Fine-grained Complexity Analysis of Two Classic TSP Variants

MARK DE BERG, KEVIN BUCHIN, and BART M. P. JANSEN, Department of Mathematics and Computer Science, TU Eindhoven
GERHARD WOEGINGER, Department of Computer Science, RWTH Aachen University

We analyze two classic variants of the Traveling Salesman Problem (TSP) using the toolkit of fine-grained complexity.

Our first set of results is motivated by the Bitonic TSP problem: given a set of \( n \) points in the plane, compute a shortest tour consisting of two monotone chains. It is a classic dynamic-programming exercise to solve this problem in \( O(n^2) \) time. While the near-quadratic dependency of similar dynamic programs for Longest Common Subsequence and Discrete Fréchet Distance has recently been proven to be essentially optimal under the Strong Exponential Time Hypothesis, we show that bitonic tours can be found in subquadratic time. More precisely, we present an algorithm that solves bitonic TSP in \( O(n \log^2 n) \) time and its bottleneck version in \( O(n \log^3 n) \) time. In the more general pyramidal TSP problem, the points to be visited are labeled \( 1, \ldots, n \) and the sequence of labels in the solution is required to have at most one local maximum. Our algorithms for the bitonic (bottleneck) TSP problem also work for the pyramidal TSP problem in the plane.

Our second set of results concerns the popular \( k \)-opt heuristic for TSP in the graph setting. More precisely, we study the \( k \)-opt decision problem, which asks whether a given tour can be improved by a \( k \)-opt move that replaces \( k \) edges in the tour by \( k \) new edges. A simple algorithm solves \( k \)-opt in \( O(n^k) \) time for fixed \( k \). For 2-opt, this is easily seen to be optimal. For \( k = 3 \), we prove that an algorithm with a runtime of the form \( O(n^{3-\varepsilon}) \) exists if and only if all-pairs shortest paths in weighted digraphs has such an algorithm. For general \( k \)-opt, it is known that a runtime of \( f(k) \cdot n^{o(k/\log k)} \) would contradict the Exponential Time Hypothesis. The results for \( k = 2, 3 \) may suggest that the actual time complexity of \( k \)-opt is \( \Theta(n^k) \). We show that this is not the case, by presenting an algorithm that finds the best \( k \)-move in \( O(n^{[2k/3]+1}) \) time for fixed \( k \geq 3 \). This implies that 4-opt can be solved in \( O(n^3) \) time, matching the best-known algorithm for 3-opt. Finally, we show how to beat the quadratic barrier for \( k = 2 \) in two important settings, namely, for points in the plane and when we want to solve 2-opt repeatedly.

CCS Concepts: • Theory of computation → Computational geometry; Parameterized complexity and exact algorithms;

Additional Key Words and Phrases: Traveling salesman problem, bitonic TSP, \( k \)-opt, fine-grained complexity

Authors’ addresses: M. de Berg, K. Buchin, and B. M. P. Jansen, Department of Mathematics and Computer Science, TU Eindhoven, P.O. Box 513, Eindhoven, the Netherlands, 5600 MB; emails: [M.T.d.Berg, K.A.Buchin, B.M.P.Jansen]@tue.nl; G. Woeginger, Department of Computer Science, RWTH Aachen University, Ahornstrasse 55, Aachen, Germany, D-52056; email: woeginger@algo.rwth-aachen.de.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1549-6325/2020/12-ART5 $15.00
https://doi.org/10.1145/3414845

1 INTRODUCTION
1.1 Motivation
We analyze two classic variants of the Traveling Salesman Problem (TSP) by applying the modern toolkit of fine-grained complexity analysis. The first TSP variant can for instance be found in Chapter 15 of the well-known textbook "Introduction to Algorithms" by Cormen et al. [19]. The chapter discusses dynamic programming, and its problem section poses the following classic exercise:

15.3 Bitonic euclidean traveling-salesman problem
In the euclidean traveling-salesman problem, we are given a set of \( n \) points in the plane, and we wish to find the shortest closed tour that connects all \( n \) points. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time.

J. L. Bentley has suggested that we simplify the problem by restricting our attention to bitonic tours, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. In this case, a polynomial-time algorithm is possible. Describe an \( \mathcal{O}(n^2) \)-time algorithm for determining an optimal bitonic tour.

This exercise already showed up in the very first edition of the book in 1991. Since then, thousands of students pondered about it and (hopefully) found the solution. One might wonder whether \( \mathcal{O}(n^2) \) runtime is best possible for this problem. As one of our main contributions, we will show that in fact it is not.

The second TSP variant concerns \( k \text{-OPT} \), a popular local search heuristic that attempts to improve a suboptimal solution by a \( k \text{-OPT move} \) (or: \( k \text{-move} \) for short), which is an operation that removes \( k \) edges from the current tour and reconnects the resulting pieces into a new tour by inserting \( k \) new edges. The cases \( k = 2 \) [20] and \( k = 3 \) have been studied extensively with respect to various aspects such as experimental performance [10, 32, 35], (smoothed) approximation ratio [17, 34], rate of convergence [17, 24], and algorithm engineering [26, 28, 38, 39]. The decision problem associated with \( k \text{-OPT} \) asks, given a tour in an edge-weighted graph, whether it is possible to obtain a tour of smaller weight by replacing \( k \) edges. There are \( \Theta(n^k) \) possibilities to choose \( k \) edges that leave the current tour, and for each choice the number of ways to reconnect the resulting pieces back into a tour is constant (for fixed \( k \)). As the weight change for each reconnection pattern can be evaluated in \( \mathcal{O}(k) \) time, this simple algorithm finds the best \( k \text{-OPT} \) improvement in time \( \mathcal{O}(n^k) \) for each fixed \( k \). The survey chapter [31] by Johnson and McGeoch extensively discusses \( k \text{-OPT} \).

On page 233 they write:

To complete our discussion of running times, we need to consider the time per move as well as the number of moves. This includes the time needed to find an improving move (or verify that none exists), together with the time needed to perform the move. In the worst case, 2-opt and 3-opt require \( \Omega(n^2) \) and \( \Omega(n^3) \) time respectively to verify local optimality, assuming all possible moves must be considered.
The two lower bounds in the last sentence are stated without further justification. It is clear that finding an improving \( k \)-move takes \( \Omega(n^k) \) time, if we require that all possible moves must be enumerated explicitly. However, one might wonder whether there are other, faster algorithmic approaches that proceed without enumerating all moves. As one of our main contributions, we will show that such faster approaches do not exist for \( k = 3 \) (under the All-Pairs Shortest Paths conjecture), but do exist for all \( k \geq 4 \).

1.2 Our Contributions

We investigate whether the long-standing runtimes of \( O(n^2) \) for bitonic tours and \( O(n^3) \) for finding \( k\text{-opt} \) improvements are optimal. Such optimality investigations usually involve two ingredients: fast algorithms and runtime lower bounds. While proving unconditional lower bounds is far out of reach, in recent years there has been an influx of techniques for establishing lower bounds on the running time of a given problem, based on a hypothesis about the best-possible running time for another problem. Recent results in this direction consider, for example, the problems of computing the longest common subsequence \([1, 14]\) of two length-\( n \) strings, the edit distance \([7, 14]\) from one length-\( n \) string to another, or the Discrete Fréchet Distance \([13]\) between two polygonal \( n \)-vertex curves in the plane. If one of these problems allows an algorithm with running time \( O(n^{2-\varepsilon}) \), then this would yield an algorithm to test the satisfiability of an \( n \)-variable CNF formula \( \phi \) in time \( (2-\varepsilon)^n \cdot |\phi|^{O(1)} \). As decades of research have not led to algorithms with such a running time for CNF-SAT, this gives evidence that the classic \( O(n^2) \)-time algorithms for these problems are optimal up to \( n^{o(1)} \) factors.

Pyramidal tours in the plane. Consider a symmetric TSP instance that is defined by an edge-weighted complete graph. For a linear ordering \( 1, \ldots, n \) of the vertices in the graph, a pyramidal tour has the form \((1, i_1, \ldots, i_r, n, j_1, \ldots, j_{n-r-2})\), where \( i_1 < i_2 < \cdots < i_r \) and \( j_1 > j_2 > \cdots > j_{n-r-2} \). A bitonic tour for a Euclidean TSP instance is pyramidal with respect to the left-to-right order on the points in the plane. Bitonic and pyramidal tours play an important role in the combinatorial optimization literature on the TSP; see References \([8, 15, 27]\). They form an exponentially large set of tours over which we can optimize efficiently, and they lead to well-solvable special cases of the TSP, even in the graph setting. Combined with a procedure for generating suitable permutations of the vertices, heuristic solutions to TSP can be obtained by computing optimal pyramidal tours with respect to the generated orders \([16]\).

We will show that the classic \( O(n^2) \) dynamic program for finding bitonic tours in the Euclidean plane is far from optimal: by an appropriate use of dynamic geometric data structures, the running time can be reduced to \( O(n \log^3 n) \). To the best of our knowledge, this presents the first improvement in finding bitonic tours since the problem was popularized in Introduction to Algorithms \([19]\) in 1991. In fact, we prove the stronger result that an optimal pyramidal tour among \( n \) points in the plane can be computed in \( O(n \log^2 n) \) time with respect to any given linear order on the points. Our techniques extend to the related bottleneck pyramidal TSP problem in the plane, where the goal is to find a pyramidal tour among the cities that minimizes the length of the longest edge. We prove that the underlying decision problem (given a linearly ordered set of points and a bottleneck value \( B \), is there a pyramidal tour of the points whose longest edge has length at most \( B \)) can be solved in \( O(n \log n) \) time, while the underlying optimization version (given a linearly ordered set of points, compute a bitonic tour that minimizes the length of the longest edge) can be solved in \( O(n \log^3 n) \) time. For the decision version of the bottleneck problem, we prove a matching \( \Omega(n \log n) \) time lower bound in the algebraic computation tree model by a reduction from Set Disjointness with integer inputs \([44]\); this reduction even applies to the bitonic setting where the points are ordered from left to right.
**k-opt in the graph setting.** The complexity of k-opt has been analyzed using the framework of parameterized complexity theory. Marx [36] proved that deciding whether there is a k-move that improves a given tour is W[1]-hard parameterized by k, giving evidence that there is no algorithm with runtime \( f(k) \cdot n^{O(1)} \). Guo et al. [29] refined this result and proved that, under the Exponential Time Hypothesis [30], there is no algorithm that determines whether a tour in a weighted complete graph can be improved by a k-move in time \( f(k) \cdot n^{o(k/\log k)} \) for any function \( f \). This lower bound shows that the exponent of \( n \) in the runtime of any k-opt algorithm must grow almost linearly with \( k \). The next question that we settle in this article is: Can one do better than \( O(n^k) \) for finding a k-opt improvement? The answer turns out to depend on the value of \( k \). For 2-opt, an easy adversarial argument shows that any deterministic algorithm must inspect all the edge weights. This gives a trivial lower bound of \( \Omega(n^2) \), matching the upper bound. For larger values of \( k \), the question becomes more interesting.

The 3-opt Detection problem asks whether the weight of a given tour can be reduced by some 3-move. We show that it is unlikely that 3-opt Detection with weights in the range \([-M, \ldots, M]\) allows an algorithm with a truly subcubic runtime of \( O(n^{3-\epsilon} \text{polylog}(M)) \) for \( \epsilon > 0 \). We prove that the Negative Edge-weighted Triangle problem (given an edge-weighted graph, is there a triangle of negative weight?) reduces to 3-opt Detection by a reduction that takes \( O(n^3) \) time and increases the size of the graph by only a constant factor. As Negative Edge-weighted Triangle is equivalent to All-Pairs Shortest Paths (APSP) in weighted digraphs with respect to having truly subcubic algorithms [42], a truly subcubic algorithm for 3-opt Detection would contradict the APSP conjecture [2, 3], which states that APSP cannot be solved in truly subcubic time. We also give a reduction in the other direction: finding a 3-opt improvement reduces to finding a negative edge-weighted triangle. Consequently, 3-opt Detection is equivalent to Negative Edge-weighted Triangle and APSP with respect to truly subcubic runtimes. This adds yet another classic problem to the growing list of such equivalent problems [2, 42].

As a final result in this direction, we design an algorithm that finds the best k-opt improvement in weighted \( n \)-vertex complete graphs in \( \tilde{O}(n^{2k/3+1}) \) time for each fixed value of \( k \). For \( k = 2 \) and \( k = 3 \), this expression simply boils down to the straightforward time complexities of \( O(n^2) \) and \( O(n^3) \) for 2-opt and 3-opt, respectively. For \( k \geq 4 \), however, our result yields a substantial improvement over the trivial \( O(n^k) \) time bound. For example, 4-opt can be solved in \( \Theta(n^3) \) time, matching the best-known algorithm for 3-opt. The algorithm mixes enumeration of partial solutions with a simple dynamic program.

**Faster 2-opt in the repeated setting and in the planar setting.** For the 2-opt problem in graphs, the runtime for finding a single tour improvement cannot be improved below the trivial \( \Theta(n^2) \). However, in the context of local search, we are often interested in repeatedly finding tour improvements. It is therefore natural to consider whether speedups can be obtained when repeatedly finding improving tours on the same TSP instance. We prove that this is indeed the case: after \( O(n^2) \) preprocessing time, one can repeatedly find the best 2-opt improvement in \( O(n \log n) \) time per iteration.

The quadratic lower bound for 2-opt applies only in the graph setting. This raises the question: Can we solve 2-opt faster for points in the plane? We show the answer is yes, by giving an algorithm for 2-opt Detection with runtime \( \tilde{O}(n^{8/5+\epsilon}) \) for all \( \epsilon > 0 \). Similarly, 3-opt Detection can be solved in expected time \( \tilde{O}(n^{80/31+\epsilon}) \).

## 2 Faster Algorithms for Pyramidal and Bottleneck Pyramidal TSP

In this section, we show that the pyramidal TSP and the bottleneck pyramidal TSP in the plane can be solved in subquadratic time. For simplicity, we only show how to compute the value of an
optimal solution; computing the actual tour can be done in a standard manner. For the bottleneck pyramidal TSP, we first consider the decision version of the problem. For this version, we also provide a lower bound in the algebraic computation-tree model of computation. Since the algorithms and data structures for the decision version do not easily extend to the minimization version, we present a different data structure for the minimization problem.

2.1 Algorithm for Pyramidal TSP

Let \( P \) be the ordered input set of \( n \) points with distinct \( x \)-coordinates in the plane. Our algorithm will consider the points in \( P \) in order, and maintain a collection of partial solutions that are locally optimal. To make this precise, define \( P_i := \{p_1, \ldots, p_i\} \) to be the first \( i \) points in \( P \). A partial solution for \( P_i \), for some \( 1 \leq i \leq n \), is a pair \( P', P'' \) of monotone paths (with respect to the order on \( P \)) that together visit all the points in \( P_i \) and that only share \( p_i \). We call a partial solution for \( P_i \) an \((i, j)\)-partial tour, for some \( 1 \leq j < i \), if one of the paths ends at \( p_i \) — this is necessarily the case in a partial solution for \( P_1 \) — and the other path ends at \( p_j \).

Our starting point is the standard dynamic-programming solution. It uses a 2-dimensional table \( A[1..n, 1..n] \), where \( A[i, j] \), for \( 1 \leq j < i \leq n \), is defined as the minimum length of an \((i, j)\)-partial tour; for \( i \leq j \leq n \) the entries \( A[i, j] \) are undefined. We can compute the entries in the table row by row, using the recursive formula

\[
A[i + 1, j] = \begin{cases} 
A[i, j] + |p_i p_{i+1}| & \text{if } 1 \leq j < i, \\
\min_{1 \leq k < i} (A[i, k] + |p_k p_{i+1}|) & \text{if } j = i,
\end{cases}
\]

(1)

where \( A[2, 1] = |p_1 p_2| \). Let us briefly verify this recurrence. For \((i + 1, j)\)-partial tours with \( j < i \), the path \( P'' \) that visits \( p_{i+1} \) must also visit \( p_i \); the other path \( P''' \) visits \( i \) such \( j < i \) and the monotonicity requirement ensures \( P''' \) cannot visit \( i \) and go back to \( j \). So for \( j < i \) any \((i + 1, j)\)-partial tour consists of an \((i, j)\)-partial tour together with the segment \( p_i p_{i+1} \). For \((i + 1, i)\)-partial tours, the predecessor of \( p_{i+1} \) cannot be \( p_i \), since a path ends at \( p_i \). Hence, an \((i + 1, i)\)-partial tour consists of an \((i, k)\)-partial tour for some \( 1 \leq k < i \) together with the segment \( p_k p_{i+1} \). The cheapest combination yields the best partial tour.

After computing the last row of \( A \), the minimum length of a pyramidal tour is given by \( \min_{1 \leq k < n} (A[n, k] + |p_k p_n|) \). There are \( O(n^2) \) entries in \( A \) of the first type that each take constant time to evaluate. There are \( O(n) \) entries of the second type that need time \( \Theta(n) \). Hence, the dynamic program can be evaluated in \( O(n^2) \) time. The next theorem shows that this dynamic-programming algorithm can be sped up significantly.

**Theorem 1.** Let \( P \) be an ordered set of \( n \) points in the plane. Then, we can compute a minimum-length pyramidal tour for \( P \) in \( O(n \log^2 n) \) time and using \( O(n) \) storage.

**Proof.** Our subquadratic algorithm is based on the following two observations. First, any two subsequent rows \( A[i, 1..n] \) and \( A[i + 1, 1..n] \) in the standard dynamic program described above are quite similar: the entries \( A[i + 1, j] \), for \( j < i \), can all be obtained from \( A[i, j] \) by adding the same value, namely, \( |p_i p_{i+1}| \). Second, the computation of \( A[i + 1, i] \) can be sped up using appropriate geometric data structures. Thus, our algorithm will maintain a data structure that implicitly represents the current row and allows for fast queries and so-called bulk updates, as described below.

Recall that \( P_i := \{p_1, \ldots, p_i\} \). The point that defines \( \min_{1 \leq k < i} (A[i, k] + |p_k p_{i+1}|) \) is the point \( p_k \in P_{i-1} \) closest to the query point \( q := p_{i+1} \) if we use the additively weighted distance function

\[
\text{dist}(p_k, q) := w_k + |p_k q|.
\]

(2)

\footnote{Some of our results can also be obtained from an alternative DP with \( n \) states. As we need the two-dimensional approach for Theorem 6, we present all our results in this setting.}
where \( w_k := A[i, k] \) is the weight of \( p_k \). Thus, we need a data structure for storing a weighted point set that supports the following operations:

- perform a nearest-neighbor query with a query point \( q \), which reports the point \( p_k \) closest to \( q \) according to the additively weighted distance function,
- perform a bulk update of the weights, which adds a given value \( \Delta \) to the weights of all the points currently stored in the data structure;
- insert a new point with a given weight into the data structure.

In Section 2.2, we describe a data structure that supports queries in \( O(\log^2 n) \) time, bulk updates in \( O(\log n) \) time, and insertions in \( O(\log^2 n) \) amortized time. Using this data structure, we can speed up the classic dynamic-programming algorithm, as described next.

Instead of computing the entire dynamic programming table \( A \) explicitly, we maintain an implicit representation of one row of the table and compute the rows one by one. The \( i \)th row of \( A \) has \( i - 1 \) well-defined entries. We define an implicit representation of row \( i \) to be an instance of the data structure storing the weighted point set \( P_{i-1} = \{p_1, \ldots, p_{i-1}\} \) such that \( w(p_j) = A[i, j] \). The first nontrivial row in \( A \) is the second row, \( A[2, 1..n] \). An implicit representation for that row consists of the point \( p_1 \) of weight \( A[2, 1] = |p_1p_2| \).

If we have an implicit representation of row \( i \), then we can efficiently obtain an implicit representation of row \( i + 1 \), as we describe next. By our choice of implicit representation, the value of \( A[i + 1, i] \) according to Equation (1) is exactly the distance from \( p_{i+1} \) to its closest neighbor in the data structure under the additively weighted distance function. Hence, the value of \( k \) that minimizes the lower expression in Equation (1) can be found by a nearest neighbor query with \( p_{i+1} \). We can therefore transform a representation of row \( i \) into a representation for row \( i + 1 \) as follows:

1. Query with point \( p_{i+1} \) to find the value \( A[i + 1, i] \) and remember this value.
2. Perform a bulk update to increase the weight of the points \( p_1, \ldots, p_{i-1} \) that are already in the structure by \( \Delta := |p_ip_{i+1}| \). Recall that for cells \( j \) with \( 1 \leq j < i \) their value in row \( i + 1 \) is obtained from their value in row \( i \) by adding \( |p_ip_{i+1}| \).
3. Insert point \( p_i \) of weight \( A[i + 1, i] \) into the structure.\(^2\)

It is easy to verify that this yields an implicit representation of row \( i + 1 \). Since a representation of the first nontrivial row can be found in constant time, and each successive row can be computed from the previous using three data structure operations that take \( O(\log^2 n) \) amortized time each, it follows that an implicit representation of the final row can be computed in \( O(n \log^2 n) \) time. The minimum cost of a pyramidal tour is \( \min_{1 \leq k < n} (A[n, k] + |p_kp_n|) \), which can be found by querying the representation of the final row with point \( p_n \).

### 2.2 Data Structure for Pyramidal TSP

We now describe the data structure needed in the proof of Theorem 1. With a slight abuse of notation, we will denote the set of points stored in the data structure by \( P \) and let \( n \) denote the number of points in the current set \( P \).

Answering nearest-neighbor queries for the weighted point set \( P \) can be done by performing point location in the additively weighted Voronoi diagram of \( P \). The additively weighted Voronoi diagram of \( P \), denoted by \( \text{AWVD}(P) \), is the subdivision of the plane into regions such that the region of a point \( p_k \) in \( P \) consists of those points \( q \in \mathbb{R}^2 \) for which \( p_k \) is the nearest neighbor of \( q \) if we

\[2\] We could also insert \( p_1 \) with weight \( A[i + 1, i] - \Delta \). This way, we would not have to subtract \( \Delta \) from the weights of \( p_1, \ldots, p_{i-1} \) in Step 2, and the bulk updates are not needed. As they are trivial in our data structure, we prefer the version that keeps the correspondence between weights and \( A[i, j] \) values.
consider additively weighted distances. The diagram AWVD($P$) consists of at most $n$ regions—at most, because some points may define an empty region—and the boundaries between the regions consist of hyperbolic arcs. The total complexity of the diagram is $O(n)$ and it can be computed in $O(n \log n)$ time [25]. Moreover, point location in a planar subdivision of complexity $O(n)$ can be done in $O(\log n)$ time with a data structure\(^4\) that uses $O(n)$ storage and $O(n \log n)$ preprocessing [23]. Thus, nearest-neighbor queries in $P$ under the additively weighted distance function can be done in $O(\log n)$ after $O(n \log n)$ preprocessing.

We first briefly review the logarithmic method [11]. It makes a static data structure $\mathcal{DS}$ semi-dynamic, as follows. Let $n$ be the number of objects—weighted points in our case—in the set $S$ currently stored in the data structure, and let $a_t \in \{0, 1\}$ be such that $n = \sum_{t=0}^{\lceil \log_2 n \rceil} a_t 2^t$. The logarithmic method maintains, for each $t$ with $a_t = 1$, a static data structure $\mathcal{DS}^{(t)}$ on a subset $S^{(t)} \subseteq S$ of size $2^t$, where the subsets $S^{(t)}$ form a partition of $S$. A query on the set $S$ can now be answered by querying each of the data structures $\mathcal{DS}^{(t)}$, and computing the final answer to the query from the $O(\log n)$ sub-answers. (The use of the logarithmic method thus requires the query problem to be such that the answer to a query on the whole set $S$ can be easily computed from the answers on the subsets $S^{(t)}$.) To insert a new object $o$, one first finds the smallest $t^*$ such that $a_{t^*} = 0$, where the $a_t$ are defined with respect to the size of $S$ before the insertion. Then all structures $\mathcal{DS}^{(0)}, \ldots, \mathcal{DS}^{(t^*-1)}$ are destroyed, and a new structure $\mathcal{DS}^{(t^*)}$ on the set $S^{(0)} \cup \cdots \cup S^{(t^*-1)} \cup \{o\}$ is constructed. The amortized insertion time is $O(\sum_{t=0}^{\lceil \log_2 n \rceil} B(2^t)/2^t)$, where $B(2^t)$ denotes the time needed to construct a data structure on a set of size $2^t$.

In our case each $\mathcal{DS}^{(t)}$ is a point-location structure for the additively weighted Voronoi diagram AWVD($P^{(t)}$) on a subset $P^{(t)} \subseteq P$. Note that we can easily find the overall nearest neighbor of a query point by taking the nearest among the $O(\log n)$ candidates found for the subsets $P^{(t)}$. Thus, our structure has $O(\log^2 n)$ query time. Since a substructure $\mathcal{DS}^{(t)}$ can be built in $O(|P^{(t)}| \log |P^{(t)}|)$ time [23, 25], the amortized time for an insertion is $O(\log^2 n)$.

It remains to deal with bulk updates, where we want to increase the weight of each of the points in our data structure by a given value $\Delta$. With the logarithmic method this is quite easy. We simply store a correction term $\Delta_t$ for each $\mathcal{DS}^{(t)}$, which indicates that the weight of each point in $P^{(t)}$ should be increased by $\Delta_t$. A bulk update with value $\Delta$ can then be performed in $O(\log n)$ time by adding $\Delta$ to each of the correction terms $\Delta_t$. Note that we can still answer queries correctly. Indeed, AWVD($P^{(t)}$) does not change when we add the same value $\Delta$ to all weights in $P^{(t)}$. Hence, we just have to make sure that when we compare the candidates found for the subsets $P^{(t)}$, we increase their weighted distances by the relevant correction term. Thus, a query still takes $O(\log^2 n)$ time. Insertions can still be done in $O(\log^2 n)$ amortized time as well; we only need to make sure that when we collect the points in the substructures $\mathcal{DS}^{(0)}, \ldots, \mathcal{DS}^{(t^*-1)}$ to be destroyed, we add the correct terms to their weights before we construct the new structure $\mathcal{DS}^{(t)}$. Hence, we obtain the claimed query time, bulk-update time, and insertion time, thus finishing the proof of Theorem 1.

\(\text{\footnote{Sometimes additively weighted Voronoi diagrams are defined based on a distance function that subtracts a positive weight from the Euclidean distance. It is easy to see that all results carry over to the case where we add weights, because we can transform the latter case to the former by subtracting the same sufficiently large value from all weights to make them negative.}}\)

\(\text{\footnote{The result by Edelsbrunner et al. [23] is stated for straight-line subdivisions, but (as remarked in the conclusions of that paper) it also applies to subdivisions with curved edges, under some mild conditions that are satisfied in our application. Alternatively, one can use the method of Sarnak and Tarjan [41], which can also be used for subdivisions with curved edges (after subdividing these edges into x-monotone pieces).}}\)

---

2.3 Algorithm for the Decision Version of Bottleneck Pyramidal TSP

In the remainder of Section 2.1, we consider the bottleneck version of the pyramidal TSP problem. The goal is to find a pyramidal tour for an ordered set \( P := \{p_1, \ldots, p_n\} \) of points in the plane such that the length of the bottleneck edge (that is, the longest edge) is minimized. We start in Section 2.3 by giving an \( O(n \log n) \) algorithm for the decision version of the problem, where we are given a value \( B \) and the question is whether there is a pyramidal tour whose bottleneck edge has length at most \( B \). The algorithm uses a data structure developed in Section 2.4. Next, we show (Section 2.5) that this is optimal by presenting an \( \Omega(n \log n) \) lower bound in the algebraic computation-tree model. Finally, we show in Section 2.6 how to solve the optimization version of the problem, using a data structure developed in Section 2.7.

The decision problem can be solved by dynamic programming, using a two-dimensional table \( A[1..n, 1..n] \), where (for \( 1 \leq j < i < n \)) we have \( A[i, j] = \text{True} \) if there is an \((i, j)\)-partial tour of cost at most \( B \) and \( A[i, j] = \text{False} \) otherwise. The dynamic program can compute the entries \( A[i, j] \) row by row with the recursive formula

\[
A[i + 1, j] = \begin{cases} 
\text{True} & \text{if } 1 \leq j < i, \\
A[i, j] \land (|p_j p_{j+1}| \leq B) & \text{if } j = i,
\end{cases}
\]

where \( A[2, 1] = \text{True} \) if \( |p_1 p_2| \leq B \). As before, we speed up the computation by using the relation between consecutive rows in the table—for \( j < i \) the entries \( A[i + 1, j] \) are all equal to \( A[i, j] \) when \( |p_j p_{j+1}| \leq B \), and they are all \text{False} otherwise—and by using appropriate geometric data structures.

Instead of computing the entries of the matrix \( A \), we will maintain a list \( \mathcal{L} \) that contains, for the current value of \( i \), all points \( p_j \) with \( 1 \leq j < i \) such that there is an \((i, j)\)-partial tour of cost at most \( B \). In other words, \( \mathcal{L} \) contains all \( p_j \) such that \( A[i, j] = \text{True} \). We initialize \( \mathcal{L} \) as an empty list, and then go over the points \( p_2, \ldots, p_n \) in order. To handle \( p_{i+1} \), we check if \( |p_i p_{i+1}| \leq B \), and we check if \( \mathcal{L} \) currently contains a point \( p_k \) such that \( |p_k p_{i+1}| \leq B \). If both conditions are satisfied, then we add \( p_i \) to \( \mathcal{L} \); if only the first condition is satisfied, then we keep \( \mathcal{L} \) as it is, if only the second condition is satisfied we first empty \( \mathcal{L} \) and then add \( p_i \) to it, and if neither condition is satisfied then we empty \( \mathcal{L} \). After having handled \( p_n-1 \) we check if \( |p_{n-1} p_n| \leq B \) and if \( \mathcal{L} \) contains a point \( p_k \) such that \( |p_k p_n| \leq B \). If both conditions are satisfied, then a pyramidal tour whose bottleneck length is at most \( B \) exists, otherwise it does not.

Since a point is added to \( \mathcal{L} \) only once, the total number of updates to \( \mathcal{L} \) is \( O(n) \). Checking the first condition obviously takes \( O(1) \) time, so it remains to describe how to check the second condition efficiently. To this end we maintain a data structure on the points in \( \mathcal{L} \) that supports two operations:

- \textit{query} the structure with a point \( p_{i+1} \) to decide if it stores a point \( p_k \) with \( |p_k p_{i+1}| \leq B \),
- \textit{insert} a new point \( p_i \) into the structure.

These operations can be performed by a semi-dynamic data structure for nearest-neighbor queries, similar to the one described earlier, which has \( O(\log^2 n) \) query time and \( O(\log^2 n) \) amortized insertion time. Below, we describe a faster data structure. The data structure is based on the following observation. Let \( D_k \) be the disk of radius \( B \) centered at the point \( p_k \), and let

\[
\mathcal{D} := \{D_k : p_k \text{ is a point in } \mathcal{L}\}.
\]

Then \( \mathcal{L} \) contains a point \( p_k \) with \( |p_k q| \leq B \) if and only if \( q \in \text{Union}(\mathcal{D}) \), where \( \text{Union}(\mathcal{D}) \) denotes the union of the disks in \( \mathcal{D} \). Theorem 3 below states that point-location queries in the union of a set of congruent disks can be done in \( O(\log n) \) time and with \( O(\log n) \) amortized update time, leading to the following result.
Theorem 2. Let \( P \) be an ordered set of \( n \) points in the plane, and let \( B > 0 \) be a given parameter. Then, we can decide in \( O(n \log n) \) time and using \( O(n) \) storage if \( P \) admits a pyramidal tour whose longest edge has length at most \( B \).

2.4 Semi-dynamic Point-location Data Structure for the Union of Congruent Disks

Let \( D \) be a set of congruent disks in the plane. We wish to maintain a data structure on \( D \) that allows us to decide if a query point \( q \) lies inside \( \text{Union}(D) \). The data structure should also allow insertions into the set \( D \). With a slight abuse of notation, we will use \( n \) to denote the number of disks in the (current) set \( D \). We will assume we have the floor function available; it is not hard to avoid the floor function, but using it simplifies the presentation. It will also be convenient to assume that the disks in \( D \) all have radius \( \sqrt{2} \), which can be ensured by appropriate scaling.

Consider the integer grid \( G \). Note that the diameter of the grid cells is \( \sqrt{2} \), so any cell containing the center of some disk \( D_i \) is completely covered by \( D_i \). We say that a grid cell\(^5 \) \( C \) is active if it contains the center of a disk \( D_i \in D \), and we say that a vertical strip \( [x, x + 1) \times (-\infty, \infty) \) is active if it contains an active grid cell. Our data structure for point location in \( \text{Union}(D) \) maintains the active strips in a balanced search tree on their \( x \)-order, and for each active strip \( \Sigma \) it maintains the active cells within \( \Sigma \) in a balanced search tree on their \( y \)-order. (These search trees could also be replaced by a hash table.) For each active cell \( C \), we maintain four partial unions, as explained next.

Let \( D(C) \subset D \) be the set of disks whose center lies in \( C \). Let \( \ell_{\text{top}}(C), \ell_{\text{bot}}(C), \ell_{\text{left}}(C), \) and \( \ell_{\text{right}}(C) \) denote the lines containing, respectively, the top, bottom, left, and right edge of \( C \). Finally, define \( U_{\text{top}}(C), U_{\text{bot}}(C), U_{\text{left}}(C), \) and \( U_{\text{right}}(C) \) to be the parts of \( \text{Union}(D(C)) \) lying, respectively, above \( \ell_{\text{top}}(C) \), below \( \ell_{\text{bot}}(C) \), to the left of \( \ell_{\text{left}}(C) \), and to the right of \( \ell_{\text{right}}(C) \). Next, we explain how we store and maintain the partial union \( U_{\text{top}}(C) \); the other three partial unions are stored and maintained in a similar manner.

Let \( p_j \) denote the center of the disk \( D_j \in D(C) \). Because the centers \( p_j \) all lie inside \( C \), they all lie below the line \( \ell_{\text{top}}(C) \). Hence, the partial union \( U_{\text{top}}(C) \) is \( x \)-monotone. Furthermore, each component of \( U_{\text{top}}(C) \) is bounded from below by a portion of the line \( \ell_{\text{top}}(C) \) and from above by circular arcs that are portions of the boundaries of the disks \( D_i \in D(C) \). The key to efficiently maintaining \( U_{\text{top}}(C) \) is the following lemma.

Lemma 2.1. Each disk \( D_i \in D(C) \) contributes at most one arc to \( \partial U_{\text{top}}(C) \). Moreover, the arc contributed by a disk \( D_i \) lies to the left of the arc contributed by a disk \( D_j \) if and only if \( p_i \) lies to the left of \( p_j \).

Proof. Define \( \gamma_i \) to be the part of \( D_i \)’s boundary above the line \( \ell_{\text{top}}(C) \). Any other disk \( D_j \in D(C) \) that covers a part of \( \gamma_i \) must contain an endpoint of \( \gamma_i \). Indeed, if \( \gamma_j \) would intersect \( \gamma_i \) twice above \( \ell_{\text{top}}(C) \), then since the centers of the disks \( D_i \) and \( D_j \) lie below \( \ell_{\text{top}}(C) \), the curvature of \( \gamma_j \) would be larger than the curvature of \( \gamma_i \), contradicting the fact that all disks have equal radius. Hence, each disk \( D_i \) can contribute at most one arc to \( \partial U_{\text{top}}(C) \), as claimed.

Now consider an arc \( \alpha_i \subset \gamma_i \) contributed by \( D_i \) and an arc \( \alpha_j \subset \gamma_j \) contributed by \( D_j \). Assume without loss of generality that \( p_i \) lies to the left of \( p_j \). Furthermore, assume \( p_i \) lies below \( p_j \), as in Figure 1(i); a similar argument applies when \( p_i \) lies above \( p_j \). Now suppose for a contradiction that \( \alpha_i \) lies to the right of \( \alpha_j \). Let \( \ell \) be the perpendicular bisector of the segment \( p_ip_j \), and let \( r_j \) be the right endpoint of \( \alpha_j \). Because the disks have equal radius, \( \alpha_i \) must lie to the left of \( \ell \) and \( \alpha_j \) must lie to the right of \( \ell \). Hence, if \( \alpha_i \) lies to the right of \( \alpha_j \), then it must lie in the triangular region.

---

\(^5\)To assign each point to a unique active cell, we assume the cells in \( G \) are closed on the left and bottom, and open on the right and top. Thus, the cells in \( G \) are of the form \([x, x + 1) \times [y, y + 1)\) for integers \( x, y \).
Lemma 2.1 gives us an easy way to store and maintain $U_{\text{top}}(C)$. We simply store the arcs comprising $\partial U_{\text{top}}(C)$ in $x$-order in a balanced search tree $T_{\text{top}}(C)$. This takes $O(n)$ storage and allows us to decide in $O(\log n)$ time for a query point $q$ if $q \in U_{\text{top}}(C)$.

Now, suppose we want to insert a new disk $D_i$ into $\mathcal{D}(C)$. As observed, $\partial D_i$ contributes at most one new arc to $\partial U_{\text{top}}(C)$. The addition of this new arc means we have to remove some existing arcs. More precisely, if $\partial D_i$ contributes a new arc, then we have to shorten two existing arcs and possibly remove one or more other arcs; see Figure 1(ii). Once we know which existing arcs are affected, the update can be done in $O((k + 1) \log n)$ time, where $k$ is the number of disappearing arcs. Since each arc is removed at most once, this gives an amortized insertion time of $O(\log n)$. It remains to describe how to check whether $D_i$ actually contributes a new arc and, if so, which existing arcs are affected. To this end, we search in $T_{\text{top}}(C)$ for an affected arc, that is, for an arc that is completely or partially covered by $D_i$. When we have such an arc, we can easily find all other affected arcs, because these arcs are neighbors in the left-to-right ordering. It remains to describe how to search for an affected arc.

Consider the arc $\alpha_j$ (contributed by some disk $D_j$) stored at the root of $T_{\text{top}}(C)$. If $D_i$ covers (a part of) $\alpha_j$, then we have found an affected arc. Otherwise, if $p_i$ lies to the left of $p_j$, then we recursively search in the left subtree of the root, and else we recursively search in the right subtree. This continues until we either find an affected arc, or we reach a leaf. In the latter case $D_i$ does not contribute a new arc to $\partial U_{\text{top}}(C)$. The correctness of this procedure is guaranteed by Lemma 2.1. We can conclude the following lemma.

**Lemma 2.2.** We can maintain $U_{\text{top}}(C)$ in a data structure using $O(n)$ storage such that we can decide in $O(\log n)$ time for a query point $q$ if $q \in U_{\text{top}}(C)$. The data structure can be maintained under insertions in $O(\log n)$ amortized time.

To summarize, our point-location data structure for Union($\mathcal{D}$) consists of the following components.

- A balanced search tree $T$ storing the active strips sorted on their $x$-order, and for each active strip $\Sigma$ a balanced search tree $T_\Sigma$ on the active cells inside that strip, sorted on $y$-order.
For each active cell $C$, the partial union $U_{\text{top}}(C)$ is stored in the data structure of Lemma 2.2. The other three partial unions $U_{\text{bot}}(C)$, $U_{\text{left}}(C)$, and $U_{\text{right}}(C)$ are stored in similar data structures.

To answer a query, we first determine the grid cell $C_q$ containing the query point $q$. If $C_q$ is active, then we know that $q \in \text{Union}(D)$. Otherwise, we determine the relevant grid cells for $q$, that is, the active cells $C$ whose distance to $q$ is at most $\sqrt{2}$; these are the only cells for which $D(C)$ can contain a disk $D_i$ such that $q \in D_i$. Note that there are only $O(1)$ such cells and that they can be found in $O(\log n)$ time using the tree $T$ and the trees $T_\ell$. For each relevant cell $C$, we then query the appropriate partial union; for example, if $q$ lies above $t_{\text{top}}(C)$, then we query $T_{\text{top}}(C)$. Now $q$ lies in $\text{Union}(D)$ if and only if $q$ lies in at least one of these partial unions.

Inserting a new disk $D_i$ is done as follows. First, we determine the grid cell $C_q$ containing the center $p_i$ of $D_i$. If $C_q$ is not yet active, then we insert $C_q$ into our structure (when necessary first creating a new active strip). Next, we insert $D_i$ into each of the four partial unions stored for $C_q$. By Lemma 2.2 the whole procedure takes $O(\log n)$ amortized time.

**Theorem 3.** We can maintain a collection $D$ of $n$ congruent disks in a data structure such that we can decide in $O(\log n)$ time if a query point $q$ lies in $\text{Union}(D)$. The data structure uses $O(n)$ storage and a new disk can be inserted into $D$ in $O(\log n)$ amortized time.

### 2.5 Lower Bound for the Decision Version of Bottleneck Pyramidal TSP

Before we turn our attention to the minimization version of the bottleneck pyramidal TSP, we first show an $\Omega(n \log n)$ time lower bound for the decision version of the problem in the Euclidean plane, in the algebraic computation-tree model \cite{9}. (The algebraic computation-tree model is similar to a real RAM, the main difference being that it excludes the floor function.) The reduction even applies to the bitonic setting where the points are ordered from left to right. This bound matches the upper bound in Theorem 3.

**Theorem 4.** The bottleneck pyramidal TSP on $n$ points in the Euclidean plane has a lower bound of $\Omega(n \log n)$ in the algebraic computation-tree model.

**Proof.** We prove the lower bound by a reduction from set disjointness for integer sets $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$, for which the lower bound is known \cite{44}. Without loss of generality, we may assume that all integers are positive. We need to construct an ordered set of points $P$ and choose a bound $B > 0$ such that $U \cap V \neq \emptyset$ if and only if $P$ admits a pyramidal tour whose longest edge has length at most $B$.

Let $M := \max U \cup V$ and $B := M + 1$. We define $P = \{p_1, \ldots, p_{2n+2}\}$ with $p_i = (0, u_i)$ for $1 \leq i \leq n$, $p_{n+1} = (0, B)$, $p_{n+2} = (B, B)$, and $p_{n+2+i} = (B, v_i)$ for $1 \leq i \leq n$ (see Figure 2).

First assume $U \cap V \neq \emptyset$ and $u_i = v_j$. The pyramidal tour that first visits all $p_i$ in order except $p_{i'}$ and $p_{n+2+j'}$ and finally $p_{n+2+j'}$ and $p_{i'}$ has only edges of length at most $B$. Conversely, assume there is a pyramidal tour whose longest edge has length at most $B$. Any tour on $P$ needs to move from the line $x = 0$ to the line $x = B$ and back. Since the corresponding edges have length at most...
B, they need to connect points in \( P \) with the same \( y \) coordinate. This implies that there has to be a \( y \neq B \) such that \((0, y)\) and \((B, y)\) are in \( P \), which in turn implies that \( U \) and \( V \) are not disjoint.

The construction above does not immediately work for the bottleneck bitonic TSP, since it uses points with the same \( x \)-coordinate. However, we can slightly perturb the points to obtain unique \( x \)-coordinates.

**Theorem 5.** The bottleneck bitonic TSP on \( n \) points in the Euclidean plane has a lower bound of \( \Omega(n \log n) \) in the algebraic computation tree model.

**Proof.** Let \( \Delta := \frac{1}{4B(n+1)} \). We use the same construction as in the previous proof, except that we slightly change the \( x \)-coordinates of the points in \( P \). Concretely, we set the \( x \)-coordinate of \( p_i \) to \( i\Delta \) for \( i \leq n + 1 \) and to \( B - (i - n + 1)\Delta \) for \( i > n + 1 \). Since \((n + 1)\Delta < 1\), the distance between any \( p_i \) and \( p_j \) with \( 1 \leq i, j, \leq n + 1 \) still is less than \( B = M + 1 \), and likewise for \( n + 2 \leq i, j \leq 2n + 2 \). Thus, if \( U \cap V \neq \emptyset \), then we obtain the same tour as in the previous proof with edges of length at most \( B \). Conversely, if there is a bitonic tour with edges of length at most \( B \), then we need to check that edges crossing the line \( x = B/2 \) do not connect points with different \( y \)-coordinates. Suppose there is such an edge, then its length would be at least

\[
\sqrt{1 + (B - 2(n + 1)\Delta)^2} = \sqrt{1 + \left(B - \frac{1}{2B}\right)^2} > \sqrt{1 + B^2} - 1 = B,
\]

a contradiction. \( \square \)

### 2.6 Algorithm for the Minimization Version of Bottleneck Pyramidal TSP

In the optimization version of the bottleneck pyramidal TSP problem the goal is to minimize the length of the bottleneck edge, that is, the length of the longest edge in the tour.

The standard dynamic-programming solution for the optimization version of the pyramidal bottleneck TSP uses a table \( A[1..n, 1..n] \) where \( A[i, j] \) is defined as the minimum value for \( B \) such that there is an \((i, j)\)-partial tour of cost at most \( B \). We have

\[
A[i + 1, j] = \begin{cases} 
\max(A[i, j], |p_ip_{i+1}|) & \text{if } 1 \leq j < i, \\
\min_{1 \leq k < i} \max(A[i, k], |p_kp_{i+1}|) & \text{if } j = i,
\end{cases}
\]

where \( A[2, 1] = |p_1p_2| \). Our strategy to speed up the dynamic-programming algorithm is similar to the strategy for the non-bottleneck version in Section 2.1: We view the values \( A[k, i] \) as the weight of the point \( p_k \) in the \( i \)th iteration of the algorithm, and we maintain the points with their weights in a suitable data structure. This time the data structure needs to support the following operations:

- perform a *query* with point \( q \), which reports the value \( \min_{p_k} \max(w_k, |p_kq|) \), where the min is over all points \( p_k \) currently in the data structure;
- perform a *bulk update* of the weights, which sets \( w_j := \max(w_j, B) \) for each point \( p_j \) currently in the data structure, for a given value \( B \);
- *insert* a new point \( p_i \) with given weight \( w_i \) into the data structure.

In Section 2.7, we describe a data structure supporting these operations with \( O(\log^3 n) \) query time, \( O(\log^3 n) \) amortized insertion time and \( O(\log n) \) time for bulk updates. The structure uses \( O(n \log n) \) storage, leading to the following theorem.

**Theorem 6.** Let \( P \) be an ordered set of \( n \) points in the plane. Then, we can compute a pyramidal tour whose bottleneck edge has minimum length in \( O(n \log^3 n) \) time and using \( O(n \log n) \) storage.
2.7 Data Structure for the Minimization Version of Bottleneck Pyramidal TSP

Below, we describe a data structure that supports queries and bulk updates. To support insertions, we then apply the logarithmic method. With a slight abuse of notation, we let $P := \{p_1, \ldots, p_n\}$ denote the weighted point set stored in the data structure. Let $W$ be the (multi-)set of the weights of the points in $P$. Our data structure is defined as follows.

- The main tree is a balanced search tree $T$ whose leaves store the weights from $W$, together with the corresponding points. For a node $v$ in $T$, let $P(v)$ denote the set of points stored in the subtree rooted at $v$. We maintain the following information at $v$.
  - Let $D(p_j, w_j)$ be the disk centered at the point $p_j$ and of radius $w_j$. We store the union of the set $\{D(p_j, w_j) : p_j \in P(v)\}$, which we denote by $U(v)$, preprocessed for point location [23]. Here the weights $w_j$ refer to the weights at the time the data structure was constructed; after a bulk update the union $U(v)$ is not changed.
  - The Voronoi diagram $VD(P(v))$ of the point set $P(v)$ (using the normal Euclidean distances), preprocessed for point location.
  - The maximum weight stored in the subtree rooted at $v$.
- We maintain $B_{\max}$, the maximum value of any of the bulk updates executed since the construction of the data structure.

Since the unions and Voronoi diagrams stored at each node (and their point-location data structures) use linear storage [23, 25], the overall amount of storage of our data structure is $O(n \log n)$. The idea behind the query procedure, which will be described below, is the following lemma. Recall that a query with a point $q$ should return the value $B(q) := \min_{p_j \in P} \max(w_j, |p_jq|)$.

**Lemma 2.3.** Let $B_1 := \min\{w_j : p_j \in P \text{ and } |p_jq| \leq w_j\}$ and $B_2 := \min\{|p_jq| : p_j \in P \text{ and } w_j < B_1\}$. Then $B(q) = \min(B_1, B_2)$.

**Proof.** Define $P_1 := \{p_j \in P : w_j \geq |p_jq|\}$ and $P_2 := \{p_j \in P : w_j < |p_jq|\}$. Note that $B_1 = \min_{p_j \in P_1} w_j$ and define $B'_2 := \min_{p_j \in P_2} |p_jq|$. Clearly $B(q) = \min(B_1, B'_2)$.

Because of the definition of $B_1$, we have $\{p_j : w_j < B_1\} \subseteq P_2$. Hence, $B_2 \geq B'_2$. Furthermore, if $B_2 > B'_2$ then the point $p_j \in P_2$ minimizing $|p_jq|$ has $w_j \geq B_1$, and so $\min(B_1, B'_2) = B_1 = \min(B_1, B_2)$ in this case. Trivially $\min(B_1, B'_2) = \min(B_1, B_2)$ also holds when $B_2 = B'_2$. Hence, $B(q) = \min(B_1, B_2)$, as claimed. \(\square\)

We now describe how to perform the three operations on $T$.

**Queries.** To answer a query, we first compute the nearest neighbor, $p_k$, of $q$ in $P$. This can be done in $O(\log n)$ time by locating $q$ in $VD(root(T))$, since $P(root(T)) = P$. If $|p_kq| \leq B_{\max}$, then we can immediately conclude that $B(q) = B_{\max}$. Otherwise, we answer the query by computing $B_1$ and $B_2$, and then returning $\min(B_1, B_2)$. Next, we explain how to compute $B_1$ and $B_2$. Note that when we have to do so, we know that $|p_jq| > B_{\max}$ for all $p_j \in P$. This implies that for any given node $v$, we can decide whether there is a point $p_j \in P(v)$ with $|p_jq| \leq w_j$ by checking if $q \in U(v)$—the bulk updates we have performed since constructing $U(v)$ do not affect the outcome.

We can compute $B_1$ as follows. We start by checking if $q \in U(root(T))$. If this is not the case, then $\{p_j \in P : |p_jq| \leq w_j\} = \emptyset$, and so we set $B_1 := \infty$. Otherwise, we walk down the tree, as follows. Suppose we are at a non-leaf node $v$. Let $\mu$ be the left child of $v$. If $q \in U(\mu)$, then we descend to the left child of $v$ (that is, we set $v := \mu$), and otherwise we proceed to the right child. Since the points of $P$ are stored in the leaves of $T$ in order of their weights, the search will end in the leaf storing the point $p_j$, with the smallest weight among the nodes $p_j$ with $|p_jq| \leq w_j$. Thus, we set $B_1 := w_j$. 

Next, we need to compute $B_2$. As observed earlier, when we have to compute $B_1$ and $B_2$, we know that $|p_j q| > B_{\max}$ for all $p_j \in P$. Hence, $B_1 > B_{\max}$. This implies that whether or not a point $p_j$ satisfies $w_j < B_1$ is not affected by the bulk updates done so far—we can use the weights at the time $T$ was constructed to find the points $p_j$ satisfying $w_j < B_1$. To compute $B_2$, we now identify a collection of $O(\log n)$ nodes $v$ such that the sets $P(v)$ contain exactly the points $p_j$ with $w_j < B_1$. This can be done by searching with $B_1$ in $T$. At each of these nodes, we compute $\min_{p_j \in P(v)} |p_j q|$ by point location in VD($P(v)$), and we set $B_2$ to be the minimum of the $O(\log n)$ values computed in this manner.

Both $B_1$ and $B_2$ are computed in $O(\log^2 n)$ time—indeed, for both we spend $O(\log n)$ at each node along a path in $T$—so the total query time (before applying the logarithmic method) is $O(\log^2 n)$ time.

**Bulk updates.** A bulk update with value $B$ is performed in $O(1)$ time by setting $B_{\max} := \max(B_{\max}, B)$; no other action is needed.

**Insertions.** Insertions are handled using the logarithmic method. This increases the time for queries and bulk updates to $O(\log^3 n)$ and $O(\log n)$, respectively. The amortized time for insertions is $O((T_B(n)/n) \log n)$, where $T_B(n)$ is the time needed to build a static structure on $n$ points. This can be done bottom-up in $O(n \log^2 n)$ time: At each node $v$, we can construct the point-location data structure on the union $U(v)$ in $O(|P(v)| \log |P(v)|)$ time [23], and we can construct the Voronoi diagram in the same amount of time [25]. (Before we can construct the point-location data structure we first need to construct $U(v)$, but this can be done in $O(|P(v)| \log |P(v)|)$ time by merging the unions from the two children of $v$.) The total preprocessing time is therefore $\sum_v O(|P(v)| \log |P(v)|)$. Since $T$ is balanced, $\sum_v |P(v)| = O(n \log n)$ and so the preprocessing time is $O(n \log^2 n)$, as claimed. We conclude that the amortized time for insertions is $O(\log^3 n)$.

## 3 THE k-OPT PROBLEM IN GENERAL GRAPHs

In this section, we change the perspective from Euclidean problems to the TSP in general graphs. A tour of an undirected graph $G$ is a Hamiltonian cycle in the graph. Depending on the context, we may treat a tour as a permutation of the vertex set or as a set of edges. We consider undirected, weighted complete graphs to model symmetric TSP inputs. The weight of a tour is simply the sum of the weights of its edges. Recall that a $k$-move of a tour $T$ is an operation that replaces a set of $k$ edges in $T$ by another set of $k$ edges from $G$ in such a way that the result is a valid tour. In degenerate cases, such an operation may delete and reinsert the same edge. The associated decision problem is defined as follows.

**k-opt Detection**

**Input:** A complete undirected graph $G$ along with a (symmetric) distance function $d : E(G) \to \mathbb{N}$, an integer $k$, and a tour $T \subseteq E(G)$.

**Question:** Is there a $k$-move that strictly improves the cost of $T$?

The optimization problem $k$-opt OPTIMIZATION is to compute, given a tour in a graph, a $k$-move that gives the largest cost improvement, or report that no improving $k$-move exists.

### 3.1 On Truly Subcubic Algorithms for 3-OPT

We say that an algorithm for $n$-vertex graphs with integer edge weights in the range $[-M, \ldots, M]$ runs in truly subcubic time if its runtime is bounded by $O(n^{3-\varepsilon} \text{polylog}(M))$ for some constant $\varepsilon > 0$. Vassilevska-Williams and Williams [42] introduced a framework for relating the truly subcubic solvability of several classic problems to each other. We use it to show that the existence of a truly
negative. The edges {a_i, b_i} and {a_j, b_k} from tour T, and we reconnect the resulting pieces by the three edges {a_i, b_j}, {a_j, b_k}, and {a_k, b_i}. The three removed edges have total length 0, while the three inserted edges have negative total length.

(⇒) Now assume that there exists an improving 3-move for tour T. This improving move cannot remove any edge {b_i, a_{i+1}} or {b_n, a_1}, as these edges have length −3M, the tour T contains no edges of positive length to potentially remove, and each edge that enters the tour has length at least −M. Consequently, the three removed edges will be {a_i, b_i}, {a_j, b_j}, and {a_k, b_k} for some i < j < k. As these three edges have total length 0, the total length of the three inserted edges must be strictly negative. The edges {a_x, b_y} and {b_x, b_y} all have length 3M, while the edges {a_x, b_y} all have length between −M and M. This implies that every inserted edge is either of the type {a_x, b_y}, or coincides with one of the removed edges. Suppose for the sake of contradiction that one of the

---

6We assume that simple arithmetic on weights can be done in constant time. The polylog(M) factors used in the framework originate from repeated executions to perform binary search on weight values.
inserted edges coincides with a removed edge \{a_k, b_k\}, so that we are actually dealing with a 2-move. Then the two inserted edges in the 2-move must be \{a_i, a_j\} and \{b_i, b_j\}, so that the new tour is by $6M$ longer than the old tour $T$. This contradiction leaves only two possibilities for the three inserted edges: either \{a_i, b_j\}, \{a_j, b_k\}, \{a_k, b_i\}, or \{a_k, b_j\}, \{a_j, b_i\} (of which the latter is actually not a valid 3-move). Since the total length of the three inserted edges is strictly negative, the three vertices $v_i, v_j, v_k$ form a triangle of strictly negative weight in $G$.

The claim shows the correctness of the reduction. It is easy to perform in $O(n^2)$ time. \hfill $\square$

In the following, we give the analogous reduction in the other direction.

**Lemma 3.2.** 3-opt Detection can be reduced to Negative Edge-weighted Triangle in time $O(n^3)$ while increasing the size of the graph and the largest weight by a constant factor.

**Proof.** Consider an instance of 3-opt Detection, which is given by a complete graph $G$ together with a tour $T$ in $G$ and a symmetric distance function $d$. Number the vertices of $G$ as $v_1, \ldots, v_n$ in the order of $T$. Let $M$ be the largest absolute value of an edge weight. To simplify the notation that we will need, we first deal with two simple cases. In $O(n^2)$ time, we check whether there is an improving 2-move in $G$. If so, then we simply output a constant-size yes-instance as the output of the reduction. In the remainder it suffices to look for a 3-move that removes three edges and replaces them by three different edges. Second, we test whether there is an improving 3-move where two of the removed edges share an endpoint. This can be done in $O(n^2)$ time: there are $n$ possibilities for the shared endpoint, which determines the first two edges to leave the tour, and $n$ options for the third edge that leaves the tour. Each option can be handled in constant time.

In the remainder it therefore suffices to produce an input of Negative Edge-weighted Triangle whose answer is yes if and only if there is a 3-move that removes three distinct edges that do not share any endpoint, and replaces them by three different edges. In the remainder of this proof, we refer to such a 3-move as a proper 3-move.

To reduce the problem of finding a proper 3-move to that of finding a negative-weighted triangle, we consider the different ways in which the three paths that are obtained from $T$ by removing three edges, can be connected back into a Hamiltonian cycle of the graph by replacing them with different edges. Consider the graph $H_3$ on vertices $a_0, a_1, b_0, b_1, c_0, c_1$ with edges $\{a_1, b_0\}, \{b_1, c_0\}$, and $\{c_1, a_0\}$, which represents an abstract tour on these vertices from which edges $\{a_0, a_1\}, \{b_0, b_1\}$, and $\{c_0, c_1\}$ have been removed; see Figure 3. The removals result in three gaps: the $a$-gap (between $a_0$ and $a_1$), the $b$-gap, and the $c$-gap. Each set of 3 edges that completes $H_3$ into a cycle without inserting any of the removed edges $\{a_0, a_1\}, \{b_0, b_1\}$, or $\{c_0, c_1\}$, can be characterized by six bits $\ell(ab), r(ab), \ell(ac), r(ac), \ell(bc), r(bc) \in \{0, 1\}$ such that the edges completing the graph into a cycle are $\{a_\ell(ab), b_r(ab)\}, \{a_r(ac), c_r(ac)\}$, and $\{b_\ell(bc), c_r(bc)\}$. The bit $\ell(ab)$ specifies, for example, whether the edge connecting the $a$-gap to the $b$-gap attaches to the left side of the $a$-gap ($\ell(ab) = 0$), or to the right side of the $a$-gap. The bit $r(ab)$ specifies whether the connection between the $a$-gap and $b$-gap attaches to the left or right side of the $b$-gap, and so on. For each set of three edges
that completes $H_3$ into a cycle without re-inserting a removed edge, consider the characterization of those three edges by six bits as above, and make a weighted 3-partite connected component with $3n$ vertices $\{x_i, y_i, z_i \mid i \in [n]\}$ and edge weights defined as follows:

- $w((x_i, y_j)) = d(v_{i+\ell(ab)}, v_{j+r(\ell(ab))}) - d(v_i, v_{j+1})$ for $1 \leq i < j - 1 \leq n$;
- $w((x_i, z_k)) = d(v_{i+\ell(ac)}, v_{j+r(\ell(ac))}) - d(v_k, v_{k+1})$ for $1 \leq i < k - 1 \leq n$;
- $w((y_j, z_k)) = d(v_{j+\ell(bc)}, v_{k+r(\ell(bc))}) - d(v_j, v_{j+1})$ for $1 \leq j < k - 1 \leq n$;
- the weight for the remaining pairs in the component is $3M$.

Observe that by this definition, the weight of the triangle $x_i, y_j, z_k$ for non-consecutive integers $i < j < k$ is exactly the net weight change when removing the edges $\{v_i, v_{i+1}\}, \{v_j, v_{j+1}\}$, and $\{v_k, v_{k+1}\}$ from the tour and replacing them as specified by the characteristic bits.

The weighted graph $G'$ is the disjoint union of the connected components built for each characteristic. The weight of edges between different components is set to $3M$.

**Claim 3.2.** The constructed instance of **Negative Edge-weighted Triangle** has a triangle of negative edge-weight, if and only if the graph $G$ allows an improving proper 3-move.

**Proof.** $(\Rightarrow)$ Assume that there exists an improving proper 3-move for tour $T$ that removes the edges $\{v_i, v_{i+1}\}, \{v_j, v_{j+1}\}$, and $\{v_k, v_{k+1}\}$, producing tour $T'$, and let $i < j < k$. Since the endpoints of the removed edges are all distinct, we have $i < j - 1, j < k - 1$, and therefore $i < k - 1$. Consider the three paths $P_1, P_2, P_3$ that result from $T$ by removing the three edges in their order along the original tour, such that $P_1$ contains vertex $v_i$. These paths are contained in tour $T'$. To find the reconnection type corresponding to this move, replace each path $P_i$ by a single edge. Relabeling the left and right endpoints of $P_1, P_2, P_3$ to $\{a_0, a_i\}$, and $\{a_j, b_0\}$, and $\{b_1, c_0\}$, respectively, we can now read off the reconnection type of the tour by seeing how the inserted edges of $T'$ connect the relabeled vertices in the contracted graph. Consider the setting of the six bits $\ell(ab), r(ab), \ell(ac), r(ac), \ell(bc), r(bc) \in \{0, 1\}$ corresponding to this way of augmenting the six-vertex graph to a cycle. In the connected component corresponding to this choice of bits, the vertices $\{x_i, y_j, z_k\}$ form a triangle. The total weight of this triangle is $d(v_{i+\ell(ab)}, v_{j+r(\ell(ab))}) + d(v_{j+\ell(ac)}, v_{j+r(\ell(ac))}) + d(v_{k+r(\ell(bc)), v_{k+r(\ell(bc))}) - d(v_i, v_{i+1}) - d(v_j, v_{j+1}) - d(v_k, v_{k+1})$. As the setting of the bits corresponds to the connection type of the 3-opt move, this is exactly the sum of the weights of the newly introduced edges minus the weights of the removed edges. As the 3-opt move gave a strict weight improvement, this value is negative and hence the vertices $\{v_i, v_j, v_k\}$ from the specified component form a triangle of negative total weight.

$(\Leftarrow)$ Assume that the vertices $v_i, v_j, v_k$ span a triangle of negative edge-weight in $G'$. Since no weight is smaller than $-M$, such a triangle cannot use a pair of weight $3M$ and therefore consists of three vertices from a connected component that was added to $G'$ on account of a specific reconnection pattern. Let $i \leq j \leq k$. Since edges between vertices of the same letter also have weight $3M$, as have edges going from larger indices to smaller ones, or between indices that differ at most one, we know that $i < j - 1$ and $j < k - 1$. Our weighting scheme ensures that removing the edges $\{v_i, v_{i+1}\}, \{v_j, v_{j+1}\}, \{v_k, v_{k+1}\}$ and reconnecting the resulting pieces according to the reconnection pattern associated to the component, improves the weight of the tour by exactly the weight of triangle $\{x_i, y_j, z_k\}$. Hence, there is an improving 3-move.

The claim proves the correctness of the reduction. Since the number of characteristics is constant, the reduction can be done in $O(n^2)$ time and blows up the graph size and largest weight by only a constant factor.
paths. The inserted edges reconnect these paths at \( k \) vertices in order, by removing the edges on vertex set \( [2^k] \). For \( T \) when \( \{ \\} \) edges are removed from a \( T \), \( \{ \\} \) is a perfect matching on vertex set \( [2^k] \) consisting of a single cycle. The first gap in \( S \), \( \{ \\} \) is a signature on vertices \( [2^k] \). Is a truly subcubic algorithm for 3-opt Detection if and only if there is such an algorithm for All-Pairs Shortest Paths on weighted digraphs.

### 3.2 A fast \( k \)-OPT Algorithm

In this section, we will prove that the \( k \)-OPT Optimization problem can be solved significantly faster than \( \Theta(n^k) \) when \( k \geq 4 \). The main idea is as follows. When \( k \) edges are removed from a tour \( T \) by a \( k \)-OPT move, the tour splits into \( k \) paths. The inserted edges reconnect these \( k \) paths back into a complete tour. We introduce the notion of a signature for a \( k \)-OPT move. It describes how the \( k \) paths are reconnected, so that the signature and the removed edges together determine the entire \( k \)-OPT move. The number of different signatures will be constant for fixed \( k \). Our algorithm for finding the best \( k \)-OPT move iterates over all possible signatures, finding the best corresponding \( k \)-OPT move for each one, and outputs the overall best. The search for the best \( k \)-OPT move with a given signature can be performed in time \( O(n^{[2k/3]+1}) \) by doing a brute force search over \( n^{[2k/3]} \) options, corresponding to \( [2k/3] \) of the \( k \) edge removals in the \( k \)-OPT move, and showing that the best way to remove the remaining \( [k/3] \) edges can be found in \( \Theta(n) \) time by a Knapack-type dynamic program. The latter step crucially relies on knowing the signature of the \( k \)-OPT move that is sought for and exploits the fact that certain choices of which edges to remove, can be optimized independently.

To formalize this approach, fix some integer \( k \) and consider the graph \( H_k \) on vertex set \( [2k] \) with edges \( \{2i, 2i + 1\} \) for \( i \in [k-1] \), along with the edge \( \{2k, 1\} \). It is instructive to think of this graph as being obtained, from a cycle (a tour) that visits the \( 2k \) vertices in order, by removing the \( k \) edges \( \{2i-1, 2i\} \) for \( i \in [k] \). Graph \( H_k \) models the combinatorial structure obtained by removing \( k \) edges from a tour and thereby splitting it into \( k \) paths; see Figure 4.

A signature \( S \) is a set of \( k \) edges on vertex set \( [2k] \) whose insertion into \( H_k \) results in a single cycle (a tour) and that does not include any edge \( \{2i-1, 2i\} \) for \( i \in [k] \). (The presence of such an edge would encode that the \( k \)-OPT move removes an edge and then reinserts the same edge.) From this definition, it follows that every signature \( S \) is a perfect matching on vertex set \( [2k] \), which implies an upper bound of \( (2k-1)!! \) on the number of distinct signatures.\(^7\) For \( i \in [k] \), we refer to the pair \( \{2i-1, 2i\} \) of nonadjacent vertices in \( H_k \) as the \( i \)th gap.

\(^7\)The symbol \( !! \) denotes the double factorial, defined as \( n!! = \prod_{k=0}^{[n/2]-1} (n - 2k) \).

Consider a tour $T \subseteq E(G)$ in a weighted complete graph $G$. Fix an arbitrary orientation and starting vertex $v_1$ of $T$. This imposes a natural ordering on the edges of $T$ as $e_1, \ldots, e_n$, starting with an edge incident on $v_1$. Let $X = \{x_1, \ldots, x_k\}$ be a set of $k$ edges from $T$, indexed in increasing order along $T$. Given a signature $S$, we can apply a $k$-opt move with signature $S$ at $X$ to obtain a new tour $T'$, as follows:

- Remove the edges $X$ from tour $T$ in $G$, thereby splitting the tour into $k$ paths.
- For each $i \in [k]$, consider the edge $x_i$ on $T$. Assign its first and second endpoints along the tour the labels $2i - 1$ and $2i$, respectively. (If both tour edges incident on a vertex are in $X$, then that vertex receives two labels.)
- For each edge $(i, j) \in S$, we have $i, j \in [2k]$: insert into $T'$ the edge between the correspondingly labeled vertices in $G$.

From this definition and the fact that $H_k \cup S$ is a single cycle, it is immediate that $T'$ is a tour. As it was obtained by replacing $k$ edges, it was obtained from $T$ by a $k$-opt move. Hence, for any set of $k$ removed edges $X$ and any signature $S$, we can apply $S$ at $X$ to obtain a new tour by a $k$-opt move. For $k$-opt moves that do not remove and then reinsert the same edge—we call these proper $k$-opt moves—the converse also holds.

**Observation 3.1.** If tour $T'$ can be obtained from $T$ by a $k$-opt move that removes a set $X$ of $k$ edges and inserts a set $Y$ of $k$ edges with $X \cap Y = \emptyset$, then there is a signature $S$ such that $T'$ can be obtained from $T$ by applying $S$ at $X$.

Such a signature can be obtained by considering the tour $T'$, labeling the first and second endpoints of the $i$th removed edge as $2i - 1$ and $2i$ for all $i \in [k]$, and then seeing which labeled vertices are connected by $Y$. If $X$ contains two edges incident on the same vertex, then there are multiple distinct signatures whose application at $X$ yields tour $T'$; see Figure 5.

We now analyze the structure of signatures. For $i, j \in [k]$, we say that gaps $i$ and $j$ in $H_k$ are independent in signature $S$, if no edge of $S$ connects a vertex of gap $i$ to a vertex of gap $j$.

**Lemma 3.3.** For any signature $S$ of a $k$-opt move, there is a set $I \subseteq [k]$ of $\lceil k/3 \rceil$ independent gaps. Given $S$ and $k$, such a set can be constructed in linear time.

**Proof.** Consider the graph $F_k'$ on vertex set $[2k]$ with edge set $S$, in which every vertex has degree one, since $S$ is a perfect matching on $[2k]$. Let $F_k$ be the graph obtained from $F_k'$ as follows: for each $i \in [k]$, identify the two vertices of the $i$th gap into one vertex, keeping any parallel edges that arise. No loops can be created in this way, because the definition of signature ensures that $S$ does not contain an edge between two vertices of the same gap.
Each of the $k$ vertices of $F_k$ corresponds to a gap, and from the definition it follows that two gaps are independent if and only if the two corresponding vertices are nonadjacent in $F_k$. Since every vertex in $F'_k$ has degree one, and $F_k$ is obtained by identifying disjoint pairs of vertices, it follows that every vertex in $F_k$ has degree two. Hence, $F$ is a 2-regular graph and therefore consists of a disjoint union of cycles, some of which may consist of two vertices connected by parallel edges. Since every cycle of length $m$ contains an independent set of size at least $\lceil m/3 \rceil$ (the worst case being a cycle of length three), we find that $F_k$ contains an independent set of size $\lceil k/3 \rceil$, which corresponds to an independent set of gaps as explained above. It can be found in linear time by omitting one vertex from each cycle and selecting every other vertex on the remaining path. \hfill \Box

**Theorem 8.** For every fixed $k \geq 3$, the $k$-opt optimization problem on an $n$-vertex graph can be solved in $O(n^{\lfloor 2k/3 \rfloor + 1})$ time and $O(n)$ space.

**Proof.** We show how to find the best proper $k$-opt move in the claimed running time. The best $k$-opt move overall can then be found by running this algorithm for all $k'$ from 2 up to $k$, which satisfies the same time bounds, since $k$ is a constant.

Observation 3.1 shows that every proper $k$-opt move can be obtained by applying an appropriate signature. Since any application of a signature to $k$ target edges yields a valid tour, it suffices to iterate over all signatures $S$, find an optimal set of edges $X$ at which to apply that signature, and output the overall best. Since the number of signatures is constant for fixed $k$, it suffices to find an optimal set of locations for a fixed signature $S$ in time $O(n^{\lfloor 2k/3 \rfloor + 1})$.

Consider an input graph $G$, together with a symmetric weight function $w$ on its edges and a starting tour $T$. For each signature $S$, we find a set $X$ of locations minimizing the cost of the tour obtained by applying $S$ at $X$. We think of this as placing $k$ gaps on edges of the old tour $T$, indicating where the removals take place. We refer to gaps by the order in which their locations occur along the original tour $T = (e_1, \ldots , e_n)$, for some arbitrary but fixed choice of starting point and orientation. A placement of the $k$ gaps is a function $f : [k] \to [n]$. Since our goal is to identify the gaps in their order along $T$, we demand that $f(1) < f(2) < \cdots < f(k)$. For a subset $I \subseteq [k]$ of gaps, an $I$-partial placement is a function $f_I : I \to [n]$, satisfying $f_I(i_1) < f_I(i_2)$ for all $i_1 < i_2 \in I$. A placement $f$ extends a partial placement $f_I$ when the placements agree on the values for $I$. The cost of a placement $f$ is the cost of the tour obtained by applying signature $S$ at $X = \{f(1), \ldots , f(k)\}$.

By Lemma 3.3, there is a set $I \subset [k]$ of at least $\lceil k/3 \rceil$ independent gaps in $S$ that can be found in $O(k)$ time. Let $\bar{I} := [k] \setminus I$ be the remaining gaps. To find an optimal placement for the $k$ gaps, we iterate over all $O(n^{\lceil k/3 \rceil}) = O(n^{\lfloor 2k/3 \rfloor})$ possible $\bar{I}$-partial placements $f_{\bar{I}}$. For each such partial placement, we will show that an optimal extension can be computed in $O(n)$ time, yielding the desired runtime.

Consider an $\bar{I}$-partial assignment $f_{\bar{I}}$. To see how the independence of the gaps $I$ can be exploited, consider a placement $f$ that extends $f_{\bar{I}}$, and let $T'$ be the corresponding tour. The cost of $T'$ can be obtained from the cost of $T$ by subtracting the total weight of the tour edges $e_{f(i)}$ for $i \in [k]$, and adding the cost of the edges whose insertion is dictated by $S$. We present an alternative way to characterize the cost change, which will be useful algorithmically.

For the index $i \in I \subseteq [k]$ of a gap, and an index $j \in [n]$ of an edge along the original tour $T$, we define the price $c[i,j]$ of placing the $i$th gap at location $j$, as follows. The signature $S$ contains two edges $e, e'$ that connect endpoints of the $i$-gap to the endpoints of other gaps. Since $i \in I$ is an independent set of gaps, edges $e$ and $e'$ each connect one endpoint of the $i$-gap to the endpoint of a gap in $\bar{I}$. The locations of gaps in $\bar{I}$ are already determined by the $\bar{I}$-partial placement $f_{\bar{I}}$. Hence, from $f_{\bar{I}}$ and $S$, we can deduce which edges $\bar{e}, \bar{e'}$ of $G$ incident on the endpoints of the $j$th tour edge would be inserted, if the $i$th gap would be placed at the $j$th tour edge $e_j$. We define $c[i,j] := w(\bar{e}) + w(\bar{e'}) - w(e_j)$ as the price of placing the $i$th gap at location $j$. Using this notation, the cost
of the tour $T'$ obtained by applying $S$ at placement $f$ extending $f_T$ can be obtained from the cost of $T$ by:

1. subtracting the weight of the edges with indices $f_T(i)$ for $i \in \bar{I}$,
2. adding the weight of all inserted edges that connect two $\bar{I}$-gaps,
3. adding the price $c[i, f(i)]$ for all $i \in I$.

For a given $\bar{I}$-partial placement, the first two parts are fixed and only the last part depends on how $f_T$ is extended to a complete placement. The definition of independent gaps crucially ensures that no inserted edges connect two gaps in $I$, so that the entire cost change of the move is indeed accounted for by the three parts.

The characterization above implies that to find an optimal extension $f$ of $f_T$, it suffices to find an extension that minimizes the total price of the $I$-gaps, under the constraint that $f(1) < f(2) < \cdots < f(k)$. We enforce the ordering constraint by searching for an $I$-partial placement (so that the values assigned to $I$-gaps are increasing), which maps any $I$-gap to a location between the first $\bar{I}$-gap before it, and the first $\bar{I}$-gap after it. We therefore assign a lower bound $\ell(i)$ and upper bound $u(i)$ for where each gap $i \in I$ may be placed. Let $\ell(i) := 0$ if $f_T$ does not assign a location to any gap with index smaller than $i$; otherwise let $\ell(i) := f_T(i^-)$ where $i^-$ is the largest index in $\bar{I}$ that is smaller than $i$. Similarly, let $u(i) := n + 1$ if $f_T$ does not assign a location to any gap with index larger than $i$; otherwise let $u(i) := f_T(i^+)$, where $i^+$ is the smallest index in $\bar{I}$ that is larger than $i$.

Using these values, we present a simple dynamic program that computes the total price of an optimal extension of $f_T$. Let $I := \{g_1, \ldots, g_j\}$. For $0 \leq i \leq t$ and $0 \leq j \leq n$ let $T[i, j]$ denote the minimum total price of a partial placement $f_i$ of gaps $\{g_1, \ldots, g_j\}$, such that every gap is assigned a location in the range $\{1, \ldots, j\}$ that lies strictly between its lower and upper bound. Here the price of a placement $f_i$ is defined as $\sum_{\ell \in [i]} c[g_{\ell}, f_{\ell}(\ell)]$. Then $T$ satisfies the following recurrence:

$$T[i, j] = \begin{cases} 0 & \text{if } i = 0, \\ +\infty & \text{if } j = 0 \land i > 0, \\ T[i, j - 1] & \text{if } j \leq \ell(g_i) \lor j \geq u(g_i), \\ \min(T[i, j - 1], T[i - 1, j - 1] + c[g_i, j]) & \text{otherwise.} \end{cases}$$

We briefly explain the recurrence. If $i = 0$, then no gaps have to be assigned a location and the price is 0. If some gaps have to be assigned a location ($i > 0$), but no locations are available ($j = 0$), then no solution exists and the price is infinite. In the third case, we cannot assign gap $g_i$ to location $j$, since it is outside the allowed range, so the optimal placement assigns gap $g_i$ to location $j - 1$ or earlier, and the optimal cost is $T[i, j - 1]$. Finally, in the fourth case, we have the option of placing gap $g_i$ at location $j$, paying its price, and placing the $i - 1$ earlier gaps at edges with smaller indices; or not placing gap $g_i$ at location $j$, but placing the first $i$ gaps into the first $j - 1$ locations.

Using this recurrence the values of $T$ can be computed by dynamic programming in $O(nk)$ time and space, which is $O(n)$, since $k$ is fixed. From the table $T$, a corresponding optimal solution can easily be obtained in the same time using standard techniques. From our discussion above, it follows that a minimum-price constrained placement of the $I$-gaps combines with the given $\bar{I}$-partial placement, into a minimum cost $k$-opt move that extends $f_T$. Since this completes any $\bar{I}$-partial placement into an optimal placement for a given signature $S$, while we iterate over all $\bar{I}$-partial placements and all signatures, this yields an optimal solution.

4 FASTER 2-OPT

In this section, we show that it is possible to beat the quadratic barrier for 2-opt in two important settings, namely, when we want to apply 2-moves repeatedly, and in the Euclidean setting in the plane.
4.1 Repeated 2-OPT

In the repeated 2-opt problem, we apply 2-opt repeatedly (e.g., until no further improvements are possible). One can considerably speed up the 2-opt computations at each of the iterations, except the first one. The following theorem gives our improvement for the 2-opt Optimization problem, where the goal is to find the best 2-move (rather than any 2-move that improves the tour).

**Theorem 9.** After \(O(n^2)\) preprocessing and using \(O(n^2)\) storage, we can repeatedly solve the 2-opt Optimization problem in \(O(n \log n)\) time per iteration.

**Proof.** Let \(T\) be the current tour, which is either the initial tour or the tour resulting from the previous operation. Note that a 2-move not only replaces a pair of edges by another pair, but that it also reverses the subpath connecting these edges. To avoid spending time on each edge of the subpath when we perform a 2-move, we borrow an idea from Chrobak et al. [18] that was also used by Fredman et al. [26]: We store the tour in a tree, and with each node \(v\) we store a Boolean \(\text{Rev}(v)\) indicating whether the subpath represented by the subtree \(T_v\) rooted at \(v\) should be reversed. (Fredman et al. [26] also use this idea to speed up 2-opt. However, their goal is only to be able to perform a 2-move efficiently, and so they only maintain one such tree for the whole tour. Our goal is to find a 2-move efficiently.) In fact (and unlike Fredman et al.), we will maintain \(n\) such trees—for each edge \(e\) in the tour, we maintain a tree \(T_e\) on the path \(P(e) := T \setminus \{e\}\)—and we augment these trees with extra information, so that we can quickly find the best edge for \(e\) to perform a 2-move with. The tree \(T_e\) is defined as follows.

Fix an arbitrary orientation for \(e\). This induces an orientation on the tour \(T\) and, hence, on the path \(P(e)\). The tree \(T(e)\) is a red-black tree storing the edges from \(P(e)\) in its leaves and storing a Boolean \(\text{Rev}(v)\) at each node \(v\). Initially the order of the edges corresponds to the order along \(P(e)\) and all Booleans \(\text{Rev}(v)\) are set to False. Later the order of the edges along \(P(e)\) may no longer correspond to the order of the leaves, but the correct order can always be restored by “pushing down” the reversals in a top-down manner. (To push down a reversal for a node \(v\) with \(\text{Rev}(v) = \text{True}\) we swap the left and right subtree of \(v\), set \(\text{Rev}(v)\) to False, and negate the Booleans \(\text{Rev}()\) of the children of \(v\). This operation is called clearing the node by Fredman et al. Note that swapping two subtrees of a node does not influence the red-black properties.) So far our tree is essentially the same as that of Chrobak et al. and Fredman et al. We now augment \(T(e)\) as follows.

Let the local orientation of an edge \(e'\) in \(P(e)\) at the leaf \(v\) where it is stored be defined as follows: if \(\text{Rev}(v) = \text{False}\) then the local orientation is the orientation along \(P(e)\) when \(T(e)\) was constructed (that is, before any reversals took place), otherwise it is the opposite orientation. The local orientation of \(e'\) at an internal node \(v\) with \(e'\) in its subtree is defined recursively: If \(\text{Rev}(v) = \text{False}\), then the local orientation of \(e'\) at \(v\) is equal to the local orientation at the relevant child of \(v\), and if \(\text{Rev}(v) = \text{True}\), then it is the reverse of that orientation. Note that the local orientation of \(e'\) at the root of \(T(e)\) is equal to the current orientation of \(e'\) in \(P(e)\). We store the following extra information at each node \(v\):

- A value \(\text{MinCost}(v)\), which is defined as the minimum over all edges \(e'\) in \(T_v\) of the cost of the 2-move defined by \(e\) and \(e'\) for the local orientation of \(e'\) at \(v\). We also store a pointer to the edge \(e'\) defining the minimum. Note that for a leaf node \(v\), the value \(\text{MinCost}(v)\) is simply the cost of the 2-move defined by \(e\) and the edge stored at \(v\).
- A value \(\text{MinRevCost}(v)\) (with the corresponding pointer), which is defined similar to \(\text{MinCost}(v)\), except that we consider the reverse of the local orientations.

Note that if \(v_1\) and \(v_2\) are the two children of \(v\), then

\[
\text{MinCost}(v) = \begin{cases} 
\min(\text{MinCost}(v_1), \text{MinCost}(v_2)) & \text{if } \text{Rev}(v) = \text{False}, \\
\min(\text{MinRevCost}(v_1), \text{MinRevCost}(v_2)) & \text{if } \text{Rev}(v) = \text{True}.
\end{cases}
\]
Similarly, \( \text{MinRevCost}(\nu) \) can be computed in \( O(1) \) time from the information at \( \nu, \nu_1, \) and \( \nu_2 \). Note that when \( \text{Rev}(\nu) \) is negated, we can just swap the values of \( \text{MinCost}(\nu) \) and \( \text{MinRevCost}(\nu) \) and propagate the change upward. For each edge \( e' \), we also maintain, for each tree \( T(e) \), a pointer to the leaf where \( e' \) is stored. Next, we show how to use the trees \( T(e) \) to perform a 2-opt iteration in near-linear time.

Finding the best 2-move in \( O(n) \) time is easy: We simply go over all trees \( T(e) \) to find the one minimizing \( \text{MinCost}(\text{root}(T(e))) \). Let \( e' \) be the edge defining this value. We now have to perform a 2-move on \( e, e' \) (assuming \( \text{MinCost}(\text{root}(T(e))) \) is negative, that is, that the 2-move actually reduces the cost of the tour). Performing the 2-move is done as follows. We first walk from the leaf storing \( e' \) back up to the root, to determine the current orientation of \( e' \). With that information we can compute the edges \( \tilde{e} \) and \( \tilde{e}' \) that should replace \( e \) and \( e' \). We then reconstruct the current path \( P(e) \) in \( O(n) \) time by clearing all nodes in \( T(e) \) in a bottom-up manner, after which we destroy the trees \( T(e) \) and \( T(e') \). Next, we build trees \( T(\tilde{e}) \) and \( T(\tilde{e}') \) from scratch. This can be done in \( O(n) \) time after constructing the paths \( P(\tilde{e}) \) and \( P(\tilde{e}') \), which we can do in \( O(n) \) time, since we have the path \( P(e) \). It remains to update the other trees. In the remainder of the proof, we show how this can be done in \( O(\log n) \) time per tree, resulting in \( O(n \log n) \) time in total for a 2-move.

We show how to update a tree \( T(f) \) in logarithmic time when a 2-move with edges \( e, e' \) is performed; see also Figure 6. Note that rotations in \( T(f) \) can still be done in \( O(1) \) time after clearing the two nodes on which the rotation is performed. Thus, standard operations on augmented red-black trees can still be performed in logarithmic time. These operations include insertions and deletions but also splits and concatenations. In a split operation in a normal red-black tree one is given a value \( X \), and the goal is to split the tree into two new trees: one containing the elements smaller than \( X \), and one containing the elements larger than \( X \). We will need to split \( T(f) \), given an edge \( e \in P(f) \), into two trees: one for the part of \( P(f) \) before \( e \), and one for the part starting at \( e \). This is possible in the usual way, provided we first clear all nodes on the path from the root of \( T(f) \) to the leaf containing \( e \). Similarly, concatenating two trees—the reverse operating from splitting—can be done in \( O(\log n) \) time. See also the paper by Chrobak et al. [18], who describe these operations (for AVL-trees) and without the extra fields \( \text{MinCost}() \) and \( \text{MinRevCost}() \). We can now update \( T(f) \) (to reflect a 2-move where edges \( e \) and \( e' \) are replaced by new edges \( \tilde{e} \) and \( \tilde{e}' \)) as follows.

We first split \( T(f) \) into two subtrees, a tree \( T_1 \) for the subpath of \( P(f) \) before \( e \), and a tree \( T_2 \) for the subpath starting at \( e \). The latter tree is then split further into a tree \( T_{2,1} \) for the subpath from \( e \) to \( e' \), and a tree \( T_{2,2} \) for the subpath behind \( e' \). We then delete \( e \) and \( e' \) from \( T_{2,1} \), reverse the

Fig. 6. Situation when a 2-move with edges \( e, e' \) has to be performed on a tree \( T(f) \). When all nodes on the search paths to \( e \) and \( e' \) are cleared, the subtrees in between the search paths together represent the subpath from \( e \) to \( e' \) along \( P(f) \).
subpath in between them by negating the Boolean $\text{Rev}(\cdot)$ at the root of $\mathcal{T}_{e,1}$, and insert $e$ as first edge of the subpath and $e'$ as last edge. We then concatenate the three subtrees again to obtain the new tree $\mathcal{T}(f)$.

We conclude that each tree $\mathcal{T}(f)$ can be updated in $O(\log n)$ time after a 2-move. \qed

It is likely that these techniques can be extended to speed up repeated 3-opt as well. As the technical details become substantially more cumbersome, we do not pursue this direction.

4.2 Faster 2-OPT and 3-OPT for Euclidean TSP in the Plane

For points in the plane (and under the Euclidean metric), we can speed up 2-opt and 3-opt computations by using suitable geometric data structures for semi-algebraic range searching, as shown below. This leads to the following theorem.\(^8\)

**Theorem 10.** For any fixed $\epsilon > 0$, 2-opt detection in the plane can be solved in $O(n^{8/5+\epsilon})$ time, and 3-opt detection in the plane can be solved in $O(n^{80/31+\epsilon})$ expected time.

**2-opt.** Suppose we are given a tour $T$ on a planar point set $P := \{p_0, p_1, \ldots, p_{n-1}\}$, where we assume without loss of generality that the points are numbered in order along $T$. The idea is to preprocess $P$ such that we can answer the following queries: given a query edge $e = p_ip_{i+1}$ of $T$, find an edge $e' = p_jp_{j+1}$ in $T$ such that performing a 2-move on $e$, $e'$ lowers the cost of $T$ (if such an edge exists). In other words, we want to find an edge $pjp_{j+1}$ such that

$$|p_ip_j| + |p_{i+1}p_{j+1}| < |p_ip_{i+1}| + |pjp_{j+1}|.$$  \(^{(4)}\)

To answer these queries, we map every edge $pjp_{j+1}$ to a point $q_j := (x(p_j), y(p_j), x(p_{j+1}), y(p_{j+1}))$ in $\mathbb{R}^4$, and we preprocess the resulting set of points in $\mathbb{R}^4$ for range queries with semi-algebraic sets,\(^9\) as described next. Given a query edge $p_ip_{i+1}$ we define a range $Q_i \subset \mathbb{R}^4$ as

$$Q_i := \{ (a_1, a_2, a_3, a_4) : |p_i(a_1, a_2)| + |p_{i+1}(a_3, a_4)| < |p_ip_{i+1}| + |(a_1, a_2)(a_3, a_4)| \}.$$  

Thus, $pjp_{j+1}$ satisfies Equation (4) if and only if $q_j \in Q_i$. We can therefore find an edge $pjp_{j+1}$ satisfying Equation (4) by performing a query with the range $Q_i$, which is (that is, can be rewritten to) a semi-algebraic set in $\mathbb{R}^4$. Semi-algebraic range searching queries can be answered in $O(n^{3/4+\epsilon})$ time after $O(n^{1+\epsilon})$ preprocessing by combining the technique from Agarwal and Matoušek [4] with a result by Koltun [33]. Alternatively, we can "dualize" the approach, by mapping each edge $pjp_{j+1}$ to a surface in $\mathbb{R}^4$ and mapping the query $pjp_{j+1}$ to a point $q_i$. By performing point location with $q_i$ in the arrangement defined by these surfaces we can then answer the queries. This takes $O(\log n)$ time after $O(n^{3+\epsilon})$ preprocessing [33]. By combining these two solutions in a standard manner, we can obtain a trade-off between preprocessing and query time—see, e.g., Reference [37] and also below, where we give some more details for the somewhat more complicated case of 3-opt. In particular, we can obtain $O(n^{1+\epsilon}/s^{1/4})$ query time using $O(s^{1+\epsilon})$ preprocessing, for any parameter $s$ with $n \leq s \leq n^4$. Setting $s = n^{8/5}$ gives $O(n^{3+\epsilon})$ query time after $O(n^{8/5+\epsilon})$ preprocessing. Thus, our 2-opt algorithm needs $O(n^{8/5+\epsilon})$ time in total.

**3-opt.** For 3-opt, we proceed similarly as for 2-opt. We preprocess the tour $T$ for the following queries: given a query edge $p_ip_{i+1}$, find a pair of edges $pjp_{j+1}, p_kp_{k+1}$ such that a 3-move involving these three edges will reduce the cost of the tour (if such a pair exists). The details are a bit more involved than for 2-opt, however.

\(^8\)Note that we do not consider the repeated version of the problem, but the single-shot version.

\(^9\)A semi-algebraic set is a subset of $\mathbb{R}^d$ obtained from a finite number of sets of the form $\{q \in \mathbb{R}^d : g(q) \geq 0\}$, where $g$ is a $d$-variate polynomial, by Boolean operations (taking unions, intersections, and complements).

Assume the points are numbered $p_0, \ldots, p_{n-1}$ in order along $T$. Define $e_i$ to be the edge $p_ip_{i+1}$, for $0 \leq i < n$, and consider a 3-move involving edges $e_i, e_j, e_k$ with $i < j < k$. The four possible triples to replace $e_i, e_j, e_k$ in a valid 3-move are

- $p_ip_j, p_{i+1}p_k, p_{j+1}p_{k+1}$ (Type I);
- $p_ip_{j+1}, p_{k}p_{i+1}, p_{j}p_{k+1}$ (Type II);
- $p_ip_{j+1}, p_kp_{i+1}, p_jp_{k+1}$ (Type III);
- $p_ip_k, p_{j+1}p_{i+1}, p_{j}p_{k+1}$ (Type IV).

Here, we have ignored the possibility that one of the edges $e_i, e_j, e_k$ re-appears in the new triple, and we thus have a 2-move; these “degenerate” 3-moves can be found as described above. Note that we may have $i + 1 = j$ and/or $j + 1 = k$. In this case some of the four 3-moves just mentioned also become degenerate, but this is not a problem. Indeed, these 3-moves still result in a valid tour, and if the tour length is reduced, we still want to find such a degenerate 3-move. We are left with the problem of deciding whether there is a 3-move of one of the four types described above that reduces the length of the tour. We explain how to do this for 3-moves of Type I; the other three types can be handled similarly.

To find an improving 3-move of Type I, we store all pairs $e_i, e_j$ with $0 \leq i < j < n - 1$ in a data structure that can answer the following queries: given an edge $e_k$, find a pair $e_i, e_j$ such that

$$j < k \text{ and } |p_ip_j| + |p_{i+1}p_k| + |p_{j+1}p_{k+1}| < |p_ip_{i+1}| + |p_jp_{j+1}| + |p_kp_{k+1}|,$$

if such a pair exists. For the moment, let us ignore the condition $j < k$. Then, we can proceed similarly as in the 2-opt case: We map every pair $e_i, e_j$ to a point

$$q_{ij} := (x(p_i), y(p_i), x(p_{i+1}), y(p_{i+1}), x(p_j), y(p_j), x(p_{j+1}), y(p_{j+1}))$$

in $\mathbb{R}^8$, and we preprocess the resulting set of points for range queries with semi-algebraic sets [5]. Given a query edge $p_kp_{k+1}$, we can now decide if there is an improving 3-move of Type I by searching with the range

$$Q_k := \{(a_1, \ldots, a_8) : |(a_1, a_2)(a_3, a_4)| + |(a_3, a_4)p_k| + |(a_2, a_5)p_{k+1}|$$

$$< |(a_1, a_2)(a_3, a_4)| + |(a_2, a_5)(a_7, a_8)| + |p_kp_{k+1}|\}.$$

The resulting data structure uses $O(n')$ space and has $O((n')^{1+\epsilon})$ expected preprocessing time and $O((n')^{7/8+\epsilon})$ query time, where $n'$ is the number of points stored in the data structure. (The preprocessing time is expected, since we are doing semi-algebraic range searching in $\mathbb{R}^8$, which requires us to use the result by Aggarwal et al. [5]. For 2-opt we were doing semi-algebraic range searching in $\mathbb{R}^4$, which can be done with earlier, deterministic techniques [4, 33].)

Alternatively, we can map every pair $e_i, e_j$ to a surface

$$\Gamma_{ij} := \{(a_1, \ldots, a_4) : |p_ip_j| + |p_{i+1}(a_1, a_2)| + |p_{j+1}(a_3, a_4)|$$

$$= |p_ip_{i+1}| + |p_jp_{j+1}| + |(a_1, a_2)(a_3, a_4)|\}$$

in $\mathbb{R}^4$, and preprocess the resulting arrangement for point location. The location of the point $(x(p_k), y(p_k), x(p_{k+1}), y(p_{k+1}))$ in this arrangement now tells us if there is an improving 3-move of Type I. This alternative would use $O((n')^{4+\epsilon})$ preprocessing time and have $O(\log n')$ query time [33].

The standard way to obtain a trade-off between preprocessing and query time is as follows. The linear-space variant is a recursively defined tree structure on the points in the input set (which is in our case the set $\{q_{ij} : 0 \leq i < j < n - 1\}$). Now, instead of continuing the recursion all the way until only constantly many points are left, we stop when the number of points falls below a suitable threshold $m$ with $1 \leq m \leq n'$. (The value of $m$ determines the trade-off.) At this point,
we dualize the problem and build the logarithmic query-time solution, which in our case uses $O(m^{4+\epsilon})$ preprocessing time. This way, we construct a "top tree" with $O(n'/m)$ leaves, each of which is associated with a "bottom tree" that needs $O(m^{3+\epsilon})$ preprocessing. The total amount of preprocessing is $O(n'm^{3+\epsilon})$.

A query is performed by first searching in the top tree. The search ends up in $O((n/m)^{7/8+\epsilon})$ leaves where the search is then continued in the corresponding bottom tree. Thus, the query time is $O((n'/m)^{7/8+\epsilon})$ (for a slightly larger $\epsilon$, which swallows the extra log-factor from searching in the bottom tree).

So far, we ignored the condition $j < k$ in Equation (5). Fortunately this condition is easy to handle, as it simply adds a so-called range restriction to the query. Range restrictions can be added at the cost of an extra log-factor in preprocessing and query time [43]. In our case these logarithmic factors are swallowed by the $O(n')$ factor that we already have, so the total structure uses $O(n'm^{3+\epsilon})$ expected preprocessing time and has $O((n'/m)^{7/8+\epsilon})$ query time, where $m$ is a parameter that we can still change to optimize performance.

Recall that $n'$, the number of points stored in the data structure, is $n^2$, and that we have to perform $n$ queries—one for each edge $e_k$. Thus, the total time of our algorithm is

$$O(n^2m^{3+\epsilon}) + n \cdot O((n^2/m)^{7/8+\epsilon}) = O(n^2m^{3+\epsilon} + n^{22/8+\epsilon}/m^{7/8}).$$

This is minimized when we set $m := n^{6/31}$, which gives a total expected runtime of $O(n^{80/31+\epsilon}) = O(n^{2.59})$.

5 CONCLUSION

Revisiting the worst-case complexity of $k$-opt and pyramidal TSP led to a number of new results on these classic problems. Some, such as the equivalence between 3-opt and APSP with respect to having truly subcubic algorithms, rely on very recent work. Other results, such as the near-linear time algorithm for finding bitonic tours, and the $k$-opt algorithm that beats the trivial $O(n^4)$ upper bound, are obtained using classic techniques. In this respect, it is surprising that these results were not found earlier. These examples show that the availability of new lower bound machinery can inspire new algorithms.

Our findings suggest several directions for further research, both theoretical and applied. An interesting open problem regarding $k$-opt Detection is whether the problem is fixed-parameter tractable when improving a given tour in an edge-weighted planar graph. This question was also asked by Marx [36] and Guo et al. [29]. Similarly, it is open whether the problem is fixed-parameter tractable when improving a given tour among points in the Euclidean plane. It would be interesting to settle the exact complexity of $k$-opt in general weighted graphs. Following the publication of our extended abstract, Cygan et al. [21, 22] showed how the running time of $k$-opt Optimization can be further improved using tree decompositions. They obtain a runtime of $O(n^{(1/4+\epsilon_k)k})$, where $\lim_{k \to \infty} \epsilon_k \to 0$. For $k = 5$ their algorithm runs in time $O(n^{3.4})$, improving on our $O(n^4)$ algorithm from Theorem 8. Can optimal 5-moves be found in time $O(n^3)$? What is the optimal dependence of the exponent on $k$ in general? (Recall that the lower bound by Guo et al. [29] shows that $k$-opt Detection cannot be solved in time $f(k) \cdot n^{o(k/\log k)}$ for any function $f$, unless ETH fails.)

Recent follow-up work by Bonnet et al. [12] considers $k$-opt Detection in bounded-degree graphs, which remains W[1]-hard. When the weight values are polylogarithmic in $n$ and the maximum degree is constant, they obtain a quasi-linear time algorithm to find an improving 8-move, and show that the case $k = 9$ does not admit quasi-linear time algorithms unless triangles in unweighted graphs can be detected in edge-linear time.

When all weights lie in the range $[-M, \ldots, M]$, one can detect a negative triangle in an edge-weighted graph in time $O(M \cdot n^\omega)$ using fast matrix multiplication [6, 40, 45]. By our reduction,
this gives an algorithm for 3-opt Detection with weights \([-M, \ldots, M]\) in time \(O(M \cdot n^\omega)\). Can similar speedups be obtained for \(k\)-opt for larger \(k\)?

Given the great industrial interest in TSP, establishing the practical applicability of these theoretical results is an important follow-up step. Several of our results rely on data structures that are efficient in theory but which are currently impractical. These include the additively weighted Voronoi diagram used for pyramidal tours on points in the plane, and the semi-algebraic range searching data structures used to speed up 2-opt Detection. In contrast, the \(O(n^{\lfloor 2k/3 \rfloor + 1})\) algorithm for finding the best \(k\)-move improvement is self-contained, easy to implement, and may have practical potential.

ACKNOWLEDGMENTS

We are grateful to Hans L. Bodlaender, Karl Bringmann, and Jesper Nederlof for insightful discussions, an anonymous referee for the observation in Footnote 1, and Christian Knauer for the observation in Footnote 2.

REFERENCES


[34] Marvin Künemann and Bodo Manthey. 2015. Towards understanding the smoothed approximation ratio of the 2-opt heuristic. In *Proceedings of the 42nd ICALP*, 859–871. DOI: https://doi.org/10.1007/978-3-662-47672-7_70


Received January 2020; accepted July 2020