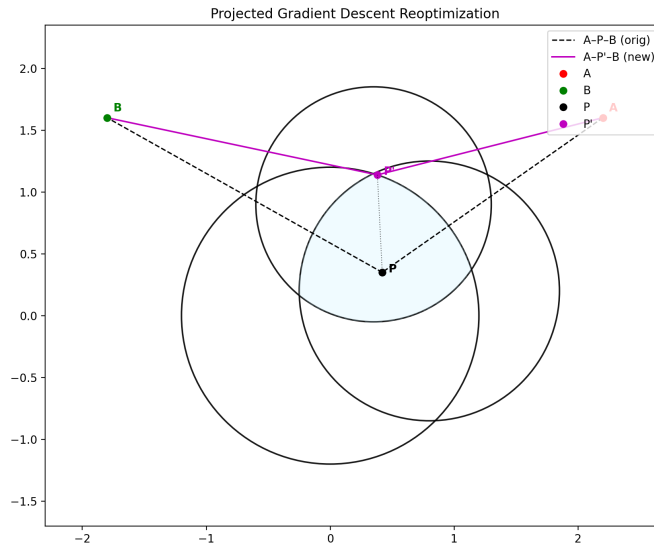


Figure 3: Illustration of the point reoptimization step. The new position  $P^*$  is obtained by projecting the current point  $P$  toward the gradient direction of decreasing tour cost, constrained to remain within all corresponding circles.

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.

## 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).