Fine-grained complexity analysis of two classic TSP variants

arXiv:1607.02725v1 [cs.DS] 10 Jul 2016

Fine-Grained Complexity Analysis of Two Classic TSP Variants

Mark de Berg* Kevin Buchin† Bart M. P. Jansen‡ Gerhard Woeginger§

Abstract

We analyze two classic variants of the Traveling Salesman Problem 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 $\mathcal{O}(n^2)$ 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 $\mathcal{O}(n \log^2 n)$ time and its bottleneck version in $\mathcal{O}(n \log^3 n)$ 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 $\mathcal{O}(n^k)$ 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 $\widetilde{\mathcal{O}}(n^{3-\varepsilon})$ 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 $\mathcal{O}(n^{\lfloor 2k/3 \rfloor + 1})$ time for fixed $k \geq 3$. This implies that 4-opt can be solved in $\mathcal{O}(n^3)$ 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.

*Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands. m.t.d.berg@tue.nl. Supported by NWO Gravity grant "Networks" (024.002.003).

†Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands. k.a.buchin@tue.nl. Supported by NWO grant "A framework for progressive, user-steered algorithms in visual analytics" (612.001.207).

‡Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands. b.m.p.jansen@tue.nl. Supported by NWO Veni grant "Frontiers in Parameterized Preprocessing" and NWO Gravity grant "Networks" (024.002.003).

§Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands. gwoegi@win.tue.nl. Supported by NWO Gravity grant "Networks" (024.002.003).

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, Leiserson, Rivest, and Stein [17]. 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-OPT, a popular local search heuristic that attempts to improve a suboptimal solution by a k-OPT *move* (or: *k-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 $k = 2$ [18] and $k = 3$ have been studied extensively with respect to various aspects such as experimental performance [9, 27, 30], (smoothed) approximation ratio [15, 29], rate of convergence [15, 19], and algorithm engineering [21, 23, 33, 34]. The decision problem associated with k-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-OPT improvement in time $\mathcal{O}(n^k)$ for each fixed k. The survey chapter [26] by Johnson and McGeoch extensively discusses k-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 $\mathcal{O}(n^2)$ for bitonic tours and $\mathcal{O}(n^k)$ for finding k-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 the problems of computing the LONGEST COMMON SUBSEQUENCE [1, 12] of two length-n strings, the EDIT DISTANCE [7, 12] from one length-n string to another, or the DISCRETE FRÉCHET DISTANCE [11] between two polygonal n-vertex curves in the plane. If one of these problems allows an algorithm with running time $\mathcal{O}(n^{2-\varepsilon})$, then this would yield an algorithm to test the satisfiability of an n-variable CNF formula ϕ in time $(2-\varepsilon)^n \cdot |\phi|^{\mathcal{O}(1)}$. As decades of research have not led to algorithms with such a running time for CNF-SAT, this gives evidence that the classic $\mathcal{O}(n^2)$-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 $i_1 < i_2 < \ldots < i_r$ and $j_1 > j_2 > \ldots > j_{n-r-2}$. 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 [8, 13, 22]. 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. 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 [14].

We will show that the classic $\mathcal{O}(n^2)$ 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 $\mathcal{O}(n \log^2 n)$. To the best of our knowledge, this presents the first improvement in finding bitonic tours since the problem was popularized in *Introduction to Algorithms* [17] in 1991. In fact, we prove the stronger result that an optimal *pyramidal* tour among n points in the plane can be computed in $\mathcal{O}(n \log^2 n)$ 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 $\mathcal{O}(n \log n)$ 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 $\mathcal{O}(n \log^3 n)$ 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 [39]; 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 [31] 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 $f(k) \cdot n^{\mathcal{O}(1)}$. Guo *et al.* [24] refined this result and proved that, under the *Exponential Time Hypothesis* [25], 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 paper is: can one do better than $\mathcal{O}(n^k)$ 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 $\mathcal{O}(n^{3-\varepsilon}\,\mathrm{polylog}(M))$ for $\varepsilon > 0$. 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 $\mathcal{O}(n^2)$ 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 in weighted digraphs (APSP) with respect to having truly subcubic algorithms [37], 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, 37].

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 $\mathcal{O}(n^{\lfloor 2k/3 \rfloor + 1})$ time for each fixed value of k. For $k = 2$ and $k = 3$, this expression simply boils down to the straightforward time complexities of $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ for 2-OPT and 3-OPT respectively. For $k \geq 4$, however, our result yields a substantial improvement over the trivial $\mathcal{O}(n^k)$ 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 $\mathcal{O}(n^2)$ preprocessing time, one can repeatedly find the best 2-OPT improvement in $\mathcal{O}(n \log n)$ 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 $\mathcal{O}(n^{8/5+\varepsilon})$ for all $\varepsilon > 0$. Similarly, 3-OPT DETECTION can be solved in expected time $\mathcal{O}(n^{80/31+\varepsilon})$.

2 Faster pyramidal TSP

In this section we show that the pyramidal TSP and the bottleneck pyramidal TSP problem 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.

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_1. 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_i—and the other path ends at p_j.

Our starting point is the standard dynamic-programming solution. It uses a 2-dimensional table[1] $A[1..n, 1..n]$, 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

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

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'' ends at index $j < i$ 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 can be found by computing $\min_{1 \leq k < n} \left(A[n, k] + |p_k p_n| \right)$. There are $\mathcal{O}(n^2)$ entries in A of the first type that each take constant time to evaluate. There are $\mathcal{O}(n)$ entries of the second type that need time $\Theta(n)$. Hence the dynamic program can be evaluated in $\mathcal{O}(n^2)$ time.

Our subquadratic algorithm is based on the following two observations. First, any two subsequent rows $A[i, 1..n]$ and $A[i+1, 1..n]$ 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 (see below).

Recall that $P_i := \{p_1, \ldots, p_i\}$. The point that defines $\min_{1 \leq k < i} \left(A[i, k] + |p_k p_{i+1}| \right)$ 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

$$\text{dist}(p_k, q) := w_k + |p_k q|, \tag{2}$$

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 Δ 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.

Answering nearest-neighbor queries for the weighted point set P can be done by performing point location in the *additively weighted Voronoi diagram* [20] of P augmented by a point location data structure [36]. This (static) data structure has size $\mathcal{O}(n)$, can be computed in $\mathcal{O}(n \log n)$ time, and allows for $\mathcal{O}(\log n)$-time queries. To allow for insertions we use the logarithmic method [10]. The logarithmic method makes a data structure semi-dynamic by storing $\mathcal{O}(\log n)$ static data structures of increasing size (resulting in an additional log-factor in the query time). The main observation is that we can handle bulk updates by storing a correction term for the weights with each of the static additively weighted Voronoi diagrams. The additively-weighted

[1] Some of our results can also be obtained from an alternative DP with n states. As we need the 2-dimensional approach for Theorem 4, we present all our results in this setting.

nearest neighbor structure does not change when adding the same constant to each point weight, which means we do not have to update the Voronoi diagrams when performing bulk updates. This leads to an implementation that supports each operation in $\mathcal{O}(\log^2 n)$ amortized time. The details are given in Appendix A. Using the data structure we obtain the following theorem.

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 $\mathcal{O}(n \log^2 n)$ time and using $\mathcal{O}(n)$ storage.*

Proof. We aim to speed up the classic dynamic-programming algorithm using the data structure described above. 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 $P_{i-1} = \{p_1, \dots, p_{i-1}\}$ 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 $A[2, 1] = |p_1 p_2|$.

If we have an implicit representation of row i, 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 (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 (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 $p_1, \dots, p_{i-1}$ that are already in the structure by $\Delta := |p_i p_{i+1}|$. 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 $|p_i p_{i+1}|$.
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 $\mathcal{O}(\log^2 n)$ amortized time each, it follows that an implicit representation of the final row can be computed in $\mathcal{O}(n \log^2 n)$ time. The minimum cost of a pyramidal tour is $\min_{1 \leq k < n} (A[n, k] + |p_k p_n|)$, which can be found by querying the representation of the final row with point p_n. □

Bottleneck pyramidal TSP. Using a similar global approach but different supporting data structures we can also solve the bottleneck version of the problem—here the goal is to minimize the length of the longest edge in the tour—in subquadratic time. For the decision version of the problem we need the following result.

Theorem 2. *We can maintain a collection $\mathcal{D}$ of n congruent disks in a data structure such that we can decide in $\mathcal{O}(\log n)$ time if a query point q lies in* $\text{Union}(\mathcal{D})$. *The data structure uses $\mathcal{O}(n)$ storage and a new disk can be inserted into $\mathcal{D}$ in $\mathcal{O}(\log n)$ amortized time.*

This result is obtained as follows; see Appendix B.1 for details. Assume the disks have radius $\sqrt{2}$ and consider the integer grid. Let $\mathcal{D}(C) \subseteq \mathcal{D}$ be the set of disks whose centers lie inside a grid cell C. To decide if $q \in \text{Union}(\mathcal{D})$ we need to test if $q \in \text{Union}(\mathcal{D}(C))$ for $O(1)$ grid cells C that are sufficiently close to q. Now consider a cell C with $\mathcal{D}(C) \neq \emptyset$. Obviously C itself is completely covered by $\text{Union}(\mathcal{D}(C))$. Let $\ell_{\text{top}}(C)$ be the line containing the top edge

[2]We could also insert p_i with weight $A[i + 1, i] - \Delta$. This way we would not have to subtract Δ from the weights of $p_1, \dots, p_{i-1}$ 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.

of C. Then the part of Union($\mathcal{D}(C)$) above $\ell_{\text{top}}(C)$—the other parts are handled similarly—is x-monotone. Moreover, we can show that each disk $D_i \in \mathcal{D}(C)$ contributes at most one arc to the boundary of Union($\mathcal{D}(C)$) above $\ell_{\text{top}}(C)$, and the left-to-right order of the contributed arcs is consistent with the left-to-right order of the corresponding disk centers. Using this fact, we can do point locations and insertions in $O(\log n)$ time.

Combining the global technique of the previous section with Theorem 2 we obtain the following theorem.

Theorem 3. *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 $\mathcal{O}(n \log n)$ time and using $\mathcal{O}(n)$ storage if P admits a pyramidal tour whose longest edge has length at most B. This problem requires $\Omega(n \log n)$ time in the algebraic computation tree model of computation.*

The algorithm for the decision version does not easily extend to solve the minimization version of the problem. We therefore design a specialized data structure—a tree storing unions of disks and (regular) Voronoi diagrams—that allows us to obtain the following result. (See Appendix B.3.)

Theorem 4. *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 $\mathcal{O}(n \log n)$ storage.*

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 $d\colon E(G) \to \mathbb{N}$, 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 $\mathcal{O}(n^{3-\varepsilon} \operatorname{polylog}(M))$ for some constant $\varepsilon > 0$. Vassilevska-Williams and Williams [37] 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 subcubic algorithm for 3-OPT is unlikely. Their framework uses a notion of subcubic reducibility based on Turing reducibility [37, §IV] that solves one instance of problem A by repeatedly solving inputs of problem B. For our applications, simple reductions suffice that transform one input of problem A into one input of problem B of roughly the same size, in $\mathcal{O}(n^2)$ time.[3] Such reductions preserve the existence of truly subcubic algorithms, so we take

[3] We 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.

this simpler viewpoint. The following problem is the starting point for our reductions.

NEGATIVE EDGE-WEIGHTED TRIANGLE **Input:** An undirected, complete graph G and a weight function $w\colon E(G) \to \mathbb{Z}$. **Question:** Does G contain a triangle whose total edge-weight is negative?

Vassilevska-Williams and Williams [37, Thm. 1.1] proved that NEGATIVE EDGE-WEIGHTED TRIANGLE has a truly subcubic algorithm if and only if the ALL-PAIRS SHORTEST PATHS problem on digraphs with non-negative integral edge weights has a truly subcubic algorithm.

Lemma 3.1. NEGATIVE EDGE-WEIGHTED TRIANGLE *can be reduced to* 3-OPT DETECTION *in time $\mathcal{O}(n^2)$ while increasing the size of the graph and the largest weight by a constant factor.*

Proof. Consider an instance (G, w) of NEGATIVE EDGE-WEIGHTED TRIANGLE, and let $v_1, \ldots, v_n$ be an enumeration of the vertices of G. Let M be the largest absolute value of an edge weight. We introduce an instance of 3-OPT DETECTION that consists of $2n$ vertices $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, where the starting tour T uses the ordering $a_1, b_1, a_2, b_2, \ldots, a_n, b_n$. The (symmetric) distances $d(\cdot, \cdot)$ between these vertices are defined as follows:

- $d(a_i, b_i) = 0$ for $1 \le i \le n$;
- $d(b_n, a_1) = -3M$, and $d(b_i, a_{i+1}) = -3M$ for $1 \le i \le n-1$;
- $d(a_i, b_j) = w(\{v_i, v_j\})$ for $1 \le i < j \le n$;
- $d(b_i, a_j) = w(\{v_i, v_j\})$ for $1 \le i < j-1 \le n-1$;
- $d(a_i, a_j) = d(b_i, b_j) = 3M$ for $1 \le i \ne j \le n$.

(For convenience, we allow distances to be negative in this construction. One easily moves to non-negative distances by adding the constant $4M$ to all distances.)

Claim 3.1. *The constructed instance of* 3-OPT DETECTION *allows an improving* 3-OPT *move, if and only if the graph G contains a triangle of negative edge-weight.*

Proof. ($\Leftarrow$) Assume that the vertices v_i, v_j, v_k span a triangle of negative edge-weight in G for $i < j < k$. We remove the three edges $\{a_i, b_i\}$, $\{a_j, b_j\}$, and $\{a_k, b_k\}$ 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.

($\Rightarrow$) 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 $\{a_x, a_y\}$ 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 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 $\{a_i, b_k\}$, $\{a_k, b_j\}$, $\{a_j, b_i\}$ (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 $\mathcal{O}(n^2)$ time. □

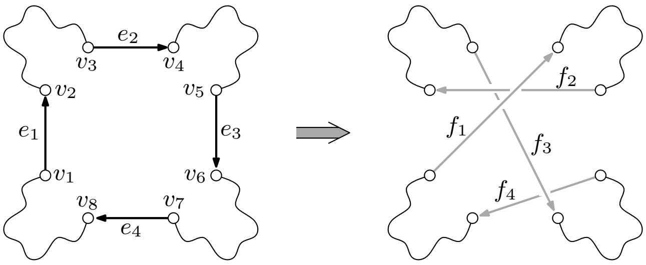

Figure 1: A 4-change with signature 4,5,7,1,2,8,3,6. Edges e_1 and e_4 are non-interfering. As we work on symmetric TSP, the graph and distance function are undirected; the arc directions merely indicate the traversal direction with respect to an arbitrary orientation of the tour.

There is an analogous reduction in the other direction, which is given as Lemma C.1 in Appendix C. Together, these lemmata show the equivalence of finding negative-weight triangles and detecting improving 3-OPT moves. From our reductions and the results of Vassilevska-Williams and Williams [37, Thm. 1.1], we obtain the following theorem.

Theorem 5. *There 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 $k \geq 4$. To this end, we first analyze the structure of k-OPT moves. Consider a k-move for a given tour $T \subseteq E(G)$, and let $e_1, \ldots, e_k$ be the removed edges with $e_i = \{v_{2i-1}, v_{2i}\}$. We assume throughout that these vertices (and edges) are indexed in such a way that T traverses the vertices v_i in order of increasing index. We assume furthermore that the vertices $v_1, \ldots, v_{2n}$ are pairwise distinct. By a simple reduction that subdivides the edges on the tour (Appendix D), it is sufficient to find an algorithm that detects an improving k-move in which the removed edges do not share any endpoints. (The arguments presented here also go through without this assumption, but the notation becomes more complicated in the equality case.) The k new edges that are inserted into T are denoted $f_1, \ldots, f_k$. The *signature* of this k-move is a permutation π of $\{1, \ldots, 2k\}$, such that vertex v_j and vertex $v_{\pi(j)}$ form the endpoints of one of the edges $f_1, \ldots, f_k$; see Fig. 1. Note that the removed edges $e_1, \ldots, e_k$ together with the signature π fully determine the k-move (and in particular determine the inserted edges $f_1, \ldots, f_k$).

Note furthermore that not every permutation π yields a feasible signature that corresponds to some k-move: First, in a feasible signature $\pi(i) = j$ always implies $\pi(j) = i$, and we will always have $\pi(i) \neq i$. Secondly, in a feasible signature the edge set that results from T by removing $e_1, \ldots, e_k$ and by inserting $f_1, \ldots, f_k$ must form a single Hamiltonian cycle—it must never form a collection of two or more cycles. It is easy to check whether a given permutation π constitutes a feasible signature, and to enumerate all feasible signatures.

We say that two of the removed edges e_i and e_j *interfere* with each other in a k-move, if there exists an inserted edge f that connects one of the endpoints of e_i to an endpoint of e_j. The following lemma states that in any k-move, there is a set of $\lceil k/3 \rceil$ pairwise non-interfering edges. This is essentially due to the fact that every k-vertex 2-regular graph (collection of cycles) contains an independent set of size at least $\lceil k/3 \rceil$; we prove it here in the k-OPT terminology.

Lemma 3.2. *For any signature π, we can find a subset $E_\pi \subseteq \{e_1, \ldots, e_k\}$ of at least $\lceil k/3 \rceil$ removed edges that are pairwise non-interfering.*

Proof. The $2k$ edges $e_1, \dots, e_k$ and $f_1, \dots, f_k$ induce a set of cycles on the vertices $v_1, \dots, v_{2k}$. If such a cycle contains an even number of removed edges, say 2ℓ, we put every other removed edge along this cycle into E_π; this yields ℓ out of 2ℓ edges for E_π. If the cycle contains only a single removed edge, we put this single edge into E_π; this yields one out of one edge for E_π. If the cycle contains an odd number of removed edges, say $2\ell+1 \geq 3$, we ignore the first removed edge and then put every other removed edge along the cycle into E_π; this yields ℓ out of $2\ell+1$ edges for E_π. The weakest contribution to E_π comes from cycles with three removed edges, which yield only one out of three edges for E_π. The claimed bound $\lceil k/3 \rceil$ follows. ☐

Theorem 6. *For every fixed $k \geq 3$, the k-*OPT *Optimization problem on an n-vertex graph can be solved in $\mathcal{O}(n^{\lfloor 2k/3 \rfloor + 1})$ time.*

Proof. For computing the best k-move for tour T, it is sufficient to compute for every feasible signature π—for fixed k there are only $\mathcal{O}(1)$ such signatures—the best k-move for tour T with that particular signature. This is done as follows. We first determine a set E_π of pairwise non-interfering edges according to the above lemma. Then we enumerate and handle all possible cases for the locations of the $\lfloor 2k/3 \rfloor$ removed edges not in E_π along T. This yields $\mathcal{O}(n^{\lfloor 2k/3 \rfloor})$ cases to handle, and every such case will be handled in $\mathcal{O}(n)$ time; note that this yields the claimed complexity. In handling a case, the positions of the removed edges not in E_π are frozen, while the edges in E_π have to be embedded into T. The cost of a k-move with signature π decomposes into two parts:

- The first part consists of the total weight of all frozen edges (which is subtracted) and the total weight of inserted edges between frozen edges (which is added).
- The second part consists of the individual contributions of the edges in E_π. For an edge $e \in E_\pi$ and an edge $e' \in T$, the cost of embedding e into e' equals the weight of the two inserted edges adjacent to e minus the weight of e'. As the edges in E_π are pairwise non-interfering, their individual cost contributions do not interact with each other.

As the cost of the first part is fixed in every considered case, our goal is to minimize the total cost of the second part. The frozen edges subdivide the tour T into a number of tour pieces, and we have to find the cheapest way of embedding the corresponding edges from E_π into such a tour piece. The following paragraph sketches a straightforward dynamic program for finding the optimal embedding for each tour piece in time proportional to the length of the piece. As the length of all tour pieces combined is $\mathcal{O}(n)$, every case is indeed handled in time $\mathcal{O}(n)$.

We are essentially dealing with the following optimization problem. There are r locations $L_1, \dots, L_r$ (the edges along tour T between two consecutive frozen edges) and s objects $O_1, \dots, O_s$ (the edges in E_π that should be embedded between the two considered frozen edges). The objects are to be embedded into the locations, so that the location of object O_i always precedes the location of object O_{i+1}. The cost of embedding object O_i into location L_j is denoted $c(i,j)$. For $1 \leq x \leq s$ and $1 \leq y \leq r$, let $V(x,y)$ denote the smallest possible cost incurred by embedding the first x objects $O_1, \dots, O_x$ into the first y locations $L_1, \dots, L_y$. As $V(x,y)$ equals the minimum of $V(x,y-1)$ and $V(x-1,y-1) + c(x,y)$, all these values $V(x,y)$ can easily be computed in $\mathcal{O}(rs)$ time. In our situation, r is the length of the considered tour piece and $s \leq k$ is a constant that does not depend on the input; hence the complexity is indeed proportional to the length of the considered tour piece. ☐

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.

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 7. *After $\mathcal{O}(n^2)$ preprocessing and using $\mathcal{O}(n^2)$ storage we can repeatedly solve the* 2-OPT OPTIMIZATION *problem in $\mathcal{O}(n \log n)$ time per iteration.*

The speedup claimed in the theorem relies on a tour representation that supports efficient 2-moves. To apply a 2-move that removes two edges e and e' and replaces them by the appropriate diagonal connections, one effectively has to reverse the part of the tour between e and e', or the part between e' and e. It can therefore take $\Omega(n)$ time to apply a 2-move to a tour represented as a sequence of vertices in an array. Chrobak *et al.* [16] give a speedup by storing the cities on the tour in an ordered balanced binary search tree. Each node in the tree stores a bit indicating whether the tour order is given by an in-order traversal of the subtree rooted there, or by the *reverse* of the in-order traversal. This allows a 2-move to be applied in $\mathcal{O}(\log n)$ time by manipulating reversal bits.

Our approach for repeated 2-OPT OPTIMIZATION is based on a similar data structure that represents tours in balanced search trees. However, instead of having only one tree that stores the current tour, we have n trees; one for each edge $e_1, \ldots, e_n$ in the current tour. A query in the tree $\mathcal{T}(e_i)$ corresponding to edge e_i can be used to determine which edge e_j yields the most profitable 2-move together with e_i. After initializing these n trees, which takes $\mathcal{O}(n^2)$ time, an iteration of 2-OPT OPTIMIZATION can be performed as follows. For each e_i on the current tour, we query in tree $\mathcal{T}(e_i)$ to find the best 2-move that removes e_i and some unknown edge e_j in $\mathcal{O}(\log n)$ time. In this way we find the best overall 2-move which removes, say, edges e_i and e_j. We can update all trees $\mathcal{T}(e_\ell)$ for $\ell \neq i, j$ by deleting e_i and e_j, and inserting the appropriate replacement edges. Using the reversal bits this can be done in $O(\log n)$ time. Trees $\mathcal{T}(e_i)$ and $\mathcal{T}(e_j)$ are destroyed; we build two new trees from scratch for the two new edges $e_{i'}$ and $e_{j'}$ that enter the tour. This gives $\mathcal{O}(n \log n)$ time per iteration.

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.

The planar case. For points in the plane (and under the Euclidean metric) we can speed up 2-OPT computations by using suitable geometric data structures for semi-algebraic range searching, as shown in Appendix E.3. (Note that we do not consider the repeated version of the problem, but the single-shot version.) A similar approach can be used to speed up 3-OPT in the Euclidean setting in the plane. This leads to the following theorem.

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

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 $\mathcal{O}(n^k)$ 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 [31] and Guo et al. [24]. 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. Is $\Theta(n^{\lfloor \frac{2k}{3} \rfloor + 1})$ the optimal running time for k-opt Detection? When all weights lie in the range $[-M, \ldots, M]$, one can detect a negative triangle in an edge-weighted graph in time $\mathcal{O}(M \cdot n^{\omega})$ using fast matrix multiplication [6, 35, 40]. By our reduction, this gives an algorithm for 3-opt Detection with weights $[-M, \ldots, M]$ in time $\mathcal{O}(M \cdot n^{\omega})$. 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 $\mathcal{O}(n^{\lfloor 2k/3 \rfloor + 1})$ 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

[1] A. Abboud, A. Backurs, and V. V. Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proc. 56th FOCS*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14. 2

[2] A. Abboud, F. Grandoni, and V. Vassilevska-Williams. Subcubic equivalences between graph centrality problems, APSP and diameter. In *Proc. 26th SODA*, pages 1681–1697, 2015. doi:10.1137/1.9781611973730.112. 3

[3] A. Abboud, V. Vassilevska-Williams, and H. Yu. Matching triangles and basing hardness on an extremely popular conjecture. In *Proc. 47th STOC*, pages 41–50, 2015. doi:10.1145/2746539.2746594. 3

[4] P. Agarwal and J. Matousek. On range searching with semialgebraic sets. *Discr. Comput. Geom*, 11:393–418, 1994. 28

[5] P. Agarwal, J. Matousek, and M. Sharir. On range searching with semialgebraic sets, II. *SIAM J. Comput.*, 42:2039–2062, 2013. doi:10.1137/120890855. 28, 29

[6] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997. doi:10.1006/jcss.1997.1388. 11

[7] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proc. 47th STOC*, pages 51–58, 2015. doi:10.1145/2746539.2746612. 2

[8] M. F. Baki and S. N. Kabadi. Pyramidal traveling salesman problem. *Computers & OR*, 26(4):353–369, 1999. doi:10.1016/S0305-0548(98)00067-7. 2

[9] J. L. Bentley. Experiments on traveling salesman heuristics. In *Proc. 1st SODA*, pages 91–99, 1990. 1

[10] J. Bently and J. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms*, 1:301–358, 1980. doi:10.1016/0196-6774(80)90015-2. 4

[11] K. Bringmann. Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. In *Proc. 55th FOCS*, pages 661–670, 2014. doi:10.1109/FOCS.2014.76. 2

[12] K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In V. Guruswami, editor, *Proc. 56th FOCS*, pages 79–97. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.15. 2

[13] R. E. Burkard, V. G. Deineko, R. van Dal, J. A. A. van der Veen, and G. J. Woeginger. Well-solvable special cases of the traveling salesman problem: A survey. *SIAM Review*, 40(3):496–546, 1998. doi:10.1137/S0036144596297514. 2

[14] J. Carlier and P. Villon. A new heuristic for the travelling salesman problem. *RAIRO – Operations Research*, 24:245–253, 1990. 2

[15] B. Chandra, H. J. Karloff, and C. A. Tovey. New results on the old k-OPT algorithm for the traveling salesman problem. *SIAM J. Comput.*, 28(6):1998–2029, 1999. doi:10.1137/S0097539793251244. 1

[16] M. Chrobak, T. Szymacha, and A. Krawczyk. A data structure useful for finding hamiltonian cycles. *Theoretical Computer Science*, 71(3):419–424, 1990. doi:10.1016/0304-3975(90)90053-K. 10, 25, 26

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. 1, 2

[18] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6:791–812, 1958. doi:10.1287/opre.6.6.791. 1

[19] M. Englert, H. Röglin, and B. Vöcking. Worst case and probabilistic analysis of the 2-opt algorithm for the TSP. *Algorithmica*, 68(1):190–264, 2014. doi:10.1007/s00453-013-9801-4. 1

[20] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987. doi:10.1007/BF01840357. 4, 14, 20, 21

[21] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. *J. Algorithms*, 18(3):432–479, 1995. doi:10.1006/jagm.1995.1018. 1, 25

[22] P. Gilmore, E. Lawler, and D. Shmoys. Well-solved special cases. In E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys, editors, *The Traveling Salesman Problem*, pages 87–143. Wiley, New York, 1985. 2

[23] F. Glover. Finding a best traveling salesman 4-Opt move in the same time as a best 2-Opt move. *J. Heuristics*, 2(2):169–179, 1996. doi:10.1007/BF00247211. 1

[24] J. Guo, S. Hartung, R. Niedermeier, and O. Suchý. The parameterized complexity of local search for TSP, more refined. *Algorithmica*, 67(1):89–110, 2013. doi:10.1007/s00453-012-9685-8. 2, 11

[25] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774. 2

[26] D. Johnson and L. McGeoch. The traveling salesman problem: A case study in local optimization. In E. Aarts and J. Lenstra, editors, *Local search in combinatorial optimization*, pages 215–310. Wiley, Chichester, 1997. 1

[27] D. S. Johnson and L. A. McGeoch. Experimental analysis of heuristics for the STSP. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 369–443. Kluwer Academic Publishers, Dordrecht, 2002. 1

[28] V. Koltun. Almost tight upper bounds for vertical decompositions in four dimensions. *J. ACM*, 51:699–730, 2004. doi:10.1145/1017460.1017461. 28, 29

[29] M. Künnemann and B. Manthey. Towards understanding the smoothed approximation ratio of the 2-opt heuristic. In *Proc. 42nd ICALP*, pages 859–871, 2015. doi:10.1007/978-3-662-47672-7_70. 1

[30] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965. doi:10.1002/j.1538-7305.1965.tb04146.x. 1

[31] D. Marx. Searching the k-change neighborhood for TSP is W[1]-hard. *Oper. Res. Lett.*, 36(1):31–36, 2008. doi:10.1016/j.orl.2007.02.008. 2, 11

[32] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discr. Comput. Geom.*, 10:157–182, 1993. doi:10.1007/BF02573972. 28

[33] I. Mavroidis, I. Papaefstathiou, and D. N. Pnevmatikatos. A fast FPGA-based 2-opt solver for small-scale euclidean traveling salesman problem. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–22, 2007. doi:10.1109/FCCM.2007.40. 1

[34] M. A. O'Neil and M. Burtscher. Rethinking the parallelization of random-restart hill climbing: a case study in optimizing a 2-opt TSP solver for GPU execution. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 99–108, 2015. doi:10.1145/2716282.2716287. 1

[35] L. Roditty and V. Vassilevska-Williams. Minimum weight cycles and triangles: Equivalences and algorithms. In *Proc. 52nd FOCS*, pages 180–189, 2011. doi:10.1109/FOCS.2011.27. 11

[36] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry (2nd ed.)*. CRC Press, 2004. 4, 14, 20, 21

[37] V. Vassilevska-Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proc. 51th FOCS*, pages 645–654, 2010. doi:10.1109/FOCS.2010.67. 3, 6, 7, 8

[38] D. Willard and G. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32:597–617, 1985. doi:10.1145/3828.3839. 30

[39] A. C.-C. Yao. Lower bounds for algebraic computation trees with integer inputs. *SIAM J. Comput.*, 20(4):655–668, 1991. doi:10.1137/0220041. 2, 18

[40] G. Yuval. An algorithm for finding all shortest paths using $n^{2.81}$ infinite-precision multiplications. *Inf. Process. Lett.*, 4(6):155–156, 1976. doi:10.1016/0020-0190(76)90085-5. 11

A Data structure for faster pyramidal TSP

In this section we describe the data structure used in 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 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 consider additively weighted distances. [4] 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 $\mathcal{O}(n)$ and it can be computed in $\mathcal{O}(n \log n)$ time [20]. Moreover, point location in a planar subdivision of complexity $\mathcal{O}(n)$ can be done in $\mathcal{O}(\log n)$ time with a data structure that uses $\mathcal{O}(n)$ storage and $\mathcal{O}(n \log n)$ preprocessing [36]. Thus nearest-neighbor queries in P under the additively weighted distance function can be done in $\mathcal{O}(\log n)$ after $\mathcal{O}(n \log n)$ preprocessing.

We first briefly review the logarithmic method. 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 $n = \sum_{t=0}^{\lfloor \log n \rfloor} a_t 2^t$. The logarithmic method maintains, for each t with $a_t = 1$, a static data structure $\mathcal{DS}^{(t)}$ on a subset $S^{(t)} \subseteq$ 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 $\mathcal{O}(\log n)$ 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 $\mathcal{O}(\sum_{t=0}^{\lfloor \log n \rfloor} B(2^t)/2^t)$, 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 $\mathcal{O}(\log n)$ candidates found for the subsets $P^{(t)}$. Thus our structure has $\mathcal{O}(\log^2 n)$ query time. Since a substructure $\mathcal{DS}^{(t)}$ can be built in $\mathcal{O}(|P^{(t)}| \log |P^{(t)}|)$ time, the amortized time for an insertion is $\mathcal{O}(\log^2 n)$.

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 Δ. With the logarithmic method this is quite easy. We simply store a *correction term* Δ_t for each $\mathcal{DS}^{(t)}$, which indicates that the weight of each point in $P^{(t)}$ should be increased by Δ_t. A bulk update with value Δ can then be performed in $\mathcal{O}(\log n)$ time by adding Δ to each of the correction terms Δ_t. Note that we can still answer queries correctly. Indeed, AWVD($P^{(t)}$) does not change when we add the same value Δ 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 $\mathcal{O}(\log^2 n)$ time. Insertions can still be done in $\mathcal{O}(\log^2 n)$ 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

[4] Sometimes additive 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 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.

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.

B Planar bottleneck pyramidal TSP

Below we consider the bottleneck version of the pyramidal TSP problem. The goal is to find a pyramidal tour for an ordered set $P := \{p_1, \dots, p_n\}$ of points in the plane such that the length of the bottleneck edge (that is, the longest edge) is minimized. We start by giving an $\mathcal{O}(n \log n)$ algorithm for 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. Next we show that this is optimal by presenting an $\Omega(n \log n)$ lower bound in the algebraic computation-tree model. Finally, we show how to solve the optimization version of the problem.

B.1 An algorithm for the decision problem

The decision problem can be solved by dynamic programming, using a 2-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

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

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 $|p_i p_{i+1}| \leq B$, and they are all 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 $p_2, \dots, p_n$ 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 we add p_i to $\mathcal{L}$, if only the first condition is satisfied 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 $\mathcal{O}(n)$. Checking the first condition obviously takes $\mathcal{O}(1)$ 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:

- *query* the structure with a point p_{i+1} to decide if it stores a point p_k with $|p_k p_{i+1}| \leq B$;
- *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 $\mathcal{O}(\log^2 n)$ query time and $\mathcal{O}(\log^2 n)$ 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 Union($\mathcal{D}$) denotes the union of the disks in $\mathcal{D}$. Theorem 2 below states that point-location queries in the union of a set of congruent disks can be done in $\mathcal{O}(\log n)$ time and with $\mathcal{O}(\log n)$ amortized update time, leading to the following result.

Theorem. *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 $\mathcal{O}(n \log n)$ time and using $\mathcal{O}(n)$ storage if P admits a pyramidal tour whose longest edge has length at most B.*

A semi-dynamic point-location data structure for the union of congruent disks. Let $\mathcal{D}$ be a set of congruent disks in the plane. We wish to maintain a data structure on $\mathcal{D}$ that allows us to decide if a query point q lies inside Union($\mathcal{D}$). The data structure should also allow insertions into the set $\mathcal{D}$. With a slight abuse of notation, we will use n to denote the number of disks in the (current) set $\mathcal{D}$. 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 $\mathcal{D}$ 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 $D_i \in \mathcal{D}$, 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 Union($\mathcal{D}$) maintains the active strips in a balanced search tree on their x-order, and for each active strip Σ it maintains the active cells within Σ 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 $\mathcal{D}(C) \subseteq \mathcal{D}$ 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 Union($\mathcal{D}(C)$) 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_i denote the center of the disk $D_i \in \mathcal{D}(C)$. Because the centers p_i 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 $D_i \in \mathcal{D}(C)$. The key to efficiently maintaining $U_{\text{top}}(C)$ is the following lemma.

Lemma B.1. *Each disk $D_i \in \mathcal{D}(C)$ 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 the left of p_j.*

Proof. Define γ_i to be the part of D_i's boundary above the line $\ell_{\text{top}}(C)$. Any other disk $D_j \in \mathcal{D}(C)$ that covers a part of γ_i must contain an endpoint of γ_i. Indeed, if γ_j would intersect γ_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 γ_j would be larger than the curvature of γ_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 $\alpha_i \subseteq \gamma_i$ contributed by D_i and an arc $\alpha_j \subseteq \gamma_j$ contributed by D_j. Assume without loss of generality that p_i lies to the left of p_j. Furthermore, assume p_i lies

[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.

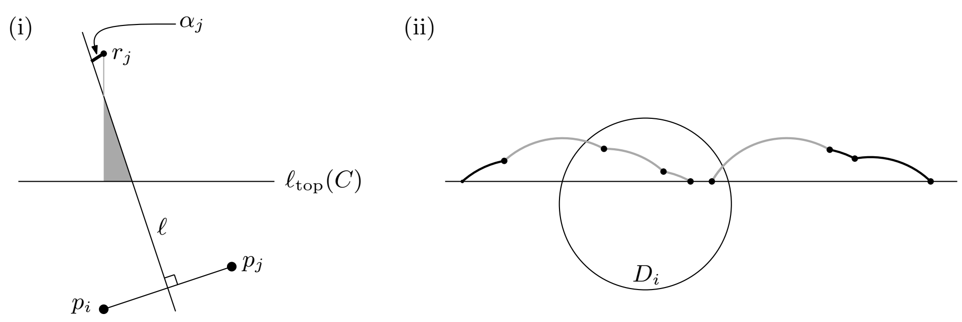

Figure 2: (i) Illustration for the proof of Lemma B.1. (ii) The addition of D_i causes several arcs to disappear from $\partial U_{top}(C)$ and two arcs to be shortened. All these arcs (indicated in gray) are consecutive in the left-to-right order.

below p_j, as in Fig. 2(i); a similar argument applies when p_i lies above p_j. Now suppose for a contradiction that α_i lies to the right of α_j. Let ℓ be the perpendicular bisector of the segment $p_i p_j$. Because the disks have equal radius, α_i must lie to the left of ℓ and α_j must lie to the right of ℓ. Hence, if α_i to lies to the right of α_j, then it must lie in the triangular region bounded by $\ell_{top}(C)$, and ℓ, and the vertical line through r_j. (In Fig. 2(i) this triangle is shown shaded.) But this region is completely contained in D_j since any point in it is closer to p_j than r_j is. Hence, we have a contradiction with the fact that α_i is an arc of $\partial U_{top}(C)$. □

Lemma B.1 gives us an easy way to store and maintain $U_{top}(C)$. We simply store the arcs comprising $\partial U_{top}(C)$ in x-order in a balanced search tree $\mathcal{T}_{top}(C)$. This takes $\mathcal{O}(n)$ storage and allows us to decide in $\mathcal{O}(\log n)$ time for a query point q if $q \in U_{top}(C)$.

Now suppose we want to insert a new disk D_i into $\mathcal{D}(C)$. As observed, ∂D_i contributes at most one new arc to $\partial U_{top}(C)$. The addition of this new arc means we have to remove some existing arcs. More precisely, if ∂D_i contributes a new arc, then we have to shorten two existing arcs and possibly remove one or more other arcs; see Fig. 2(ii). Once we know which existing arcs are affected, the update can be done in $\mathcal{O}((k+1)\log n)$ time, where k is the number of disappearing arcs. Since each arc is removed at most once, this given an amortized insertion time of $\mathcal{O}(\log n)$. 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 $\mathcal{T}_{top}(C)$ 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 α_j (contributed by some disk D_j) stored at the root of $\mathcal{T}_{top}(C)$. If D_i covers (a part of) α_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 $\partial U_{top}(C)$. The correctness of this procedure is guaranteed by Lemma B.1. We can conclude the following lemma.

Lemma B.2. *We can maintain $U_{top}(C)$ in a data structure using $\mathcal{O}(n)$ storage such that we can decide in $\mathcal{O}(\log n)$ time for a query point q if $q \in U_{top}(C)$. The data structure can be maintained under insertions in $\mathcal{O}(\log n)$ amortized time.*

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

- A balanced search tree $\mathcal{T}$ storing the active strips sorted on their x-order, and for each active strip Σ a balanced search tree $\mathcal{T}_\Sigma$ 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 B.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, we know that $q \in \text{Union}(\mathcal{D})$. 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 $\mathcal{D}(C)$ can contain a disk D_i such that $q \in D_i$. Note that there are only $\mathcal{O}(1)$ such cells and that they can be found in $\mathcal{O}(\log n)$ time using the tree $\mathcal{T}$ and the trees $\mathcal{T}_\Sigma$. For each relevant cell C, we then query the appropriate partial union; for example, if q lies above $\ell_{\text{top}}(C)$ we query $\mathcal{T}_{\text{top}}(C)$. Now q lies in $\text{Union}(\mathcal{D})$ if and only if q lies in at least of one 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, 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 B.2 the whole procedure takes $\mathcal{O}(\log n)$ amortized time.

Theorem 2. *We can maintain a collection $\mathcal{D}$ of n congruent disks in a data structure such that we can decide in $\mathcal{O}(\log n)$ time if a query point q lies in* $\text{Union}(\mathcal{D})$*. The data structure uses $\mathcal{O}(n)$ storage and a new disk can be inserted into $\mathcal{D}$ in $\mathcal{O}(\log n)$ amortized time.*

B.2 A lower bound for the decision problem

Below we show an $\Omega(n \log n)$ time lower bound for the decision version of the bottleneck pyramidal TSP in the Euclidean plane, in the algebraic computation-tree model. 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 9. *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 $U = \{u_1, \dots u_n\}$ and $V = \{v_1, \dots v_n\}$, for which the lower bound is known [39]. 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 $P = \{p_1, \dots, p_{2n+2}\}$ 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 Fig 3).

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 10. *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.*

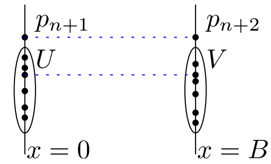

Figure 3: Lower bound construction: A TSP tour needs to move between the lines $x = 0$ and $x = B$ at least twice, which is possible with a bottleneck of B exactly if U and V have at least one element in common.

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 $U \cap V \neq \emptyset$, 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, 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

$$\sqrt{1 + (B - 2(n+1)\Delta)^2} = \sqrt{1 + (B - \frac{1}{2B})^2} > \sqrt{1 + B^2 - 1} = B,$$

a contradiction. □

B.3 An algorithm for the optimization problem

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

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

where $A[2, 1] = |p_1 p_2|$. Our strategy to speed up the dynamic-programming algorithm is similar to the strategy for the non-bottleneck version in Section 2: 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 $\min_{p_k} \max(w_k, |p_k q|)$, 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.

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

Theorem 4. *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 $\mathcal{O}(n \log n)$ storage.*

The data structure. 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 $\mathcal{T}$ whose leaves store the weights from W, together with the corresponding points. For a node ν in $\mathcal{T}$, let $P(\nu)$ denote the set of points stored in the subtree rooted at ν. We maintain the following information at ν.
  - 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 $\{D(p_j, w_j) : p_j \in P(\nu)\}$, preprocessed for point location. Here the weights w_j refer to the weights at the time the data structure was constructed; after a bulk update the union $U(\nu)$ is not changed. We denote this union by $U(\nu)$.
  - The Voronoi diagram $\mathrm{VD}(P(\nu))$ of the point set $P(\nu)$ (using the normal Euclidean distances), preprocessed for point location.
  - The maximum weight stored in the subtree rooted at ν.
- 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 [20, 36], the overall amount of storage of our data structure is $\mathcal{O}(n \log n)$. 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 $B(q) := \min_{p_j \in P} \max(w_j, |p_j q|)$.

Lemma B.3. *Let $B_1 := \min\{w_j : p_j \in P$ and $|p_j q| \leq w_j\}$ and $B_2 := \min\{|p_j q| : p_j \in P$ and $w_j < B_1\}$. Then $B(q) = \min(B_1, B_2)$.*

Proof. Define $P_1 := \{p_j \in P : w_j \geq |p_j q|\}$ and $P_2 := \{p_j \in P : w_j < |p_j q|\}$. Note that $B_1 = \min_{p_j \in P_1} w_j$ and define $B_2' := \min_{p_j \in P_2} |p_j q|$. 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 $|p_j q|$ 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. □

We now describe how to perform the three operations on $\mathcal{T}$.

Queries. To answer a query we first compute the nearest neighbor, p_k, of q in P. This can be done in $\mathcal{O}(\log n)$ time by locating q in $\mathrm{VD}(root(\mathcal{T}))$, since $P(root(\mathcal{T})) = P$. If $|p_k q| \leq B_{\max}$ 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 $|p_j q| > B_{\max}$ for all $p_j \in P$. This implies that for any given node ν we can decide whether there is a point $p_j \in P(\nu)$ with $|p_j q| \leq w_j$ by checking if $q \in U(\nu)$—the bulk updates we have performed since constructing $U(\nu)$ do not affect the outcome.

We can compute B_1 as follows. We start by checking if $q \in U(root(\mathcal{T}))$. If this is not the case then $\{p_j \in P : |p_j q| \leq w_j\} = \emptyset$ and so we set $B_1 := \infty$. Otherwise we walk down the tree, as follows. Suppose we are at a non-leaf node ν. Let μ be the left child of ν. If $q \in U(\mu)$ then we descend to the left child of ν (that is, we set $\nu := \mu$) and otherwise we proceed to the right child. Since the points of P are stored in the leaves of $\mathcal{T}$ 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 $|p_j q| \leq w_j$. 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 $\mathcal{T}$ was constructed to find the points p_j satisfying $w_j < B_1$. To compute B_2 we now identify a collection of $\mathcal{O}(\log n)$ nodes ν such that the sets $P(\nu)$ contain exactly the points p_j with $w_j < B_1$. This can be done by searching with B_1 in $\mathcal{T}$. At each of these nodes we compute $\min_{p_j \in P(\nu)} |p_j q|$ by point location in $\mathrm{VD}(P(\nu))$, and we set B_2 to be the minimum of the $\mathcal{O}(\log n)$ values computed in this manner.

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

Bulk updates. A bulk update with value B is performed in $\mathcal{O}(1)$ 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 $\mathcal{O}(\log^3 n)$ and $\mathcal{O}(\log n)$, respectively. The amortized time for insertions is $\mathcal{O}(T_B(n) \log n)$, where $T_B(n)$ is the time needed to build a static structure on n points. This can be done bottom-up in $\mathcal{O}(n \log^2 n)$ time: At each node ν we can construct the point-location data structure on the union $U(\nu)$ in $\mathcal{O}(|P(\nu)| \log |P(\nu)|)$ time [36], and we can construct the Voronoi diagram in the same amount of time [20]. (Before we can construct the point-location data structure we first need to construct $U(\nu)$, but this can be done in $\mathcal{O}(|P(\nu)| \log |P(\nu)|)$ time by merging the unions from the two children of ν.) We conclude that the amortized time for insertions is $\mathcal{O}(\log^3 n)$.

C On truly subcubic algorithms for 3-OPT: Missing proof

Lemma C.1. 3-OPT DETECTION *can be reduced to* NEGATIVE EDGE-WEIGHTED TRIANGLE *in time $\mathcal{O}(n^2)$ 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 $\mathcal{O}(n^2)$ time we check whether there is an improving 2-move in G. If so, 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. Secondly, we test whether there is an improving 3-move where two of the removed edges share an endpoint. This can be done in $\mathcal{O}(n^2)$ 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 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 $\{c_0, a_1\}$, 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 4. 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 this graph 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 6 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 $\{a_{\ell(ab)}, b_{r(ab)}\}$, $\{a_{\ell(ac)}, c_{r(ac)}\}$, and $\{b_{\ell(bc)}, c_{r(bc)}\}$.

Figure 4: Left: the 6-vertex template graph with the a-gap, b-gap, and c-gap. Right: the thick edges give one possibility for completing the graph into a cycle, with characteristic $\ell(ab) = 0, r(ab) = 1, \ell(ac) = 1, r(ac) = 1, \ell(bc) = 0, r(bc) = 0$.

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 3 edges that completes the graph into a cycle without re-inserting a removed edge, 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:

- $w(\{x_i, y_j\}) = d(v_{i+\ell(ab)}, v_{j+r(ab)}) - d(v_i, v_{i+1})$ for $1 \leq i < j - 1 \leq n$;
- $w(\{x_i, z_k\}) = d(v_{i+\ell(ac)}, v_{j+r(ac)}) - d(v_k, v_{k+1})$ for $1 \leq i < k - 1 \leq n$;
- $w(\{y_j, z_k\}) = d(v_{j+\ell(bc)}, v_{k+r(bc)}) - d(v_j, v_{j+1})$ 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 C.1. *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 3 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_1. 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 $\{c_0, a_0\}$, and $\{a_1, b_0\}$, 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 6 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 $d(v_{i+\ell(ab)}, v_{j+r(ab)}) + d(v_{i+\ell(ac)}, v_{j+r(ac)}) + d(v_{j+\ell(bc)}, v_{k+r(bc)}) - d(v_i, v_{i+1}) - d(v_j, v_{j+1}) - d(v_k, v_{k+1})$. 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 $\mathcal{O}(n^2)$ time and blows up the graph size and largest weight by only a constant factor. □

D A fast k-OPT algorithm: missing proof

In this section we present an elementary reduction which shows that to find optimal an k-move, it suffices to find a k-move where the removed edges do not share any endpoints.

Lemma D.1. *For any $k \geq 3$, an instance (G, T, d) of k-OPT OPTIMIZATION can be reduced in time $\mathcal{O}(n^2)$ to an instance (G', T', d'), such that:*

1. *$|V(G')| = 2|V(G)|$,*
2. *If the distances under d lie in the range $[-M, \dots, +M]$, then the distances under d' lie in the range $[-2kM, \dots, +M]$.*
3. *Instance (G', T', d') has an optimal k-move in which the removed edges do not share any endpoints.*
4. *Given an optimal k-move in (G', T', d'), one can find an optimal k-move in (G, T, d) in time $\mathcal{O}(k)$.*

Proof. Consider an instance of k-OPT OPTIMIZATION, which is given by a complete graph G together with a tour T in G and a symmetric distance function d. The goal is to find a k-move that improves tour T the most. Number the vertices of G as $v_1, \dots, v_n$ in the order of T. Let M be the largest absolute value of an edge weight. Intuitively, the graph G' is obtained by subdividing all the edges on the current tour with a new vertex. One half of each subdivided edge will have very small weight (so that it is never beneficial to remove it from the tour), whereas the other half has weight equal to the weight of the original undivided edge. This will ensure that an optimal k-move in the resulting instance removes only disjoint edges from the tour.

Formally, the instance (G', T', d') is produced as follows. Graph G' consists of vertices $a_1, b_1, a_2, b_2, \dots, a_n, b_n$ and the initial tour T' visits the vertices in this order. The (symmetric) distances $d'(\cdot, \cdot)$ between these vertices are defined as follows:

- $d'(a_i, a_j) = d'(b_i, b_j) = d(v_i, v_j)$ for $1 \leq i, j \leq n$;
- $d'(a_i, b_i) = -2kM$ for $1 \leq i \leq n$;
- $d'(a_i, b_j) = d(v_i, v_j)$ for $1 \leq i, j \leq n$ with $i \neq j$.

The first two properties of the reduction follow immediately from these definitions. Let us consider how a k-move that improves T by removing the edges E and adding the edges F, translates into a k-move improving T'. We may assume without loss of generality that $E \cap F = \emptyset$, since $E \Delta F$ is also a valid k-move with the same effect. Removing the edges E from T splits the tour T into $|E|$ paths. For each edge $\{v_i, v_{i+1}\} \in E$ (modulo n), we remove the edge $\{b_i, a_{i+1}\}$ from T' to split it into $|E|$ paths; note that $d'(b_i, a_{i+1}) = d(v_i, v_{i+1})$. Let E' denote the corresponding set of removed edges of T'. Every edge in F connects two endpoints of paths of $T - E$. If v_i is an endpoint of a path in $T - E$, then either a_i or b_i is an endpoint

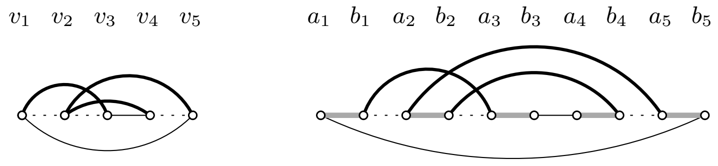

Figure 5: Illustration for the reduction of Lemma D.1. Left: illustration of an instance (G, T, d) with five vertices where $T = (v_1, v_2, v_3, v_4, v_5)$. The 3-move that removes edges $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_4, v_5\}\}$ is shown. Removed edges E are dotted, inserted edges F are thick. Right: the instance (G', T', d') resulting from the reduction by subdividing edges. The edges with very small weight are gray; they cannot be removed by an improving k-move. The 3-move for T has a natural analogue in T', where the removed edges share no endpoints.

of a path in $T' - E'$. (In the special case that both tour edges incident on v_i are in E, there is a path in $T' - E'$ consisting of only a_i and b_i and both vertices are endpoints.) For each inserted edge F between endpoints p, q of paths in $T - E$, insert into F' the edge between the corresponding endpoints of the paths in $T' - E'$, which has the same weight (see Figure 5). It follows that replacing E' by F' changes the weight of tour T' in the same way as replacing E by F does for tour T. Let $\textsc{opt}$ denote the optimal cost improvement achieved by a k-move for T, and let $\textsc{opt}'$ denote the optimal improvement for T'. Applying this transformation to an optimal k-move for T shows that $\textsc{opt}' \geq \textsc{opt}$ and yields a k-move for T' with profit $\textsc{opt}$ for which the removed edges share no endpoints. To prove the third property, it suffices to show that $\textsc{opt}' \leq \textsc{opt}$, implying that such a move is also optimal for T'. This will be implied by our proof of the fourth property, which we now present.

Consider a tour T_1' in G' obtained by applying an optimal k-move to T'. We claim that tour T_1' contains all edges $\{a_i, b_i\}$ for $1 \leq i \leq n$. To see this, observe that those are the only edges of weight $-2kM$, and the other edges have weight at least $-M$. If one of these edges $\{a_i, b_i\}$ disappears from the tour (increasing its weight by $2kM$), then at best the other $k-1$ removed edges of weight $\leq M$ decrease the weight by $(k-1)M$, causing a net weight increase of $(k+1)M$ due to removals. Inserting k distinct new edges into the tour decreases the weight by at most kM, since there are no new edges of weight $-2kM$ to introduce and the smallest weight under d is at least $-M$. Hence any k-move that removes an edge of weight $-2kM$ is not optimal since it increases the weight of the tour; the empty k-move that performs no changes is better. It follows that T_1' contains the n edges $\{a_i, b_i\}$ for $1 \leq i \leq n$. Since vertices a_i and b_i have the same distances to the rest of the vertices for all i, in this tour T_1' we can "contract" all edges $\{a_i, b_i\}$ for $1 \leq i \leq n$ to obtain a tour T_1 in G whose cost difference with T is the same as the difference between T_1' and T, and which can also be obtained by a k-move. It follows that the optimal weight improvement by applying a k-move to T' is bounded by the optimal weight improvement by applying a k-move to T, showing that $\textsc{opt}' \leq \textsc{opt}$ (and therefore $\textsc{opt} = \textsc{opt}'$) and proving the third property. The k-move used to obtain T_1 can easily be extracted from the k-move used to obtain T_1'. Any removed or inserted edge in the k-move producing T_1' connects two vertices with distinct indices i, j in the range $1 \ldots n$; the k-move to produce T_1 removes or inserts the corresponding edge $\{v_i, v_j\}$. This completes the proof of the fourth property. □

E Faster 2-OPT: Additional details

E.1 The repeated case

In this section we provide the proof of Theorem 7.

Theorem 7. *After $\mathcal{O}(n^2)$ preprocessing and using $\mathcal{O}(n^2)$ storage we can repeatedly solve the*

2-OPT OPTIMIZATION *problem in* $\mathcal{O}(n \log n)$ *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.* [16] that was also used by Fredman *et al.* [21]: we store the tour in a tree, and with each node ν we store a Boolean $\text{Rev}(\nu)$ indicating whether the subpath represented by the subtree $\mathcal{T}_\nu$ rooted at ν should be reversed. (Fredman *et al.* [21] 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 $\mathcal{T}(e)$ 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 $\mathcal{T}(e)$ 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 $\mathcal{T}(e)$ is a red-black tree storing the edges from $P(e)$ in its leaves and storing a Boolean $\text{Rev}(\nu)$ at each node ν. Initially the order of the edges corresponds to the order along $P(e)$ and all Booleans $\text{Rev}(\nu)$ 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 ν with $\text{Rev}(\nu) = \text{True}$ we swap the left and right subtree of ν, set $\text{Rev}(\nu)$ to FALSE, and negate the Booleans $\text{Rev}(\cdot)$ of the children of ν. 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 $\mathcal{T}(e)$ as follows.

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

- A value $\mathit{MinCost}(\nu)$, which is defined as the minimum over all edges e' in $\mathcal{T}_\nu$ of the cost of the 2-move defined by e and e' for the local orientation of e' at ν. We also store a pointer to the edge e' defining the minimum.
- A value $\mathit{MinRevCost}(\nu)$ (with the corresponding pointer) which is defined similarly as $\mathit{MinCost}(\nu)$, except that we consider the reverse of the local orientations.

Note that if ν_1 and ν_2 are the two children of ν then

$$\mathit{MinCost}(\nu) = \begin{cases} \min(\mathit{MinCost}(\nu_1), \mathit{MinCost}(\nu_2)) & \text{if } \text{Rev}(\nu) = \text{False} \\ \min(\mathit{MinRevCost}(\nu_1), \mathit{MinRevCost}(\nu_2)) & \text{if } \text{Rev}(\nu) = \text{True}. \end{cases} \quad (3)$$

Similarly, $\mathit{MinRevCost}(\nu)$ can be computed in $\mathcal{O}(1)$ time from the information at ν, ν_1, and ν_2. Note that when $\text{Rev}(\nu)$ is negated, we can just swap the values of $\mathit{MinCost}(\nu)$ and $\mathit{MinRevCost}(\nu)$ and propagate the change upward. For each edge e' we also maintain, for each tree $\mathcal{T}(e)$, a pointer to the leaf where e' is stored. Next we show how to use the trees $\mathcal{T}(e)$ to perform a 2-OPT iteration in near-linear time.

Finding the best 2-move in $\mathcal{O}(n)$ time is easy: we simply go over all trees $\mathcal{T}(e)$ to find the one minimizing $\mathit{MinCost}(\mathit{root}(\mathcal{T}(e)))$. Let e' be the edge defining this value. We now have to

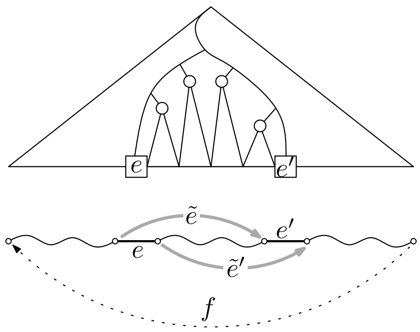

Figure 6: Situation when a 2-move with edges e, e' has to be performed on a tree $\mathcal{T}(f)$. When all nodes on the search paths to e and e' are cleared, the subtrees in between the search paths together represent the subpath from e to e' along $P(f)$.

perform a 2-move on e, e' (assuming $\mathit{MinCost}(\mathit{root}(\mathcal{T}(e))$ 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 destroy the trees $\mathcal{T}(e)$ and $\mathcal{T}(e')$, and build trees $\mathcal{T}(\tilde{e})$ and $\mathcal{T}(\tilde{e}')$ from scratch. The latter can be done in $\mathcal{O}(n)$ time after constructing the path $P(\tilde{e})$ and $P(\tilde{e}')$, which we can do in $\mathcal{O}(n)$ time. It remains to update the other trees. In the remainder of the proof we show how this can be done in $\mathcal{O}(\log n)$ time per tree, resulting in $\mathcal{O}(n \log n)$ time in total for a 2-move.

We show how to update a tree $\mathcal{T}(f)$ in logarithmic time when a 2-move with edges e, e' is performed; see also Fig. 6. Note that rotations in $\mathcal{T}(f)$ can still be done in $\mathcal{O}(1)$ 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 $\mathcal{T}(f)$, 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 $\mathcal{T}(f)$ to the leaf containing e. Similarly, concatenating two trees—the reverse operating from splitting—can be done in $\mathcal{O}(\log n)$ time. See also the paper by Chrobak *et al.* [16], who describe these operations (for AVL-trees) and without the extra fields $\mathit{MinCost}(\cdot)$ and $\mathit{MinRevCost}(\cdot)$. We can now update $\mathcal{T}(f)$ (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 $\mathcal{T}(f)$ into two subtrees, a tree $\mathcal{T}_1$ for the subpath of $P(f)$ before e, and a tree $\mathcal{T}_2$ for the subpath starting at e. The latter tree is then split further into a tree $\mathcal{T}_{2,1}$ for the subpath from e to e', and a tree $\mathcal{T}_{2,2}$ for the subpath behind e'. We then delete e and e' from $\mathcal{T}_{2,1}$, reverse the subpath in between them by negating the Boolean $\textsc{Rev}(\cdot)$ at the root of $\mathcal{T}_{2,1}$, and insert $\tilde{e}$ as first edge of the subpath and $\tilde{e}'$ 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 $\mathcal{O}(\log n)$ time after a 2-move. □

E.2 Repeated 3-OPT

The approach above can also be used to speed up 3-OPT computations in the repeated setting. To this end we maintain a data structure $\mathcal{DS}(e, e')$ for each pair e, e' of edges in the tour, which allows us to quickly find the edge e'' that gives the best 3-move with e, e'. This data structure, which is very similar to the one for 2-OPT, is defined as follows.

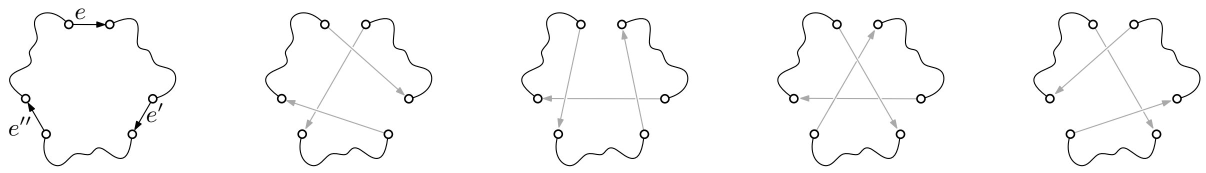

Figure 7: A triple e, e', e'' in the current tour (left) has at most four different 3-moves. ("At most" because if one or more of the subpaths between these edges are empty, then some of these 3-moves will degenerate into 2-moves.)

Let e, e' be a pair of edges from the current tour. To define $\mathcal{DS}(e, e')$ it is convenient to consider the moment at which $\mathcal{DS}(e, e')$ was created, which is the first moment e and e' both appear in the tour. Let T_{init} be this initial tour (for the pair e, e'). Fix an orientation for the edge e. This orientation determines the orientation of all other edges in T_{init}, including the edge e'. We call this orientation of e' its *initial orientation*. Note that when the tour changes due to a 3-move, the orientation of e' may change: instead of having the oriented edges e, e' in the tour we may have $e, rev(e')$ in the tour, where $rev(e')$ is the reverse of the edge e'. To deal with this, we will actually maintain two data structures for the pair of (undirected) edges: a tree $\mathcal{T}(e, e')$ and a tree $\mathcal{T}(e, rev(e'))$. Moreover, we will maintain a Boolean indicating whether the current tour uses e' or $rev(e')$. It is easily checked that these Booleans can be maintained without affecting the overall time bound.

Now consider $\mathcal{T}(e, e')$; the tree $\mathcal{T}(e, rev(e'))$ is similar. Note that the orientations of e and e' are fixed. Hence, if we consider a third edge e'' in the current tour T and we know the orientation of e'', then we also know which are the valid 3-moves for the triple e, e', e''; see Fig. 7. Thus we can define the tree $\mathcal{T}(e, e')$ in a similar fashion we defined the tree $\mathcal{T}(e)$ in the 2-OPT setting. To this end, let $P(e) := T \setminus \{e\}$ be the path resulting from removing e from the tour T. As before, we store $P(e)$ in a red-black tree with Booleans $\text{REV}(\nu)$ at each node ν, and we define the local orientation of an edge e'' at a given node ν, where ν must be such that e'' is stored in the subtree rooted at ν. We augment the nodes of $\mathcal{T}(e, e')$ with the following extra information.

- A value $MinCost(\nu)$, which is defined as the minimum over all edges $e'' \neq e'$ in $\mathcal{T}_\nu$ of the minimum cost of a valid 3-move defined by e, e', e''' for the local orientation of e' at ν and the fixed orientations of e and e'. (As mentioned, for each e'' there are up to four types of valid 3-moves.) We also store a pointer to the edge e'' defining the minimum and the type of the 3-move.
- A value $MinRevCost(\nu)$ (with the corresponding pointer and type) which is defined similarly as $MinCost(\nu)$, except that we consider the reverse of the local orientations.

Note that the recurrence relation (3) still holds.

An iteration of the repeated 3-OPT algorithm now proceeds as follows. For each pair of edges e, e' from the current tour we find the cheapest 3-move involving e, e' by considering the relevant tree—either $\mathcal{T}(e, e')$ or $\mathcal{T}(e, rev(e'))$, depending on the orientation of e' in the current tour. Recall that the cheapest 3-move is stored at the root of the tree, so it can be found in $O(1)$ time. This gives us $O(n^2)$ 3-moves to consider—one for each pair e, e'. If the best of these 3-moves has negative cost, we perform that 3-move and update our data structures. The latter is done as follows.

Let e, e', e'' be the old edges in the tour and let $\tilde{e}, \tilde{e}', \tilde{e}''$ be the new edges that replace them. We first destroy all data structures defined by any of the old edges e, e', e'', that is, all data structures $\mathcal{DS}(f, f')$ such that $\{f, f'\} \cap \{e, e', e''\} \neq \emptyset$. Next we build (from scratch) all data structures defined by the new edges $\tilde{e}, \tilde{e}', \tilde{e}''$. Since any edge is involved in $n - 1$ pairs, the total number of data structures we destroy and create is $O(n)$. Since building a data structure can be done in $O(n)$ time, this takes $O(n^2)$ time in total.

Ir remains to update the trees $\mathcal{T}(f, f')$ and $\mathcal{T}(f, rev(f'))$ for $\{f, f'\} \cap \{e, e', e''\} = \emptyset$. This can be done similarly to the 2-OPT case. More precisely, to update $\mathcal{T}(f, f')$ we proceed as follows. First we split $\mathcal{T}(f, f')$ into four subtrees by deleting the edges e, e', e''. Then we reverse one or more of the resulting subpaths by negating the Boolean $\text{REV}(\cdot)$ at the root of the corresponding subtree; which paths have to be reversed depends on the specific type of 3-move we have to perform. Finally, we insert the new edges $\tilde{e}, \tilde{e}', \tilde{e}''$ into the relevant subtrees, and we concatenate all subtrees to form the new tree $\mathcal{T}(f, f')$. Thus updating the tree$\mathcal{T}(f, f')$—and, similarly, updating $\mathcal{T}(f, rev(f'))$—takes $O(\log n)$ time. Since we have to do this for $O(n^2)$ pairs f, f' we spend $O(n^2 \log n)$ time in total.

E.3 The planar case

We now turn our attention to the planar setting. (Note that we do not consider the repeated version of the problem, but the single-shot version.) We focus on the problem of detecting any 2-move or 3-move that lowers the cost of the tour, although similar results are possible for the finding the best change.

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

2-OPT. Suppose we are given a tour T on a planar point set $P := \{p_0, p_1, \dots, p_{n-1}\}$, 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 $p_i p_{i+1}$ of T, find an edge $p_j p_{j+1}$ 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 $p_j p_{j+1}$ such that

$$|p_i p_j| + |p_{i+1} p_{j+1}| < |p_i p_{i+1}| + |p_j p_{j+1}|. \tag{4}$$

To answer these queries we map every edge $p_j p_{j+1}$ 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 [5]. Given a query edge $p_i p_{i+1}$ we define a range $Q_i \subset \mathbb{R}^4$ as

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

Thus $p_j p_{j+1}$ satisfies (4) if and only if $q_j \in Q_i$. We can therefore find an edge $p_j p_{j+1}$ satisfying (4) by performing a query with the range Q_i, which is a semi-algebraic set. In $\mathbb{R}^4$, semi-algebraic range-searching queries can be answered in $\mathcal{O}(n^{3/4+\varepsilon})$ time after $\mathcal{O}(n^{1+\varepsilon})$ preprocessing [4, 28]. Alternatively, we can "dualize" the approach, by mapping each edge $p_j p_{j+1}$ to a surface in $\mathbb{R}^4$ and mapping the query $p_i p_{i+1}$ 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 $\mathcal{O}(\log n)$ time after $\mathcal{O}(n^{4+\varepsilon})$ preprocessing [28]. By combining these two solutions in a standard manner, we can obtain a trade-off between preprocessing and query time—see e.g. [32] and also below, where we give some more details for the somewhat more complicated case of 3-OPT. In particular, we can obtain $\mathcal{O}(n^{3/5+\varepsilon})$ query time after $\mathcal{O}(n^{8/5+\varepsilon})$ preprocessing. Thus our 2-OPT algorithm needs $\mathcal{O}(n^{8/5+\varepsilon})$ 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 $p_i p_{i+1}$, find a pair of edges $p_j p_{j+1}, p_k p_{k+1}$ 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.

Assume the points are numbered $p_0, \dots, p_{n-1}$ in order along T. Define e_i to be the edge $v_i v_{i+1}$, for $0 \le i < n$, 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

- $p_i p_j, p_{i+1} p_k, p_{j+1} p_{k+1}$ (Type I);
- $p_i p_{j+1}, p_k p_{i+1}, p_j p_{k+1}$ (Type II);
- $p_i p_{j+1}, p_k p_j, p_{i+1} p_{k+1}$ (Type III);
- $p_i p_k, p_{j+1} p_{i+1}, p_j p_{k+1}$ (Type IV).

Here we have ignored the possibility that one of the edges e_1, e_2, e_3 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 $0 \le i < j < n-1$ in a data structure that can answer the following queries: given an edge e_k, find a pair e_i, e_j such that

$$j < k \quad \text{and} \quad |p_i p_j| + |p_{i+1} p_k| + |p_{j+1} p_{k+1}| < |p_i p_{i+1}| + |p_j p_{j+1}| + |p_k p_{k+1}|, \tag{5}$$

if such a pair exists. For the moment, let's 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 $p_k p_{k+1}$ we can now decide if there is an improving 3-move of Type I by searching with the range

$$\begin{aligned} Q_k \;:=\; \{\, (a_1, \dots, a_8) : \;& |(a_1,a_2)(a_5,a_6)| + |(a_3,a_4)p_k| + |(a_7,a_8)p_{k+1}| \\ & < |(a_1,a_2)(a_3,a_4)| + |(a_5,a_6)(a_7,a_8)| + |p_k p_{k+1}| \}. \end{aligned}$$

The resulting data structure uses $\mathcal{O}(n')$ space, and has $\mathcal{O}((n')^{1+\varepsilon})$ expected preprocessing time and $\mathcal{O}((n')^{7/8+\varepsilon})$ query time, where n' is the number of points stored in the data structure.

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

$$\begin{aligned} \Gamma_{ij} \;:=\; \{(a_1, \dots, a_4) : \;& |p_i p_j| + |p_{i+1}(a_1,a_2)| + |p_{j+1}(a_3,a_4)| \\ & = |p_i p_{i+1}| + |p_j p_{j+1}| + |(a_1,a_2)(a_3,a_4)|\} \end{aligned}$$

in $\mathbb{R}^4$, and preprocess the resulting arrangement for point location. Performing a point-location query with the point $(x(p_k), y(p_k), x(p_{k+1}), y(p_{k+1}))$ now tells us if there is an improving 3-move of Type I. This alternative would use $\mathcal{O}((n')^{4+\varepsilon})$ preprocessing time and have $\mathcal{O}(\log n')$ query time [28].

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 $\{q_{ij} : 0 \le i < j < n-1\}$). 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 $1 \le m \le n'$. (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 $\mathcal{O}(m^{4+\varepsilon})$ preprocessing time. This way we construct a "top tree" with $\mathcal{O}(n'/m)$ leaves, each of which is associated with a “bottom tree” that needs $\mathcal{O}(m^{4+\varepsilon})$ preprocessing. The total amount of preprocessing is $\mathcal{O}(n'm^{3+\varepsilon})$.

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

So far we ignored the condition $j < k$ in (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 time and query time [38]. In our case these logarithmic factors are swallowed by the $\mathcal{O}(n^\varepsilon)$ factor that we already have, so the total structure uses $\mathcal{O}(n'm^{3+\varepsilon})$ expected preprocessing time and has $\mathcal{O}((n'/m)^{7/8+\varepsilon})$ 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

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

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