Improving TSP Tours Using Dynamic Programming over Tree Decompositions*†

Marek Cygan[1], Łukasz Kowalik[2], and Arkadiusz Socała[3]

1 **University of Warsaw, Poland**
`cygan@mimuw.edu.pl`
2 **University of Warsaw, Poland**
`a.socala@mimuw.edu.pl`
3 **University of Warsaw, Poland**
`kowalik@mimuw.edu.pl`

Abstract

Given a traveling salesman problem (TSP) tour H in graph G a k-move is an operation which removes k edges from H, and adds k edges of G so that a new tour H' is formed. The popular k-OPT heuristic for TSP finds a local optimum by starting from an arbitrary tour H and then improving it by a sequence of k-moves.

Until 2016, the only known algorithm to find an improving k-move for a given tour was the naive solution in time $O(n^k)$. At ICALP'16 de Berg, Buchin, Jansen and Woeginger showed an $O(n^{\lfloor 2/3k \rfloor+1})$-time algorithm.

We show an algorithm which runs in $O(n^{(1/4+\epsilon_k)k})$ time, where $\lim_{k\to\infty} \epsilon_k = 0$. It improves over the state of the art for every $k \geq 5$. For the most practically relevant case $k = 5$ we provide a slightly refined algorithm running in $O(n^{3.4})$ time. We also show that for the $k = 4$ case, improving over the $O(n^3)$-time algorithm of de Berg et al. would be a major breakthrough: an $O(n^{3-\epsilon})$-time algorithm for any $\epsilon > 0$ would imply an $O(n^{3-\delta})$-time algorithm for the All Pairs Shortest Paths problem, for some $\delta > 0$.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases TSP, treewidth, local search, XP algorithm, hardness in P

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.30

1 Introduction

In the Traveling Salesman Problem (TSP) one is given a complete graph $G = (V, E)$ and a weight function $w : E \to \mathbb{N}$. The goal is to find a Hamiltonian cycle in G (also called a *tour*) of minimum weight. This is one of the central problems in computer science and operation research. It is well known to be NP-hard and has been researched from different perspectives, most notably using approximation [1, 4, 25], exponential-time algorithms [13, 16] and heuristics [24, 20, 5].

In practice, TSP is often solved by means of local search heuristics where we begin from an arbitrary Hamiltonian cycle in G, and then the cycle is modified by means of some local

* The work of M. Cygan and Ł. Kowalik is a part of the project TOTAL that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 677651). A. Socała is supported by the National Science Centre of Poland, grant number 2013/09/B/ST6/03136.

† A full version of the paper is available at `http://arxiv.org/abs/1703.05559`.

changes in a series of steps. After each step the weight of the cycle should improve; when the algorithm cannot find any improvement it stops. One of the most successful examples of this approach is the k-opt heuristic, where in each step an improving k-move is performed. Given a Hamiltonian cycle H in a graph $G = (V, E)$ a *k-move* is an operation that removes k edges from H and adds k edges of G so that the resulting set of edges H' is a new Hamiltonian cycle. The k-move is *improving* if the weight of H' is smaller than the weight of H. The k-opt heuristic has been introduced in 1958 by Croes [5] for $k = 2$, and then applied for $k = 3$ by Lin [21] in 1965. Then in 1972 Lin and Kernighan designed a complicated heuristic which uses k-moves for unbounded values of k, though restricting the space of k-moves to search to so-called sequential k-moves. A variant of this heuristic called LKH, implemented by Helsgaun [14], solves optimally instances up to 85 900 cities. Among other modifications, the variant searches for non-sequential 4- and 5-moves. From the theory perspective, the quality of the solutions returned by k-opt, as well as the length of the sequence of k-moves needed to find a local optimum, was studied, among others, by Johnson, Papadimitriou and Yannakakis [15], Krentel [18] and Chandra, Karloff and Tovey [3]. More recently, smoothed analysis of the running time and approximation ratio was investigated by Manthey and Veenstra [19] and Künnemann and Manthey [22].

In this paper we study the k-opt heuristic but we focus on its basic ingredient, namely on finding a single improving k-move. The decision problem k-OPT DETECTION is to decide, given a tour H in an edge weighted complete graph G, if there is an improving k-move. In its optimization version, called k-OPT OPTIMIZATION, the goal is to find a k-move that gives the largest weight improvement, if any. Unfortunately, this is a computationally hard problem. Namely, Marx [23] has shown that k-OPT DETECTION is $W[1]$-hard, which means that it is unlikely to be solvable in $f(k)n^{O(1)}$ time, for any function f. Later Guo, Hartung, Niedermeier and Suchý [12] proved that there is no algorithm running in time $n^{o(k/\log k)}$, unless Exponential Time Hypothesis (ETH) fails. This explains why in practice people use exhaustive search running in $O(n^k)$ time for every fixed k, or faster algorithms which explore only a very restricted subset of all possible k-moves.

Recently, de Berg, Buchin, Jansen and Woeginger [8] have shown that it is possible to improve over the naive exhaustive search. For every fixed $k \geq 3$ their algorithm runs in time $O(n^{\lfloor 2k/3 \rfloor + 1})$ and uses $O(n)$ space. In particular, it gives $O(n^3)$ time for $k = 4$. Thus, the algorithm of de Berg et al. is of high practical interest: the complexity of the $k = 4$ case now matches the complexity of $k = 3$ case, and hence it seems that one can use 4-opt in all the applications where 3-opt was fast enough. De Berg et al. show also that a progress for $k = 3$ is unlikely, namely k-OPT DETECTION has an $O(n^{3-\epsilon})$-time algorithm for some $\epsilon > 0$ iff ALL PAIRS SHORTEST PATHS problem can be solved in $O(n^{3-\delta})$-time algorithm for a $\delta > 0$.

Our Results. In this paper we extend the line of research started in [8]: we show an algorithm running in time $O(n^{(1/4+\epsilon_k)k})$ and using space $O(n^{(1/8+\epsilon_k)k})$ for every fixed k, where $\lim \epsilon_k = 0$. We are able to compute the values of ϵ_k for $k \leq 10$. These values show that our algorithm improves the state of the art for every $k = 5, \ldots, 10$ (see Table 1). A different adjustment of parameters of our algorithm results in time $O(n^{k/2+3/2})$ and additional space of $O(\sqrt{n})$, which improves the state of the art for every $k \geq 8$.

We also show a good reason why we could not improve over the $O(n^3)$-time algorithm of de Berg et al. for 4-OPT OPTIMIZATION: an $O(n^{3-\epsilon})$-time algorithm for some $\epsilon > 0$ would imply that ALL PAIRS SHORTEST PATHS can be solved in time $O(n^{3-\delta})$ for some $\delta > 0$. Note that although the family of 4-moves contains all 3-moves, it is still possible that there is no improving 3-move, but there is an improving 4-move. Thus the previous lower bound

Table 1 New running times for $k = 5, \ldots, 10$.

k	5	6	7	8	9	10
previous algorithm [8]	$O(n^4)$	$O(n^5)$	$O(n^5)$	$O(n^6)$	$O(n^7)$	$O(n^7)$
our algorithm	$O(n^{3.4})$	$O(n^4)$	$O(n^{4.25})$	$O(n^{4\frac{2}{3}})$	$O(n^5)$	$O(n^{5.2})$

of de Berg et al. does not imply our lower bound, though our reduction is essentially an extension of the one by de Berg et al. [8] with a few additional technical tricks.

We also devote special attention to the $k = 5$ case of k-OPT OPTIMIZATION problem, hoping that it can still be of a practical interest. Our generic algorithm works in $O(n^{3.67})$ time in this case. However, we show that it can be further refined, obtaining the $O(n^{3.4})$ running time. We suppose that similar improvements of order $n^{\Omega(1)}$ should be also possible for larger values of k. In Table 1 we present the running times for $k = 5, \ldots, 10$.

Our Approach. Our algorithm applies dynamic programming on a tree decomposition. This is a standard method for dealing with some sparse graphs, like series-parallel graphs or outerplanar graphs. However, in our case we work with complete graphs. The trick is to work on an implicit structure, called dependence graph D. Graph D has k vertices which correspond to the k edges of H that are chosen to be removed. A subset of edges of D corresponds to the pattern of edges to be added (as we will see the number of such patterns is bounded for every fixed k, and one can iterate over all patterns). The dependence graph can be thought of as a sketch of the solution, which needs to be embedded in the input graph G. Graph D is designed so that if it has a separator S, such that $D - S$ falls apart into two parts A and B, then once we find an optimal embedding of $A \cup S$ for some fixed embedding of S, one can forget about the embedding of A. This intuition can be formalized as dynamic programming on a tree decomposition of D, which is basically a tree of separators in D. The idea sketched above leads to an algorithm running in time $O(n^{(1/3+\epsilon_k)k})$ for every fixed k, where $\lim \epsilon_k = 0$. The reason for the exponent in the running time is that D is of maximum degree 4 and hence it has treewidth at most $(1/3 + \epsilon_k)k$, as shown by Fomin et al. [9].

The further improvement to $O(n^{(1/4+\epsilon_k)k})$ is obtained by yet another idea. We partition the n edges of H into $n^{1/4}$ buckets of size $n^{3/4}$ and we consider all possible distributions of the k edges to remove into buckets. If there are many nonempty buckets, then graph D has fewer edges, because some dependencies are forced by putting the corresponding edges into different buckets. As a result, the treewidth of D decreases and the dynamic programming runs faster. The case when there are few nonempty buckets does not give a large speed-up in the dynamic programming, but the number of such distributions is small.

2 Preliminaries

Throughout the paper let $w_1, w_2, \ldots, w_n$ and $e_1, \ldots, e_n$ be sequences of respectively subsequent vertices and edges visited by H, so that $e_i = \{w_i, w_{i+1}\}$ for $i = 1, \ldots, n-1$ and $e_n = \{w_n, w_1\}$. For $i = 1, \ldots, n-1$ we call w_i the *left endpoint* of e_i and w_{i+1} the *right endpoint* of e_i. Also, w_n is the left endpoint of e_n and w_1 is its right endpoint.

We work with undirected graphs in this paper. An edge between vertices u and v is denoted either as $\{u, v\}$ or shortly as uv.

For a positive integer i we denote $[i] = \{1, \ldots, i\}$.

2.1 Connection patterns and embeddings

Formally, a k-move is a pair of sets (E^-, E^+), both of cardinality k, where $E^- \subseteq \{e_1, \ldots, e_n\}$, $E^+ \subseteq E(G)$, and $E(H) \setminus E^- \cup E^+$ is a Hamiltonian cycle. This is the most intuitive definition of a k-move, however it has a drawback, namely it is impossible to specify E^+ without specifying E^- first. For this reason instead of listing the edges of E^+ explicitly, we will define a connection pattern, which together with E^- expressed as an *embedding* fully specifies a k-move.

A *k-embedding* (or shortly: *embedding*) is any function $f : [k] \to [n]$. A *connection k-pattern* (or shortly: *connection pattern*)[1] is any perfect matching in the complete graph on the vertex set $[2k]$. We call a connection pattern *valid* when one obtains a single k-cycle from M by identifying vertex $2i$ with vertex $(2i + 1) \bmod 2k$ for every $i = 1, \ldots, k$.

Let us show that every pair (E^-, E^+) that defines a k-move has a corresponding pair of an embedding and a connection pattern, consequently giving an intuitive explanation of the above definition of embeddings and connection patterns. Consider a move $Q = (E^-, E^+)$. Let $E^- = \{e_{i_1}, \ldots, e_{i_k}\}$, where $i_1 < i_2 < \cdots < i_k$. For every $j = 1, \ldots, k$, let v_{2j-1} and v_{2j} be the left and right endpoint of e_{i_j}, respectively. An *embedding* of the k-move Q is the function $f_Q : [k] \to [n]$ defined as $f_Q(j) = i_j$ for every $j = 1, \ldots, k$. Note that f_Q is increasing. A *connection pattern* of Q is every perfect matching M in the complete graph on the vertex set $[2k]$ such that $E^+ = \{\{v_i, v_j\} \mid \{i, j\} \in M\}$. Note that at least one such matching always exists, and if E^- contains two incident edges then there is more than one such matching. Note also that M is valid, because otherwise after applying the k-move Q we do not get a Hamiltonian cycle.

Conversely, consider a pair (f, M), where f is an increasing embedding and M is a valid connection pattern. We define $E^-_f = \{e_{f(j)} \mid j = 1, \ldots, k\}$. For every $j = 1, \ldots, k$, let v_{2j-1} and v_{2j} be the left and right endpoint of $e_{f(j)}$, respectively. Then we also define $E^+_{(f,M)} = \{v_i v_j \mid \{i, j\} \in M\}$. It is easy to see that $(E^-_f, E^+_{(f,M)})$ is a k-move.

Because of the equivalence shown above, in what follows we abuse the notation slightly and a k-move Q can be described both by a pair of edges to remove and add (E^-_Q, E^+_Q) and by an embedding-connection pattern pair (f_Q, M_Q). The *gain* of Q is defined as $\mathrm{gain}(Q) = w(E^-_Q) - w(E^+_Q)$. Given a connection pattern M and an embedding f, we can also define an M-gain of f, denoted by $\mathrm{gain}_M(f) = \mathrm{gain}(Q)$, where Q is the k-move defined by (f, M). Note that k-OPT OPTIMIZATION asks for a k-move with maximum gain.

2.2 Tree decomposition and nice tree decomposition

To make the paper self-contained, in this section we recall the definitions of tree and path decompositions and state their basic properties which will be used later in the paper. The content of this section comes from the textbook of Cygan et al. [6].

A *tree decomposition* of a graph G is a pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where T is a tree whose every node t is assigned a vertex subset $X_t \subseteq V(G)$, called a bag, such that the following three conditions hold:

(T1) $\bigcup_{t \in V(T)} X_t = V(G)$.

(T2) For every $uv \in E(G)$, there exists a node t of T such that $u, v \in X_t$.

(T3) For every $u \in V(G)$, the set $\{t \in V(T) \mid u \in X_t\}$ induces a connected subtree of T.

[1] We note that the notion of connection pattern of a k-move was essentially introduced by de Berg et al. [8] under the name of 'signature', though they used a permutation instead of a matching, which we find more natural.

The *width* of tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, denoted by $w(\mathcal{T})$, equals $\max_{t \in V(T)} |X_t| - 1$. The *treewidth* of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum possible width of a tree decomposition of G. When E is a set of edges and $V(E)$ the set of endpoints of all edges in E, by $\mathrm{tw}(E)$ we denote the treewidth of the graph $(V(E), E)$.

A *path decomposition* is a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$, where T is a path. Then $\mathcal{T}$ is more conveniently represented by a sequence of bags $(X_1, \ldots, X_{|V(T)|})$, corresponding to successive vertices of the path. The *pathwidth* of a graph G, denoted by $\mathrm{pw}(G)$, is the minimum possible width of a path decomposition of G.

In what follows we frequently use the notion of *nice tree decomposition*, introduced by Kloks [17]. These tree decompositions are more structured, making it easier to describe dynamic programming over the decomposition. A tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ can be rooted by choosing a node $r \in V(T)$, called the root of T, which introduces a natural parent-child and ancestor-descendant relations in the tree T. A rooted tree decomposition $(T, \{X_t\}_{t \in V(T)})$ is *nice* if $X_r = \emptyset$, $X_\ell = \emptyset$ for every leaf ℓ of T, and every non-leaf node of T is of one of the following three types:

- **Introduce node**: a node t with exactly one child t' such that $X_t = X_{t'} \cup \{v\}$ for some vertex $v \notin X_{t'}$.
- **Forget node**: a node t with exactly one child t' such that $X_t = X_{t'} \setminus \{w\}$ for some vertex $w \in X_{t'}$.
- **Join node**: a node t with two children t_1, t_2 such that $X_t = X_{t_1} = X_{t_2}$.

A path decomposition is nice when it is nice as tree decomposition after rooting the path in one of the endpoints. (Note that it does not contain join nodes.)

▶ **Proposition 1** (see Lemma 7.4 in [6]). *Given a tree (resp. path) decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G of width at most k, one can in time $O(k^2 \cdot \max(|V(T)|, |V(G)|))$ compute a nice tree (resp. path) decomposition of G of width at most k that has at most $O(k|V(G)|)$ nodes.*

We say that (A, B) is a *separation* of a graph G if $A \cup B = V(G)$ and there is no edge between $A \setminus B$ and $B \setminus A$. Then $A \cap B$ is a *separator* of this separation.

▶ **Lemma 2** (see Lemma 7.3 in [6]). *Let $(T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of a graph G and let ab be an edge of T. The forest $T - ab$ obtained from T by deleting edge ab consists of two connected components T_a (containing a) and T_b (containing b). Let $A = \bigcup_{t \in V(T_a)} X_t$ and $B = \bigcup_{t \in V(T_b)} X_t$. Then (A, B) is a separation of G with separator $X_a \cap X_b$.*

3 The algorithm

In this section we present our algorithms for k-OPT OPTIMIZATION. The brute-force algorithm verifies all possible k-moves. In other words, it iterates over all possible valid connection patterns and increasing embeddings. The brilliant observation of Berg et al. [8] is that we can iterate only over all possible connection patterns, whose number is bounded by $(2k)!$. In other words, we fix a valid connection pattern M and from now on, our goal is to find an increasing embedding $f : [k] \to [n]$ which, together with M, defines a k-move giving the largest weight improvement over all k-moves with connection pattern M. Instead of doing this by enumerating all $\Theta(n^k)$ embeddings, Berg et al. [8] fix carefully selected $\lfloor 2/3k \rfloor$ values of f in all $n^{\lfloor 2/3k \rfloor}$ possible ways, and then show that the optimal choice of the remaining values can be found by a simple dynamic programming running in $O(nk)$ time. Our idea is to find the optimal embedding for a given connection pattern using a more efficient approach.

3.1 Basic setup

Informally speaking, instead of guessing some values of f, we guess an *approximation* of f defined by appropriate bucketing. For each approximation b, finding an optimal embedding consistent with b is done by a dynamic programming over a tree decomposition. We stress that even without bucketing (i.e, by using a single trivial bucket of size n) our algorithm works in $n^{(1/3+\epsilon_k)k}$ time. Therefore bucketing is used to further improve the running time, but it is not essential to perform the dynamic programming on a tree decomposition.

More precisely, we partition the set $[n]$, corresponding to the edges of H, into buckets. Each bucket is an interval $\{i, i+1, \ldots, j\} \subseteq [n]$, for some $1 \le i \le j \le n$. Let n_b be the number of buckets and let B_j denote the j-th bucket, for $j = 1, \ldots, n_b$. A *bucket assignment* is any nondecreasing function $b : [k] \to [n_b]$.

Unless explicitly modified, we use all buckets of the same size $\lceil n^\alpha \rceil$, for a constant α which we set later. Then, for $j = 1, \ldots, b$ the j-th bucket is the set $B_j = \{(j-1)\lceil n^\alpha \rceil + 1, \ldots, j\lceil n^\alpha \rceil\} \cap [n]$.

Given a bucket assignment b we define the set

$$O_b = \{\{i, i+1\} \subset [k] \mid b(i) = b(i+1)\}.$$

▶ **Definition 3** (b-monotone partial embedding)**.** Let $f : S \to [n]$ be a partial embedding for some $S \subseteq [k]$. We say that f is b-monotone when

(M1) for every $i \in S$ we have $f(i) \in B_{b(i)}$, and

(M2) for every $\{i, i+1\} \in O_b$, if $\{i, i+1\} \subseteq S$, then $f(i) < f(i+1)$.

Note that a b-monotone embedding $f : [k] \to [n]$ is always increasing, but a b-monotone partial embedding does not even need to be non-decreasing (this seemingly artificial design simplifies some of our proofs). In what follows, we present an efficient dynamic programming algorithm which, given a valid connection pattern M and a bucket assignment b finds a b-monotone embedding of maximum M-gain. To this end, we need to introduce the gain of a partial embedding. Let $f : S \to [n]$ be a b-monotone partial embedding, for $S \subseteq [k]$. For every $j \in S$, let v_{2j-1} and v_{2j} be the left and right endpoint of $e_{f(j)}$, respectively. We define

$$E_f^- = \{e_{f(i)} \mid i \in S\}$$

$$E_f^+ = \{\{v_{i'}, v_{j'}\} \mid i, j \in S, i' \in \{2i-1, 2i\}, j' \in \{2j-1, 2j\}, \{i', j'\} \in M\}.$$

Then, $\text{gain}_M(f) = w(E_f^-) - w(E_f^+)$.

Note that $\text{gain}_M(f)$ does not necessarily represent the *actual* cost gain of the choice of the edges to remove represented by f. Indeed, assume that for some pair $i, j \in [k]$ there are $i' \in \{2i-1, 2i\}$ and $j' \in \{2j-1, 2j\}$ such that $\{i', j'\} \in M$. Then we say that i *interferes* with j, which means that we plan to add an edge between an endpoint of the i-th deleted edge and the j-th deleted edge. Note that if $i \in S$ (the i-th edge is chosen) and $j \notin S$ (the j-th edge is not chosen yet) this edge to be added is not known yet, and its cost is not represented in $\text{gain}_M(f)$. However, the value of $f(i)$ influences this cost. Consider the following set of interfering pairs:

$$I_M = \{\{i, j\} \mid i \text{ interferes with } j\}.$$

Note that I_M is obtained from M by identifying vertex $2i-1$ with vertex $2i$ for every $i = 1, \ldots, k$ (and the new vertex is simply called i). In particular, this implies that every connected component of the graph $([k], I_M)$ is a cycle or a single edge.

3.2 Dynamic programming over tree decomposition

Now we define the graph $D_{M,b}$, called *the dependence graph*, where $V(D_{M,b}) = [k]$ and $E(D_{M,b}) = O_b \cup I_M$. The vertices of $D_{M,b}$ correspond to the k edges to be removed from H (i.e., j corresponds to the j-th deleted edge in the sequence $e_1, \ldots, e_n$). The edges of $D_{M,b}$ correspond to dependencies between the edges to remove (equivalently, elements of the domain of an embedding). The edges from O_b are *order dependencies*: edge $\{i, i+1\}$ means that the $(i+1)$-th deleted edge should appear further on H than the i-th deleted edge. In O_b there are no edges between the last element of a bucket and the first element of the next bucket, because the corresponding constraint is forced by the assignment to buckets. The edges from I_M are *cost dependencies* (resulting from interference explained in Section 3.1).

The goal of this section is a proof of the following theorem.

▶ **Theorem 4.** *Let M be a valid connection k-pattern and let $b : [k] \to [n]$ be a bucket assignment, where every bucket is of size $\lceil n^\alpha \rceil$. Then, a b-monotone embedding of maximum M-gain can be found in $O(n^{\alpha(\mathrm{tw}(D_{M,b})+1)}k^2 + 2^k)$ time.*

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a nice tree decomposition of $D_{M,b}$ with minimum width. Such a decomposition can be found in $O^*(1.7347^k)$ time by an algorithm of Fomin and Villanger [11], though for practical purposes a simpler $O^*(2^k)$-time algorithm is advised by Bodlaender et al. [2]. For every $t \in V(T)$ we denote by V_t the union of all the bags in the subtree of T rooted in t.

For every node $t \in V(T)$, and for every b-monotone function $f : X_t \to [n]$, we will compute the following value.

$$T_t[f] = \max_{\substack{g: V_t \to [n] \\ g|_{X_t} = f \\ g \text{ is } b\text{-monotone}}} \mathrm{gain}_M(g).$$

Then, if r is the root of T, and $\emptyset$ denotes the unique partial embedding with empty domain, then $T_r[\emptyset]$ is the required maximum M-gain of a b-monotone embedding. The embedding itself (and hence the corresponding k-move) can be also found by using standard DP techniques. The values of $T_t[f]$ are computed in a bottom-up fashion. Let us now present the formulas for computing these values, depending on the kind of node in the tree T.

Leaf node. When t is a leaf of T, we know that $X_t = V_t = \emptyset$, and we just put $T_t[\emptyset] = 0$.

Introduce node. Assume $X_t = X_{t'} \cup \{i\}$, for some $i \notin X_{t'}$ where node t' is the only child of t. Denote $\Delta E_f^+ = E_f^+ \setminus E_{f|_{X_{t'}}}^+$. Then, we claim that for every b-monotone function $f : X_t \to [n]$,

$$T_t[f] = T_{t'}[f|_{X_{t'}}] + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u, v\}). \tag{1}$$

We show that (1) holds by showing the two relevant inequalities. Let g be a function for which the maximum from the definition of $T_t[f]$ is attained. Let $g' = g|_{V_{t'}}$. Note that g' is b-monotone because g is b-monotone. Hence, $\mathrm{gain}_M(g') \leq T_{t'}[f|_{X_{t'}}]$. It follows that $T_t[f] = \mathrm{gain}_M(g) = \mathrm{gain}_M(g') + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u, v\}) \leq T_{t'}[f|_{X_{t'}}] + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u, v\})$.

Now we proceed to the other inequality. Assume g' is a function for which the maximum from the definition of $T_{t'}[f|_{X_{t'}}]$ is attained. Let $g : V_t \to [n]$ be the function such that

$g|_{V_{t'}} = g'$ and $g(i) = f(i)$. Let us show that g is b-monotone. The condition $(M1)$ is immediate, since g' and f are b-monotone. For $(M2)$, consider any $\{j, j+1\} \in O_b$ such that $\{j, j+1\} \subseteq V_t$. If $i \notin \{j, j+1\}$ then $g(j) < g(j+1)$ by b-monotonicity of g', so assume $i \in \{j, j+1\}$. Then $\{j, j+1\} \subseteq X_t$, for otherwise $X_t \cap X_{t'}$ does not separate j from $j+1$, a contradiction with Lemma 2. For $\{j, j+1\} \subseteq X_t$, we have $g(j) < g(j+1)$ since $f(j) < f(j+1)$. Hence g is b-monotone, which implies $T_t[f] \geq \text{gain}_M(g)$. Then it suffices to observe that $\text{gain}_M(g) = \text{gain}_M(g') + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u,v\}) = T_{t'}[f|_{X_{t'}}] + w(e_{f(i)}) - \sum_{\{u,v\} \in \Delta E_f^+} w(\{u,v\})$. This finishes the proof that (1) holds.

Forget node. Assume $X_t = X_{t'} \setminus \{i\}$, for some $i \in X_{t'}$ where node t' is the only child of t. Then the definition of $T_t[f]$ implies that

$$T_t[f] = \max_{\substack{f': X_{t'} \to [n] \\ f'|_{X_t} = f \\ f' \text{ is } b\text{-monotone}}} T_{t'}[f']. \tag{2}$$

Join node. Assume $X_t = X_{t_1} = X_{t_2}$, for some nodes t, t_1 and t_2, where t_1 and t_2 are the only children of t. Then, we claim that for every b-monotone function $f : X_t \to [n]$ the following holds,

$$T_t[f] = T_{t_1}[f] + T_{t_2}[f] + \left(w(E_f^-) - w(E_f^+) \right), \tag{3}$$

which we prove by using arguments very similar to the ones used for the introduce nodes, and hence due to space limitations the proof is omitted and can be found in the full version [7].

Running time. Since $|V(T)| = O(k)$, in order to complete the proof of Theorem 4 it suffices to prove the following lemma.

▶ **Lemma 5.** *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a nice tree decomposition of D. Let t be a node of T. For every $i \in X_t$ let s_i be the size of the bucket assigned to i. Then, all the values of T_t can be found in time $O(k \prod_{i \in X_t} s_i)$. In particular, if all buckets are of size $\lceil n^\alpha \rceil$, then t can be processed in time $O(kn^{\alpha|X_t|})$.*

Proof. Obviously, in every leaf node the algorithm uses only $O(1)$ time.

For an introduce node, observe that evaluation of the formula (1) takes $O(k)$ time for every f, since $|\Delta E_f^+| \leq 2$ (the factor $O(k)$ is needed to read off a single value from the table). By $(M1)$, each value $f(i)$ of a b-monotone function f can be fixed in s_i ways, so the number of b-monotone functions $f : X_t \to [n]$ is bounded by $\prod_{i \in X_t} s_i$. Hence all the values of T_t are computed in time $O(k \prod_{i \in X_t} s_i)$, which is $O(kn^{\alpha|X_t|})$ when all buckets are of size $\lceil n^\alpha \rceil$.

For a forget node, a direct evaluation of (2) for all b-monotone functions $f : X_t \to [n]$ takes $O(k \prod_{i \in X_{t'}} s_i)$ time, where t' is the only child of t.

Finally, for a join node a direct evaluation of (3) takes $O(k)$ time, since $|E_f^-| \leq k$ and $|E_f^+| \leq k$. Hence all the values of T_t are computed in time $O(k \prod_{i \in X_t} s_i)$. ◀

3.3 An algorithm running in time $O(n^{(1/3+\epsilon)k})$ for k large enough

We will make use of the following theorem due to Fomin, Gaspers, Saurabh, and Stepanov [9].

▶ **Theorem 6** (Fomin et al. [9]). *For any $\epsilon > 0$, there exists an integer n_ϵ such that for every graph G with $n > n_\epsilon$ vertices,*

$$\mathrm{pw}(G) \leq \frac{1}{6}n_3 + \frac{1}{3}n_4 + \frac{13}{30}n_5 + \frac{23}{45}n_6 + n_{\geq 7} + \epsilon n,$$

where n_i is the number of vertices of degree i in G for any $i \in \{3, \dots, 6\}$ and $n_{\geq 7}$ is the number of vertices of degree at least 7.

We actually use the following corollary, which is rather immediate.

▶ **Corollary 7.** *For any $\epsilon > 0$, there exists an integer n_ϵ such that for every multigraph G with $n > n_\epsilon$ vertices and m edges where for every vertex $v \in V(G)$ we have $2 \leq \deg_G(v) \leq 4$, the pathwidth of G is at most $(m-n)/3 + \epsilon n$.*

Proof. The corollary follows from Theorem 6 by the following chain of equalities.

$$\begin{aligned}\frac{1}{6}n_3 + \frac{1}{3}n_4 &= \frac{1}{3}\left(\frac{1}{2}n_3 + n_4\right) = \frac{1}{3}\left(\frac{1}{2}(2n_2 + 3n_3 + 4n_4) - (n_2 + n_3 + n_4)\right) \\ &= \frac{1}{3}\left(\frac{1}{2}\sum_{v \in V(G)} \deg_G(v) - n\right) = \frac{1}{3}(m-n). \end{aligned} \tag{4}$$

◀

Let $P_k = \{\{i, i+1\} \mid i \in [k-1]\}$.

▶ **Lemma 8.** *For any $A \subseteq P_k$ we have $\mathrm{pw}(I_M \cup A) \leq |A|/3 + \epsilon_k k$, where $\lim_{k\to\infty} \epsilon_k = 0$.*

Proof. Although $([k], I_M \cup A)$ may not be of minimum degree 2, we may consider the edge multiset I'_M of the graph obtained from $([k], I_M)$ by replacing every single edge component $\{u, v\}$ by a 2-cycle uvu. Then I'_M is a cycle cover, so every vertex in multigraph $([k], I'_M \cup A)$ has degree between 2 and 4. Hence, by Corollary 7, for some sequence ϵ_k with $\lim_{k\to\infty} \epsilon_k = 0$ we have that $\mathrm{pw}(I_M \cup A) = \mathrm{pw}(I'_M \cup A) \leq (|I'_M| + |A| - k)/3 + \epsilon_k k \leq |A|/3 + \epsilon_k k$. ◀

By Lemma 8 it follows that the running time in Theorem 4 is bounded by $O(n^{(\frac{\alpha}{3}+\epsilon)k})$. If we do not use the buckets at all, i.e., $\alpha = 1$ and we have one big bucket of size n, we get the $O(n^{(1/3+\epsilon)k})$ bound. By iterating over all at most $(2k)!$ connection patterns we get the following result, which already improves over the state of the art for large enough k.

▶ **Theorem 9.** *For every fixed integer k, k-OPT OPTIMIZATION can be solved in time $O(n^{(1/3+\epsilon_k)k})$, where $\lim_{k\to\infty} \epsilon_k = 0$.*

3.4 An algorithm running in time $O(n^{(1/4+\epsilon)k})$ for k large enough

Let $\mathcal{M}_k$ be the set of all valid connection k-patterns.

▶ **Lemma 10.** *k-OPT OPTIMIZATION can be solved in time $2^{O(k \log k)} n^{c(k)}$, where*

$$c(k) = \max_{M \in \mathcal{M}_k} \min_{\alpha \in [0,1]} \max_{A \subseteq P_k} \left((1-\alpha)(k - |A|) + \alpha(\mathrm{tw}(I_M \cup A) + 1)\right). \tag{5}$$

Proof. We perform the algorithm from Theorem 4 for each possible valid connection pattern M and every bucket assignment b, with all the buckets of size $\lceil n^{\alpha_M} \rceil$, for some $\alpha_M \in [0,1]$. Let us bound the total running time. Let $A \subseteq P_k$ and consider a bucket assignment b such that $O_b = A$. There are $n^{(1-\alpha_M)(k-|A|)}$ such bucket assignments, and by Theorem 4 for each of them the algorithm uses time $O(n^{\alpha_M(\mathrm{tw}(I_M\cup A)+1)}k^2+2^k)$. Hence the total running time is bounded by

$$\sum_{M\in\mathcal{M}_k}\sum_{A\subseteq P_k}\sum_{\substack{b:[k]\to[\lceil n/\lceil n^{\alpha_M}\rceil\rceil]\\ b\text{ nondecreasing}\\ O_b=A}} O(n^{\alpha_M(\mathrm{tw}(I_M\cup A)+1)}k^2+2^k) =$$

$$O(2^k)\sum_{M\in\mathcal{M}_k}\sum_{A\subseteq P_k} n^{(1-\alpha_M)(k-|A|)}\cdot n^{\alpha_M(\mathrm{tw}(I_M\cup A)+1)} \tag{6}$$

For every $M\in\mathcal{M}_k$, the optimal value of α_M can be found by a simple LP (see Section 3.6). The claim follows. ◀

▶ **Theorem 11.** *For every fixed integer k, k-OPT OPTIMIZATION can be solved in time $O(n^{(1/4+\epsilon_k)k})$, where $\lim_{k\to\infty}\epsilon_k=0$.*

Proof. Fix the same value $\alpha=3/4$ for every connection pattern M. By Lemma 8 we have $(1-\alpha)(k-|A|)+\alpha(\mathrm{tw}(I_M\cup A)+1)\le(\frac{1}{4}+\frac{3}{4k}+\frac{3}{4}\epsilon'_k)k$. The claim follows by Lemma 10, after putting $\epsilon_k=\frac{3}{4k}+\frac{3}{4}\epsilon'_k$. ◀

3.5 Saving space

The algorithm from Theorem 11, as described above, uses $O(n^{(1/4+\epsilon_k)k})$ space. However, a closer look reveals that the space can be decreased to $O(n^{(1/8+\epsilon_k)k})$. This is done by exploiting some properties of the specific tree decomposition of graphs of maximum degree 4, described by Fomin et al. [9], which we used in Theorem 6.

This decomposition is obtained as follows. Let D be a k-vertex graph of maximum degree 4. As long as D contains a vertex v of degree 4, we remove v. As a result we get a set of removed vertices S and a subgraph $D'=D-S$ of maximum degree 3. Then we construct a tree decomposition $\mathcal{T}'$ of D', of width at most $(1/6+\epsilon_k)k$, given in the paper of Fomin and Høie [10]. The tree decomposition $\mathcal{T}$ of D is then obtained by adding S to every bag of $\mathcal{T}'$. An inductive argument (see [9]) shows that the width of $\mathcal{T}$ is at most $\frac{1}{3}k_4+\frac{1}{6}k_3+\epsilon_k k$.

Assume we are given a partial b-monotone embedding $f_0:S\to[n]$, where S is the set of removed vertices mentioned in the previous paragraph. Consider the dynamic programming algorithm from Theorem 4, which finds a b-monotone embedding of maximum M-gain, for a given bucket assignment b and connection pattern M. It is straightforward to modify this algorithm so that it computes a b-monotone embedding of maximum M-gain that extends f_0. The resulting algorithm runs in time $O(n^{\alpha(\mathrm{tw}(D-S)+1)}k^2)$ and uses space $O(n^{\alpha(\mathrm{tw}(D-S)+1)})$. Recalling that $\alpha=3/4$ and $\mathrm{tw}(D-S)\le(1/6+\epsilon_k)k$, we get the space bound of $O(n^{(1/8+\epsilon_k)k})$. Repeating this for each of $n^{\alpha|S|}$ embeddings of S takes time $O(n^{\alpha(|S|+\mathrm{tw}(D-S)+1)})$ instead of $O(n^{\alpha(\mathrm{tw}(D)+1)})$ from Theorem 4. However, as explained above, the bound on $\mathrm{tw}(D)$ from Theorem 6 used in the proof of Theorem 11 is also a bound on $|S|+\mathrm{tw}(D-S)$, so the time of the whole algorithm is still bounded by $O(n^{(1/4+\epsilon_k)k})$.

Another interesting observation is that if we build set S by picking an arbitrary vertex of every edge in O_b, then $D':=D-S$ contains no edges of O_b, so it has maximum degree at most 2. It follows that $\mathrm{tw}(D')\le 2$. Thus, in Lemma 10 we can bound $\mathrm{tw}(I_M\cup A)\le|A|+2$ and for $\alpha=1/2$ we get the running time of $O(n^{k/2+3/2})$. By using the approach of fixing all embeddings of S described above, we get the space of $O(n^{\alpha\mathrm{tw}(D')})=O(n^{3/2})$ which is less than the $\Theta(n^2)$ space needed to store all the distances of the TSP instance. The additional space can be further improved to $O(n^{1/2})$, details in the full version [7].

3.6 Small values of k

The value of $c(k)$ in Lemma 10 can be computed using a computer programme for small values of k, by enumerating all connection patterns and using formula (5) to find optimum α. We used a C++ implementation (see `http://www.mimuw.edu.pl/~kowalik/localtsp/localtsp.cpp` for the source code) including a simple $O(2^k)$ dynamic programming for computing treewidth described in the work of Bodlaender et al. [2]. For every valid connection pattern M our program finds the value of $\min_{\alpha\in[0,1]} \max_{A\subseteq P_k} ((1-\alpha)(k-|A|)+\alpha(\mathrm{tw}(I_M\cup A)+1))$ by solving a simple linear program, as follows.

$$
\begin{aligned}
\text{minimize}\quad & v \\
\text{subject to}\quad & v \geq (1-\alpha)(k-s) + \alpha \max_{\substack{A\subseteq P_k\\|A|=s}} (\mathrm{tw}(I_M\cup A)+1), \quad s=0,\ldots,k-1 \\
& \alpha\in[0,1]
\end{aligned}
$$

We get running times for $k=5,\ldots,10$ as described in Table 1, except that for $k=5$ the running time is $n^{3\frac{2}{3}}$. Because of the practical relevance we investigated the $k=5$ case by hand. A closer look reveals that the source of hardness of this case is a single (up to isomorphism) graph $([5], I_M\cup A)$ of treewidth 3. It turns out that using a different bucket partition design one can decrease the running time to $O(n^{3.4})$. The full argument proving the theorem below requires extensive case analysis, and does not fit in the page limit of the present conference version. It can be found in the full version [7].

▶ **Theorem 12.** 5-OPT OPTIMIZATION *can be solved in time* $O(n^{3.4})$.

4 Lower bound for $k = 4$

In this section we show a hardness result for 4-OPT OPTIMIZATION. More precisely, we work with the decision version, called 4-OPT DETECTION, where the input is the same as in 4-OPT OPTIMIZATION and the goal is to determine if there is a 4-move which improves the weight of the given Hamiltonian cycle. To this end, we reduce the NEGATIVE EDGE-WEIGHTED TRIANGLE problem, where the input is an undirected, complete graph G, and a weight function $w: E(G)\to\mathbb{Z}$. The goal is to determine whether G contains a triangle whose total edge-weight is negative.

▶ **Lemma 13.** *Every instance $I=(G,w)$ of* NEGATIVE EDGE-WEIGHTED TRIANGLE *can be reduced in $O(|V(G)|^2)$ time into an instance $I'=(G',w',C)$ of* 4-OPT DETECTION *such that G contains a triangle of negative weight iff I' admits an improving 4-move. Moreover, $|V(G')|=O(|V(G)|)$, and the maximum absolute weight in w' is larger by a constant factor than the maximum absolute weight in w.*

Proof. Let $V(G)=\{v_1,\ldots,v_n\}$, $V_{\mathsf{up}}=\{a_1,b_1,\ldots,a_n,b_n\}$, $V_{\mathsf{down}}=\{a'_1,b'_1,\ldots,a'_n,b'_n\}$ and $V(G')=V_{\mathsf{up}}\,\dot\cup\,V_{\mathsf{down}}$. Let W be the maximum absolute value of a weight in w. Then let $M_1=5W+1$ and $M_2=21M_1+1$ and let

$$
w'(u,v)=\begin{cases}
0 & \text{if } (u,v) \text{ is of the form } (a_i,b'_i)\\
w(v_i,v_j) & \text{if } (u,v) \text{ is of the form } (a_i,b_j) \text{ for } i<j \text{ or } (a'_i,b_j) \text{ for } j<i\\
M_1 & \text{if } (u,v) \text{ is of the form } (a_i,b_i)\\
-3M_1 & \text{if } (u,v) \text{ is of the form } (a'_i,b'_i)\\
-M_2 & \text{if } (u,v) \text{ is of the form } (b_i,a_{i+1}) \text{ or } (b'_i,a'_{i+1}) \text{ or } (a_1,a'_1) \text{ or } (b_n,b'_n)\\
M_2 & \text{in other case.}
\end{cases}
$$

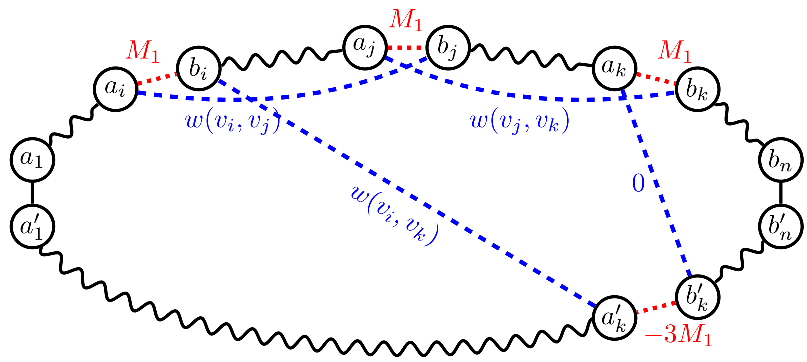

Figure 1 A simplified view of the instance (G', w', C) together with an example of a 4-move. The added edges are marked as blue (dashed) and the removed edges are marked as red (dotted).

Note that the cases are not overlapping. (Note also that although some weights are negative, we can get an equivalent instance with nonnegative weights by adding M_2 to all the weights.) The construction is illustrated in Fig. 1

If there is a negative triangle v_i, v_j, v_k for some $i < j < k$ in G then we can improve C by removing edges $(a_i, b_i), (a_j, b_j), (a_k, b_k)$ and (a'_k, b'_k) and inserting edges $(a_i, b_j), (a_j, b_k), (a_k, b'_k)$ and (a'_k, b_i). The total weight of the removed edges is $M_1 + M_1 + M_1 + (-3M_1) = 0$ and the total weight of the inserted edges is $w(v_i, v_j) + w(v_j, v_k) + 0 + w(v_k, v_i) < 0$ hence indeed the cycle is improved.

The proof in the other direction is presented in a shortened form due to space constraints (see the full version [7] for a more elaborate proof). Let us assume that C can be improved by removing 4 edges and inserting 4 edges. Note that all the edges of weight $-M_2$ belong to C and all the edges of weight M_2 do not belong to C. By the way the weights M_1 and M_2 are defined, we treat edges of weights $\pm M_2$ as fixed, i.e., they cannot be inserted or removed from the cycle in any improving 4-move. Note that the edges of C that can be removed are only the edges of the form (a_i, b_i) (of weights M_1) and (a'_i, b'_i) (of weights $-3M_1$).

All the edges of weight $-3M_1$ already belong to C, and in the next step we prove that we cannot remove more than one edge of the weight $-3M_1$ from C. Also, if we do remove one edge of the weight $-3M_1$ (i.e., of the form (a'_i, b'_i)) from C we need to remove also three edges of the weights M_1 (i.e., of the form (a_j, b_j)) in order to compensate the loss of $3M_1$.

Next, we investigate the possible locations of removed edges in an improving 4-move. We show, that if any edge is removed, then exactly three edges of the form (a_i, b_i) and exactly one edge of the form (a'_j, b'_j) have to be removed. Note that this implies also that the total weight of the removed edges has to be equal to zero.

Clearly the move has to remove at least one edge in order to improve the weight of the cycle. Let us assume that the removed edges are (a_i, b_i), (a_j, b_j) and (a_k, b_k) for some $i < j < k$ and (a'_ℓ, b'_ℓ) for some ℓ. We argue that in order to obtain a Hamiltonian cycle one of the inserted edges has to be the edge (a'_ℓ, b_i). Also the vertex b_j has to be connected with something but the vertex a'_ℓ is already taken and hence it has to be connected with the vertex a_i. Similarly the vertex b_k has to be connected with a_j because a'_ℓ and a_i are already taken. Thus a_k has to be connected with b'_ℓ and this means that $k = \ell$. The total weight change of the move is negative and therefore the total weight of the added edges has to be negative. Thus we have $w(v_i, v_j) + w(v_j, v_k) + w(v_k, v_i) = w'(a_i, b_j) + w'(a_j, b_k) + w'(a'_k, b_i) + w'(a_k, b'_k) < 0$. So v_i, v_j, v_k is a negative triangle in (G, w). ◀

▶ **Theorem 14.** *If there is* $\epsilon > 0$ *such that* 4-OPT DETECTION *admits an algorithm in time* $O(n^{3-\epsilon} \cdot \text{polylog}(M))$*, then there is* $\delta > 0$ *such that both* NEGATIVE EDGE-WEIGHTED TRIANGLE *and* ALL PAIRS SHORTEST PATHS *admit an algorithm in time* $O(n^{3-\delta} \cdot \text{polylog}(M))$*, where in all cases we refer to* n*-vertex input graphs with integer weights from* $\{-M, \ldots, M\}$*.*

Proof. The first part of the claim follows from Lemma 13, while the second part follows from the reduction of ALL PAIRS SHORTEST PATHS to NEGATIVE EDGE-WEIGHTED TRIANGLE by Vassilevska-Williams and Williams (Theorem 1.1 in [26]). ◀

References

1. Sanjeev Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.
2. Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, 2012.
3. Barun Chandra, Howard J. Karloff, and Craig A. Tovey. New results on the old k-opt algorithm for the traveling salesman problem. *SIAM J. Comput.*, 28(6):1998–2029, 1999.
4. Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976.
5. Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.
6. Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
7. Marek Cygan, Lukasz Kowalik, and Arkadiusz Socala. Improving TSP tours using dynamic programming over tree decomposition. *CoRR*, abs/1703.05559, 2017. URL: `http://arxiv.org/abs/1703.05559`.
8. Mark de Berg, Kevin Buchin, Bart M. P. Jansen, and Gerhard J. Woeginger. Fine-grained complexity analysis of two classic TSP variants. In *ICALP*, volume 55 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
9. Fedor V. Fomin, Serge Gaspers, Saket Saurabh, and Alexey A. Stepanov. On two techniques of combining branching and treewidth. *Algorithmica*, 54(2):181–207, 2009.
10. Fedor V. Fomin and Kjartan Høie. Pathwidth of cubic graphs and exact algorithms. *Inf. Process. Lett.*, 97(5):191–196, 2006.
11. Fedor V. Fomin and Yngve Villanger. Finding Induced Subgraphs via Minimal Triangulations. In Jean-Yves Marion and Thomas Schwentick, editors, *27th International Symposium on Theoretical Aspects of Computer Science*, volume 5 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 383–394, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.STACS.2010.2470`.
12. Jiong Guo, Sepp Hartung, Rolf Niedermeier, and Ondrej Suchý. The parameterized complexity of local search for tsp, more refined. *Algorithmica*, 67(1):89–110, 2013.
13. Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
14. Keld Helsgaun. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000. `doi:10.1016/S0377-2217(99)00284-2`.
15. David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. How easy is local search? *J. Comput. Syst. Sci.*, 37(1):79–100, 1988.
16. Richard M Karp. Dynamic programming meets the principle of inclusion and exclusion. *Operations Research Letters*, 1(2):49–51, 1982.

17 Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

18 Mark W. Krentel. On finding and verifying locally optimal solutions. *SIAM J. Comput.*, 19(4):742–749, 1990.

19 Marvin Künnemann and Bodo Manthey. Towards understanding the smoothed approximation ratio of the 2-opt heuristic. In *ICALP (1)*, volume 9134 of *Lecture Notes in Computer Science*, pages 859–871. Springer, 2015.

20 S. Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973. doi:10.1287/opre.21.2.498.

21 Shen Lin. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269, 1965.

22 Bodo Manthey and Rianne Veenstra. Smoothed analysis of the 2-opt heuristic for the TSP: polynomial bounds for gaussian noise. In *ISAAC*, volume 8283 of *Lecture Notes in Computer Science*, pages 579–589. Springer, 2013.

23 Dániel Marx. Searching the k-change neighborhood for TSP is w[1]-hard. *Oper. Res. Lett.*, 36(1):31–36, 2008.

24 Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991. doi:10.1137/1033004.

25 András Sebö and Jens Vygen. Shorter tours by nicer ears: 7/5-approximation for the graph-tsp, 3/2 for the path version, and 4/3 for two-edge-connected subgraphs. *Combinatorica*, 34(5):597–629, 2014.

26 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654. IEEE Computer Society, 2010.