



HAL
open science

Faster Treewidth-based Approximations for Wiener Index

Giovanna K Conrado, Amir K Goharshady, Pavel Hudec, Pingjiang Li,
Harshit J Motwani

► **To cite this version:**

Giovanna K Conrado, Amir K Goharshady, Pavel Hudec, Pingjiang Li, Harshit J Motwani. Faster Treewidth-based Approximations for Wiener Index. 2024. hal-04327333v3

HAL Id: hal-04327333

<https://hal.science/hal-04327333v3>

Preprint submitted on 27 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.


L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Faster Treewidth-based Approximations for Wiener Index

Giovanna Kobus Conrado ✉ 

Amir Kafshdar Goharshady ✉ 

Pavel Hudec ✉ 

Pingjiang Li ✉ 

Harshit Jitendra Motwani ✉ 

Department of Computer Science and Engineering

Department of Mathematics

Hong Kong University of Science and Technology (HKUST)

Clear Water Bay, New Territories, Hong Kong

Abstract

The Wiener index of a graph G is the sum of distances between all pairs of its vertices. It is a widely-used graph property in chemistry, initially introduced to examine the link between boiling points and structural properties of alkanes, which later found notable applications in drug design. Thus, computing or approximating the Wiener index of molecular graphs, i.e. graphs in which every vertex models an atom of a molecule and every edge models a bond, is of significant interest to the computational chemistry community.

In this work, we build upon the observation that molecular graphs are sparse and tree-like and focus on developing efficient algorithms parameterized by treewidth to approximate the Wiener index. We present a new randomized approximation algorithm using a combination of tree decompositions and centroid decompositions. Our algorithm approximates the Wiener index within any desired multiplicative factor $(1 \pm \epsilon)$ in time $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k/\epsilon^2)$, where n is the number of vertices of the graph and k is the treewidth. This time bound is almost-linear in n .

Finally, we provide experimental results over standard benchmark molecules from PubChem and the Protein Data Bank, showcasing the applicability and scalability of our approach on real-world chemical graphs and comparing it with previous methods.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Computational Chemistry, Treewidth, Wiener Index


Digital Object Identifier 10.4230/LIPIcs.SEA.2024.18

Acknowledgements The research was partially supported by the Hong Kong Research Grants Council ECS Project Number 26208122. G.K. Conrado and P. Hudec were supported by the Hong Kong PhD Fellowship Scheme (HKPFS). Authors are ordered alphabetically.

1 Introduction

MOTIVATION. The Wiener index of a graph G is the sum of the distances between all pairs of vertices in G . Besides being a natural problem to compute, it is also a well-studied graph invariant with applications in computational chemistry and biology. Indeed, it is one of computational chemistry's oldest and most important topological indices [60].


HISTORY. In chemistry, the Wiener index was first considered by Harry Wiener in [63]. It was initially studied to establish connections between alkanes' boiling points and the underlying graphs' structural properties. This study later motivated the development of other topological indices in computational chemistry. Further development of QSAR (Quantitative

 © Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Pavel Hudec, Pingjiang Li and Harshit Jitendra Motwani;

licensed under Creative Commons License CC-BY 4.0

22nd International Symposium on Experimental Algorithms (SEA 2024).

Editor: Leo Liberti; Article No. 18; pp. 18:1–18:17

 Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

44 Structure-Activity Relationship) and QSPR (Quantitative Structure-Property Relationship)
45 models led to the discovery of positive correlations of even more chemical and physical
46 properties to the Wiener index [48, 60, 61, 65]. Due to its simplicity and usefulness, the
47 Wiener index was also studied by computer scientists and mathematicians [31, 57]. The use
48 of neural networks in chemical graph theory has led to a renewed interest in topological
49 indices and their application in molecular mining, toxicity detection, and computer-aided
50 drug discovery. Several studies have been conducted on this topic, such as [10, 30, 32, 44, 64].
51 Given the significance of the Wiener index for chemists and the abundance of large molecules,
52 it is imperative to develop faster algorithms for computing it. Indeed, there are many previous
53 works in this direction [12, 22, 29, 40, 50].

54 **PARAMETERIZED ALGORITHMS.** Parameterized algorithms aim to tackle computationally-
55 intractable problems and identify subsets of instances that can be solved efficiently [26]. In
56 parameterized complexity, we consider an additional parameter k along with the input size
57 n for measuring the runtime. This is in contrast to classical complexity theory, which only
58 considers the input size of the problem. Many parameterized algorithms focus on NP-hard
59 problems and provide runtime bounds that depend polynomially on the size of the problem
60 but have non-polynomial dependence on the parameter k . If we know that k is small in
61 real-world instances, this leads to solutions that are effectively polynomial-time, i.e. they
62 take polynomial time on all the real-world instances where this parameter is small.

63 **FIXED-PARAMETER TRACTABLE (FPT).** Given an input of size n and a parameter k , an
64 algorithm with a running time of $O(f(k) \cdot n^c)$, for some constant c and computable function
65 f , is called *Fixed-Parameter Tractable (FPT)* [26]. The intuition is the same as above. If the
66 parameter k is small in all real-world instances of the problem, then the algorithm would
67 in practice have a polynomial runtime. Crucially, the degree c of this polynomial does not
68 depend on either k or n .

69 **TREewidth.** Treewidth is one of the most important structural parameters of graphs and has
70 been extensively studied in combinatorics and graph theory. Intuitively speaking, it measures
71 the tree-likeness of a graph [9]. Trees and forests have a treewidth of 1 and cliques on n
72 vertices have treewidth $n - 1$. The main advantage of treewidth in algorithm design arises
73 when we are designing parameterized algorithms for NP-hard problems by considering it as
74 the parameter of the problem. Many families of commonly-studied graphs, such as trees, cacti,
75 series-parallel graphs, outer-planar graphs, control-flow graphs of structured programs, and
76 conflict graphs of Bitcoin transactions have bounded treewidth [7, 9, 26, 18, 59, 13, 49, 25].
77 This allows efficient dynamic programming techniques using the tree decomposition of the
78 graph [7, 37, 17, 3, 2, 39, 24]. See Section 2 for a formal definition.

79 **TREewidth OF MOLECULES.** Extending this idea, computational chemists and biologists
80 have also explored the treewidth of various important classes of molecules [66, 68]. In our
81 experimental results (Section 4), we observe that a significant majority of molecules in the
82 PubChem repository [34] have a treewidth of at most 10. Even large proteins from the
83 Protein Data Bank [54] are observed to have a treewidth of at most 5. Since a significant
84 fraction of molecules have bounded treewidth, exploring and designing treewidth-based
85 parameterized algorithms for computational problems in chemistry and biology is a natural
86 step. In fact, the same has been done in several works in the literature [4, 12, 23, 62, 67].
87 We extend this line of research by presenting significantly faster treewidth-based approaches
88 for approximating the Wiener index.

89 **OUR CONTRIBUTION.** In this paper, we introduce a novel randomized algorithm that
90 approximates the Wiener index of a graph using its tree decomposition. The unique aspect

91 of our algorithm is the incorporation of both tree and centroid decompositions. This idea
 92 significantly enhances efficiency in answering distance queries within the graph. This is then
 93 plugged directly into an established randomized algorithm to approximate the Wiener index,
 94 obtaining the same approximation guarantees by an asymptotically faster method. Both
 95 theoretical analysis and experimental results demonstrate that our algorithm outperforms
 96 current methods in calculating the Wiener index for molecular graphs, which are commonly
 97 encountered in computational chemistry and biology.

98 **COMPARISON WITH PREVIOUS RESULTS.** Table 1 compares the runtime complexity of our
 99 algorithm with previous methods. Here, n is the number of vertices in the graph, k is the
 100 treewidth, and ϵ is the error in the approximation, i.e. we are reporting the runtime for a
 101 $(1 \pm \epsilon)$ -approximation of the Wiener index. We refer to Section 4 for a detailed experimental
 102 evaluation of our algorithm on datasets from PubChem [34] and the Protein Data Bank [54].

103 The most classical approach to compute the Wiener index is simply performing an all-pairs
 104 shortest path computation using Floyd-Warshall and then summing up the distances. This
 105 will lead to a time complexity of $O(n^3)$. In [12], the authors provided the first parameterized
 106 algorithm for the Wiener index based on treewidth. Their algorithm is a divide-and-conquer
 107 method based on orthogonal range searching and repeatedly finds small cuts using the tree
 108 decomposition. They achieve a runtime bound of $O(n \cdot \log^{k-1} n)$. Note that this is not
 109 FPT. In [21], an FPT algorithm was provided based on dynamic programming on the tree
 110 decomposition. This algorithm has a quadratic dependence on n . For unweighted graphs,
 111 given that a graph with n vertices and treewidth k has $O(n \cdot k)$ edges, running a BFS from
 112 each vertex would lead to a total runtime of $O(n^2 \cdot k)$. Finally, [40] provides an algorithm on
 113 general graphs, not using any parameters, that approximates the average pairwise distance
 114 within a factor of $(1 \pm \epsilon)$ with a probability of at least $2/3$ by taking a random sample of
 115 the distances between pairs of vertices. Note that the Wiener index is n^2 times the average
 116 distance. Thus, this algorithm is directly applicable to our setting, as well. Our algorithm
 117 builds upon the classical approximation of [40] and uses a tree decomposition and a centroid
 118 decomposition to speed up the sampling.

119 **SIMILAR WORKS.** Our distance query results are similar to those of [53, 41, 6, 1, 15, 19, 20,
 120 14, 16]. However, unlike previous works that obtain a balanced tree decomposition, i.e. a tree
 121 decomposition with height $O(\log n)$, our approach looks at the centroid decomposition of a
 122 tree decomposition. This centroid decomposition is not necessarily a valid tree decomposition
 123 of the original graph, but it has the same set of bags as the tree decomposition. Hence,
 124 unlike several previous works, our approach does not increase the width in order to obtain a
 125 balanced tree.

126 **2 Preliminaries**

127 In this section, we introduce the Wiener index and define some basic concepts of parameterized
 128 complexity. We refer to [26] for more details. This is followed by a short presentation of the
 129 classical approximation algorithm of [40], which forms the basis of our approach.

WIENER INDEX [63]. The Wiener Index of an undirected graph $G = (V, E)$ is defined as the
 all-pairs sum of distances among vertices of the graph. Formally,

$$W(G) := \sum_{u,v \in V} d(u,v).$$

130 Additionally, we define the average distance between pairs of vertices in G as $\bar{d}(G) :=$
 131 $W(G)/n^2$.

18:4 Faster Treewidth-based Approximations for Wiener Index

■ **Table 1** Comparison of Different Algorithms for Computing the Wiener Index. Here, n denotes the number of vertices, k denotes the treewidth, and ϵ represents the error of approximation.

Algorithm	Time Complexity	Type	Ref.
Floyd-Warshall	$O(n^3)$	Exact	[33]
Orthogonal Range Searching	$O(n \cdot \log^{k-1} n)$	Exact Parameterized	[12]
Treewidth-based Dynamic Programming	$O(n^2 \cdot k^2)$	Exact Parameterized	[21]
BFS	$O(n^2 \cdot k)$	Exact Parameterized	[51, 69]
Classical Approximation	$O(n^{5/2}/\epsilon^2)$	Randomized Approximation	[40]
Our Algorithm	$O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k/\epsilon^2)$	Parameterized Randomized Approximation	Sec. 3

132 ► **Remark 1.** In this work we assume that our graphs are connected, unweighted, and
 133 undirected. In the context of molecular graphs, all types of covalent bonds—be they single,
 134 double, or triple—are represented as a single undirected edge in the corresponding graph.
 135 For a disconnected graph, the Wiener index is simply $+\infty$. However, in some applications,
 136 the Wiener index of a disconnected graph is defined as the sum of the Wiener indices of its
 137 connected components. In such cases, each connected component can be processed separately.
 138 Our algorithm can easily be extended to weighted graphs, as well.

139 **TREE DECOMPOSITION (TD)** [43, 55, 56]. A *tree decomposition* of a given graph $G = (V, E_G)$
 140 is a tree $T = (\mathcal{B}, E_T)$ satisfying the following conditions:

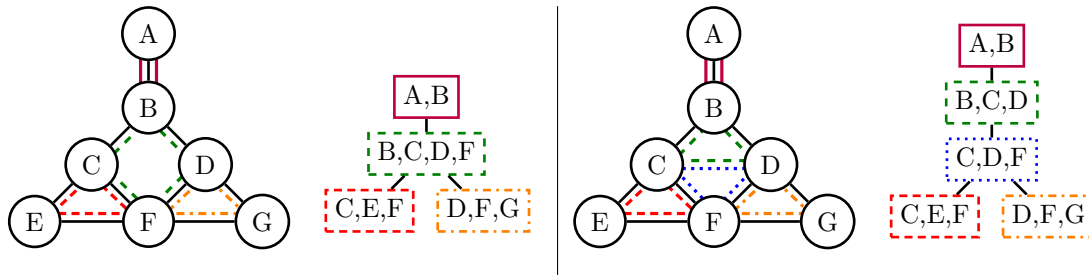
- 141 ■ Every node $b \in \mathcal{B}$ of T , which is called a *bag*, contains a subset of vertices $V_b \subseteq V$.
- 142 ■ The bags cover the entire vertex set V of G , i.e. $\bigcup_{b \in \mathcal{B}} V_b = V$. In other words, every vertex
 143 appears in at least one bag.
- 144 ■ For every edge in the original graph G , there exists a bag that contains both endpoints
 145 of the edge. More formally, for every $e = \{u, v\} \in E_G$, there is a bag $b \in \mathcal{B}$, s.t. $u, v \in V_b$.
- 146 ■ Every vertex $v \in V$ appears in a connected subtree of T , meaning that the set $\mathcal{B}_v =$
 147 $\{b \in \mathcal{B} \mid v \in V_b\}$ forms a connected subgraph of T .

148 ► **Remark 2.** An equivalent statement of the last condition above is that for every three bags
 149 $b_1, b_2, b_3 \in \mathcal{B}$, if b_3 is on the unique path from b_1 to b_2 in T , then $V_{b_1} \cap V_{b_2} \subseteq V_{b_3}$.

150 **TREewidth** [55]. The *width* of a tree decomposition T is defined as $w(T) := \max_{b \in \mathcal{B}} |V_b| - 1$,
 151 i.e. the size of the largest bag minus one. Furthermore, the *treewidth* of the graph G , denoted
 152 as $\text{tw}(G)$, is defined as the minimum width amongst all possible tree decompositions of G .

153 Intuitively speaking, treewidth measures the structural likeness of a graph to a tree.
 154 Specifically, the smaller the treewidth of a graph, the more tree-like it appears, in the sense
 155 that a graph of treewidth k can be decomposed into small parts (bags), each of size at most
 156 $k + 1$, which are connected to each other in a tree-like manner T . Figure 1 showcases an
 157 illustration containing two distinct tree decompositions of a graph G , each having a different
 158 width. Since only forests have treewidth of 1, the tree decomposition on the right is optimal,
 159 and $\text{tw}(G) = 2$.

160 Treewidth is a parameter indicating graph sparsity, providing an upper bound on the
 161 number of edges. Specifically, in a graph with n vertices and treewidth k , the number of edges
 162 is $O(n \cdot k)$. More precisely, the number of edges is less than or equal to $n \cdot k - k \cdot (k + 1/2)$ [52].
 163 Additionally, we have the following ubiquitous lemma:



■ **Figure 1** A Graph G and Two Tree Decompositions of G of Width 3 (left) and 2 (right).

164 ► **Lemma 3** (Cut Lemma [26]). Let $T = (\mathcal{B}, E_T)$ be a tree decomposition of $G = (V, E_G)$.
 165 Consider two vertices $u, v \in V$ and two arbitrary bags $b_u, b_v \in \mathcal{B}$ such that $u \in b_u$ and $v \in b_v$. If
 166 $b \in \mathcal{B}$ is a bag on the unique path from b_u to b_v in T , then any path from u to v in G will
 167 intersect V_b . Additionally, if $e = \{b_1, b_2\} \in E_T$ is an edge on the unique path from b_u to b_v in
 168 T , then any path from u to v in G will intersect $V_{b_1} \cap V_{b_2}$.

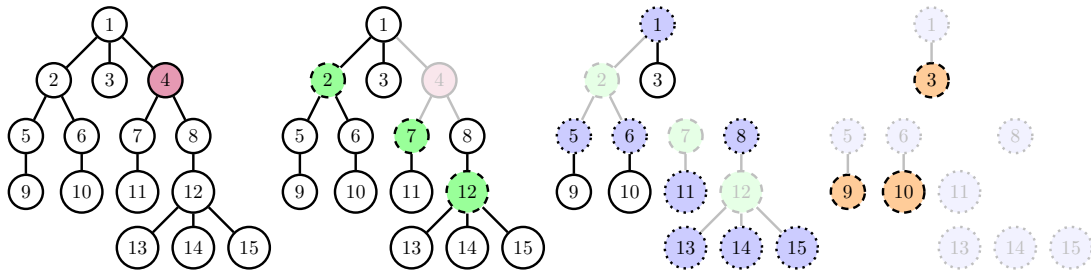
169 COMPUTING TREE DECOMPOSITIONS. In general, computing the treewidth of a given
 170 graph is an NP-hard problem. However, for small values of k , it is well-known that we
 171 can decide whether the treewidth of a given graph is at most k and also compute as an
 172 optimal tree decomposition with $O(n)$ bags by a linear-time FPT algorithm (parameterized
 173 by the treewidth itself and depending exponentially on k) [8]. Additionally, there are many
 174 well-optimized tools for this task. Thus, in the sequel, we assume without loss of generality
 175 that an optimal tree decomposition of our graph is given as a part of the input.

176 CENTROID [45]. Consider a tree $T = (V_T, E_T)$ with n vertices. We define a *centroid node* of
 177 T as a node whose removal breaks the tree down into several subtrees such that no resulting
 178 subtree has a size greater than $n/2$. In other words, a centroid is a $1/2$ -separator of T . It is
 179 well-known that every tree has at least one centroid node, which can be obtained in linear
 180 time by dynamic programming.

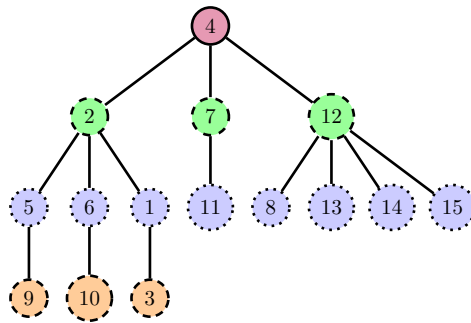
181 CENTROID DECOMPOSITION (CD) [11, 27]. A *centroid decomposition* of T is another tree
 182 T' on the same set of vertices as T , recursively defined as follows:

- 183 ■ When $|V_T| = 1$, we simply have $T' = T$.
- 184 ■ For a more complex tree, we first identify a centroid node r of T , then position this node
 185 as the root of T' .
- 186 ■ Once we have selected a centroid node r and removed it from T , we end up separating
 187 the original tree into several connected subtrees. Let us denote these as T_1, T_2, \dots, T_m .
 188 For each subtree T_i , we find a centroid decomposition T'_i with a root r_i . We make each
 189 r_i a child of r .

190 Figure 2 shows the steps of computing a centroid decomposition. Each color corresponds
 191 to a distinct layer of the centroid decomposition, with the node representing the centroid
 192 of the similarly colored dotted subtree. In this illustration, the node 4 is identified as the
 193 centroid of the initial tree. Following the removal of node 4, nodes 2, 7, and 12 are selected
 194 as the centroids of each resulting subtree. Subsequent centroids are determined in a recursive
 195 manner. The final centroid decomposition is shown in Figure 3.



■ **Figure 2** A Graph G and the Steps of Building its Centroid Decomposition. Each step highlights the centroid vertex of each of the current components of the graph.



■ **Figure 3** The Resulting Centroid Decomposition of G .

196 PROPERTIES OF CDS. The height of a CD is bounded by $O(\log n)$, where n is the number
 197 of vertices in the original tree. This is because with every new layer added to the centroid
 198 decomposition, each connected component splits into several parts, each no larger than $1/2$
 199 the size of the original component. Consequently, we can append at most $O(\log n)$ layers to
 200 the centroid decomposition. Additionally, CDs satisfy the following useful lemma:

201 ► **Lemma 4** (Proof in Appendix A). *Let $u, v \in V_T$ be two vertices of the original tree T and l be
 202 their lowest common ancestor in the centroid decomposition T' . The unique path connecting
 203 u and v in T must visit l .*

204 COMPUTING CENTROID DECOMPOSITIONS. Given a tree T with n vertices, there are a
 205 variety of algorithms in the literature that compute a centroid decomposition T' of T in
 206 $O(n)$. Examples include [11, 27].

207 LOWEST COMMON ANCESTOR QUERIES. Consider a rooted tree T with n vertices. Suppose
 208 we have q offline queries, each providing two vertices $u, v \in T$ and asking for their lowest
 209 common ancestor. The classical algorithm of Gabow and Tarjan [35] solves this problem and
 210 answers all queries in $O(n + q)$.

211 APPROXIMATION ALGORITHM OF [40]. The work [40] provides an elegant and simple
 212 approximation algorithm for the average distance $\bar{d}(G)$ between pairs of vertices. Since the
 213 Wiener index is simply $n^2 \cdot \bar{d}(G)$, the same algorithm can be reused for our problem. Given
 214 a graph G and an error bound ϵ as the input, the algorithm in [40] works as follows:

- 215 1. Uniformly select $\Theta(\sqrt{n}/\epsilon^2)$ pairs of vertices.
- 216 2. Find the distance between each selected pair of vertices.
- 217 3. Output the average of the computed distances.

218 Surprisingly, this algorithm provides a $(1 \pm \epsilon)$ -approximation of $\bar{d}(G)$ with probability $2/3$.

219 ► **Theorem 5** ([40], Theorem 5.1). *Given G and ϵ as input, the algorithm above outputs a*
220 *$(1 \pm \epsilon)$ -approximation of $\bar{d}(G)$ with probability at least $2/3$.*

221 As a direct corollary, a $(1 \pm \epsilon)$ -approximation of the Wiener index can be computed in the
222 same time complexity by simply multiplying the result of this algorithm by n^2 .

223 COMPLEXITY ANALYSIS. For general graphs, each distance query can take $O(n^2)$ time.
224 Thus, the total runtime of the algorithm above is $O(n^{5/2}/\epsilon^2)$. However, if the underlying
225 graph G is guaranteed to have small treewidth k , then it can have at most $O(n \cdot k)$ edges.
226 Thus, each distance query can be answered in $O(n \cdot k)$ by a BFS. This reduces the runtime
227 to $O(n^{3/2} \cdot k/\epsilon^2)$.

228 In this work, we build upon this simple and classical randomized algorithm and use the
229 treewidth to obtain a faster algorithm for distance queries. This allows us to reduce the
230 runtime dependence on n to almost-linear.

231 3 Our Algorithm

232 In this section, we present our treewidth-based algorithm. Our algorithm follows the same
233 steps as the approximation algorithm of [40], except that we exploit the tree decomposition
234 to perform distance queries faster. Our main novel idea is to look not only at a tree
235 decomposition of the underlying graph but also at a centroid decomposition of this tree
236 decomposition. Thus, our algorithm exploits the desirable properties of both types of
237 decomposition, as formalized by the lemma below:

238 ► **Lemma 6.** *Let $G = (V, E_G)$ be a graph, $T = (\mathcal{B}, E_T)$ a tree decomposition of G and*
239 *$T' = (\mathcal{B}, E_{T'})$ a centroid decomposition of T . Consider two vertices $u, v \in V$ and arbitrary*
240 *bags $b_u, b_v \in \mathcal{B}$ such that $u \in b_u$ and $v \in b_v$. Let l be the lowest common ancestor of b_u and b_v*
241 *in the centroid decomposition T' . Any path that goes from u to v in G intersects V_l .*

242 **Proof.** Consider a path π_T from b_u to b_v in the tree decomposition T . By Lemma 4, we have
243 $l \in \pi_T$. By Lemma 3, any bag in π_T intersects every path from u to v in G . This is illustrated
244 in Figure 4. ◀

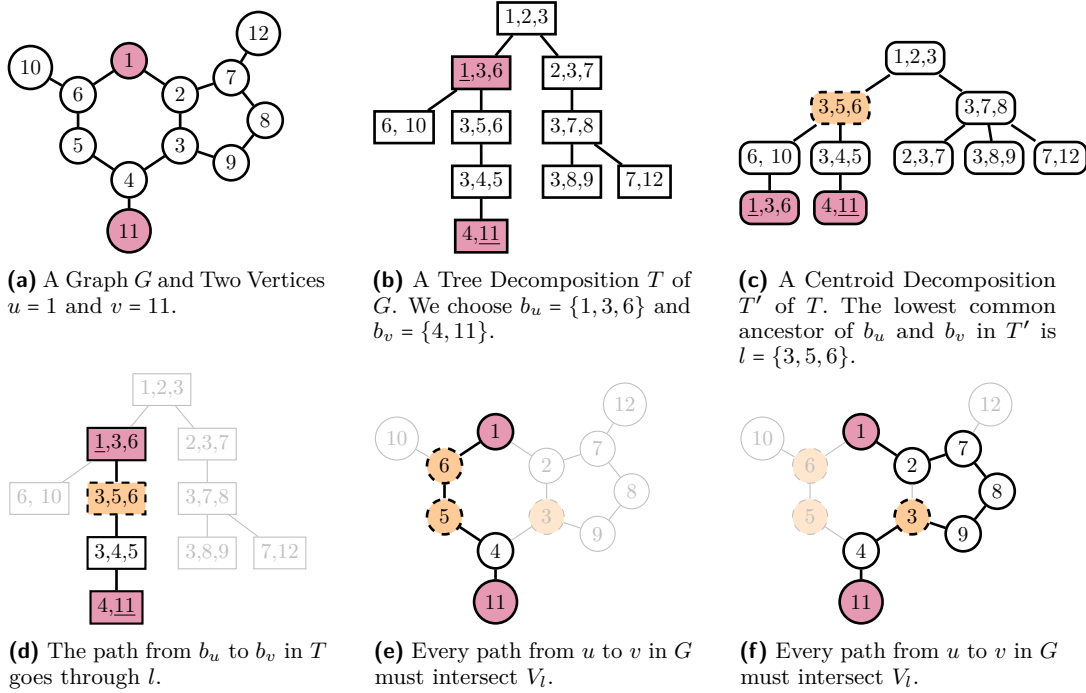
Based on the lemma above, if we precompute the distances from each vertex appearing
in a bag l of the centroid decomposition T' to the vertices appearing in descendants of l in
 T' , then we can answer distance queries in $O(k)$. In other words, to find the distance from u
to v , we first find two bags b_u and b_v containing them, then compute $l = \text{lca}(b_u, b_v)$. Now, we
know that every path from u to v has to go through l , thus

$$d_G(u, v) = \min_{w \in V_l} (d_G(u, w) + d_G(w, v)).$$

245 Here, d_G denotes the distance in our graph G .

246 OUR ALGORITHM FOR WIENER INDEX. Based on the discussion above, given $\epsilon > 0$, a graph
247 $G = (V, E_G)$ and a tree decomposition $T = (\mathcal{B}, E_T)$ of G with width k , our algorithm turns
248 G into a weighted graph and takes the following steps:

- 249 ■ **Step 1 (Centroid Decomposition).** Compute a centroid decomposition T' of the tree
250 decomposition T .
- 251 ■ **Step 2 (Local Precomputation).** For every two vertices $u, v \in V$, if there is a bag $b \in \mathcal{B}$
252 that contains both of them, i.e. $u, v \in V_b$, then compute the distance $d_G(u, v)$ and add a
253 direct edge with weight $d_G(u, v)$ between u and v .



■ **Figure 4** An Illustration of Lemma 6.

254 ■ **Step 3 (Ancestor-Descendant Precomputation).** Let $b_1, b_2 \in \mathcal{B}$ be two bags such
 255 that b_1 is an ancestor of b_2 in the centroid decomposition T' . For every $u \in V_{b_1}$ and $v \in V_{b_2}$,
 256 compute the distance $d_G(u, v)$ and add a direct edge with weight $d_G(u, v)$ between u and
 257 v .

258 ■ **Step 4 (Sampling).** Uniformly select $\Theta(\sqrt{n}/\epsilon^2)$ pairs of vertices of G as in the algorithm
 259 of [40].

260 ■ **Step 5 (Distance Queries).** For each pair of vertices $(u, v) \in V^2$ selected in the
 261 previous step, compute $d_G(u, v)$.

262 ■ **Step 6 (Output).** Output the average of all the distances obtained in the previous step.
 263 For Step 1, we can rely on previous algorithms that compute centroid decompositions,
 264 such as [11, 27]. Steps 4 and 6 are straightforward. We now provide details of Steps 2, 3,
 265 and 5, followed by correctness proofs and runtime analyses.

266 **DETAILS OF STEP 2.** This step is inspired by and similar to [21, 5, 36, 38]. Given the graph
 267 $G = (V, E_G)$ and its tree decomposition $T = (\mathcal{B}, E_T)$, our goal is to create shortcut edges
 268 between any pair of vertices that appear in the same bag. We provide a recursive procedure
 269 as follows:

- 270 i. Choose a leaf bag ℓ of the tree decomposition T .
- 271 ii. Perform an all-pairs shortest-path algorithm, such as Floyd-Warshall, in $G[V_\ell]$, i.e. only
 272 on the vertices and edges in ℓ . If a path of length d is found between u and v , add a
 273 direct $\{u, v\}$ edge with weight d to G .
- 274 iii. Let $T^* = T - \ell$ and $G^* = G - \{v \in V_\ell \mid \nexists b \in \mathcal{B} \ b \neq \ell \wedge v \in V_b\}$. In other words, we are
 275 removing the leaf bag ℓ from our tree decomposition and also removing any vertex that
 276 appeared only in this bag from the graph G .
- 277 iv. Run the algorithm recursively on (G^*, T^*) . This causes more shortcut edges to be added
 278 in G .

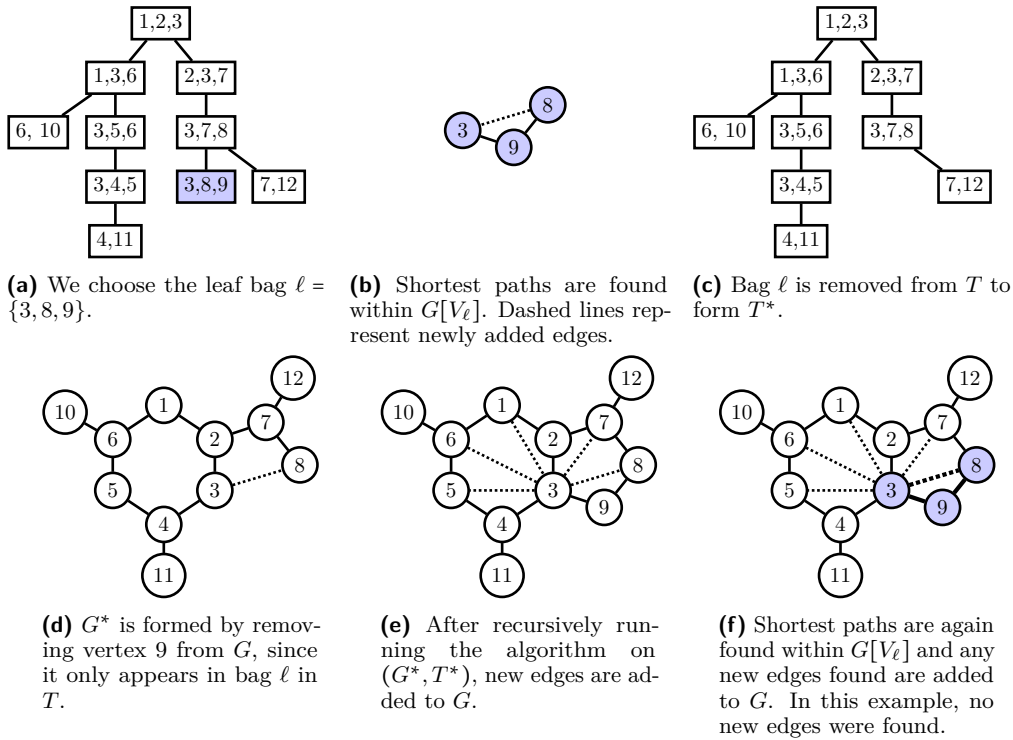


Figure 5 An Example of Step 2 on the Graph and Decomposition of Figure 4.

279 v. Repeat Step ii, i.e. perform another all-pairs shortest-path in $G[V_{\ell}]$ and add the resulting
 280 shortcut edges to G .

281 Figure 5 provides an example of this step.

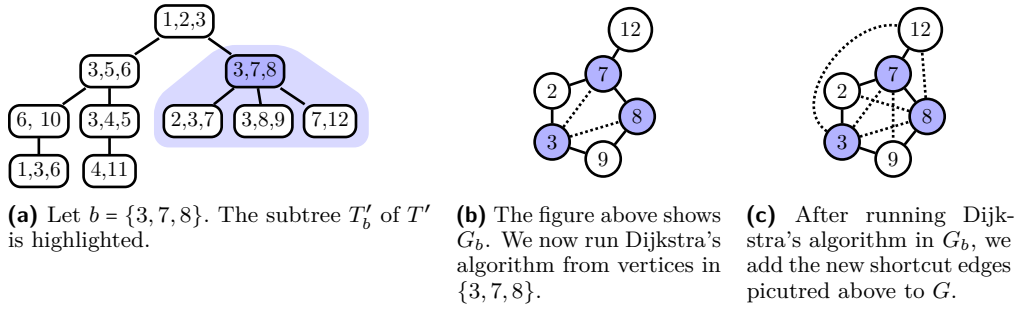
282 ► **Lemma 7** (Proof in Appendix B). *The procedure above runs in time $O(n \cdot k^3)$. After its*
 283 *execution, T is still a valid tree decomposition of G , and for every pair of vertices $u, v \in V$, if*
 284 *there exists a bag $b \in \mathcal{B}$ containing both of them, then there is a direct (shortcut) edge from u*
 285 *to v with weight $d_G(u, v)$.*

286 ► **Remark 8.** Throughout our algorithm, we always keep at most one edge, i.e. the edge with
 287 minimum weight, between every pair $\{u, v\}$ of vertices.

DETAILS OF STEP 3. In this step, we process our centroid decomposition T' in a bottom-up manner. For every bag $b \in \mathcal{B}$, we consider the subtree T'_b of the centroid decomposition T' , consisting of b and all of its descendants in T' . Let G_b be the induced subgraph of G that contains all the vertices in T'_b , i.e.

$$G_b = G \left[\bigcup_{b' \in T'_b} V_{b'} \right].$$

288 For every vertex $v \in V_b$ that appears in the bag b , our algorithm runs a shortest-path
 289 computation, such as Dijkstra's algorithm [28], from b in the graph G_b and finds its distances
 290 to all other vertices of G_b , adding the corresponding shortcut edges. See Figure 6 for an
 291 example.



■ **Figure 6** An Example of Step 3 on the Graph and Decompositions of Figure 4.

292 ▶ **Lemma 9.** *The procedure above runs in $O(n \cdot \log n \cdot k^3)$ time. After its execution, for*
 293 *every two bags $b_1, b_2 \in \mathcal{B}$ such that b_1 is an ancestor of b_2 in the centroid decomposition T'*
 294 *and every two vertices $u \in V_{b_1}$ and $v \in V_{b_2}$, we have a shortcut edge from u to v with weight*
 295 *$d_G(u, v)$.*

296 **Proof.** Let α_b and δ_b be the number of ancestors and descendants of b in T' , respectively.
 297 The graph G_b has $O(\delta_b \cdot k)$ vertices and thus $O(\delta_b \cdot k^2)$ edges. Moreover, we perform $O(k)$
 298 Dijkstras over this graph, one for each vertex in the bag b . Our graph is weighted at this
 299 point, but all edge weights and distances are non-negative integers less than n . Thus, Dijkstra
 300 runs in linear time on the number of vertices and edges. Intuitively, instead of keeping a
 301 priority queue of vertices in our Dijkstra, we can simply keep an array $A[n]$ of queues where
 302 $A[i]$ contains all vertices of distance i to the source. When we find that a particular vertex
 303 has distance i to the source, we simply add it to $A[i]$. We then process the vertices in each
 304 $A[i]$ in the order of increasing i and make sure not to process a vertex more than once.

Based on the points above, our total runtime is

$$\sum_{b \in \mathcal{B}} O(\delta_b \cdot k^3) = \sum_{b \in \mathcal{B}} O(\alpha_b \cdot k^3) = O(n \cdot \log n \cdot k^3).$$

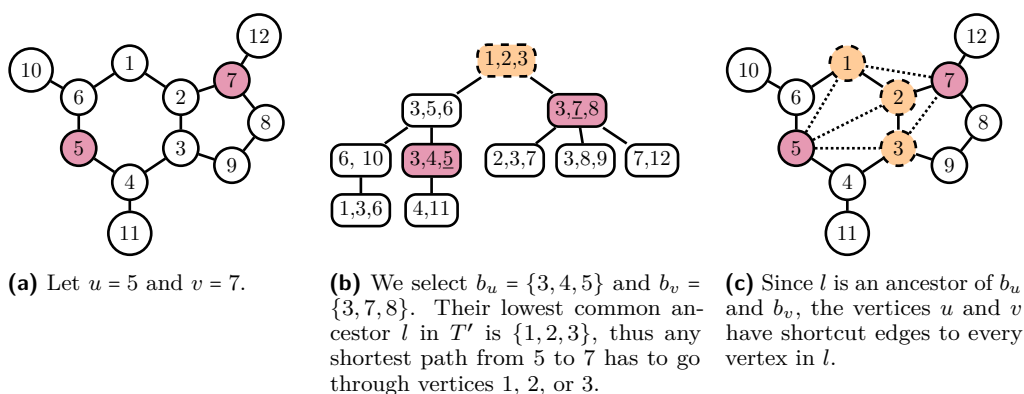
305 The latter equality is because every vertex has $O(\log n)$ ancestors.

306 For the second part, consider a shortest path π from u to v in G . Let π_T be the path
 307 from b_1 to b_2 in the tree decomposition T . By Lemma 3, π intersects the vertices of every
 308 bag b in π_T . Without loss of generality, we can assume that π stays in these bags, i.e. it only
 309 visits vertices in $\cup_{b \in \pi_T} V_b$. Note that if π leaves π_T , then it has to reenter it, but the exit and
 310 entry vertices are in the same bag and, by Lemma 7, there is already a shortcut edge between
 311 them. Additionally, since b_1 is an ancestor of b_2 in the centroid decomposition T' , there was a
 312 point in the construction of T' when b_1 was chosen as the centroid of a connected component
 313 containing b_2 . Thus, all the bags in π_T were also in the same connected component. Hence,
 314 every b is a descendant of b_1 . Therefore, the entire path π is included in G_b and the Dijkstra
 315 from u finds the shortest path to v and adds the corresponding shortcut edge. ◀

DETAILS OF STEP 5. Suppose our goal is to compute $d_G(u, v)$. We first pick two bags b_u
 and b_v such that $u \in b_u$ and $v \in b_v$. We then find the lowest common ancestor $l = \text{lca}(b_u, b_v)$.
 By Lemma 4, every path from u to v has to intersect V_l . Thus, we compute

$$d_G(u, v) = \min_{w \in V_l} (d_G(u, w) + d_G(w, v)).$$

316 Note that since l is an ancestor of both b_u and b_v , we have the distances needed on the RHS
 317 as weights of direct shortcut edges. This is illustrated in Figure 7



■ **Figure 7** An Example of Step 5 on the Graph and Decompositions of Figure 4.

318 ▶ **Lemma 10.** *The procedure above returns the correct distances in time $O(n + k \cdot \sqrt{n}/\epsilon^2)$.*

319 **Proof.** Correctness is already argued above. Since the centroid decomposition T' has $O(n)$
 320 bags, preprocessing and answering offline lowest common ancestor queries takes $O(n +$
 321 $\sqrt{n}/\epsilon^2)$ [35]. For each of the \sqrt{n}/ϵ^2 queries generated in Step 4, we should compute the
 322 minimum of $O(k)$ values since $|V_l| \leq k + 1$. ◀

323 Finally, the following is our main theorem in this work:

324 ▶ **Theorem 11.** *Given an $\epsilon > 0$, an undirected unweighted graph $G = (V, E_G)$ with n vertices
 325 and a tree decomposition $T = (\mathcal{B}, E_T)$ of G with $O(n)$ bags and width k , our algorithm runs
 326 in time $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k/\epsilon^2)$ and produces a $(1 \pm \epsilon)$ -approximation of the Wiener index
 327 $W(G)$ with probability at least $2/3$.*

328 **Proof.** Correctness of the approximation ratio and success probability follows from Theorem 5
 329 since our algorithm is the same as [40] except for how we answer distance queries. Step 1 takes
 330 $O(n)$ using well-known algorithms such as [11, 27]. Step 2 takes $O(n \cdot k^3)$ based on Lemma 7.
 331 Step 3 takes $O(n \cdot \log n \cdot k^3)$ as shown in Lemma 9. Step 4 simply takes $O(\sqrt{n}/\epsilon^2)$ samples
 332 from the uniform distribution and Step 5 takes $O(n + k \cdot \sqrt{n}/\epsilon^2)$ time as per Lemma 10.
 333 Finally, Step 6 takes $O(\sqrt{n}/\epsilon^2)$ time. Summing these up leads to the desired asymptotic
 334 time complexity. ◀

335 4 Experimental Results

336 In this section, we present our experimental results, comparing the runtimes of our algorithm
 337 with previous approaches. We implemented the main algorithms in C++ and provided the
 338 same inputs, i.e. graph G , tree decomposition T and $\epsilon = 0.1$ to all of them. To obtain this
 339 input, we first used pysmiles [46], RDKit [47] and NetworkX [42] for preprocessing molecular
 340 data and turning them into graphs. We employed the FlowCutter algorithm [58] for tree
 341 decompositions, limiting iterations to $20 + \log n$, and obtained results in under 1 second. All
 342 our experiments were conducted on an Intel Core i5 (2.3 GHz, Quad-core) Machine with 8
 343 GB of RAM running MacOS. We enforced a time limit of 1000 seconds per instance.

344 **BENCHMARKS.** We used the following datasets for our experiments: (i) PubChem [34] and
 345 (ii) Protein Data Bank (PDB). Specifically, we report results on 1049 randomly-selected
 346 protein molecules from the PDB database and 1,311,229 molecules from PubChem.

■ **Table 2** Statistics of the PDB Benchmarks

	Minimum	Maximum	Average
Number of Vertices	132	90507	6651
Number of Edges	134	98828	6820
Treewidth	2	5	3.13

■ **Table 3** Statistics of the PubChem Benchmarks

Metric	Minimum	Maximum	Average
Number of Vertices	2	568	21
Number of Edges	1	643	22
Treewidth	1	16	1.8

347 PDB. The Protein Data Bank (PDB) [54] is an extensive repository of three-dimensional
 348 structural data for large biological molecules, including proteins, DNA and RNA. We randomly
 349 selected 1049 protein molecules from this database. Table 2 shows some statistics about these
 350 molecules. We observed that even the large molecules in this dataset have small treewidth.

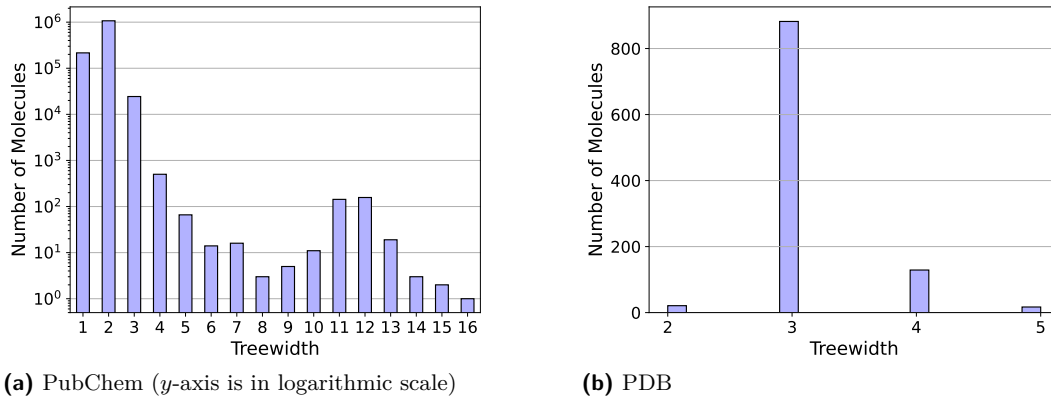
351 PUBCHEM. PubChem [34] is an open chemistry database of the National Institutes of Health
 352 (NIH). It includes information on chemical structures, identifiers, chemical and physical
 353 properties, and biological activities of small molecules. As benchmarks, we took the following
 354 datasets from PubChem: Common Chemistry CAS, Nature Catalysis, Wikipedia, Nature
 355 Communications, Wiley, Springer Nature, Nature Chemistry, Nature Portfolio Journals,
 356 Springer Materials, Drug and Medication, Nature Synthesis, Nature Chemical Biology,
 357 KEGG, DrugBank. Collectively, these datasets contained 1,311,229 molecules at the time of
 358 writing. See Table 3 for the statistics over this set of benchmarks.

359 TREEWIDTH OF THE MOLECULES. As mentioned in Tables 2 and 3, we observed that the
 360 chemical compounds in both benchmark suites exhibit small treewidth. Figure 8 provides a
 361 histogram for each benchmark suite. Notably, the vast majority of PubChem compounds
 362 have a treewidth of less than 10, with very few molecules having treewidths of up to 16. In
 363 addition, the large molecules in the PDB dataset also have bounded treewidths of at most 5.

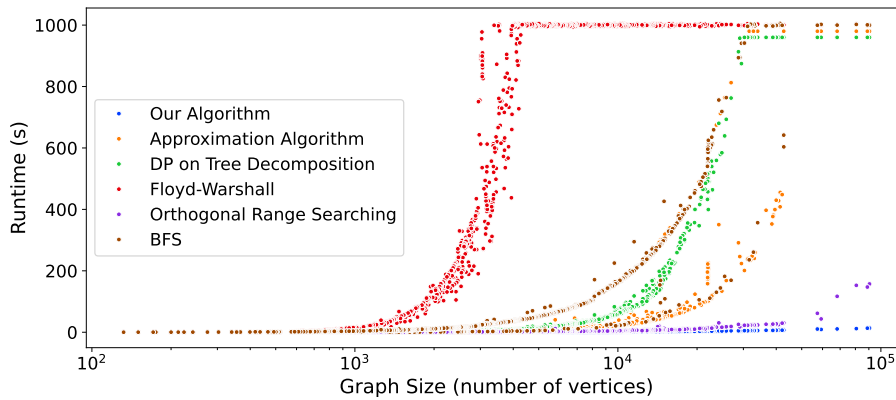
364 RESULTS. Figure 9 compares the performance of our algorithm and previous methods over the
 365 PDB dataset, whereas Table 4 provides the same comparison for PubChem. Our approach’s
 366 better asymptotic complexity leads to significant gains in efficiency when considering the
 367 large graphs in PDB. However, no benefit is observed over the PubChem molecules, since
 368 they are all small and every algorithm can handle them in under 1 ms.

369 5 Conclusion

370 In this work, we considered the problem of computing the Wiener index, i.e. sum of all
 371 pairwise vertex distances, of a graph with n vertices and treewidth k . We provided a
 372 novel algorithm using a combination of tree decompositions and centroid decompositions,
 373 which achieves an almost-linear FPT runtime of $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k/\epsilon^2)$ and outputs a
 374 $(1 \pm \epsilon)$ -approximation of the Wiener index with probability at least $2/3$. To the best of our
 375 knowledge, this is the first sub-quadratic time FPT algorithm for this problem. We also
 376 showed that many real-world molecular graphs have small treewidth and thus our algorithm
 377 is applicable in practice.



■ **Figure 8** Treewidth Distribution in Our Benchmarks



■ **Figure 9** Runtime Comparison of the Algorithms of Table 1 over PDB Benchmarks. Each dot corresponds to one benchmark molecule.

■ **Table 4** Runtime Comparison of the Algorithms of Table 1 over PubChem Benchmarks. All times are in milliseconds.

Algorithm	Maximum	Minimum	Average
Our Algorithm	1.425	0.187	0.296454
Approximation Algorithm	1.283	0.203	0.297342
DP on Tree Decomposition	1.256	0.192	0.296152
Floyd-Warshall	2.261	0.199	0.288638
Orthogonal Range Searching	1.121	0.199	0.292404
BFS	1.097	0.205	0.290523

378 — References

- 379 1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. A
380 hub-based labeling algorithm for shortest paths in road networks. In *SEA*, volume 6630, pages
381 230–241, 2011.
- 382 2 Ali Ahmadi, Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer,
383 Roodabeh Safavi, and Đorđe Zikelić. Algorithms and hardness results for computing cores of
384 markov chains. In *FSTTCS*, volume 250, pages 29:1–29:20, 2022.
- 385 3 Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. Efficient
386 approximations for cache-conscious data placement. In *PLDI*, pages 857–871, 2022.
- 387 4 Tatsuya Akutsu and Hiroshi Nagamochi. Comparison and enumeration of chemical graphs.
388 *Computational and structural biotechnology journal*, 5(6):e201302004, 2013.
- 389 5 Ali Asadi, Krishnendu Chatterjee, Amir Kafshdar Goharshady, Kiarash Mohammadi, and
390 Andreas Pavlogiannis. Faster algorithms for quantitative analysis of mcs and mdps with small
391 treewidth. In *ATVA*, pages 253–270, 2020.
- 392 6 Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in
393 contraction hierarchies. *Theor. Comput. Sci.*, 645:112–127, 2016.
- 394 7 Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *ICALP*,
395 volume 317, pages 105–118, 1988.
- 396 8 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth.
397 *SIAM J. Comput.*, 25(6):1305–1317, 1996.
- 398 9 Hans L Bodlaender et al. A tourist guide through treewidth. 1992.
- 399 10 Danail Bonchev. *Chemical graph theory: introduction and fundamentals*, volume 1. CRC Press,
400 1991.
- 401 11 Gerth Stølting Brodal, Rolf Fagerberg, Christian N. S. Pedersen, and Anna Östlin. The
402 complexity of constructing evolutionary trees using experiments. In *ICALP*, volume 2076,
403 pages 140–151, 2001.
- 404 12 Sergio Cabello and Christian Knauer. Algorithms for graphs of bounded treewidth via
405 orthogonal range searching. *Computational Geometry*, 42(9):815–824, 2009.
- 406 13 Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. The
407 treewidth of smart contracts. In *SAC*, pages 400–408, 2019.
- 408 14 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Prateesh Goyal, Rasmus Ibsen-Jensen,
409 and Andreas Pavlogiannis. Faster algorithms for dynamic algebraic queries in basic rsm with
410 constant treewidth. *ACM Trans. Program. Lang. Syst.*, 41(4):23:1–23:46, 2019.
- 411 15 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas
412 Pavlogiannis. Algorithms for algebraic path properties in concurrent systems of constant
413 treewidth components. In *POPL*, pages 733–747, 2016.
- 414 16 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas
415 Pavlogiannis. Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In
416 *ESOP*, volume 12075, pages 112–140, 2020.
- 417 17 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis.
418 Efficient parameterized algorithms for data packing. *Proc. ACM Program. Lang.*, 3(POPL):53:1–
419 53:28, 2019.
- 420 18 Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. JTDec: A
421 tool for tree decompositions in soot. In *ATVA*, pages 59–66, 2017.
- 422 19 Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Kafshdar Goharshady, and Andreas
423 Pavlogiannis. Algorithms for algebraic path properties in concurrent systems of constant
424 treewidth components. *ACM Trans. Program. Lang. Syst.*, 40(3):9:1–9:43, 2018.
- 425 20 Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Optimal tree-
426 decomposition balancing and reachability on low treewidth graphs. 2014.
- 427 21 Shiva Chaudhuri and Christos D Zaroliagis. Shortest paths in digraphs of small treewidth.
428 part i: Sequential algorithms. *Algorithmica*, 27:212–226, 2000.

- 429 22 Victor Chepoi and Sandi Klavžar. The wiener index and the szeged index of benzenoid systems
430 in linear time. *Journal of chemical information and computer sciences*, 37(4):752–755, 1997.
- 431 23 Giovanna K Conrado, Amir K Goharshady, Harshit J Motwani, and Sergei Novozhilov.
432 Parameterized algorithms for topological indices in chemistry. *arXiv preprint arXiv:2303.13279*,
433 2023.
- 434 24 Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Kerim Kochekov, Yun Chen Tsai, and
435 Ahmed Khaled Zaher. Exploiting the sparseness of control-flow and call graphs for efficient and
436 on-demand algebraic program analysis. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1993–2022,
437 2023.
- 438 25 Giovanna Kobus Conrado, Amir Kafshdar Goharshady, and Chun Kit Lam. The bounded
439 pathwidth of control-flow graphs. *Proc. ACM Program. Lang.*, 7(OOPSLA2):292–317, 2023.
- 440 26 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin
441 Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 2015.
- 442 27 Davide della Giustina, Nicola Prezza, and Rossano Venturini. A new linear-time algorithm for
443 centroid decomposition. In *SPIRE*, pages 274–282, 2019.
- 444 28 E Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*,
445 1:269–271, 1959.
- 446 29 Andrey A Dobrynin, Ivan Gutman, Sandi Klavžar, and Petra Žigert. Wiener index of hexagonal
447 systems. *Acta Applicandae Mathematica*, 72:247–294, 2002.
- 448 30 Alexander G Dossetter, Edward J Griffen, and Andrew G Leach. Matched molecular pair
449 analysis in drug discovery. *Drug Discovery Today*, 18(15-16):724–731, 2013.
- 450 31 Roger C Entringer, Douglas E Jackson, and DA Snyder. Distance in graphs. *Czechoslovak*
451 *Mathematical Journal*, 26(2):283–296, 1976.
- 452 32 Ernesto Estrada and Eugenio Uriarte. Recent advances on the role of topological indices in
453 drug discovery research. *Current Medicinal Chemistry*, 8(13):1573–1588, 2001.
- 454 33 Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- 455 34 National Center for Biotechnology Information. Pubchem database. <https://pubchem.ncbi.nlm.nih.gov>.
- 456
- 457 35 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of
458 disjoint set union. In *STOC*, pages 246–251, 1983.
- 459 36 Amir Kafshdar Goharshady. *Parameterized and Algebro-geometric Advances in Static Program*
460 *Analysis*. PhD thesis, Institute of Science and Technology Austria, Klosterneuburg, Austria,
461 2020.
- 462 37 Amir Kafshdar Goharshady, Mohammad Reza Hooshmandasl, and M. Alambardar Meybodi.
463 $[1, 2]$ -sets and $[1, 2]$ -total sets in trees with algorithms. *Discret. Appl. Math.*, 198:136–146,
464 2016.
- 465 38 Amir Kafshdar Goharshady and Fatemeh Mohammadi. An efficient algorithm for computing
466 network reliability in small treewidth. *Reliab. Eng. Syst. Saf.*, 193:106665, 2020.
- 467 39 Amir Kafshdar Goharshady and Ahmed Khaled Zaher. Efficient interprocedural data-flow
468 analysis using treedepth and treewidth. In *VMCAI*, volume 13881, pages 177–202, 2023.
- 469 40 Oded Goldreich and Dana Ron. Approximating average parameters of graphs. *Random*
470 *Structures & Algorithms*, 32(4):473–493, 2008.
- 471 41 Siddharth Gupta, Adrian Kosowski, and Laurent Viennot. Exploiting hopsets: Improved
472 distance oracles for graphs of constant highway dimension and beyond. In *ICALP*, volume
473 132, pages 143:1–143:15, 2019.
- 474 42 Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and
475 function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos,
476 NM (United States), 2008.
- 477 43 Rudolf Halin. S-functions for graphs. *Journal of geometry*, 8:171–186, 1976.
- 478 44 Christoph Helma. *Predictive toxicology*. CRC Press, 2005.
- 479 45 Camille Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathem-*
480 *atik*, 70:185–190, 1869.

- 481 46 Peter C Kroon. pysmiles: A python library for parsing smiles strings. [https://pypi.org/
482 project/pysmiles/](https://pypi.org/project/pysmiles/).
- 483 47 Gregory Landrum. Rdkit: Open-source cheminformatics. <https://www.rdkit.org>.
- 484 48 Jerzy Leszczynski. *Handbook of computational chemistry*, volume 3. Springer Science &
485 Business Media, 2012.
- 486 49 Mohsen Alambardar Meybodi, Amir Kafshdar Goharshady, Mohammad Reza Hooshmandasl,
487 and Ali Shakiba. Optimal mining: Maximizing bitcoin miners' revenues from transaction fees.
488 In *Blockchain*, pages 266–273, 2022.
- 489 50 Bojan Mohar and Tomaž Pisanski. How to compute the wiener index of a graph. *Journal of
490 mathematical chemistry*, 2(3):267–277, 1988.
- 491 51 Edward F Moore. The shortest path through a maze. In *Proc. of the International Symposium
492 on the Theory of Switching*, pages 285–292, 1959.
- 493 52 Jaroslav Nešetřil and Patrice Ossona De Mendez. Structural properties of sparse graphs. In
494 *Building Bridges: Between Mathematics and Computer Science*, pages 369–426. Springer, 2008.
- 495 53 Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Xuemin Lin, and Ying Zhang. When hierarchy
496 meets 2-hop-labeling: efficient shortest distance and path queries on road networks. *VLDB J.*,
497 32(6):1263–1287, 2023.
- 498 54 RCSB. Protein data bank. <https://www.rcsb.org>.
- 499 55 Neil Robertson and Paul D Seymour. Graph minors. iii. planar tree-width. *Journal of
500 Combinatorial Theory, Series B*, 36(1):49–64, 1984.
- 501 56 Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width.
502 *Journal of algorithms*, 7(3):309–322, 1986.
- 503 57 L'ubomír Šoltés. Transmission in graphs: a bound and vertex removing. *Mathematica Slovaca*,
504 41(1):11–16, 1991.
- 505 58 Ben Strasser and KIT algorithms group. Flowcutter: Software for computing flow-based
506 balanced graph cuts. <https://github.com/kat-algo/flow-cutter-pace17>.
- 507 59 Mikkel Thorup. All structured programs have small tree width and good register allocation.
508 *Information and Computation*, 142(2):159–181, 1998.
- 509 60 Nenad Trinajstić. *Chemical graph theory*. Routledge, 2018.
- 510 61 Stephan Wagner and Hua Wang. *Introduction to chemical graph theory*. CRC Press, 2018.
- 511 62 Pengfei Wan, Jianhua Tu, Shenggui Zhang, and Binlong Li. Computing the numbers of
512 independent sets and matchings of all sizes for graphs with bounded treewidth. *Applied
513 Mathematics and Computation*, 332:42–47, 2018.
- 514 63 Harry Wiener. Structural determination of paraffin boiling points. *Journal of the American
515 chemical society*, 69(1):17–20, 1947.
- 516 64 Jun Xu and Arnold Hagler. Chemoinformatics and drug discovery. *Molecules*, 7(8):566–600,
517 2002.
- 518 65 Ling Xue and Jurgen Bajorath. Molecular descriptors in chemoinformatics, computational
519 combinatorial chemistry, and virtual screening. *Combinatorial chemistry & high throughput
520 screening*, 3(5):363–372, 2000.
- 521 66 Atsuko Yamaguchi, Kiyoko F Aoki, and Hiroshi Mamitsuka. Graph complexity of chemical
522 compounds in biological pathways. *Genome Informatics*, 14:376–377, 2003.
- 523 67 Atsuko Yamaguchi, Kiyoko F Aoki, and Hiroshi Mamitsuka. Finding the maximum common
524 subgraph of a partial k-tree and a graph with a polynomially bounded number of spanning
525 trees. *Information Processing Letters*, 92(2):57–63, 2004.
- 526 68 Atsuko Yamaguchi and Kiyoko F Aoki-Kinoshita. Chemical compound complexity in biological
527 pathways. *Quantitative Graph Theory: Mathematical Foundations and Applications*, page 471,
528 2014.
- 529 69 Konrad Zuse. Der plankalkül. 1972.

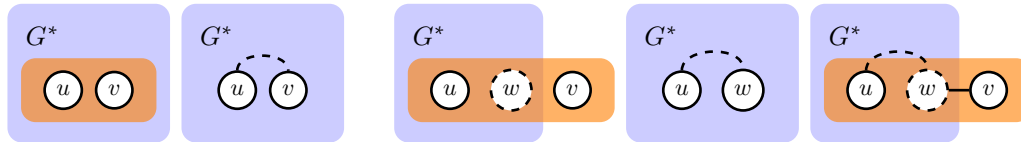
A Proof of Lemma 4

530

531 **Proof.** We prove this lemma through induction on the size n of the tree. If n is at most 3,
 532 the lemma holds trivially. Now assume that the lemma holds for all trees with a size less
 533 than n . Let us consider a general tree of size n . In the first step, we identify a centroid node
 534 of T , denoted as c . Removing c breaks T into several connected components. If any two
 535 vertices $u, v \in V_T$ are in the same connected component T_i , then in the corresponding centroid
 536 decomposition T' , they will appear in T'_i as per the definition of centroid decomposition. By
 537 the induction hypothesis, their path must cross their lowest common ancestor in T'_i . In case
 538 they belong to different connected components, say T'_i and T'_j , any path from T'_i and to T'_j
 539 must traverse the node c . In this scenario, their lowest common ancestor would be the root
 540 c , as the remaining nodes on the path from u to v are either in T_i or T_j and, hence, cannot
 541 be a common ancestor. ◀

B Proof of Lemma 7

542



(a) When u and v appear in G^* , a shortcut edge will be calculated during the recursive call on G^* .

(b) If v is not in G^* , its path to u must contain a vertex w that is in the same bag ℓ as u and v and that also appears in G^* . A shortcut edge from u to w will be added during the processing of G^* and thus the path from u to v can be calculated in Step v.

■ **Figure 10** An Illustration of Lemma 7.

543 **Proof.** We run the Floyd-Warshall algorithm twice on each bag of the tree decomposition,
 544 once in Step ii and once in v. Since each bag has $k + 1$ vertices and the tree decomposition
 545 has $O(n)$ bags, the total runtime is $O(n \cdot k^3)$. The procedure above adds new shortcut edges
 546 only between pairs of vertices that were already in the same bag, thus the tree decomposition
 547 remains valid.

548 We prove the last property by induction on $|\mathcal{B}|$. If $|\mathcal{B}| = 1$, then the first Floyd-Warshall
 549 in Step ii adds all the necessary shortcut edges. Otherwise, let $u, v \in V_\ell$ be two vertices that
 550 appear in the leaf bag ℓ and let $p \in \mathcal{B}$ be the parent of ℓ in T . If there is a path between u
 551 and v that is entirely within V_ℓ , then Step ii adds a shortcut edge summarizing this path.
 552 Thus, if $u', v' \in G^*$, then $d_{G^*}(u, v) = d_G(u, v)$. Moreover, both u and v have to appear in p ,
 553 since they each appear in a connected subtree. Hence, by induction hypothesis, the recursive
 554 call in Step iv adds the required shortcut edge between u and v . Now consider the case
 555 where either u or v (or both) are not in G^* . Take a shortest path π from u to v in G . If π
 556 is entirely within V_ℓ , then Step ii adds the shortcut edge. Otherwise, we use Lemma 3 to
 557 break π down as $\pi = \pi_1 \cdot w_1 \cdots w_2 \cdot \pi_2$ where π_1 is the longest prefix of π that only contains
 558 vertices from $V_\ell \setminus V_p$ and π_2 is the longest such suffix. By Lemma 3, we have $w_1, w_2 \in V_\ell \cap V_p$.
 559 Since they are both in $V_p \subseteq V_{G^*}$, Step iv adds a shortcut edge from w_1 to w_2 . Hence, Step v
 560 adds a shortcut edge from u to v with the correct weight. Finally, if u and v are vertices
 561 that appear in the same bag $b \neq \ell$, then the recursive call on (G^*, T^*) adds a shortcut edge
 562 between them. ◀