New Applications of Nearest-Neighbor Chains: Euclidean TSP and Motorcycle Graphs

Nil Mamano[a]
Department of Computer Science,
University of California, Irvine, USA
nmamano@uci.edu

[a] Corresponding author

Alon Efrat
Department of Computer Science,
University of Arizona, Tucson, USA
alon@cs.arizona.edu

David Eppstein
Department of Computer Science,
University of California, Irvine, USA
eppstein@uci.edu

Daniel Frishberg
Department of Computer Science,
University of California, Irvine, USA
dfrishbe@uci.edu

Michael T. Goodrich
Department of Computer Science,
University of California, Irvine, USA
goodrich@uci.edu

Stephen Kobourov
Department of Computer Science,
University of Arizona, Tucson, USA
kobourov@cs.arizona.edu

Pedro Matias
Department of Computer Science,
University of California, Irvine, USA
pmatias@uci.edu

Valentin Polishchuk
Communications and Transport Systems,
ITN, Linköping University, Sweden
valentin.polishchuk@liu.se

Abstract

We show new applications of the nearest-neighbor chain algorithm, a technique that originated in agglomerative hierarchical clustering. We use it to construct the greedy multi-fragment tour for Euclidean TSP in $O(n \log n)$ time in any fixed dimension and for Steiner TSP in planar graphs in $O(n\sqrt{n} \log n)$ time; we compute motorcycle graphs, a central step in straight skeleton algorithms, in $O(n^{4/3+\varepsilon})$ time for any $\varepsilon > 0$.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Computational geometry

Keywords and phrases Nearest-neighbors, Nearest-neighbor chain, motorcycle graph, straight skeleton, multi-fragment algorithm, Euclidean TSP, Steiner TSP

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.51

Related Version A full version of the paper is available on `https://arxiv.org/abs/1902.06875`.

Funding *David Eppstein*: Supported in part by NSF grants CCF-1618301 and CCF-1616248.
Michael T. Goodrich: Supported in part by NSF grant 1815073.
Stephen Kobourov: Supported in part by NSF grants CCF-1740858, CCF-1712119, DMS-1839274, and DMS-1839307.

1 Introduction

The *nearest-neighbor chain* (NNC) technique is used for agglomerative hierarchical clustering [36, 35], and has only seen one other use besides it, in stable matching [22]. In this paper, we use it to speed up an algorithm for Euclidean TSP called multi-fragment heuristic. We also use it to speed up the computation of motorcycle graphs, which is a central step in algorithms for computing straight skeletons. In the full version of the paper [32], we follow this approach for two additional problems: a special case of stable matching and a

geometric coverage problem. All the mentioned problems share a property with agglomerative hierarchical clustering, which we call global-local equivalence, and which is the key to using the NNC algorithm. First, we review the NNC algorithm in its original context.

1.1 Prior work: NNC in hierarchical clustering

Given a set of points, the *agglomerative hierarchical clustering* problem is defined procedurally as follows: each point starts as a base cluster, and the two closest clusters are repeatedly merged until there is only one cluster left. This creates a *hierarchy*, where any two clusters are either nested or disjoint. A key component of hierarchical clustering is the function used to measure distances between clusters. Popular metrics include minimum distance (or single-linkage), maximum distance (or complete-linkage), and centroid distance.

We call two clusters *mutually nearest neighbors* (MNNs) if they are the nearest neighbor of each other. Consider this alternative, non-deterministic procedure: instead of repeatedly merging the two overall closest clusters, merge any pair of MNNs. The non-determinism comes from the fact that there can be multiple pairs of MNNs, and thus several valid choices. Clearly, this may merge clusters in a different order. Nonetheless, if the cluster-distance metric satisfies a property called *reducibility*, this procedure results in the same hierarchy [9, 10, 35]. A cluster-distance metric $d(\cdot, \cdot)$ is reducible if for any clusters A, B, C: if A and B are MNNs, then

$$d(A \cup B, C) \geq \min\left(d(A, C), d(B, C)\right). \tag{1}$$

In words, the new cluster $A \cup B$ resulting from merging A and B is *not* closer to other clusters than both A and B were. The relevance of this property is that, if, say, C and D are MNNs, merging A and B does not break that relationship. The net effect is that MNNs can be merged in any order and produce the same result. Many commonly used metrics are reducible, including minimum–, maximum–, and average–distance, but others such as centroid and median distance are not.

The NNC algorithm exploits this reducibility property, which was originally observed by Bruynooghe [10]. We briefly review the algorithm for hierarchical clustering, since we discuss it in detail later in the context of the new problems. For extra background on NNC for hierarchical clustering, see [36, 35]. The basic idea is to maintain a stack (called *chain*) of clusters. The first cluster is arbitrary. The chain is always extended with the nearest neighbor (NN) of the current cluster at the top of the chain. Note that the distance between clusters in the chain keeps decreasing, so (with an appropriate tie breaking rule) no repeated clusters or "cycles" occur, and the chain inevitably reaches a pair of MNNs. At this point, the MNNs are merged and removed from the chain. Crucially, after a merge happens, the rest of the chain is not discarded. Due to reducibility, every cluster in the chain still points to its NN, so the chain is still valid. The process continues from the new top of the chain.

The algorithm is efficient because each cluster is added to the chain only once, since it stays there until it is merged with another cluster. As we will see in detail for our problems, this bounds the number of iterations to be linear in the input size, with the cost of each iteration dominated by a NN computation.

Recently, the NNC algorithm was used for the first time outside of the domain of hierarchical clustering [23]. It was used in a stable matching problem where the two sets to be matched are point sets in a metric space, and each agent in one set ranks the agents in the other set by distance, with closer points being preferred. In this setting, there is an analogous phenomenon: the stable matching is unique, and it can be obtained in two ways; by repeatedly matching the closest pair (from different sets), or by repeatedly matching MNNs. They used the NNC algorithm to solve the problem efficiently.

1.2 Our Contributions

Our new observation is that this equivalence between merging closest pairs and MNNs is not unique to hierarchical clustering and stable matching: it also applies to the problems in this paper. We coin the term *global-local equivalence* for it. The main thesis of this paper is that NNC is an efficient algorithm for problems with global-local equivalence, which may include even more problems than those discussed here and in the full version of the paper.

Section 3 speeds up the multi-fragment heuristic for Euclidean TSP with a variant of NNC that uses a new data structure that we describe in Section 2, the *soft nearest-neighbor data structure*. We extend this result to Steiner TSP in Section 3.3. Section 4 is on motorcycle graphs. Each section contains background on the corresponding problems.

2 The Soft Nearest-Neighbor Data Structure

Throughout this section, we consider points in $\mathbb{R}^\delta$, for some fixed dimension δ, and distances measured under any L_p metric $d(\cdot,\cdot)$. We begin with a formal definition of the structure and the main result of this section.

▶ **Definition 2.1** (Dynamic soft nearest-neighbor data structure)**.** *Maintain a dynamic set of points, P, subject to insertions, deletions, and soft nearest-neighbor queries: given a query point q, return either of the following:*

- *The nearest neighbor of q in P: $p^* = \arg\min_{p\in P} d(q,p)$.*
- *A pair of points p, p' in P satisfying $d(p,p') < d(q,p^*)$.*

We make a general position assumption: the distances between q and the points in P are all unique. If this is not the case, the query point q can be perturbed slightly.

▶ **Theorem 2.2.** *In any fixed dimension, and for any L_p metric, there is a dynamic soft nearest-neighbor data structure that maintains a set of n points with $O(n \log n)$ preprocessing time and $O(\log n)$ time per operation (queries and updates).*

We label the two types of answers to soft nearest-neighbor (SNN) queries as *hard* or *soft*. A "standard" NN data structure is a special case of a SNN structure that always gives hard answers. However, in light of Theorem 2.2, a standard NN structure would not be as efficient as a SNN structure. For comparison, the best dynamic NN structure in $\mathbb{R}^2$ requires $O(\log^4 n)$ time per operation [12].

Given a point set P and a point q, let p_i^* denote the i-th closest point to q in P. In our implementation, we use the following data structure.

▶ **Definition 2.3** (Dynamic ε-approximate k nearest-neighbor (k-ANN) data structure)**.** *Maintain a dynamic set of points, P, subject to insertions, deletions, and ε-approximate k nearest-neighbor queries: given a query point q and an integer k with $1 \le k \le |P|$, return k points $p_1, \ldots, p_k \in P$ such that, for each p_i, $d(q,p_i) \le (1+\varepsilon)d(q,p_i^*)$, where $\varepsilon > 0$ is a constant known at construction time*[1]*. We assume that $p_1, \ldots, p_k$ are sorted by non-decreasing distance from q.*

We reduce each SNN query to a single k-ANN query with constant ε and k. Once we show this reduction, Theorem 2.2 will follow from the following result by Arya et al. [4]:

[1] Some approximate nearest-neighbor data structures [4] do not need to know ε at construction time, and, in fact, allow ε to be part of the query and to be different for each query. Clearly, such data structures are also valid for our needs.

▶ **Lemma 2.4** ([4]). *In any fixed dimension, for any L_p metric, and for any constant $\varepsilon > 0$, there is a dynamic ε-approximate k nearest-neighbor data structure with $O(n \log n)$ preprocessing time and $O(\log n)$ time per operation (query and updates) for constant k.*

2.1 Soft nearest-neighbor implementation

We maintain the point set P in a dynamic k-ANN structure initialized with an approximation factor ε that will be determined later. The chosen ε will depend only on the metric space. In what follows, q denotes an arbitrary query point and p_i^* the i-th closest point to q in P. Using this notation, $p^* = p_1^*$. Recall the assumption that the points returned by a k-ANN structure are sorted by distance from the query point, so only the first one can be p^*. Queries rely on the following lemma.

▶ **Lemma 2.5.** *Let $p_1, \ldots, p_k$ be the answer given by a k-ANN structure, initialized with approximation factor ε, to a query (q, k). If $p_1 \neq p^*$, then, for each p_i in $p_1, \ldots, p_k$, $d(q, p_i) \leq (1+\varepsilon)^{i-1} d(q, p_1)$.*

Proof. For $i = 1$, the claim is trivial. For $i = 2, \ldots, k$, note that $d(q, p_i^*) \leq d(q, p_{i-1})$. This is because there are at least i points within distance $d(q, p_{i-1})$ of q: $p^*, p_1, \ldots, p_{i-1}$. Thus, $d(q, p_i) \leq (1+\varepsilon) d(q, p_i^*) \leq (1+\varepsilon) d(q, p_{i-1})$. The claim follows by induction on i. ◀

Let $S(q, r_1, r_2)$ denote a closed shell centered at q with inner radius r_1 and outer radius r_2 (i.e., $S(q, r_1, r_2)$ is the difference between two balls centered at q, the bigger one of radius r_2 and the smaller one of radius r_1).

We call a pair (ε, k) *valid SNN parameters* (for a given metric space) if any set of k points inside a shell with inner radius 1 and outer radius $(1+\varepsilon)^{k-1}$ contains two points, p and p', satisfying $d(p, p') < 1/(1+\varepsilon)$.

Suppose that (ε^*, k^*) are valid parameters. Initially, we construct the k-ANN structure using ε^* for the approximation factor. Then we answer queries as in Algorithm 1.

Algorithm 1 Soft nearest-neighbor query.

```
Ask query (q, k*) to the k-ANN structure initialized with ε*.
Measure the distance between each pair of the k* returned points, p_1, ..., p_{k*}.
if any pair (p_i, p_j) satisfies d(p_i, p_j) < d(q, p_1)/(1 + ε*) then
    return p_i, p_j.
else
    return p_1.
```

▶ **Lemma 2.6.** *If (ε^*, k^*) are valid SNN parameters, Algorithm 1 is correct.*

Proof. The algorithm considers two cases. First, if a pair p_i, p_j of points returned by the k-ANN structure satisfies $d(p_i, p_j) < d(q, p_1)/(1+\varepsilon^*)$, p_i and p_j are a valid soft answer to the query, as $d(q, p_1)/(1+\varepsilon^*) \leq d(q, p^*)$.

In the alternative case, no pair among the returned points is at distance $< d(q, p_1)/(1+\varepsilon^*)$. Consider the shell $S(q, d(q, p_1), (1+\varepsilon^*)^{k^*-1} d(q, p_1))$. If we scale distances so that $d(q, p_1) = 1$, then this shell has inner radius 1 and outer radius $(1+\varepsilon^*)^{k^*-1}$. Given that (ε^*, k^*) are valid SNN parameters, if all k^* of the returned points were inside this shell, at least two of them would be at a distance smaller than $1/(1+\varepsilon^*)$. However, without the scaling, this distance

would be smaller than $d(q, p_1)/(1+\varepsilon^*)$, which corresponds to the first case. Thus, at least one of the returned points lies outside the shell. In particular, the farthest point from q, p_{k^*}, satisfies $d(q, p_{k^*}) > (1+\varepsilon^*)^{k^*-1} d(q, p_1)$. By the contrapositive of Lemma 2.5, we have that $p^* = p_1$. ◀

As a side note, a SNN structure always returns a hard answer when queried from a point that is part of the closest pair of the set of points it maintains, as there is no *closer* pair. In this way, a SNN structure can be used to find the closest pair in $(\mathbb{R}^\delta, L_p)$, for constant δ, in $O(n \log n)$ time by querying from every point. This matches the known runtimes in the literature [7].

We left open the issue of finding valid parameters (ε^*, k^*), which we defer to Appendix A. In particular, it is not hard to see that they exist in any metric space $(\mathbb{R}^\delta, L_p)$ (see Lemma A.1).

3 Multi-Fragment Euclidean TSP

Given an undirected, complete graph G with uniquely and positively weighted edges, the *traveling salesperson problem* asks to find a cycle passing through all the nodes of minimum weight. In this section, we consider a classic greedy heuristic for constructing TSP tours, the *multi-fragment* algorithm.

Given two disjoint paths, p and p', in G, we define the cost of connecting them, $cost(p, p')$, as the weight of the cheapest edge between an endpoint of p and an endpoint of p'. We use $p \cup p'$ to denote the path resulting from connecting p and p' into a single path along that edge.

The multi-fragment algorithm works as follows. A path generally has two endpoints, but, in this algorithm, each node starts as a single-node path. While there is more than one path, we connect the two paths such that the cost of connecting them is minimum. We call this the closest pair. Once there is a single path left, we connect its endpoints to complete the tour. We call the tour resulting from this process the *multi-fragment tour*, and the problem of constructing this tour *multi-fragment TSP*.

Euclidean TSP is the variant of TSP where the nodes are points in the plane, and the weights of the edges are the Euclidean distance between the points. Thus, here the goal is to find a closed polygonal chain, called a *tour*, through all the points of shortest length. The problem is NP-hard even in this geometric setting [27], but a polynomial-time approximation scheme is known [3].

The multi-fragment algorithm was proposed by Bentley [5] specifically in the geometric setting. Its approximation ratio is $O(\log n)$ [37, 8]. Nonetheless, it is used in practice due to its simplicity and empirical support that it generally performs better than other heuristics [19, 30, 33, 34, 6].

We are interested in the complexity of computing the multi-fragment tour in the geometric setting. A straightforward implementation of the multi-fragment algorithm is similar to Kruskal's minimum spanning tree algorithm: sort the $\binom{n}{2}$ pairs of points by increasing distances and process them in order: for each pair, if the two points are endpoints of separate paths, connect them. The runtime of this algorithm is $O(n^2 \log n)$. Eppstein [21] uses dynamic closest pair data structures to compute the multi-fragment tour in $O(n^2)$ time (for arbitrary distance matrices). Bentley [5] gives a K-d tree-based implementation and says that it appears to run in $O(n \log n)$ time on uniformly distributed points in the plane. We give a NNC-type algorithm that compute the multi-fragment tour for Euclidean TSP in $O(n \log n)$ in any fixed dimensions. We do not know of any prior worst-case subquadratic algorithm.

3.1 Global-local equivalence for multi-fragment TSP

In this section, we prove global–local equivalence for multi-fragment TSP in general graphs. That is, this result is not restricted to the Euclidean setting.

We say two paths are mutually nearest neighbors if the cost of connecting them is lower than the cost of connecting either with a third path. We consider an alternative algorithm to the multi-fragment algorithm, which connects MNNs paths instead of the closest pair. We use CP-MF to refer to the multi-fragment algorithm, and MNN-MF to refer to this new algorithm.

Clearly, CP-MF is a special case of MNN-MF. We show that any run of MNN-MF outputs the same solution as CP-MF.

Note the similarity between multi-fragment TSP and hierarchical clustering. Instead of merging clusters, we merge paths. The difference is that, in a cluster, it does not matter in which order the points were added when determining the distance to another cluster. In contrast, in a path, it is important which points are the endpoints.

We can see that in multi-fragment TSP we have a notion equivalent to reducibility in agglomerative hierarchical clustering (Equation 1).

▶ **Lemma 3.1** (Reducibility in multi-fragment TSP). *Let a, b, and c be paths in an undirected, complete graph G with positively weighted edges. Then, $cost(a \cup b, c) \geq \min(cost(a, c), cost(b, c))$.*

Proof. The cost of connecting paths is defined as the minimum weight among the edges connecting their endpoints. The claim is clear given that two endpoints of $a \cup b$ are a subset of the four endpoints of a and b. ◀

Given that we have reducibility, we can adapt the proof of global-local equivalence for agglomerative hierarchical clustering presented by Müllner [35] to multi-fragment TSP. The proof is in Appendix B.

▶ **Lemma 3.2** (Global-local equivalence in multi-fragment TSP). *Let G be an undirected, complete graph with uniquely and positively weighted edges. Then, CP-MF and MNN-MF output the same solution.*

3.2 Soft nearest-neighbor chain for multi-fragment Euclidean TSP

In general graphs, computing the multi-fragment tour with CP-MF requires $O(n^2 \log n)$ time, where the bottleneck is to sort the $\Theta(n^2)$ edges from cheapest to most expensive. Given that we have global-local equivalence (Lemma 3.2), this can be improved to $O(n^2)$ by implementing MNN-MF via a straightforward adaptation of the NNC algorithm. NNC builds a chain of paths in order to find MNN paths. We only need to spell out how to find the "nearest neighbor" of a path. We can find it in $O(n)$ time by scanning through the adjacency list of each endpoint. Thus, we can finish the $O(n)$ iterations of the algorithm in $O(n^2)$ time.

Similarly, in the geometric setting, for points in $\mathbb{R}^2$, we can pair NNC with the dynamic NN data structure from [12], which runs in $O(\log^4 n)$ amortized time per operation. Thus, we can compute the multi-fragment tour in $O(n \log^4 n)$ time. However, we do not go into the details of these results and jump directly to our main result, which further improves the runtime in the geometric setting:

▶ **Theorem 3.3.** *The multi-fragment tour of a set of n points in any fixed dimension, and under any L_p metric, can be computed in $O(n \log n)$ time.*

We use a variation of the NNC algorithm that uses a SNN structure instead of the usual NN structure, which we call the *soft nearest-neighbor chain* algorithm (SNNC). For this, we need a SNN structure for paths in the plane (polygonal chains) instead of points. That is, a structure that maintains a set of (possibly single-point) paths, and, given a query path q, returns the closest path to q or two paths in the set which are closer to each other than q to its closest path in the set. The distance between paths is measured as the minimum distance between an endpoint of one and an endpoint of the other, so, for the purposes of this data structure, only the coordinates of the endpoints are important.

A soft nearest-neighbor structure for paths

We simulate a SNN structure for paths with a SNN structure for points. Given a set of paths, we maintain the set of path endpoints in the SNN structure for points. Updates are straightforward: we add or remove both endpoints of the path. Given a query path q with endpoints $\{q_1, q_2\}$, we do a SNN query from each endpoint of the path. If both answers are hard (assuming that the path has two distinct endpoints, otherwise, just the one), then we find the true NN of the path, and we can return it. However, there is a complication with soft answers: the two points returned could be the endpoints of the same path. Thus, it could be the case that we find two closer points, but not two closer paths, as we need. The solution is to modify the specification of the SNN structure for points so that soft answers, instead of returning two points closer to each other than the query point to its NN, return three pairwise closer points. We call this a *three-way* SNN structure. In the context of using the structure for paths, this guarantees that even if two of the three endpoints belong to the same path, at least two different paths are involved.

Lemma 3.4 shows how to obtain a three-way SNN structure for points, Algorithm 2 shows the full algorithm for answering SNN queries about paths using a three-way SNN structure for points, and Lemma 3.5 shows its correctness.

▶ **Lemma 3.4.** *In any fixed dimension and for any L_p metric, there is a three-way SNN structure with $O(n \log n)$ preprocessing time and $O(\log n)$ operation time (queries and updates).*

Proof. Recall the implementation of the SNN structure from Section 2. To obtain a three-way SNN structure, we need to change the values of ε and k to make the shell smaller and k bigger, so that if there are k points in a shell of inner radius 1 and outer radius $(1+\varepsilon)^k$, then there must be at least three points at pairwise distance less than 1. The method described in Section A for finding valid parameters in (R^δ, L_p) also works here. It only needs to be modified so that the area (or surface) of the shell is accounted for twice. Since k and ε are still constant, this does not affect the asymptotic runtimes in Theorem 2.2. ◀

▶ **Lemma 3.5.** *In any fixed dimension, and for any L_p metric, we can maintain a set of n paths in a SNN structure for paths with $O(n \log n)$ preprocessing time and $O(\log n)$ operation time (queries and updates).*

Proof. All the runtimes follow from Lemma 3.4: we maintain the set of path endpoints in a three-way SNN structure S. The structure S can be initialized in $O(n \log n)$ time. Updates require two insertions or deletions to S, taking $O(\log n)$ time each. Algorithm 2 for queries clearly runs in $O(\log n)$ time. We argue that it returns a valid answer. Let q be a query path with endpoints $\{q_1, q_2\}$, and consider the three possible cases:

Algorithm 2 Soft-nearest-neighbor query for paths.

Let q_1 and q_2 be the endpoints of the query path, q.
Let S be a three-way SNN structure containing the set of path endpoints.
Query S with q_1 and q_2.
if both answers are hard **then**
  Let p_1 and p_2 be the respective answers.
  return the closest path to the query path among the paths with endpoints p_1 and p_2.
else if one answer is hard and the other is soft **then**
  Let p be the hard answer to q_1 and (a, b, c) the soft answer to q_2 (wlog). Let P and P' be the two closest paths among the paths with endpoints a, b, and c.
  if $d(p, q) < d(P, P')$ **then**
    return the path with endpoint p.
  else
    return (P, P').
else (both answers are soft)
  Let (a_1, b_1, c_1) and (a_2, b_2, c_2) be the answers to q_1 and q_2.
  return the closest pair of paths among the paths with endpoints $a_1, b_1, c_1, a_2, b_2, c_2$.

- Both answers are hard. In this case, we find the closest path to each endpoint, and, by definition, the closest of the two is the NN of q.
- One answer is soft and the other is hard. Let p be the hard answer to q_1 and (a, b, c) the soft answer to q_2 (wlog). Let P and P' be the two closest paths among the paths with endpoints a, b, and c. If $d(p, q) < d(P, P')$, then, the path with p as endpoint must be the NN of q, because there is no endpoint closer than $d(P, P')$ to q_2. Otherwise, P, P' is a valid soft answer, as they are closer to each other than either endpoint of q to their closest endpoints.
- Both answers are soft. Assume (wlog) that the NN of q is closer to q_1 than q_2. Then, the soft answer to q_1 gives us two paths closer to each other than q to its NN, so we return a valid soft answer. ◀

The soft nearest-neighbor chain algorithm

We use a SNN for paths (Lemma 3.5). In the context of this algorithm, let us think of a SNN answer, hard or soft, as being a set of two paths. If the answer is hard, then one of the paths returned in the answer is the query path itself, and the remaining path is its NN. Now, we can establish a comparison relationship between SNN answers (independently of their type): given two SNN answers $\{a, b\}$ and $\{c, d\}$, we say that $\{a, b\}$ is *better* than $\{c, d\}$ if and only if $d(a, b) < d(c, d)$.

The algorithm is given in Algorithm 3. The input is a set of points, which are interpreted as single-point paths and added to the SNN structure for paths. We assume unique distances between the points. The algorithm maintains a stack (the chain) of *nodes*, where each node consists of a pair of paths. In particular, each node in the chain is the best SNN answer among two queries for the two paths in the predecessor node (when querying from a path, we remove it from the structure temporarily, so that the answer is not the path itself).

Figure 1 shows a snapshot of the algorithm. Nodes 3 and 5 are soft answers, whereas nodes 2 and 4 are hard answers. The pair of paths in the fifth node are the overall closest pair, so the SNN structures will return that pair when queried from each of them. The algorithm will connect them, remove the fifth node from the chain, and continue from the fourth node.

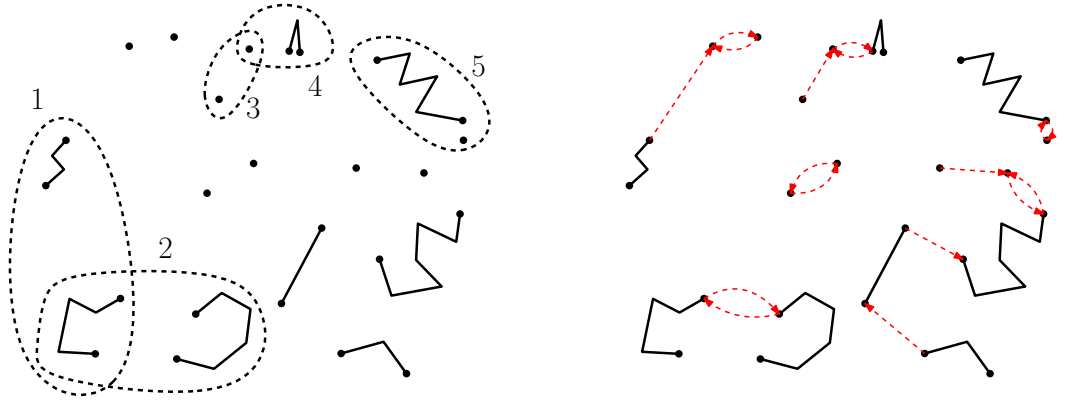

Figure 1 Left: a set of paths, including some single-point paths, and a possible chain, where the nodes are denoted by dashed lines and appear in the chain according to the numbering. **Right:** Nearest-neighbor graph of the set of paths. For each path, a dashed/red arrow points to its NN. Further, the arrows start and end at the endpoints determining the minimum distance between the paths.

Algorithm 3 Soft nearest-neighbor chain algorithm for multi-fragment Euclidean TSP.

Initialize an empty stack (the chain).
Initialize a SNN structure S for paths (Lemma 3.5) with the set of input points as single-point paths.
while there is more than one path in S **do**
  if the chain is empty **then**
    add a node containing an arbitrary pair of paths from S to it.
  Let $U = \{u, v\}$ be the node at the top of the chain.
  Remove u from S, query S with u, and re-add u to S.
  Remove v from S, query S with v, and re-add v to S.
  Let A be the best answer.
  if $A \neq U$ **then**
    Add A to the chain.
  else
    Remove u and v from S and add $u \cup v$.
    Remove U from the chain.
    if the chain is not empty and the new last node, V, contains u or v **then**
      Remove V from the chain.
Connect the two endpoints of the remaining path in S.

Correctness and runtime analysis

▶ **Lemma 3.6.** *The following invariants hold at the beginning of each iteration of Algorithm 3:*

1. *Each input point is an endpoint or a vertex of exactly one path in S.*
2. *If node R appears after node $\{s, t\}$ in the chain, then R is better than $\{s, t\}$.*
3. *Every path in S appears in at most two nodes in the chain, in which case they are consecutive nodes.*
4. *The chain only contains paths in S.*

Proof.

1. The claim holds initially. Each time two paths u, v are replaced by $u \cup v$, one endpoint of each becomes a vertex in the new path $u \cup v$, and the other endpoints become endpoints of $u \cup v$.
2. We show it for the specific case where R is immediately after $\{s, t\}$ in the chain, which suffices. Note that $R \neq \{s, t\}$, or it would not have been added to the chain. We distinguish between two cases:
   - s and t were MNNs when R was added. Then, R had to be a soft answer from s or t, which would have to be better than $\{s, t\}$.
   - s and t were not MNNs when R was added. Then, s had a closer path than t (wlog). Thus, whether the answer for s was soft or hard, the answer had to be better than $\{s, t\}$.
3. Assume, for a contradiction, that a path p appears in two non-consecutive nodes, $X = \{p, x\}$ and $Z = \{p, z\}$ (this covers the case where p appears more than twice). Let Y be the successor of X. By Invariant 2, Z is better than Y. It is easy to see that if z_1 and z_2 are the two endpoints of path z, then z_1 and z_2 were endpoints of paths since the beginning of the algorithm. Thus, the answer for p when X was at the top of the chain had to be a pair at distance at most $\min(d(p, z_1), d(p, z_2))$. However, $\min(d(p, z_1), d(p, z_2)) = d(p, z)$, contradicting that Z is better than Y.
4. Clearly, each node in the chain contains paths that were present in S at the time the node was added. Therefore, the invariant could only break when removing paths from S. In the algorithm, paths are removed from S when merging the paths in the top node. Thus, if a path p is removed from S, it means that p is in the top node. By Invariant 3, besides the top node, p can only occur in the second-from-top node. In the algorithm, when we merge the paths in the top node, we remove the top node from the chain, as well as its predecessor if has a path in common with the top node. ◀

▶ **Lemma 3.7.** *Paths connected in Algorithm 3 are MNNs in the set of paths in the SNN structure.*

Proof. Let $\{u, v\}$ be the node at the top of the chain at some iteration of Algorithm 3. Let A the best SNN answer among the queries from u and v. In the algorithm, u and v are connected when $A = \{u, v\}$. Thus, we need to show that if A is $\{u, v\}$, then u and v are MNNs. We show the contrapositive: if u and v are *not* MNNs, then A is not $\{u, v\}$. If u and v are not MNN, at least one of them, u (wlog), has a closer path than the other, v, so the answer for u is *better* than $\{u, v\}$. ◀

Proof of Theorem 3.3. We show that Algorithm 3 computes the multi-fragment tour in $O(n \log n)$ time.

For its correctness, note that the output is a single cycle that visits every vertex (Invariant 1). This cycle is constructed by only merging pairs of paths that are MNNs (Lemma 3.7). Thus, Algorithm 3 implements MNN-MF. By global-local equivalence (Lemma 3.2), this produces the multi-fragment tour.

For the runtime, note that the chain is acyclic in the sense that each node contains a path from the current set of paths in S (Invariant 4) not found in previous nodes (Invariant 3). Thus, the chain cannot grow indefinitely, so, eventually, paths get connected. The main loop does not halt until there is a single path.

If there are n points at the beginning, there are $n - 1$ connections between different paths in total, and $2n - 1$ different paths throughout the algorithm. This is because each connection removes two paths and adds one new path. At each iteration, either two paths are connected or one node is added to the chain. There are $n - 1$ iterations of the first kind,

each of which triggers the removal of one or two nodes in the chain. Thus, the total number of nodes removed from the chain is at most $2n - 2$. Since every node added is removed, the number of nodes added to the chain is also bounded by $2n - 2$. Thus, the total number of iterations of the second kind is also at most $2n - 2$, and the total number of iterations is at most $3n - 3$. Therefore, the total running time is $O(P(n) + nT(n))$, where $P(n)$ and $T(n)$ are the preprocessing and operation time of a SNN structure for paths. By Lemma 3.5, this can be done in $O(n \log n)$ time. ◀

3.3 Steiner TSP

In the traditional, non-Euclidean setting, a TSP instance consists of a complete graph with arbitrary distances. We remark that global-local equivalence (Lemma 3.2) still holds in this general setting. In this context, the nearest neighbor of a path can be found in $O(n)$ time by iterating through the adjacency lists of both endpoints, where n is the number of nodes. Using this linear search, we can easily compute the multi-fragment tour in $O(n^2)$ time with a NNC-based algorithm. It is a simpler version of Algorithm 3 that only has to handle hard answers and does not need any sophisticated data structures. This improves upon the natural way to implement the multi-fragment heuristic, which is to sort the $\Theta(n^2)$ edges by weight. Sorting requires $\Theta(n^2 \log n)$ time.

This is the first use of NNC in a graph-theoretical setting, but the fact of the matter is that the NNC algorithm can be used in any setting where we can find nearest neighbors efficiently. Consider the related Steiner TSP problem [16]: given a weighted, undirected graph and a set of k node sites $P \subseteq V$, find a minimum-weight tour (repeated vertices and edges allowed) in G that goes at least once through every site in P. Nodes not in P do not need to be visited. For instance, G could represent a road network, and the sites could represent the daily drop-off locations of a delivery truck.

Recently, Eppstein et al. [24] gave a NN structure for graphs from graph families with sublinear separators, which is the same as the class of graphs with polynomial expansion [18]. For instance, planar graphs have $O(\sqrt{n})$-size separators[2]. This data structure maintains a subset of nodes P of a graph G, and, given a query node q in G, returns the node in P closest to q. It allows insertions and deletions to and from the set P. We cite their result in Lemma 3.8.

▶ **Lemma 3.8** ([24]). *Given an n-node weighted graph from a graph family with separators of size $S(n) = n^c$, with $0 < c < 1$, which can be constructed in $O(n)$ time, there is a dynamic[3] nearest-neighbor data structure requiring $O(nS(n))$ space and preprocessing time and that answers queries in $O(S(n))$ time and updates in $O(S(n) \log n)$ time.*

As mentioned, one way to implement the multi-fragment heuristic is to sort the $\binom{k}{2}$ pairs of sites by increasing distances, and process them in order: for each pair, if the two sites are endpoints of separate paths, connect them. The bottleneck is computing the distances. Running Dijkstra's algorithm from each site in a sparse graph, this takes $O(k(n \log n))$ (or $O(kn)$ in planar graphs [28]). When k is $\Theta(n)$, this becomes $O(n^2 \log n)$. We do not know of any prior faster algorithm to compute the multi-fragment tour for Steiner TSP.

[2] Other important families of sparse graphs with sublinear separators include k-planar graphs [17], minor-closed graph families [31], and graphs that model road networks (better than, e.g., k-planar graphs) [25].

[3] They ([24]) use the term *reactive* for the data structure instead of dynamic, to distinguish from other types of updates, e.g., edge insertions and deletions.

Since we have global-local equivalence, we can use the NNC algorithm to construct the multi-fragment tour in $O(P(n) + kT(n))$ time, where $P(n)$ and $T(n)$ are the preprocessing and operation time of a nearest-neighbor structure. Thus, using the structure from [24], we get:

▶ **Theorem 3.9.** *The multi-fragment tour for the steiner TSP problem can be computed in $O(nS(n) + kS(n)\log n)$-time in weighted graphs from a graph family with separators of size $S(n) = n^c$, with $0 < c < 1$.*

Finally, in graphs of bounded treewidth, which have separators of size $O(1)$, the data structure from [24] achieves $P(n) = O(n \log n)$ and $T(n) = O(\log^2 n)$, so we can construct a multi-fragment tour in $O(n \log n + k \log^2 n)$.

4 Motorcycle Graphs

An important concept in geometric computing is the straight skeleton [2]. It is a tree-like structure similar to the medial axis of a polygon, but which consists of straight segments only. Given a polygon, consider a shrinking process where each edge moves inward, at the same speed, in a direction perpendicular to itself. The straight skeleton of the polygon is the trace of the vertices through this process. Some of its applications include computing offset polygons [20] and medical imaging [15]. It is a standard tool in geometric computing software [11].

The fastest algorithms for computing straight skeletons consist of two phases [14, 29], neither of which dominates the other in computation time [13]. We focus on the first phase, which is a motorcycle graph computation induced by the reflex vertices of the polygon.

The motorcycle graph problem can be described as follows (see Figure 2, top) [20]. The input consists of n points in the plane, with associated directions and speeds (the motorcycles). Consider the process where all the motorcycles start moving at the same time, in their respective directions and speeds. Motorcycles leave a trace behind that acts as a "wall" such that other motorcycles crash and stop if they reach it. Some motorcycles escape to infinity while others crash against the traces of other motorcycles. The motorcycle graph is the set of traces.

Most existing algorithms rely on three-dimensional ray-shooting queries. This is because if time is seen as the third dimension, the position of a motorcycle starting to move from (x, y), at speed s, in the direction (u, v), forms a ray (if it escapes) or a segment (if it crashes) in three dimensions, starting at $(x, y, 0)$ in the direction $(u, v, 1/s)$. In particular, the impassable traces left behind by the motorcycles correspond to infinite vertical "curtains" – wedges or trapezoidal slabs, depending on whether they are bounded below by a ray or a segment.

Thus, ray-shooting queries help determine which trace a motorcycle would reach first, if any. Of course, the complication is that as motorcycles crash, their potential traces change. Early algorithms handle this issue by computing the crashes in chronological order [20, 14]. The best previously known algorithm, by Vigneron and Yan [39], is the first that computes the crashes in non-chronological order. It runs in $O(P(n)+n(T(n)+\log n)\log n)$ time, where $P(n)$ and $T(n)$ are the preprocessing time and operation time (maximum between query and update) of a dynamic ray-shooting data structure for curtains in $\mathbb{R}^3$.

We propose a NNC-based algorithm which improves the number of ray-shooting operations needed from $O(n \log n)$ to $3n$. Besides the ray-shooting structure, Vigneron and Yan also use range searching data structures, which do not increase the asymptotic runtime but make the algorithm more complex. Our algorithm only uses a ray-shooting data structure.

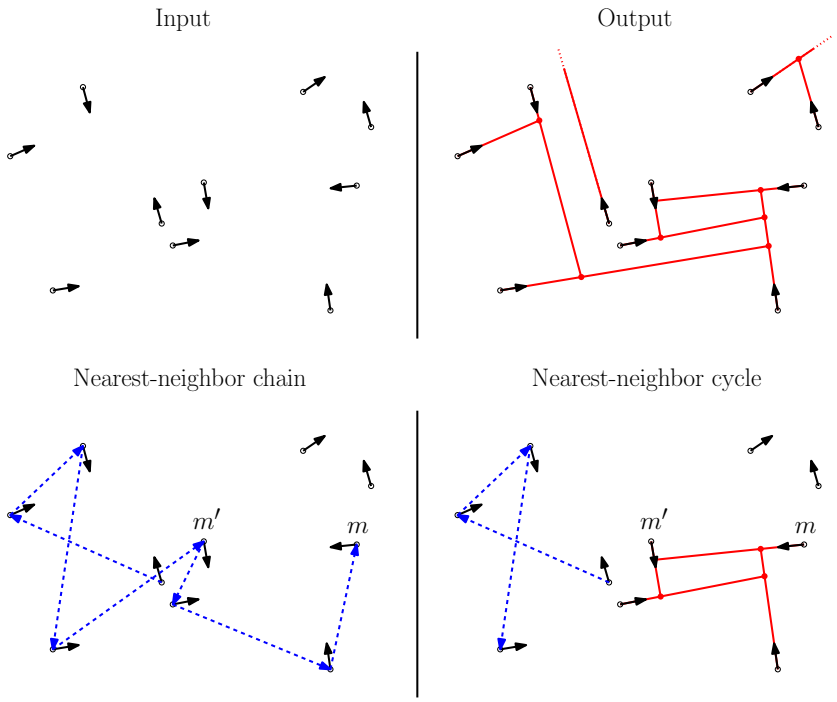

Figure 2 Top: an instance input with uniform velocities and its corresponding motorcycle graph. **Bottom:** snapshots of the NNC algorithm before and after determining all the motorcycles in a NN cycle found by the chain: the NN of the motorcycle at the top, m, is m', which is already in the chain. Note that some motorcycles in the chain have as NNs motorcycles against the traces of which they do not crash in the final output. That is expected, because these motorcycles are still undetermined (e.g., as a result of clipping the curtain of m', the NN of its predecessor in the chain changes).

Agarwal and Matoušek [1] give a ray-shooting data structure for curtains in $\mathbb{R}^3$ which achieves $P(n) = O(n^{4/3+\varepsilon})$ and $T(n) = O(n^{1/3+\varepsilon})$ for any $\varepsilon > 0$. Using this structure, both our algorithm and the algorithm of Vigneron and Yan [39] run in $O(n^{4/3+\varepsilon})$ time for any $\varepsilon > 0$. However, if both algorithms use the same ε in the ray-shooting data structure, then our algorithm is asymptotically faster by a logarithmic factor.

4.1 Algorithm description

Initially, we label all motorcycles as *undetermined*, meaning that their final location is still unknown. They change to *determined* during the algorithm. We use a dynamic three-dimensional ray-shooting data structure to maintain a set of curtains, one for each motorcycle. Determined motorcycles have wedges or slabs as curtains, corresponding to their final trajectory. Undetermined motorcycles have wedge curtains as if they were to escape. When a motorcycle goes from undetermined to determined, the curtain is "clipped" from a wedge to a slab at the point where it crashes.

For an undetermined motorcycle m, we define its nearest neighbor to be the motorcycle, determined or not, against which m would crash next according to the set of curtains in the data structure. We can find the NN of a motorcycle m with one ray-shooting query. Motorcycles that escape may have no NN. Note that m may actually not crash against the trace of its NN, say m', if m' is undetermined and happens to crash early.

Our main structure is a chain (a stack) of undetermined motorcycles. It starts (and restarts, if it becomes empty) with an arbitrary motorcycle, and it is repeatedly extended with the NN of the motorcycle m at the top of the chain, until one of the following cases is reached: *(a)* m has no NN (it escapes), *(b)* the NN of m is determined, or *(c)* the NN of m is already in the chain. In Case (a), we label m as determined and remove it from the chain. In Case (b), we clip m's curtain, mark it as determined, and remove it and the previous motorcycle (if any) from the chain, as m may not be its NN anymore after the clipping.

In contrast to typical applications of the NNC algorithm, here "proximity" is not symmetric: there may be no "mutually nearest-neighbors". In fact, the only case where two motorcycles are MNNs is the degenerate case where two motorcycles reach the same point simultaneously. That said, mutually nearest neighbors have an appropriate analogue in the asymmetric setting: *nearest-neighbor cycles*, $m_1 \to m_2 \to \cdots \to m_k \to m_1$. Case (c) corresponds to finding such a cycle. If we find a nearest-neighbor cycle of undetermined motorcycles, then each motorcycle in the cycle crashes into the next motorcycle's trace. It is easy to see from our definition of nearest neighbors that no motorcycle outside the cycle would "interrupt" the cycle by making one of them crash early. Thus, in Case (c), we can determine all the motorcycles in the cycle at once (this can be seen as a type of chronological global-local equivalence; See Figure 2, bottom). We clip their curtains appropriately and we remove them and the prior motorcycle (if any) from the chain.

The process continues until all motorcycles are determined. Note that we only modify the curtain of the newly determined motorcycles. Thus, if at some step we determine the motorcycle (or motorcycles) at the top of the chain, only the NN of the would-be top motorcycle in the chain may have changed. This is why the would-be top motorcycle is removed from the chain in Cases (b) and (c). The rest of the chain remains consistent.

4.2 Analysis

Clearly, every motorcycle eventually becomes determined, and we have already argued in the algorithm description that irrespective of whether it becomes determined through Case (a), (b), or (c), its final position is correct. Thus, we move on to the complexity analysis. Each "clipping" update can be seen as an update to the ray-shooting data structure: we remove the wedge and add the slab.

▶ **Theorem 4.1.** *The NNC algorithm computes the motorcycle graph in time $O(P(n)+nT(n))$, where $P(n)$ and $T(n)$ are the preprocessing time and operation time (maximum between query and update) of a dynamic, three-dimensional ray-shooting data structure.*

Proof. Each step of the algorithm requires one ray-shooting query from the motorcycle at the top of the chain. At each iteration, either a motorcycle is added to the chain, or at least one motorcycle is determined.

Motorcycles begin as undetermined and, once they become determined, they remain so. This bounds the number of Cases (a–c) to n. In Cases (b) and (c), one undetermined motorcycle may be removed from the chain. Thus, the number of undetermined motorcycles removed from the chain is at most n. It follows that there are at most $2n$ iterations where a motorcycle is added to the chain.

Overall, the algorithm takes at most $3n$ iterations, so it needs no more than $3n$ ray-shooting queries and at most n "clipping" updates where we change a triangular curtain into a slab. It follows that the runtime is $O(P(n) + nT(n))$. ◀

See Appendix C for additional results in special cases and some remarks.

5 Conclusions

Before this paper, NNC had been used only in agglomerative hierarchical clustering and stable matching problems based on proximity. This paper adds: its first use without a NN structure (multi-fragment TSP, which uses our new soft NN structure); its first use in a graph-theoretical framework (Steiner TSP, Subsection 3.3); and its first use in problems without symmetric distances (motorcycle graphs, where it finds nearest-neighbor cycles).

The full version of the paper considers two additional problems [32]. The NNC technique is versatile enough that it seems likely that it will find more uses.

References

1 Pankaj K. Agarwal and Jiří Matoušek. Ray Shooting and Parametric Search. *SIAM Journal on Computing*, 22(4):794–806, 1993.

2 Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A novel type of skeleton for polygons. In *J. UCS The Journal of Universal Computer Science*, pages 752–761. Springer, 1996.

3 Sanjeev Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.

4 Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

5 Jon Jouis Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.

6 Jon Louis Bentley. Experiments on Traveling Salesman Heuristics. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 91–99, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

7 Sergei N. Bespamyatnikh. An Optimal Algorithm for Closest-Pair Maintenance. *Discrete & Computational Geometry*, 19(2):175–195, February 1998.

8 Judith Brecklinghaus and Stefan Hougardy. The approximation ratio of the greedy algorithm for the metric traveling salesman problem. *Operations Research Letters*, 43(3):259–261, 2015.

9 Michel Bruynooghe. New methods in automatic classification of numerous taxonomic data. *Statistics and data analysis*, 2(3):24–42, 1977.

10 Michel Bruynooghe. Classification ascendante hiérarchique des grands ensembles de données: un algorithme rapide fondé sur la construction des voisinages réductibles. *Les cahiers de l'analyse de données*, 3:7–33, 1978.

11 Fernando Cacciola. A CGAL implementation of the Straight Skeleton of a Simple 2D Polygon with Holes. *2nd CGAL User Workshop*, January 2004. URL: `http://www.cgal.org/UserWorkshop/2004/straight_skeleton.pdf`.

12 Timothy M. Chan. Dynamic Geometric Data Structures via Shallow Cuttings. In Gill Barequet and Yusu Wang, editors, *35th International Symposium on Computational Geometry (SoCG 2019)*, volume 129 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.SoCG.2019.24`.

13 Siu-Wing Cheng, Liam Mencel, and Antoine Vigneron. A Faster Algorithm for Computing Straight Skeletons. *ACM Trans. Algorithms*, 12(3):44:1–44:21, April 2016.

14 Siu-Wing Cheng and Antoine Vigneron. Motorcycle Graphs and Straight Skeletons. *Algorithmica*, 47(2):159–182, February 2007.

15 F. Cloppet, J. M. Oliva, and G. Stamon. Angular bisector network, a simplified generalized Voronoi diagram: application to processing complex intersections in biomedical images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):120–128, January 2000.

16 Gérard Cornuéjols, Jean Fonlupt, and Denis Naddef. The traveling salesman problem on a graph and some related integer polyhedra. *Mathematical Programming*, 33(1):1–27, September 1985.

17 Vida Dujmović, David Eppstein, and David R. Wood. Structure of graphs with locally restricted crossings. *SIAM J. Discrete Mathematics*, 31(2):805–824, 2017.

18 Zdeněk Dvořák and Sergey Norin. Strongly sublinear separators and polynomial expansion. *SIAM Journal on Discrete Mathematics*, 30(2):1095–1101, 2016.

19 Mehdi El Krari, Belaïd Ahiod, and Bouazza El Benani. An Empirical Study of the Multi-fragment Tour Construction Algorithm for the Travelling Salesman Problem. In Ajith Abraham, Abdelkrim Haqiq, Adel M. Alimi, Ghita Mezzour, Nizar Rokbani, and Azah Kamilah Muda, editors, *Proceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016)*, pages 278–287, Cham, 2017. Springer International Publishing.

20 D. Eppstein and J. Erickson. Raising Roofs, Crashing Cycles, and Playing Pool: Applications of a Data Structure for Finding Pairwise Interactions. *Discrete & Computational Geometry*, 22(4):569–592, December 1999.

21 David Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *Journal of Experimental Algorithmics (JEA)*, 5:1, 2000.

22 David Eppstein, Michael T. Goodrich, Doruk Korkmaz, and Nil Mamano. Defining equitable geographic districts in road networks via stable matching. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 52. ACM, 2017.

23 David Eppstein, Michael T. Goodrich, and Nil Mamano. Algorithms for stable matching and clustering in a grid. In *International Workshop on Combinatorial Image Analysis*, pages 117–131. Springer, 2017.

24 David Eppstein, Michael T. Goodrich, and Nil Mamano. Reactive Proximity Data Structures for Graphs. In Michael A. Bender, Martín Farach-Colton, and Miguel A. Mosteiro, editors, *LATIN 2018: Theoretical Informatics*, pages 777–789, Cham, 2018. Springer International Publishing.

25 David Eppstein and Siddharth Gupta. Crossing patterns in nonplanar road networks. In *25th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, September 2017.

26 Michael T. Goodrich and Roberto Tamassia. Dynamic Ray Shooting and Shortest Paths via Balanced Geodesic Triangulations. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, pages 318–327, New York, NY, USA, 1993. ACM.

27 Christos H. Papadimitriou. The Euclidean traveling salesman problem is NP-complete. *Theoretical Computer Science*, 4:237–244, June 1977.

28 Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

29 Stefan Huber and Martin Held. A fast straight-skeleton algorithm based on generalized motorcycle graphs. *International Journal of Computational Geometry & Applications*, 22(05):471–498, 2012.

30 David S. Johnson and Lyle A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, Chichester, United Kingdom, 1997.

31 Ken-ichi Kawarabayashi and Bruce Reed. A separator theorem in minor-closed classes. In *51st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 153–162, 2010.

32 Nil Mamano, Alon Efrat, David Eppstein, Daniel Frishberg, Michael Goodrich, Stephen Kobourov, Pedro Matias, and Valentin Polishchuk. Euclidean TSP, Motorcycle Graphs, and Other New Applications of Nearest-Neighbor Chains, 2019. `arXiv:1902.06875`.

33 Alfonsas Misevicius and Andrius Blazinskas. Combining 2-OPT, 3-OPT and 4-OPT with K-Swap-Kick perturbations for the traveling salesman problem. *17th International Conference on Information and Software Technologies*, 2011.

34 Pablo Moscato and Michael G. Norman. On the Performance of Heuristics on Finite and Infinite Fractal Instances of the Euclidean Traveling Salesman Problem. *INFORMS Journal on Computing*, 10(2):121–132, 1998. doi:10.1287/ijoc.10.2.121.

35 Daniel Müllner. Modern hierarchical, agglomerative clustering algorithms. *arXiv e-prints*, September 2011. arXiv:1109.2378.

36 Fionn Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

37 Hoon Liong Ong and J. B. Moore. Worst-case analysis of two travelling salesman heuristics. *Operations Research Letters*, 2(6):273–277, 1984.

38 Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

39 Antoine Vigneron and Lie Yan. A Faster Algorithm for Computing Motorcycle Graphs. *Discrete Comput. Geom.*, 52(3):492–514, October 2014.

A Choice of Parameters

We left open the issue of finding valid SNN parameters. Recall that (ε, k) are valid if any set of k points inside a shell with inner radius 1 and outer radius $(1+\varepsilon)^{k-1}$ contains two points, p and p', satisfying $d(p,p') < 1/(1+\varepsilon)$. To simplify the question, we can scale distances by $(1+\varepsilon)$, so that the inner radius is $1+\varepsilon$, the outer radius $(1+\varepsilon)^k$, and the required distance between the two points is < 1. As a further simplification, we can reduce the inner radius to be 1. This makes the shell grow, and thus, if (ε, k) are valid parameters with this change, then they are also valid under the original statement. Hence, to clarify, the goal of this section is to show how to find, for any metric space $(\mathbb{R}^\delta, L_p)$, a pair of parameters (ε, k) such that any set of k points inside a shell with inner radius 1 and outer radius $(1+\varepsilon)^k$ contains two points, p and p', satisfying $d(p,p') < 1$.

This question is related to the *kissing number* of the metric space, which is the maximum number of points that can be on the surface of a unit sphere all at pairwise distance ≥ 1. For instance, it is well known that the kissing number is 6 in $(\mathbb{R}^2, L_2)$ and 12 in $(\mathbb{R}^3, L_2)$. It follows that, in $(\mathbb{R}^2, L_2)$, $(\varepsilon^* = 0, k^* = 7)$ are valid SNN parameters. Of course, we are interested in $\varepsilon^* > 0$. Thus, our question is more general in the sense that our points are not constrained to lie on a ball, but in a shell (and, to complicate things, the width of the shell, $(1+\varepsilon)^k - 1$, depends on the number of points).

▶ **Lemma A.1.** *There are valid SNN parameters in any metric space* $(\mathbb{R}^\delta, L_p)$.

Proof. Consider a shell with inner radius 1 and outer radius $1+c$, for some constant $c > 0$. A set of points in the shell at pairwise distance ≥ 1 corresponds to a set of disjoint balls of radius $1/2$ centered inside the shell. Consider the volume of the intersection of the shell with such a ball. This volume is lower bounded by some constant, v, corresponding to the case where the ball is centered along the exterior boundary. Since the volume of the shell, v_s, is itself constant, the maximum number of disjoint balls of radius $1/2$ that fit in the shell is constant smaller than v_s/v. This is because no matter where the balls are placed, at least v volume of the shell is inside any one of them, so, if there are more than v_s/v balls, there must be some region in the shell inside at least two of them. This corresponds to two points at distance < 1.

Set k to be v_s/v, and ε to be the constant such that $(1+\varepsilon)^k = 1+c$. Then, (ε, k) are valid parameters for $(\mathbb{R}^\delta, L_p)$. ◀

The dependency of k-ANN structures on $1/\varepsilon$ is typically severe. Thus, for practical purposes, one would like to find a valid pair of parameters with ε as big as possible. The dependency on k is usually negligible in comparison, and, in any case, k cannot be too large because the shell's width grows exponentially in k. Thus, we narrow the question to optimizing ε: what is the largest ε that is part of a pair of valid parameters?

We first address the case of $(\mathbb{R}^2, L_2)$, where we derive the optimal value for ε analytically. We then give a heuristic, numerical algorithm for general $(\mathbb{R}^\delta, L_p)$ spaces.

Parameters in $(\mathbb{R}^2, L_2)$

Let $\varepsilon_\varphi \approx 0.0492$ be the number such that $(1+\varepsilon_\varphi)^{10} = \varphi$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. The valid SNN parameters with largest ε for $(\mathbb{R}^2, L_2)$ are $(\varepsilon^* < \varepsilon_\varphi, k^* = 10)$ (ε^* can be arbitrarily close to ε_φ, but must be smaller). This follows from the following observations.

- The kissing number is 6, so there are no valid parameters with $k < 6$.
- The thinnest annulus (i.e., 2D shell) with inner radius 1 such that 10 points can be placed inside at pairwise distance ≥ 1 has outer radius $\varphi = (1+\varepsilon_\varphi)^{10}$. Figure 3, top, illustrates this fact. In other words, if the outer radius is any smaller than φ, two of the 10 points would be at distance < 1. Thus, any valid pair with $k = 10$ requires ε to be smaller than ε_φ, but any value smaller than ε_φ forms a valid pair with $k = 10$.
- For $6 \leq k < 10$ and for $k > 10$, it is possible to place k points at pairwise distance > 1 in an annulus of inner radius 1 and outer radius $(1+\varepsilon_\varphi)^k$, and they are not packed "tightly", in the sense that k points at pairwise distance > 1 can lie in a thinner annulus. This can be observed easily; Figure 3 (bottom) shows the cases for $k = 9$ and $k = 11$. Cases with $k < 9$ can be checked one by one; in cases with $k > 11$, the annulus grows at an increasingly faster rate, so placing k points at pairwise distance > 1 of each other becomes increasingly "easier". Thus, for any $k \neq 10$, any valid pair with that specific k would require an ε smaller than ε_φ.

Parameters in $(\mathbb{R}^\delta, L_p)$

For other L_p spaces, we suggest a numerical approach. We can do a binary search on the values of ε to find one close to optimal. For a given value of ε, we want to know if there is any k such that (ε, k) are valid. We can search for such a k iteratively, trying $k = 1, 2, \ldots$ (the answer will certainly be "no" for any k smaller than the kissing number). Note that, for a fixed k, the shell has constant volume. As in Lemma A.1, let v be the volume of the intersection between the shell and a ball of radius $1/2$ centered on the exterior boundary of the shell. As argued before, if kv is bigger than the shell's volume, then (ε, k) are valid parameters. For the termination condition, note that if in the iterative search for k, k reaches a value where the volume of the shell grows more than v in a single iteration, no valid k for that ε will be found, as the shell grows faster than the new points cover it.

Besides the volume check, one should also consider a lower bound on how much of the shell's surface (both inner and outer) is contained inside an arbitrary ball. We can then see if, for a given k, the amount of surface contained inside the k balls is bigger than the total surface of the shell, at which point two balls surely intersect. This check finds better valid parameters than the volume one for relatively thin shells, where the balls "poke" out of the shell on both sides.

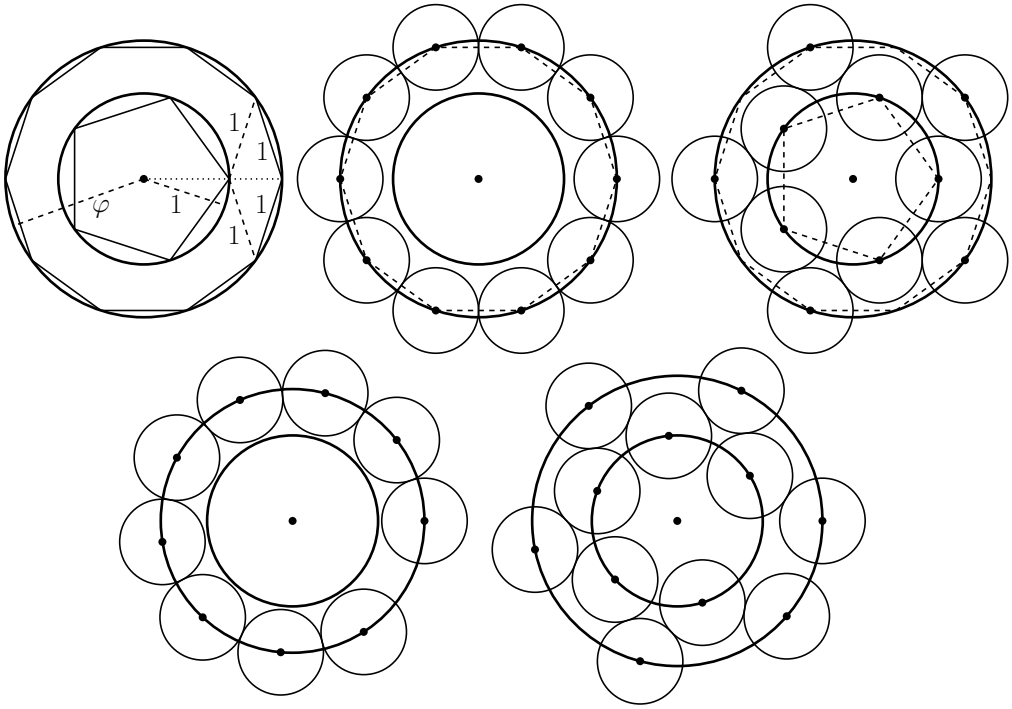

Figure 3 Top: The first figure shows two concentric circles of radius 1 and φ with an inscribed pentagon and decagon, respectively, and some proportions of these shapes. The other figures show two different ways to place 10 points at pairwise distance ≥ 1 inside an annulus of inner radius 1 and outer radius $(1+\varepsilon_\varphi)^{10} = \varphi$. Disks of radius 1/2 around each point are shown to be non-overlapping. In one case, the points are placed on the vertices of the decagon. In the other, they alternate between vertices of the decagon and the pentagon. In both cases, the distance between adjacent disks is 0. Thus, these packings are "tight", i.e., if the annulus were any thinner, there would be two of the 10 points at distance < 1. **Bottom:** 9 and 11 points at pairwise distance ≥ 1 inside annuli of radius $(1+\varepsilon_\varphi)^9$ and $(1+\varepsilon_\varphi)^{11}$, respectively. These packings are not tight, meaning that, for $k = 9$ and $k = 11$, a valid value of ε would have to be smaller than ε_φ.

B Proof of Global-Local Equivalence in Multi-Fragment TSP

In this section, we show the proof for Lemma 3.2. First we introduce another lemma.

▶ **Lemma B.1.** *Let G be an undirected, complete graph with uniquely and positively weighted edges. Then, any pair of paths p, p' that is or becomes MNNs during CP-MF remains MNNs until they are connected.*

Proof. By definition of MNNs, it is cheaper to connect p and p' to each other than to any third path. By reducibility (Lemma 3.1), this does not change if CP-MF connects other paths, as the resulting path will be further from p and from p' than at least one of the preexisting paths before the merge. Eventually, p and p' become the overall cheapest pair to connect, and CP-MF connects them. ◀

Proof of Lemma 3.2. Let $L = l_1, l_2, \ldots, l_m$ be the sequence of pairs of paths merged by a *specific* run of MNN-MF. Now, consider an algorithm that is a hybrid between MNN-MF and CP-MF. Let $hybrid_j$ be an algorithm that starts merging the same pairs as in L up to and including l_{j-1}, and then switches to CP-MF, that is, it switches to merging closest pairs.

The main claim is that, no matter at which iteration the switch happens, the final solution is the same. In the extremes, if the switch happens before the first pick, the method is simply CP-MF. If the switch happens after the last pick, when no more elements can be picked, the solution is L. If we can prove the main claim, it follows that the solution of CP-MF is L.

Let $i \in \{1, \ldots, m\}$. We show that the hybrid that switches before iteration i ($hybrid_i$) finds the same solution as the greedy that switches after iteration i ($hybrid_{i+1}$). The main claim follows by induction on i.

We label the pairs merged by $hybrid_i$ starting at iteration i with g_i, g_{i+1}, and so on. We label the pairs merged by $hybrid_{i+1}$ starting at iteration $i+1$ with g'_{i+1}, g'_{i+2}, and so on. Figure 4, Left, shows this setup. Figure 4, Right, shows what actually happens with the paths merged by the two hybrid methods.

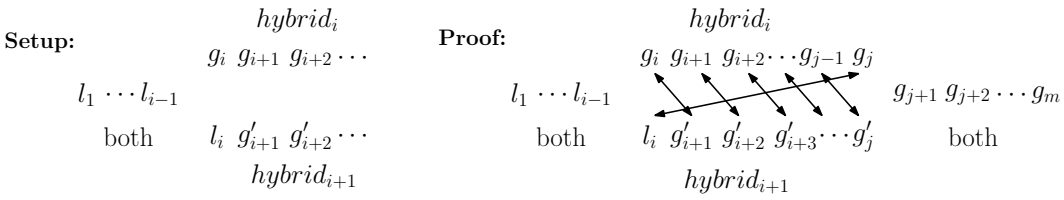

Figure 4 Left: the sequence of path pairs merged by $hybrid_i$ and $hybrid_{i+1}$. They agree up to and including iteration $i-1$, and then $hybrid_i$ switches to CP-MF an iteration early. **Right:** what we prove about the pairs merged by the two hybrid methods. The arrows indicate that these pairs are actually the same. They coincide up to and including iteration $i-1$ and after iteration j.

First, note that $hybrid_i$ *eventually* merges l_i. That is, $l_i = g_j$ for some $j \geq i$. This follows from Lemma B.1.

Further, $g'_{k+1} = g_k$ for all $k \in \{i, \ldots, j-1\}$. This can be shown by induction on k. By Lemma B.1 again, the path resulting from merging the paths in l_i is not closer to any of the paths involved in any of the pairs $g'_{i+1}, \ldots, g'_j$ than one of the two paths in l_i. It follows that "hoisting" the merge of l_i before them does not change the fact that g'_k is the closest pair at iteration k.

After iteration j, both $hybrid_i$ and $hybrid_{i+1}$ have merged exactly the same pairs, so they have the same partial solutions, and both are now the same algorithm, CP-MF. Thus, from that point on, they coincide in their merges, and finish with the same solution. ◀

C Motorcycle Graphs: Special Cases and Remarks

Consider the case where all motorcycles start from the boundary of a simple polygon with $O(n)$ vertices, move through the inside of the polygon, and also crash against the edges of the polygon. In this setting, the motorcycle trajectories form a connected planar subdivision. There are dynamic ray-shooting queries for connected planar subdivisions that achieve $T(n) = O(\log^2 n)$ [26]. Vigneron and Yan used this data structure in their algorithm to get a $O(n \log^3 n)$-time algorithm for this case [39]. Our algorithm brings this down to $O(n \log^2 n)$. Furthermore, their other data structures require that coordinates have $O(\log n)$ bits, while we do not have this requirement.

Vigneron and Yan also consider the case where motorcycles can only go in C different directions. They show how to reduce $T(n)$ to $\min(O(C \log^2 n, C^2 \log n)$, leading to a $O(n \log^2 nC \min(\log n, C))$ algorithm for motorcycle graphs in this setting. Using the same data structures, the NNC algorithm improves the runtime to $O(n \log nC \min(\log n, C))$.

A remark on the use of our algorithm for computing straight skeletons: degenerate polygons where two shrinking reflex vertices collide gives rise to a motorcycle graph problem where two motorcycles collide head on. To compute the straight skeleton, a new motorcycle should emerge from the collision. Our algorithm does not work if new motorcycles are added dynamically (such a motorcycle could, e.g., disrupt a NN cycle already determined), so it cannot be used in the computation of straight skeletons of degenerate polygons.

As a side note, the NNC algorithm for motorcycle graphs is reminiscent of Gale's top trading cycle algorithm [38] from the field of economics. That algorithm also works by finding "first-choice" cycles. We are not aware of whether they use a NNC-type algorithm to find such cycles; if they do not, they certainly can; if they do, then at least our use is new in the context of motorcycle graphs.