Faster Treewidth-based Approximations for Wiener Index

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

Faster Treewidth-based Approximations for Wiener Index

Giovanna K. Conrado[1], Amir K. Goharshady[1], Pavel Hudec[1], Pingjiang Li[1], and Harshit J. Motwani[1]

[1]Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong

December 6, 2023

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 discovered to examine the link between the boiling points and structural properties of alkanes, but 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 decomposition and centroid decomposition. Our algorithm approximates the Wiener index within a multiplicative factor of $(1+\epsilon)$ in time $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot \log n \cdot (\log\log n + k)/\epsilon^2)$, where n is the number of vertices of the graph, k is the treewidth, and ϵ is the error of approximation. Note that the time bound is almost-linear in n.

Finally, we provide experimental results over the 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 the previous state-of-the-art methods.

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 [1].

In chemistry, the Wiener index was first considered by Harry Wiener in [2]. It was initially studied to establish the connection 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 structure-activity relationship) and QSPR (quantitative structure-property relationship) models led to the discovery of positive correlations of even more chemical and physical properties to the Wiener index [3, 1, 4, 5]. Due to its simplicity and usefulness, the Wiener index was also studied by computer scientists and mathematicians [6, 7].

Although discovered almost 75 years ago, the Wiener index is still relevant in computational chemistry today. The use of neural networks in chemical graph theory has led to a renewed interest in topological indices and their application in molecular mining, toxicity detection, and computer-aided drug discovery. Several studies have been conducted on this topic, such as [8, 9, 10, 11, 12].

Given the significance of the Wiener index for chemists and the abundance of large molecules, it is imperative to develop faster algorithms for computing it. Indeed, there are many previous works in this direction [13, 14, 15, 16, 17].

Our goal in this paper is to contribute to the ongoing effort of creating faster algorithms for calculating the Wiener index. Our focus has been on developing a randomized approximation algorithm that utilizes the tree and centroid decomposition of the graph.

PARAMETERIZED ALGORITHMS. Parameterized algorithms are principally aimed at tackling computationally intractable problems [18]. These algorithms consider an additional parameter k along with the input size for measuring the runtime, unlike the classical algorithms, which only consider the input size of the problem. Most parameterized algorithms focus on NP-hard problems. In particular, an efficient parameterized algorithm for such problems would have a polynomial dependence on the size of the problem and non-polynomial dependence on the parameter k. This leads to solutions that are effectively polynomial-time, i.e. they take polynomial time on all the real-world instances where this parameter is small.

FIXED-PARAMETER TRACTABLE. Given an input of size n and a parameter k, an algorithms with a running time of $O(f(k) \cdot n^c)$, for some constant c and computable function f, is called fixed-parameter tractable (FPT) [18]. The intuition is the same as above. If the parameter k is small in all real-world instances of the problem, then the algorithm would in practice have a polynomial runtime. Crucially, the degree of this polynomial should not depend on either k or n.

TREEWIDTH. Treewidth is one of the most important structural parameters of graphs that has been extensively studied in combinatorics and algorithms. Intuitively speaking, it measures the tree-likeness of the graph [19]. Trees and forests have a treewidth of 1, and cliques on n vertices have treewidth $n - 1$. The main advantage of treewidth in algorithm design arises when we are designing parameterized algorithms for NP-hard problems by considering it as the parameter of the problem. Many families of commonly-studied graphs, such as trees, cacti, series-parallel graphs, outer-planar graphs, and control-flow graphs of structured programs have bounded treewidth [20, 19, 18]. This allows for the development of efficient dynamic programming techniques using the tree decomposition of the graph [20].

TREEWIDTH OF MOLECULES. Extending this idea, chemists and biologists have also explored the treewidth of various important classes of molecules [21, 22]. In our own experimental results (Section 5), we observe that a significant fraction of molecules in the PubChem repository [23] (nearly 99.9%) have a treewidth of at most 10. Even large proteins from the Protein Data Bank [24] are observed to have a treewidth of 3. Since a significant fraction of molecules has bounded treewidth, exploring and designing treewidth-based parameterized algorithms for computational problems in chemistry and biology is a natural step. In fact, the same has been done in several works in the literature [25, 13, 26, 27, 28]. We continue on this line of research by developing faster treewidth-based approximations for computing the Wiener index.

OUR CONTRIBUTION. In this paper, we introduce a novel randomized algorithm that approximates the Wiener index of a graph using its tree decomposition. The unique aspect of our algorithm is the incorporation of centroid decomposition into the tree decomposition, which significantly enhances efficiency in answering distance queries within the graph. This method is subsequently integrated with an established randomized algorithm to approximate the Wiener index.

Both theoretical analysis and experimental results demonstrate that our algorithm outperforms current methods in calculating the Wiener index for molecular graphs, which are commonly encountered in computational chemistry and biology. The observed speed improvements with our methodology indicates that the techniques presented herein have promising applications in the fields of computational chemistry and biology.

Table 1 compares the runtime complexity of our algorithm with previous methods. Here, n

Algorithm	Time Complexity	Type	Ref.
Floyd-Warshall	$O(n^3)$	Exact	[29]
Ortho. Range Searching	$O(n \cdot \log^{k-1} n)$	Exact	[13]
DP on Tree Decomp.	$O(n^2 \cdot k^2)$	Exact	[30]
Breadth-first Search (BFS)	$O(n^2 \cdot k)$	Exact	[31]
Classical Approximation	$O(n^{3/2} \cdot k/\epsilon^2)$	Randomized Approx.	[16]
Our Algorithm	$O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot \log n \cdot (\log\log n + k)/\epsilon^2)$	Randomized Approx.	Sec. 4

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.

is the number of vertices in the graph, k is the treewidth, and ϵ is the error in the approximation. We refer to Section 5 for a detailed experimental evaluation of our algorithm on datasets from PubChem [23] and the Protein Data Bank [24].

ORGANIZATION. In Section 2, we formalize the problem and introduce the parameters. In Section 3, we give a brief overview of the previous algorithms used to compute the Wiener Index of a graph. Section 4 presents our algorithm. Finally, in Section 5, we present the experimental results.

2 Preliminaries

In this section, we first define the Wiener Index along with its related terminology. We then define tree decompositions and treewidth. Finally, we provide a short overview of the concept of centroid decomposition. For the purpose of our discussion in this section, we have only included the essential information. For a more thorough exposition of these topics, we refer the reader to [18].

WIENER INDEX. The Wiener Index of a graph $G = (V, E)$ is defined as the all-pairs sum of shortest paths in the graph. Formally,

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

Remark 1. *We will assume, without loss of generality, that our graphs are connected, unweighted, and undirected. It is pertinent to mention that in the context of molecular graphs, all types of covalent bonds—be they single, double, or triple—are represented as a single undirected edge in the corresponding graph. For a disconnected graph, the Wiener index is simply $+\infty$. However, in some applications, the Wiener index of a disconnected graph is defined as the sum of Wiener indices of its connected components. In such cases, each connected component can be processed separately.*

Remark 2. *We will use the definition of the Wiener Index without the* $1/2$ *factor multiplied before taking the summation, which mainly occurs in undirected graphs as $d(u, v) = d(v, u)$.*

TREE DECOMPOSITIONS. A *tree decomposition* of a given graph $G = (V, E_G)$ is a tree $T = (\mathcal{B}, E_T)$ such that it satisfies the following conditions:

- Every node $b \in \mathcal{B}$ of T, which is called a *bag*, contains a subset of vertices $V_b \subseteq V$.
- 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 appears in at least one bag.
- For every edge in the original graph G, there exists a bag that contains both endpoints of the edge. More formally, for every $e = (u, v) \in E_G$, there is a bag $b \in \mathcal{B}$, s.t. $u, v \in V_b$
- Every vertex $v \in V$ appears in a connected subtree of T, meaning that the set $\mathcal{B}_v = \{b \in \mathcal{B} \mid v \in V_b\}$ forms a connected subgraph of T.

Remark 3. *An equivalent statement of the last condition above is that for every three bags $b_1, b_2, b_3 \in \mathcal{B}$, if b_3 is on the unique path from b_1 to b_2 in T, then $V_{b_1} \cap V_{b_2} \subseteq V_{b_3}$.*

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

Figure 1 showcases an illustration containing two distinct tree decompositions of a graph G, each having a different width. Since only forests have a treewidth of 1, the tree decomposition on the right is optimal and $\mathrm{tw}(G) = 2$.

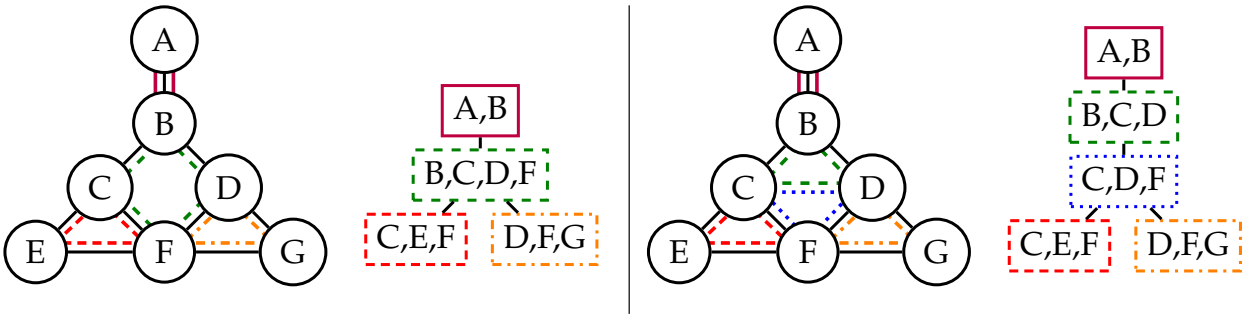


Figure 1: A graph G and two tree decompositions of G of width 3 (left) and 2 (right).

Intuitively speaking, treewidth measures the structural likeness of a graph to a tree. Specifically, the smaller the treewidth of a graph, the more tree-like it appears. A graph of treewidth k can be decomposed into small parts (bags), each of size at most $k+1$, which are connected to each other in a tree-like manner T.

Treewidth is a parameter indicating graph sparsity, in that it provides an upper bound on the number of edges. Specifically, in a graph with n vertices and treewidth k, the number of edges is $O(k \cdot n)$. More precisely, the number of edges is less than or equal to $n \cdot k - k \cdot (k + 1/2)$ [32].

CENTROID DECOMPOSITIONS. Consider a tree $T = (V_T, E_T)$ with n vertices. We define a *centroid node* of T as a node whose removal breaks the tree down into several subtrees such that no resulting subtree has a size of more than $n/2$. In other words, a centroid is a 1/2 separator of T. It is well-known that every tree has at least one centroid node, which can be obtained easily by a dynamic programming on its BFS tree.

A *centroid decomposition* of T is another tree T' on the same set of vertices as T, recursively defined as follows:

- When $|V_T| = 1$, we simply have $T' = T$.
- For a more complex tree, we first identify a centroid node r of T, then position this node as the root of T'.
- Once we have selected a centroid node r and removed it from T, we end up separating the original tree into several connected subtrees. Let us denote these as $T_1, T_2, \ldots, T_m$. For each subtree T_i, we find a centroid decomposition T_i' and add the root of T_i as a child of r.

As an example, Figure 2 shows the steps of computing a centroid decomposition. Each color corresponds to a distinct layer of the centroid decomposition, with the node representing the centroid of the similarly colored dotted subtree. In this illustration, node 4 is identified as the centroid of the initial tree. Following the removal of node 4, nodes 2, 7 and 12 are selected as the centroids of each resulting subtree. Subsequent centroids are determined in a recursive manner.

The height of a centroid decomposition is bounded by $O(\log n)$. This is because with every new layer added to the centroid decomposition, a connected component splits into several

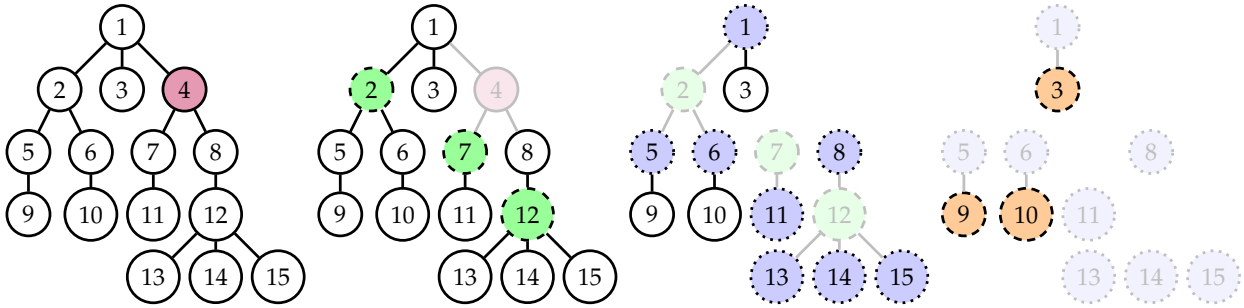


Figure 2: A graph G and the steps of building its centroid decomposition. Each step highlights the centroid vertex of each of the current components of the graph.

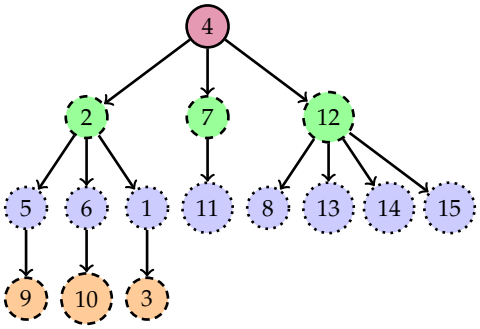


Figure 3: Resulting centroid decomposition of G.

parts, each no larger than 1/2 the size of the original component. Consequently, we can append at most $O(\log n)$ layers to the centroid decomposition, thus limiting its height to $O(\log n)$. Additionally, centroid decompositions satisfy the following useful lemma:

Lemma 1. *Let $u, v \in V_T$ be two vertices of the original tree T and l be their lowest common ancestor in the centroid decomposition T'. The unique path connecting u and v in T must visit l.*

Proof. We will prove this lemma through induction based on the size of tree. If the tree size is at most 3, the lemma holds trivially.

Now will assume that the lemma holds for all trees with a size less than n, let's consider a general tree of size n. In the first step, we identify a centroid node of T, denoted as c. The node c fragments T into several connected components. For any arbitrary vertices $u, v \in V_T$, if they exist in the same connected component, let's say T_i, then in the corresponding centroid decomposition T', they will appear in T'_i as per the definition of centroid decomposition. According to the inductive hypothesis, their path must cross their lowest common ancestor in T'_i.

In case they belong to different connected components, say T'_i and T'_j, any path originating from T'_i and terminating at T'_j must traverse the node c. This is because c separates T_i and T_j. In this scenario, their lowest common ancestor would be the root c, as the remaining nodes on the path from u to v are either in T_i or T_j, and hence, cannot serve as the common ancestor. Therefore, the lemma also holds in this case, hence completing the proof. □

3 Existing Approaches

In this section, we provide an overview of previously-known algorithms for computing the Wiener index of a graph. Of course, the most trivial approach is to compute all-pairs distances using the Floyd-Warshall algorithm and then simply sum up the distances. This leads to cubic

runtime of $O(n^3)$, which is not desirable for large molecules. We thus introduce more efficient approaches from the literature that utilize tree decompositions to improve this runtime using parameterization, approximation and randomization. The techniques discussed in this section are incorporated into our own algorithm, which is explained in Section 4.

3.1 Orthogonal Range Searching on Tree Decompositions

We briefly mention the key ideas of the algorithm for computing the Wiener index based on orthogonal range searching on tree decomposition from [13, Section 3].

The authors presented a divide and conquer algorithm that utilizes tree decomposition to compute the Wiener index in $O(n \cdot \log^{k-1} n)$ time. Note that this time is not FPT, since the dependence on the treewidth appears in the exponent. The main idea of the algorithm is to find small separators in the graph and then recursively compute the Wiener index of the subgraphs induced by the removing the separators. One of the algorithm's crucial components is using orthogonal range searching to handle and represent distances in the graph efficiently.

The basic process of performing the divide and conquer task is that in each iteration, we try to divide the vertices into two balanced subsets. We denote these subsets by A and B and their intersection is called a "portal" and denoted as S. Formally, every path from $A \setminus B$ to $B \setminus A$ should intersect S. We can recursively compute the Wiener indices of A and B. Then, if S is small, we can find the distances from every vertex in S to every vertex in A and B. Finally, for every $b \in B \setminus A$ and $a \in A \setminus B$, the path from b to a has to intersect S. Thus, we can find the sum of distances from b to all vertices in A by an orthogonal range query of dimension $|S|$. Hence, we need to ensure that (i) our portal $|S|$ is small, and (ii) removing it breaks the graph into connected components with no more than $n/2 + O(1)$ vertices.

Assuming we have bounded treewidth, we rely on our tree decomposition to find small cuts that break the graph into parts that are each no larger than $n/2 + O(1)$. A well-known property of the tree decompositions is the following: Let $e = (b_1, b_2)$ be an edge in the tree decomposition T of the graph G. Removing the edge e breaks T into two connected components T_1 and T_2, containing b_1 and b_2 respectively. Let $S = V_{b_1} \cap V_{b_2}$. The set S is a portal (cut) in G that separates $A := V_{T_1} = \bigcup_{b \in T_1} V_b$ and $B := V_{T_2} = \bigcup_{b \in T_2} V_b$. Note that we can also choose any bag V_b as our set S since replacing a bag with two copies of itself keeps the tree decomposition valid.

Ideally, we would like each resulting connected component after removing S to be no larger than half the size of our original graph, so that our recursion depth is logarithmic. In order to make the two vertex sets A and B balanced, we can use a centroid decomposition. We first find a centroid bag c. We then remove V_c from the graph G and let A be the largest connected component in the resulting graph. Similarly, we let B be the rest of the graph and add V_c to both A and B. We first compute the Wiener index of A and each connected component of B. Now, we can use the orthogonal range searching technique to calculate the distance sum. Let $S = \{s_1, \ldots, s_k\}$. For every vertex in $b \in B$ and $s_i \in S$, we want to find the number of vertices $a \in A$ such that the shortest path from b to a goes through the vertex s_i. This is where the orthogonal range search takes place. Since we need to compare the distance difference for every vertex in S, the dimension is of size k, and therefore, the runtime has the factor $O(\log^{k-1} n)$. We have to also be mindful of double-counting. See [13] for details. The total runtime of this approach is $O(n \cdot \log^{k-1} n)$.

3.2 Dynamic Programming Algorithm based on Tree Decomposition

In this section, we highlight a well-established dynamic programming algorithm, which utilizes tree decomposition for calculating the Wiener index. This algorithm, originally detailed in [30], serves as a foundational reference for our exposition. We provide a thorough analysis of its steps, focusing on the key components that are integral to understanding and adapting

the methodology for our algorithm's context. We denote the initial graph as $G = (V, E)$ and the tree decomposition of minimum width as $T(\mathcal{B}, E_T)$, with $w(T) = \text{tw}(G)$.

The algorithm is as follows:

1. **Initialize Variables and Data Structures:**
   - Let the initial graph be $G(V, E)$.
   - Obtain the tree decomposition of minimum width, denoted as $T(\mathcal{B}, E_T)$, where $w(T) = \text{tw}(G)$.
   - Create a stack for reverse order processing.
2. **Preprocessing Phase:**
   - Select an arbitrary leaf bag, b, from the set $\mathcal{B}$.
   - Apply the Floyd-Warshall Algorithm within the bag b to compute the all-pair shortest distances treating the vertices within this bag as an entire vertex set.
   - Mark the bag b as processed and remove it from $\mathcal{B}$.
   - Push the bag b into the stack for reverse processing later.
   - Repeat the above steps until all bags have been processed.
   - Process each bag again in the reverse order (by popping from the stack) and execute the Floyd-Warshall algorithm to update distance values considering vertices from other bags.
3. **Updating Distance Values:**
   - For every leaf bag b and vertices $u, v \in b$, check if the shortest path exists within bag b or traverses through other bags.
   - If the shortest path is within the bag, add a direct edge between u and v with weight equal to the shortest path distance.
4. **Computing Final Distances After Preprocessing:**
   - After the preprocessing phase is complete, iterate through all the vertices in the graph.
   - For each vertex v_i, initiate a Breadth-First Search (BFS) from a bag containing v_i and spread out across the tree decomposition T.
   - Each time a bag b is encountered during BFS, compute $\text{dis}(v_i, v_j)$ for every vertex $v_j \in b$ by comparing it with $\text{dis}(v_i, v_k) + \text{dis}(v_k, v_j)$ for all $v_k \in V_b$, and assign $\text{dis}(v_i, v_j)$ as the minimum value amongst these.

Detailed Explanation of Algorithmic Steps. The first step is to preprocess to achieve the all-pair shortest distances within each bag. We initially select an arbitrary leaf bag, b from the set $\mathcal{B}$ and apply the Floyd-Warshall algorithm to compute in-bag distances, treating the vertices within this bag as the entire vertex set. Subsequently, the bag b is marked as processed and removed. This procedure continues until all bags have been handled. Ultimately, we execute the Floyd-Warshall algorithm once more within each bag, but this time, in the reversed order of the previous sequence in which the bags were chosen.

The fundamental concept deriving from this preprocessing step is that every path from vertex u to vertex v, represented as $P = \langle u, ..., v \rangle$, corresponds to a path in T. This path begins from a bag containing u and terminates at a bag containing v, encountering the intermediate vertices in the same sequence as the original path. This fact can be straightforwardly established via the definition of tree decomposition and inductive reasoning.

Hence, for every leaf bag b and vertices $u, v \in b$, the shortest path must either exist within bag b itself or traverse through other bags before eventually returning to b. If the former scenario is valid, we add a direct edge connecting vertices u and v, assigning it a weight equal to the calculated shortest path distance. This addition process is effectively an update to the distance value in the context of the Floyd-Warshall Algorithm.

When we temporarily remove such a leaf bag, we effectively eliminate the vertices that only appear in that specific leaf bag. Suppose there is an arbitrary vertex s of this kind that appears later in the shortest path from vertex r to t. This path must follow the form $\langle r, ..., s_1, s, s_2, ..., t \rangle$, where s_1, s_2 are two neighbors of s. Here, the path $\langle s_1, s, s_2 \rangle$ represents the shortest distance path from s_1 to s_2, and we handle this and update $\text{dis}(s_1, s_2)$ when we process the leaf bag. Consequently, removing the leaf bag doesn't impact the correctness of the following steps.

We perform the Floyd-Warshall Algorithm a second time because, when initially processing a bag, we might not know the shortest distance between two vertices as they may intersect vertices in another bag. However, when we process for the second time in reverse order, we update the corresponding distance values as necessary.

Once we complete the preprocessing phase, we will have all the distance values for vertices that reside within a common bag. Subsequently, we iterate through all the vertices. For each vertex v_i, we initiate a Breadth-First Search (BFS) from a bag and spread out across T. Each time we encounter a bag b, we compute $\text{dis}(v_i, v_j)$ for every vertex $v_j \in b$. This computation is achieved by comparing it with $\text{dis}(v_i, v_k) + \text{dis}(v_k, v_j)$ for all $v_k \in b$, and we assign $\text{dis}(v_i, v_j)$ as the minimum value amongst these.

PROOF OF CORRECTNESS. To prove the correctness of this approach, we need to demonstrate that for every edge $\text{edge}(b_i, b_j) \in E_T$, the intersection of b_i and b_j is nonempty, i.e., $b_i \cap b_j \neq \emptyset$. Suppose, for the sake of contradiction, that there exists an edge $\text{edge}(b_i, b_j)$ where $b_i \cap b_j = \emptyset$. Upon removing this edge, T would be divided into two connected components that share no common vertex. If they did share a common vertex, based on the definition of tree decomposition, this vertex would have to appear in a connected component of T and be present in both b_i and b_j, as $\text{edge}(b_i, b_j)$ is the sole edge connecting the two components. This contradicts our initial assumption that our graph G is connected. Therefore, we can conclude that $b_i \cap b_j \neq \emptyset$.

Let b_u be a bag containing u. By above argument, we can prove that for every neighboring bag of b_u, a vertex v must exist in b_u and its neighboring bag, this can be used to calculate $\text{dis}(u, v)$. Through induction, we can correctly compute all other distance values, thereby validating the correctness of the algorithm.

RUNTIME ANALYSIS. Given that for any graph with n vertices, a tree decomposition exists with $O(n)$ bags of minimum width, we can conclude that for an n-vertex graph G with a treewidth $\text{tw}(G) = k$, the preprocessing step can be executed in $O(n \cdot k^3)$ time, which is in linear FPT. The computation phase can be performed in $O(n^2 \cdot k^2)$. Therefore, the overall runtime is bounded by $O(n^2 \cdot k^2)$. While an $O(n^2)$ runtime is not yet satisfactory, we do have a linear FPT preprocessing stage, which will be a crucial step for our own algorithm in Section 4.

3.3 Breadth-first Search (BFS)

We revisit the breadth-first search (BFS) method for computing the Wiener index of a graph. It is a well-established fact that BFS requires $O(n + m)$ time to calculate the shortest path distances from a given source vertex to all other vertices in a graph, with n and m representing the number of vertices and edges, respectively. Consequently, we can execute BFS from each vertex in the graph and sum the distances to compute the Wiener index. This method requires $O(n^2 + n \cdot m)$ time, which, in the worst case, is cubic. However, for graphs with bounded treewidth, the number of edges m is bounded by $O(n \cdot k)$, where k denotes the treewidth, as discussed in Section 2. Hence, the runtime for this approach is $O(n^2 \cdot k)$.

3.4 An Approximation Algorithm

In this section, we briefly explain the key points of the approximation algorithm used to compute the Wiener index as presented in [16]. The algorithm randomly selects $\Theta(\sqrt{n}/\epsilon^2)$ pairs of distance queries, calculates the average distances among these pairs and uses this value as an approximation for the average value between all vertex pairs. The Wiener index is then scaled by n^2.

The algorithm is as follows:

1. **Initialization of Variables and Data Structures:**
   - Let the initial graph be $G(V, E)$.
   - Let ϵ be the desired approximation ratio.
2. **Computation of Wiener Index:**
   - Select uniformly and at random $\Theta(\frac{\sqrt{n}}{\epsilon^2})$ pairs of vertices in V. Let the set of these pairs be S.
   - For every pair $(v_1, v_2) \in S$, compute their distance in G using a Breadth-first search, in $O(n \cdot k)$ time.
   - Compute the average distance between the pairs and return the result multiplied by n^2.

The running time of the algorithm is the total time required to compute all distances between pairs of vertices in the set S. This equates to $\Theta\left(\frac{\sqrt{n}}{\epsilon^2}\right)$ instances of an $O(n \cdot k)$ algorithm, resulting in a final running time of $O\left(\frac{n^{3/2} \cdot k}{\epsilon^2}\right)$.

The authors of [16] proved that their method yields a $(1+\epsilon)$-approximation with a probability of at least $\frac{2}{3}$. Naturally, repeating the algorithm increases the success probability arbitrarily close to 1. A significant feature of their technique is the requirement of only $\Theta\left(\frac{\sqrt{n}}{\epsilon^2}\right)$ distance queries. However, they address general graphs without parameterizing by treewidth, thereby not leveraging tree decomposition for distance queries. Instead, each query is answered using Breadth-First Search (BFS). This leaves room for further optimization for graphs that have small treewidth.

4 Our Algorithm

We are now prepared to present our algorithm, which integrates concepts from the algorithms discussed in the previous section and enhances distance query speed through centroid decomposition.

First, we provide an overview of the algorithm's steps, followed by a detailed explanation of each step. The algorithm is outlined as follows:

1. Compute all-pairs distances for each bag in the tree decomposition T of G. Specifically, for every pair (u, v) of vertices that appear in the same bag, add a direct shortcut edge from u to v whose weight is the distance from u to v.
2. Compute the centroid decomposition T' of T.
3. For every vertex $v \in V$ of the original graph G, consider the highest bag b in T' that contains v. Compute distance values from v to all vertices appearing in the descendants of b in T'.

4. Select $\Theta(\sqrt{n}/\epsilon^2)$ pairs of vertices uniformly at random and compute their distance values based on the precomputed distances in the previous steps. Return the average distance multiplied by n^2 as the result.

DETAILED EXPLANATION OF ALGORITHMIC STEPS. The first step is to compute all-pairs shortest paths within each bag using the preprocessing step of the algorithm described in Section 3.2.

For the next step, we compute a centroid decomposition T' of our tree decomposition $T = (\mathcal{B}, E_T)$. We need to find the centroid root for every sub-tree for centroid decomposition. Suppose T has n bags and for each bag $b \in \mathcal{B}$, we calculate the size for the subtree rooted at b in the following procedure.

- If b is a leaf bag, then $\text{size}(b) = 1$
- Otherwise, $\text{size}(b) = 1 + \sum \text{size}(c_i)$, where c_i are children of b.

In order to check whether b is a centroid bag, we check that (i) the size of all the subtrees rooted at its children are less than $n/2$, and (ii) the size of the subtree rooted at b is at least $n/2$. The second condition is checking whether, after removing b, the size of the component containing its ancestors is at most $n/2$. The running time for finding the centroid node is linear in the subtree size by our procedure because we can run a BFS to get to the leaf node and calculate the size backward. Suppose every layer has $b_1, ..., b_i$ nodes and each corresponds to the centroid of $T_1, ..., T_i$ subtree. All those subtrees are pairwise disjoint based on the property of centroid decomposition, and based on previous analysis, we have the whole running time for finding b_1 to b_i is $\sum_{j=1}^{i} O(\text{size}(T_i)) = O(n)$. Therefore, each layer takes $O(n)$ time to construct, and as the centroid decomposition can have at most $O(\log n)$ layers, the overall running time is $O(n \cdot \log n)$.

The next step is to compute distances based on the centroid decomposition. For every bag b and vertex $v \in V_b$, we compute all the distance values from v to the vertices appearing in any of the bags in the subtree of the centroid decomposition T' rooted at b.

For example, in Figure 3, we calculate the distance between vertices in bag 12 and all the vertices that appear in any of the bags $8, 13, 14$ and 15.

Consider the way our centroid decomposition T' is constructed. At each step, we find a centroid bag c, put it as the root of T' and remove it from the tree decomposition T. This breaks T into several connected components $T_1, T_2, \ldots, T_m$. Let $V(T_i)$ be the set of vertices in G that appear in any of the bags in T_i. Since T is a tree decomposition, V_c is a cut in G separating the $V(T_i)$'s. In other words, if $u, v \in V(T_i) \cup V_c$ then the shortest path from u to v never exits $V(T_i) \cup V_c$. Moreover, since we have added shortcut edges to our graph, if $u, v \in V(T_i)$, then there is a shortest path from u to v that never exits $V(T_i)$.

Based on observations above, we can compute the distance in a bottom-up fashion on T'. Suppose we want to find the distance from $v \in V_c$ to a vertex $u \in V(T_i) \cup V_c$. If $u \in V_c$, then we already have a shortcut edge that gives us the distance. Otherwise, let b_i be the bag in T_i that is connected to c in T. The path from v to u has to intersect $V_{b_i} \cap V_c$. So, for every w in $V_{b_i} \cap V_c$, we compute $\text{dis}(v, w) + \text{dis}(w, u)$ and take the minimum. Here, $\text{dis}(v, w)$ is known since v and w appear in the same bag c and $\text{dis}(w, u)$ can be found by a recursive distance query (see below) on T_i since the shortest path from w to u does not leave T_i.

In the final step of our algorithm, we perform the randomized algorithm discussed in Section 3.4. More precisely, we randomly select $\Theta(\sqrt{n}/\epsilon^2)$ different pairs of vertices and query their distance values with the help of our centroid decomposition. Note that the same query mechanism is also used in our preprocessing above:

Lemma 2. *Given a pair (u, v) of vertices, the running time to answer a distance query from u to v is $O(\log \log n + k)$, where n is the number of vertices in G and k is the width of the tree decomposition.*

Proof. Let us consider any pair of vertices (u, v). The initial step is to identify the highest bags b_u, b_v in the centroid decomposition T' containing them. We can precompute and keep this information.

Based on the cut property of tree decompositions, it is evident that any path from vertex u to vertex v must traverse through the vertices that are present in the bags lying on the path between the highest bags containing u and v, denoted as b_u and b_v respectively.

Referring to Lemma 1, this path will pass through the lowest common ancestor of b_u and b_v, which we will denote by $\text{lca}(b_u, b_v)$ within T'.

After determining the distances from every vertex w in the bag $\text{lca}(b_u, b_v)$ to u and v, our objective is to calculate $dis(u, v)$. This can be computed through the formula $dis(u, v) = \min_{w \in \text{lca}(b_u, b_v)}(dis(u, w) + dis(w, v))$.

This computation can be completed in $O(k)$ time. For the remainder of the proof, we will demonstrate that it is possible to find $\text{lca}(b_u, b_v)$ in $O(\log \log n)$ time.

To find the lowest common ancestor efficiently, we will do further preprocessing on our centroid decomposition. For every bag b in the centroid decomposition, we define a vector called $\text{anc}(b)$, which holds the ancestor information of b. Formally, $\text{anc}(b)$ is defined as follows:

- $\text{anc}(b)[0]$ is set to the parent of the bag b, denoted as P_b. However, if b is the root bag in the centroid decomposition, $\text{anc}(b)[0]$ is set to -1.
- For $i > 0$, $\text{anc}(b)[i]$ is defined recursively as $\text{anc}(\text{anc}(b)[i-1])[i-1]$. This means that $\text{anc}(b)[i]$ stores the ancestor of bag b that is 2^i steps away.

We can construct the *anc* vector by using a dynamic programming approach. Initially, we compute the ancestor that is 2^0 steps away, then iteratively use the information about the 2^{i-1}-th ancestor to compute the 2^i-th ancestor. Given that the centroid decomposition has at most $O(\log n)$ levels, we only need to calculate $O(\log \log n)$ values for *anc*.

This construction process should be completed prior to executing the distance query. It is worth noting that the construction of *anc* vectors takes $O(n \cdot \log \log n)$ time.

We now focus on finding the lowest common ancestor of b_u and b_v, denoted as $\text{lca}(b_u, b_v)$. The first step involves ensuring that b_u and b_v are at the same height in the centroid decomposition. Let us assume that b_u is closer to the root than b_v, and let the difference in distances from b_u and b_v to the root bag be d. We can replace b_v with its d-th ancestor. Utilizing the *anc* information, this can be accomplished through binary search, which takes $O(\log d) = O(\log \log n)$ time since $d \leq \log n$. For the rest of the discussion, we will assume that b_u and b_v are at the same level, meaning they are equidistant from the root bag in T'.

The main idea of the algorithm lies in iterating through $\text{anc}(b_u)$ and $\text{anc}(b_v)$ until the first common ancestor at the same index is found. In other words, we need to find the minimum index i such that $\text{anc}(b_u)[i] = \text{anc}(b_v)[i]$. We can use a binary search to determine such i, hence finding the $\text{lca}(b_u, b_v)$. The running time for this step is bounded by $\log \log n$

Therefore, combining this with the time taken to iterate through every vertex in the bag, the total running time is $O(\log \log n + k)$. □

An additional aspect of consideration is that the aforementioned randomized algorithm has a 2/3 probability of achieving a $(1 + \epsilon)$-approximation result. Consequently, to reduce the probability of failure to $O(1/n)$, we can execute the algorithm $\Theta(\log n)$ times, selecting the median of the results as the final output. It follows from the standard techniques using Chernoff bounds and properties of median that the likelihood of not getting a result within a $(1 + \epsilon)$-approximation ratio is $O(1/n)$.

RUNTIME ANALYSIS. Let us examine the time it takes for the entire algorithm to run. It takes $O(n \cdot k^3)$ time during the preprocessing to get all-pairs distances in each bag. Additionally, it takes $O(n \cdot \log n)$ time to get the centroid decomposition of T, and $O(n \cdot \log n \cdot k^3)$ to calculate the distance from the bag to all other bags in its subtree in centroid decomposition T'. When

it comes to distance queries, we perform $\Theta(\sqrt{n} \cdot \log n/\epsilon^2)$ distance queries in total, and each query takes $O(\log\log n + k)$ time to calculate. Therefore, the distance query takes $O(\sqrt{n} \cdot \log n \cdot (\log\log n + k)/\epsilon^2)$ time. Finding the median is simple, as we only have $\Theta(\log n)$ amount of data to sort and grab the median, taking $O(\log n \cdot \log\log n)$ time, which is negligible compared to other parts. Our runtime is $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot \log n \cdot (\log\log n + k)/\epsilon^2)$, which is almost linear FPT treating both k and $1/\epsilon$ as parameters.

5 Experimental Results

In this section, we present our experimental results, comparing the runtimes of our algorithm with previous approaches. We have used the following datasets for our experiments:

- PubChem Dataset [23].
- Protein Data Bank (PDB) Dataset [24].

In recent studies [26], it has been observed that the chemical compounds in the PubChem database exhibit bounded treewidth. Notably, more than 99.9% of these compounds have a treewidth of less than 10, with very rare compounds having treewidth of around 20. In addition to this, we have observed that large protein molecules in the PDB dataset also have bounded treewidth of 3. These discoveries are intriguing and suggest a promising direction for future research in parameterized algorithms for chemical graph theory.

In Figure 4, we have presented a comparison of the performances of algorithms on well-known large protein molecules. We present a comparison of running time concerning the number of vertices in the molecule. This experimentation result serves as a proof of concept of the theory presented in the earlier section and supports our claim of using parameterized algorithms for proteins and molecules arising in computational chemistry and biology.

Note that we have set the cutoff time to be 1000 seconds in the experimentation.

In Figure 5, we showcase a comparative analysis of algorithmic performances on molecules derived from the handpicked important data sources in PubChem database. This comparison evaluates the running time relative to the number of vertices of molecules. We present two versions of the data: the original raw dataset and a smoothed counterpart. A near-uniform performance across all algorithms was observed for molecules on the smaller end of the spectrum. The timings are captured in milliseconds, so they are susceptible to minor fluctuations. To enhance clarity and mitigate noise, we employ a rolling median on vertex size, using a window of size 5. Furthermore, we focus on molecules with vertex counts surpassing 300 to keep our visualizations pertinent and legible.

In Figure 6, we provide the distribution of treewidth for the PubChem datasets we are using. Most molecules have a treewidth less than 10, with only a small fraction having a treewidth around 15. This observation is consistent with the findings reported in [26].

To ensure accurate comparisons, we have selected different values for epsilon based on the dataset source. Specifically, we have set epsilon to 0.3 for the Protein Data Bank dataset and 0.1 for the PubChem dataset. Additionally, timing measurements are reported in milliseconds for small molecules and seconds for large molecules, with the median time calculated across all observations.

Based on our experimental results, it is apparent that our algorithm exhibits superior performance for large molecules. Moreover, compared to alternative algorithms, its performance remains competitive when applied to small molecules. Specifically, the orthogonal range searching is the only algorithm whose runtime is competitive with our algorithm. It is worth noting that while the orthogonal range searching operates as an exact algorithm, it does not offer fixed-parameter tractable (FPT) time. Contrarily, our methodology utilizes a randomized approximation approach while maintaining FPT time.

EXPERIMENTATION SPECIFICS. We implemented above mentioned algorithms in C++. For the preprocessing of molecular data, we integrated certain Python libraries, notably pysmiles [33] and RDKit [34]. Our initial step involved acquiring molecular data from the PDB (Protein Data Bank) and PubChem databases. Subsequently, we employed the mentioned Python libraries to transform this molecular data into graphs utilizing NetworkX [35]. Later, we transitioned these graphs into the conventional .gr format, which is compatible with tree-decomposition solvers. For the computation of tree-decompositions of these graphs, we selected FlowCutter [36]. We conducted experiments on the system with the following configuration: Intel Core i5 machine with 8GB of RAM running MacOS Version 10.13.6.

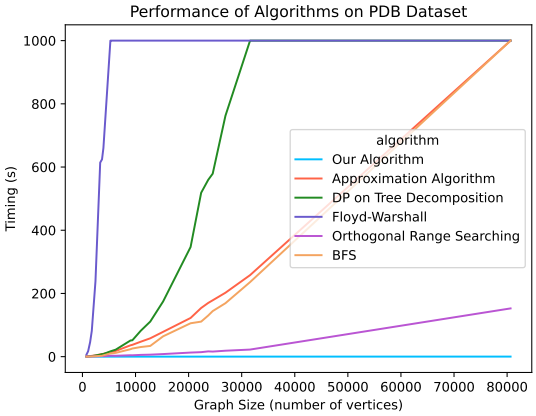
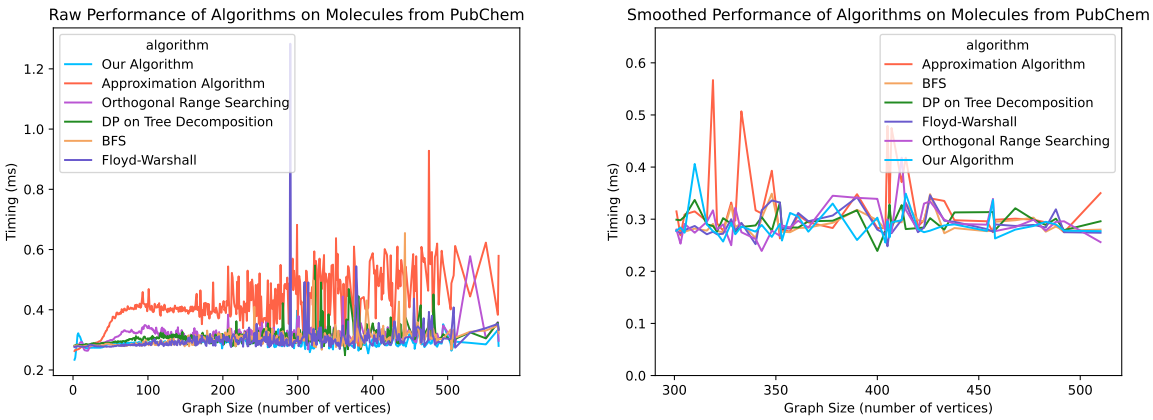


Figure 4: Performance on Protein Data Bank (PDB) Dataset with cuttoff time of 1000 seconds



(a) Algorithm timings vs. graph size (original data). (b) Smoothed algorithm timings vs. graph size.

Figure 5: Performance Comparison on PubChem Dataset: Timing vs. Number of Vertices. The left graph displays original data, while the right graph shows data with a rolling median applied for clarity.

6 Conclusion

We presented a novel randomized approximation algorithm for computing the Wiener index of a graph. Our algorithm is based on tree and centroid decomposition, achieving a runtime of $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot \log n \cdot (\log \log n + k)/\epsilon^2)$, which is a significant improvement over the existing algorithms. We have also provided experimental evidence supporting our claim on

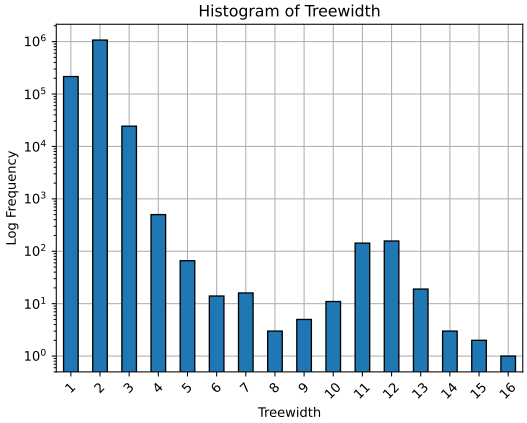


Figure 6: Treewidth Distribution of PubChem Datasets

molecules from well-known chemical data repositories like PubChem and the Protein Data Bank.

References

[1] N. Trinajstic, Chemical graph theory, Routledge, 2018.

[2] H. Wiener, Structural determination of paraffin boiling points, Journal of the American chemical society 69 (1) (1947) 17–20.

[3] J. Leszczynski, Handbook of computational chemistry, Vol. 3, Springer Science & Business Media, 2012.

[4] S. Wagner, H. Wang, Introduction to chemical graph theory, CRC Press, 2018.

[5] L. Xue, J. Bajorath, Molecular descriptors in chemoinformatics, computational combinatorial chemistry, and virtual screening, Combinatorial chemistry & high throughput screening 3 (5) (2000) 363–372.

[6] R. C. Entringer, D. E. Jackson, D. Snyder, Distance in graphs, Czechoslovak Mathematical Journal 26 (2) (1976) 283–296.

[7] L. Šoltés, Transmission in graphs: a bound and vertex removing, Mathematica Slovaca 41 (1) (1991) 11–16.

[8] D. Bonchev, Chemical graph theory: introduction and fundamentals, Vol. 1, CRC Press, 1991.

[9] A. G. Dossetter, E. J. Griffen, A. G. Leach, Matched molecular pair analysis in drug discovery, Drug Discovery Today 18 (15-16) (2013) 724–731.

[10] E. Estrada, E. Uriarte, Recent advances on the role of topological indices in drug discovery research, Current Medicinal Chemistry 8 (13) (2001) 1573–1588.

[11] C. Helma, Predictive toxicology, CRC Press, 2005.

[12] J. Xu, A. Hagler, Chemoinformatics and drug discovery, Molecules 7 (8) (2002) 566–600.

[13] S. Cabello, C. Knauer, Algorithms for graphs of bounded treewidth via orthogonal range searching, Computational Geometry 42 (9) (2009) 815–824.

[14] V. Chepoi, S. Klavžar, The wiener index and the szeged index of benzenoid systems in linear time, Journal of chemical information and computer sciences 37 (4) (1997) 752–755.

[15] A. A. Dobrynin, I. Gutman, S. Klavžar, P. Žigert, Wiener index of hexagonal systems, Acta Applicandae Mathematica 72 (2002) 247–294.

[16] O. Goldreich, D. Ron, Approximating average parameters of graphs, Random Structures & Algorithms 32 (4) (2008) 473–493.

[17] B. Mohar, T. Pisanski, How to compute the wiener index of a graph, Journal of mathematical chemistry 2 (3) (1988) 267–277.

[18] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, S. Saurabh, Parameterized algorithms, Springer, 2015.

[19] H. L. Bodlaender, et al., A tourist guide through treewidth (1992).

[20] H. L. Bodlaender, Dynamic programming on graphs with bounded treewidth, in: ICALP, Vol. 317, 1988, pp. 105–118.

[21] A. Yamaguchi, K. F. Aoki, H. Mamitsuka, Graph complexity of chemical compounds in biological pathways, Genome Informatics 14 (2003) 376–377.

[22] A. Yamaguchi, K. F. Aoki-Kinoshita, Chemical compound complexity in biological pathways, Quantitative Graph Theory: Mathematical Foundations and Applications (2014) 471.

[23] N. C. for Biotechnology Information, Pubchem database, `https://pubchem.ncbi.nlm.nih.gov`, [Online; accessed [29-June-2023]] (Accessed in [2023]).

[24] RCSB, Protein data bank, `https://www.rcsb.org`, [Online; accessed [29-June-2023]] (Accessed in [2023]]).

[25] T. Akutsu, H. Nagamochi, Comparison and enumeration of chemical graphs, Computational and structural biotechnology journal 5 (6) (2013) e201302004.

[26] G. K. Conrado, A. K. Goharshady, H. J. Motwani, S. Novozhilov, Parameterized algorithms for topological indices in chemistry, arXiv preprint arXiv:2303.13279 (2023).

[27] P. Wan, J. Tu, S. Zhang, B. Li, Computing the numbers of independent sets and matchings of all sizes for graphs with bounded treewidth, Applied Mathematics and Computation 332 (2018) 42–47.

[28] A. Yamaguchi, K. F. Aoki, H. Mamitsuka, Finding the maximum common subgraph of a partial k-tree and a graph with a polynomially bounded number of spanning trees, Information Processing Letters 92 (2) (2004) 57–63.

[29] R. W. Floyd, Algorithm 97: shortest path, Communications of the ACM 5 (6) (1962) 345.

[30] S. Chaudhuri, C. D. Zaroliagis, Shortest paths in digraphs of small treewidth. part i: Sequential algorithms, Algorithmica 27 (2000) 212–226.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms, MIT press, 2022.

[32] J. Nešetřil, P. O. De Mendez, Structural properties of sparse graphs, in: Building Bridges: Between Mathematics and Computer Science, Springer, 2008, pp. 369–426.

[33] pysmiles: A python library for parsing smiles strings, https://pypi.org/project/pysmiles/, accessed: [29-June-2023].

[34] Rdkit: Open-source cheminformatics, https://www.rdkit.org, accessed: [29-June-2023].

[35] A. Hagberg, P. Swart, D. S Chult, Exploring network structure, dynamics, and function using networkx, Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States) (2008).

[36] Flowcutter: Software for computing flow-based balanced graph cuts, https://github.com/kit-algo/flow-cutter-pace17, accessed: [29-June-2023].

Authors' addresses:

Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong

Giovanna K. Conrado: gkc@connect.ust.hk
Amir K. Goharshady: goharshady@cse.ust.hk
Pavel Hudec: phudec@connect.ust.hk
Pingjiang Li: pliav@connect.ust.hk
Harshit J. Motwani: csemotwani@ust.hk