Distances in Graphs

Anne Christiono, Albert Jiang, Kevin Xiao, Hua Wang

March 8, 2022

1 Introduction

Harry Wiener proposed the Wiener Index in 1947, marking the creation of one of the oldest molecular topological index. The importance of the calculation of this index is found in its correlation to crucial chemical properties involving density, viscosity, and boiling points–specifically of alkane molecules–among many other attributes [7]. Instead of spending large amounts of resources and time on laboratory experiments, scientists can refer to topological indices such as the Wiener Index. The classification of the Wiener Index served as one of the catalysts of the creation of thousands of additional topological indices, including the Estrada index and the Szeged index.

More specifically, the Wiener index is important in mathematical chemistry in the creation of regression models like quantitative structure-activity relationships (QSAR) and quantitative structure-property relationships (QSPR) [5]. These models are applied in a multitude of ways, including toxicity prediction [2], regulation, and risk assessment, and are not limited to strictly chemical use

The complexity of applicable graphs varies significantly, depending on whether or not double bonds or atom types are considered (beyond just a "node"). Formulas for the calculation of the Wiener Index of chemical compounds like fullerene molecules [4] have already been studied. Furthermore, there exist algorithms of varying complexity for the calculation of general graphs and trees. This paper is designed to expand upon certain special graphs to efficiently calculate the Wiener Index.

Extremal problems, or extremal graph theory, studies the circumstances that yields a minimum or maximum amount [3]. In this paper, we will also be examining the properties of a graph with N vertices that yields the minimum and maximum Wiener Index for all graphs with N vertices and proving why these types of graphs produce the minimum or maximum value. For example, a star graph produces the minimum Wiener Index. A star is characterized by having a central node with edges connecting it to every other vertex of the graph. In contrast, a path, which is a type of graph that can be described as a sequence of vertices that are adjacent to the those next to it, is the graph yielding the maximum Wiener Index [6]. A proof on why these two circumstances minimize or maximize the Wiener Index is explained in Section 6.

2 Preliminaries

Definition 2.1 (Graph)**.** A *graph* is a collection of vertices and edges, and a graph G is often expressed as $G = (V, E)$.

A *directed graph* is a graph in which each edge has a specific direction. A *weighted graph* is another type of graph, where a weight is assigned to each edge. In a *weighted directed graph*, each edge has both a weight and direction. Finally, *vertex-weighted graphs* have a weight assigned to each vertex.

Definition 2.2 (Tree)**.** A *tree* is a connected graph with no cycles. A tree with n vertices would have $n - 1$ edges.

Definition 2.3 (Adjacency Matrix)**.** If n is the number of vertices in graph G, the resulting *adjacency matrix* of G has size $n \times n$. If there exists edge (v, u), it will be marked, generally as a "1", in the v^{th} row and u^{th} column. Note that if the graph is undirected, the adjacency matrix is symmetrical.

A *transition matrix* is similar to an adjacency matrix, although specifically for weighted graphs. For every edge (v, u) in graph G, the v^{th} row and u^{th} column of the transition matrix contains the weight of (v, u).

Definition 2.4 (Distance)**.** The *distance* between two vertices v and u, often expressed as $d(v, u)$, is the length of the shortest path between them. If the graph is weighted, then the distance refers to the least weighted distance.

The notation $d_G(v)$ describes the sum of the distances of all nodes in graph G to a vertex v. Similarly, $d_{C_1}(v)$ denotes the sum of the distances of all nodes in component C_1 to vertex v.

Definition 2.5 (Component)**.** A *component* of a graph G is a maximal induced subgraph of G where there exists a path between any pair of vertices of the subgraph. In Figure 1, C_1 and C_2 are both components; vertices u_1^1 and u_1^2 are nodes in C_1 while u_2^1 is a node in C_2.

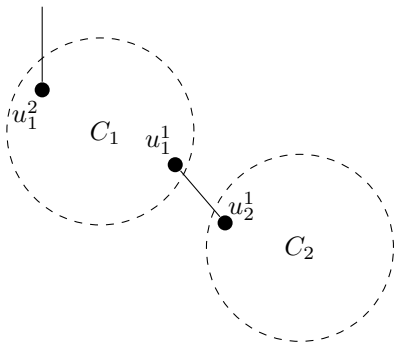

Figure 1: An example of components and cut-edges.

In a connected graph, a *cut-edge*, also known as a *bridge*, is an edge such that if it is removed, the graph becomes disconnected and it will increase the number of a graph's components. Notice that in a tree, every edge is a cut-edge. In Figure 1, edge $u_1^1u_2^1$ is an example of a cut-edge. The edge connected to u_1^2 is also a cut-edge connecting C_1 to another component not shown. The components of a graph after removing its cut-edges are often referred to as *pseudo-components*.

For an edge e in a (rooted) graph with a root node, $n_1(e)$ describes the number of vertices connected on the parent's side of the edge, including the parent node. Similarly, $n_2(e)$ describes the number of vertices on the child side of edge e, including the child node. For example, in edge $u_1^1u_2^1$ in Figure 1, node u_1^1 is the parent node and u_2^1 is the child node. $n_2(u_1^1u_2^1)$ is equal to the number of nodes in C_2 and $n_1(u_1^1u_2^1)$ is equal to every other node in the graph that can reach u_1^1 through a series of edges.

Definition 2.6 (Wiener Index)**.** The *Wiener Index* of a graph refers to the total sum of distances between all possible pairs of vertices in a graph.

The amount that an edge e would contribute to the total Wiener Index of a general graph is denoted as $W(e)$ and in an unweighted and undirected graph, $W(e) = n_1(e) \cdot n_2(e)$. This is potentially helpful when we consider the Wiener index of trees or "treelike" structures.

In a *general unweighted graph* G, the Wiener Index of G is described as

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

In the case of an *edge weighted graph*, the above equation would be accurate in computing the Wiener Index of the graph. For each pair of vertices (u, v), $d(u, v)$ is redefined as the least weighted distance between vertices u and v.

In a *general vertex-weighted graph* G, its Wiener Index, denoted as $VWW(G)$, would be

$$VWW(G) = \sum_{(u,v)} d((u,v)) \cdot w(u) \cdot w(v).$$

A *star* with is the type of tree that yields that minimum possible Wiener Index. It is demonstrated as a node in the "center" that has an edge connecting it to every other edge in the graph. Note that since this is a tree, there are no additional edges connecting any pair of vertices that does not contain the center node.

Meanwhile, a *path* is the type of tree that yields that maximum possible Wiener Index. It can be arranged in a sequence of vertices where each vertex is adjacent

(i.e. connected by an edge) to the vertex next to it.

A *pseudo-star* or *pseudo-path* is simply a star or path, respectively, where each vertex is actually a pseudo-component.

3 Algorithms for computing the Wiener index of general graphs

In this section, we provide pseudo-code for Floyd-Warshall, Matrix multiplication, and BFS algorithms.

3.1 Floyd-Warshall

The most straightforward way of computing the Wiener index of a graph is probably the Floyd-Warshall algorithm. For completeness, a complete pseudo-code is provided in Algorithm 1.

Algorithm 1: Using the Floyd-Warshall algorithm to calculate the shortest distance between all pairs of nodes in a weighted directed graph

Definitions: N – number of vertices
Input: $N \times N$ adjacency matrix $adj[][]$ containing weight of each edge
Result: Matrix of the shortest distance between every pair of vertices.

Label every vertex from 1 to N. Let $dist[][]$ be an $N \times N$ matrix of minimum distance between every pair of vertices. Initialize values in dist to be some infinitely large constant.

```
for each vertex v from 1 to N do
|  dist[v][v] = 0
end
for each edge (u, v) do
|  dist[u][v] = weight of (u, v) as listed in G
end
for k from 1 to N do
|  for i from 1 to N do
|  |  for j from 1 to N do
|  |  |  if dist[i][j] > dist[i][k] + dist[k][j] then
|  |  |  |  dist[i][j] = dist[i][k] + dist[k][j]
|  |  end
|  end
end
```

The Wiener Index of this graph can then be calculated by taking the sum of all values in the dist matrix.

Time Complexity Analysis: $O(N^3)$

As stated, Algorithm 1 may be applied to weighted and directed graphs. To apply this algorithm to vertex-weighted graphs, simply multiply the distance between two nodes by the weights of those two nodes each time we take the distance. For example, $dist[j][k]$ would be replaced with $dist[j][k] \cdot w[j] \cdot w[k]$ and $dist[j][i]$ would be replaced with $dist[j][i] \cdot w[j] \cdot w[i]$, where the array $w[]$ of length N stores the weights of all N vertices.

3.2 Matrix Multiplication

Muller's Matrix-Multiplication algorithm seems to improve upon the Floyd-Warshall algorithm, as it has better time complexity. The complete pseudo-code is shown in Algorithm 2.

Algorithm 2: Using Muller's Matrix-Multiplication algorithm to calculate all pairwise shortest distances in an unweighted graph.

Definitions: N – number of vertices,
Input: $N \times N$ adjacency matrix D
Result: Matrix of the shortest distance between every pair of vertices, which the Wiener Index can be easily calculated from.

```
for i from 1 to N do
    for j from 1 to N do
        if D[i, j] = 0 and i != j then
            D[i, j] = N;
    end
end
for ℓ from 1 to N-1 do
    for j from 1 to N do
        for i from 1 to N do
            if D[i, j] = ℓ then
                for W from 1 to N do
                    D[W, j] = min(D[W, j], D[W, i] + ℓ);
                end
        end
    end
end
```

Now, the Wiener Index is equal to the sum of all $D[i, j]$.
Time Complexity Analysis: $O(N^4)$ if we judge by the loops only. With amortized analysis, this algorithm is $O(N^3)$.

However, the Matrix-Multiplication algorithm cannot be easily adopted for weighted graphs.

Remark 1. Why can't this algorithm be used for weighted graphs? Hypothesis: The first loop of the main portion of the algorithm (i.e. for ℓ from 1 to N-1)

is looping through all possible *lengths* of paths; in an unweighted graph of N nodes, this is guaranteed to be in the range [0, N-1] (and since the only case where distance between two nodes is 0 is when the two nodes are equal, we don't need to worry about 0 in the loop). The fact that ℓ represents the length of the path is implied because (a) it's the letter ℓ (ℓ for length, maybe), and (b) in the third for-loop there is a specific check that D[i, j] = ℓ, which suggests that in each loop, we are specifically trying to minimize paths by using sub-paths of length ℓ.

3.3 BFS

The BFS algorithm appears to be the most efficient approach by far, illustrated in Algorithm 3.

Remark 2. As in the case of the Matrix-Multiplication algorithm, the BFS algorithm cannot be used to handle weighted graphs. This is a fundamental limitation of Breadth-First Search; since it processes nodes in order of breadth, or the number of edges that they are from the original node, BFS is unsuitable for weighted graphs (as when edges can have many different weights, the assumption that the shortest path is the path with the least number of edges is not necessarily true anymore). To mitigate this, we would be forced to use a shortest-paths algorithm suited for a weighted graph, such as the Floyd-Warshall Algorithm.

4 Computing the Wiener Index of Trees

There exist algorithms that efficiently compute the Wiener Index $W(T)$ for any tree T. In this section, we present one such algorithm that runs in $O(N)$ time, the lowest time complexity possible.

4.1 Preliminaries

At the root of the algorithm is the following relation:

Proposition 4.1.

$$W(T) = \sum_{e \in E(T)} weight(e) \cdot subtree(e) \cdot (N - subtree(e))$$

where $E(T)$ is the set of all edges in T, $weight(e)$ is the weight of the edge, $subtree(e)$ is the number of nodes in the subtree of either the starting or ending node on edge e, and N is the total number of nodes in the graph.

Proof. To prove this relation, we use a contribution counting argument: consider the total contribution of every edge e to W(T), and take their sum to find the

Algorithm 3: Using BFS to calculate distance matrix for the calculation of Wiener Index in an unweighted graph

Definitions: N – number of vertices, E – number of edges
Input: Adjacency list $G[n][]$ storing vertex and weight (1 or 0 for adjacent/non-adjacent), queue Q
Result: $N \times N$ Distance Matrix $M_d[][]$

Label all the vertices from 1 to n. Let $D[n]$ be a single array of the distance matrix, corresponding to the shortest distance from the chosen vertex v to every other vertex. Trivially, D[v]=0, and all other values of D can be set to some infinitely large constant

```
while Q is not empty do
    vertex c- first element of Q
    pop first element from Q
    for element k in G[c][] do
        tk- adjacent vertex from k;
        tw- adjacent weight from k;
        if D[tk] > D[c] + tw then
            D[tk] = D[c] + tw;
            push tk to end of Q
    end
end
```

The sum of all D[] values for all N vertices divided by 2 (due to overcounting) yields the wiener index

Chemical graphs of compounds like isobutane or propane are typically constrained to $E = N - 1$ due to their tree-like structure. In compounds such as pyrene that are connected by a hexagonal pattern, E is constrained to $c \times N$, where c is a small constant.

Time Complexity Analysis: $O(N \times E)$ for a general graph, $O(N^2)$ for typical chemical graphs

total $W(T)$. Firstly, observe that if e was removed from the tree, then the tree would now consist of two separate components. In other words, removing e would disconnect every path that starts in the first component and ends in the other, which means that edge e **must** lie on any path between nodes in these two components. But how many paths are there between the two components? If we let n_1 equal the number of nodes in one of the components and n_2 equal the number of nodes in the other component, then there are clearly $n_1 \cdot n_2$ paths (we can enumerate all paths by simply selecting one of the n_1 nodes to be the starting point of the path, and one of the n_2 nodes to be the ending point). Therefore, edge e lies on $n_1 \cdot n_2$ paths.

There are two final observations to make. Firstly, is that $n_1 + n_2 = N$, so $n_1 = N - n_2$ and the total number of paths that edge e lies on can be rewritten as $n_1(N - n_1)$. Secondly, observe that n_1 and n_2, by definition, must equal the sizes of the two subtrees of nodes at the end of edge e. Now, if we let $weight(e)$ = the weight of edge e, it is obvious that edge e's total contribution to $W(T)$ is $weight(e) \cdot subtree(e) \cdot (N - subtree(e))$. Then, finding the total $W(T)$ can be done by summing up the above expression for all edges $e \in E(T)$, which leads to the relation above. □

4.2 The algorithm

Now, we may conceptualize the algorithm. The algorithm has the following steps

1. Root the tree T at an arbitrary node R.
2. For each node, compute the size of that node's subtree.
3. For each edge, calculate the contribution of their unique edge to $W(T)$, and sum these.

Computing the size of each node's subtree is a well-known problem that can be done in $O(N)$ time by using recursion, where N is the size of the tree.

Algorithm 4: Computing the Wiener Index of a Tree in O(N) time

Definitions: *subsize*[] – the subtree sizes of each node, will be calculated during the algorithm

Input: *curr* – the root of the tree (can be any node), *par* – the parent of *curr*, which is set to -1 at the beginning of the algorithm

Result: W, the Wiener Index of the tree

void `dfs(`*curr, par*`)`**:**

```
set subsize[curr] to 1
for all nodes X adjacent to curr do
    if X = par then
        continue
    add subtree[i] to subtree[curr]
    add subtree[i] · (N − subtree[i]) to W
end
```

Time Complexity Analysis: The time complexity is O(N), since throughout the execution of the algorithm each node is visited exactly once, and each edge is traversed exactly once. There are O(N) nodes and O(N) edges, so the time complexity is equal to O(N) + O(N) = O(N).

4.3 Vertex (or edge) weighted trees

Also, we may modify the above approach to accommodate the vertex-weighted (and edge-weighted) case.

Remark 3. In the case where the edges are weighted, it suffices to additionally multiply each edge's contribution by its edge weight. This requires hardly any modification to the above algorithm; simply replace the line (add $subtree[i] \cdot (N - subtree[i])$ to W) with (add $edgeWeight \cdot subtree[i] \cdot (N - subtree[i])$ to W).

Likewise, in the case where vertices are weighted, only our calculation for *subtree*[] needs to change. In this scenario, each node represents a number of nodes equal to its node weight. Therefore, it suffices to change the line (set $subsize[curr]$ to 1) to (set $subsize[curr]$ to $nodeweight[curr]$).

5 Graphs with many cut-edges

In this section we present our study of graphs with many cut-edges.

5.1 Preliminaries

Let G be a graph with pseudo-components C_i ($1 \leq i \leq n$ and cut-edges e_j ($1 \leq j \leq n-1$). Let T be the underlying tree structure where v_i corresponds to C_i, with the weight $w(v_i) = |V(C_i)|$. For convenience, sometimes we use

$n_v(vu)$ (as opposed to $n_1(vu)$ or $n_2(vu)$)to denote the number of vertices in the component containing v after the edge vu is removed from the graph.

- Then the contributions from the cut-edges to $W(G)$ is exactly

$$VWW(T) = \sum_{j=1}^{n-1} n_1(e_j) \cdot n_2(e_j). \tag{1}$$

- The contribution to the Wiener index from each C_i is simply its own (weighted) Wiener index, for a total of

$$\sum_{i=1}^{n} W(C_i). \tag{2}$$

- The contribution to the Wiener index from edges in each C_i, as part of the distance between vertices from different C_i, C_j, can be computed through two separate approaches:
  - $d_{C_i}(u_i)$: This represents, for paths that start and end in different pseudo-components, the contributions of edges in the beginning or end pseudo-component. Therefore, this should be counted $n_{u_j}(u_iu_j))$ times for each of the cut-edges "adjacent to" C_i. Hence we have

$$\sum_{i=1}^{n} \sum_{s=1}^{k} d_{C_i}(u_i^s) \cdot \left(N - n_{u_i^s}(u_i^s u_j^{s'})\right). \tag{3}$$

    Note that $N = |V(G)|$ is the total number of vertices, and $N - n_{u_i^s}(u_i^s u_j^{s'})$ is equivalent to $\sum_{s'} n_{u_j^{s'}}(u_i^s u_j^{s'})$ over all s' such that $u_i^s u_j^{s'}$ is a cut edge.
  - $d_{C_i}(u_i^s u_i^t)$: This represents paths that go through multiple pseudo-components, and should be counted $n_{u_j^{s'}}(u_i^s u_j^{s'})) \cdot n_{u_\ell^{t'}}(u_i^t u_\ell^{t'}))$ times for each pair of cut-edges "adjacent to" C_i. Hence we have

$$\sum_{i=1}^{n} \sum_{1 \le s < t \le k} d_{C_i}(u_i^s u_i^t) \cdot \left(N - n_{u_i^s}(u_i^s u_j^{s'})\right) \cdot \left(N - n_{u_i^t}(u_i^t u_\ell^{t'}))\right). \tag{4}$$

Here the superscripts s (or s') are used to distinguish endpoints of different cut-edges in the same pseudo-component. See Figure 2 for an example.

Summing up (1), (2), (3) and (4), we have the Wiener index of G in the following theorem.

Theorem 5.1. *Given a graph G with pseudo-components C_i ($1 \le i \le n$ and cut-edges e_j ($1 \le j \le n-1$), T as the underlying tree structure (where v_i corresponds to C_i) with the weight $w(v_i) = |V(C_i)|$, the Wiener index of G can be computed through the following expression, within $\sum_{i=1}^{c} |C_i|^2$ time:*

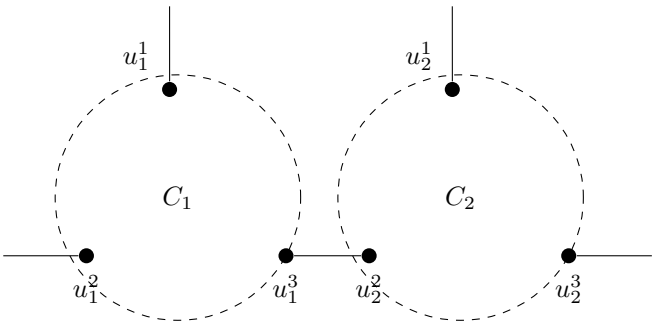

Figure 2: Examples of the pseudo-components, cut-edges, and their end vertices.

$$W(G) = \sum_{i=1}^{n} W(C_i) + VWW(T) + \sum_{i=1}^{n} \sum_{s=1}^{k} d_{C_i}(u_i^s) \cdot \left(N - n_{u_i^s}(u_i^s u_j^{s'})\right)$$
$$+ \sum_{i=1}^{n} \sum_{1 \le s < t \le k} d_{C_i}(u_i^s u_i^t) \cdot \left(N - n_{u_i^s}(u_i^s u_j^{s'})\right) \cdot \left(N - n_{u_i^t}(u_i^t u_\ell^{t'}))\right)$$

Proof. Again, this expression finds the total Wiener Index by summing up the contribution of 4 separate components to the total Wiener Index. Specifically:

- Term 1 is the contribution from paths that start and end in the same pseudo-component. Therefore, all of the other 3 terms are components of the contribution from paths that start and end in different pseudo-components.
- Term 2 is the contribution of the cut-edges to paths that start/end in different pseudo-components, since any such path must go through at least one cut-edge.
- Term 3 counts the contribution from edges on the path that are inside either the first or the last pseudo-component, on paths that start/end in different pseudo-components.
- Term 4 counts the contribution from edges on the path that are inside any intermediate (not first or last) pseudo-components, on paths that start/end in different pseudo-components.

Clearly, Term 1 is true. The sum of all shortest paths that start and end in the same pseudo-component is obviously the sum of the Wiener Indexes for each individual pseudo-component.

Term 2 is also true by definition. Remember that a cut-edge uv lies on $n_u(uv) * n_v(uv)$ paths, and that we can efficiently compute this quantity for all cut-edges in $O(N)$ time by condensing the pseudo-component graph into a

vertex-weighted tree, then finding the sum of the lengths of all paths in this vertex-weighted tree. As seen in the previous section, this means that the contribution of the cut-edges is equivalent to the $VWW(T)$, which is the vertex-weighted Wiener Index of the underlying tree structure of the original graph.

Term 3. Remember that this term is part of the contribution of shortest paths that start/end in different PC's. This term considers any pseudo-component C_i that is either the start or end of such a path. If we looked at the portion of the whole path that is within C_i, we would see the path start at some node n then immediately move to one of the exit-nodes u_i^s before leaving using a cut-edge. Moreover, we know that the number of paths that contain this path portion is equal to the number of nodes reachable from any cut-edge adjacent to C_i (since we fixed the starting/ending point at node n, we just need to enumerate the other endpoint of the path). The number of nodes reachable from any cut-edge adjacent to C_i can be expressed as $\sum_{s=1}^{k} N - n_{u_i^s}(u_i^s u_j^{s'})$. Then, summing this quantity up over all nodes in C_i, then over all pseudo-components themselves, results in Term 3.

Term 4. This term is also part of the contribution of shortest paths that start/end in difference PC's (like Term 3). But this term considers pseudo-components C_i that are in the middle of the path. If we looked at the portion of the whole path that is within C_i, we would see the path eventually enter C_i through one of the cut-edges adjacent to it, then move to another adjacent cut-edge and exit there. Obviously, we need to consider every pair of cut-edges. For a given pair of cut-edges

NOTE: New explanation is above, old one is below

First, note that (2) is obvious. To justify (1), simply remember that T is the underlying tree structure of the graph, where the pseudo-components of the original graph G are represented as individual nodes connected by cut-edges. Therefore, to find all cut-edges' contribution to $W(T)$, we can simply use the algorithm discussed in the previous section that finds the Wiener Index of a tree. This yields the expression directly above.

To justify (3), visualize the tree structure, T. Remember that nodes in T correspond to pseudo-components in G, while edges between nodes in T correspond to cut-edges in G. This means that in order to move between two pseudo-components C_i and C_j, we must cross one of the cut-edges adjacent to C_i (cut-edges are the only ways out of pseudo-components). Let's say that this cut-edge is between nodes u_i^s and $u_j^{s'}$, which themselves are in pseudo-components C_i and C_j respectively. Then, if we were to remove this cut-edge, the total number of nodes in the component of $u_j^{s'}$ would just be $n_{u_j^{s'}}(u_i^s u_j^{s'})$. Multiplying this quantity by the contribution of edges in C_i gives us $d_{C_i}(u_i^s) \cdot n_{u_j^{s'}}(u_i^s u_j^{s'})$. However, this only gives us the contribution of the edges for paths that leave C_i through the cut-edge $u_i^s u_j^{s'}$. In order to count the full contribution, we clearly just need to sum this quantity up over all cut-edges adjacent to the current pseudo-component C_i, and then sum that over all pseudo-components C_1 to C_n. This results in the above expression.

Similarly, to justify (4), consider all paths that pass *through* a pseudo-

component C_i. Clearly, any such path will enter through one of the cut-edges adjacent to C_i, and leave through another cut-edge adjacent to C_i. Let the two cut-edges be from nodes u_i^s to $u_j^{s'}$ and from u_i^t to $u_j^{t'}$, where u_i^s and u_i^t are the two endpoints in C_i. Then, it is obvious that there are $n_{u_j^{s'}}(u_i^s u_j^{s'})) \cdot n_{u_\ell^{t'}}(u_i^t u_\ell^{t'}))$ paths that pass through both of these cut-edges, from simply multiplying the number of nodes in the component of $u_j^{s'}$ with the number of nodes in the component of $u_j^{t'}$. And in a weighted graph, the total contribution will be equal to the number of paths passing through these two edges multiplied by the length of the shortest path between the two endpoints of the cut-edges located in C_i ($d_{C_i}(u_i^s u_i^t)$). This results in the 4th term.

□

5.2 The CEPC algorithm

We are now ready to present the complete CEPC Algorithm (shown in Algorithm 5) for computing the Wiener index $W(G)$, as stated in Theorem 5.1.

Algorithm 5: Computing the Wiener index for graphs with many cut-edges

Definitions: N- number of vertices, E- number of edges
Input: Adjacency list $adj[N][]$
Result: Wiener Index

Part 1 [Determining overall tree structure]: DFS from an arbitrary vertex (1 for simplicity). At every vertex in the DFS, mark the "time" that the vertex is visited in cur[v]. low[v] will store the minimum of the cur[] values of all of the neighbors of v except for its parent. It can be shown that if low[v]=cur[v] for some vertex v, then the the edge from v to its parent is a cut-edge. At the beginning of every instance of the DFS, add the vertex to a stack. At the end of every instance of the DFS, if low[v]=cur[v], pop the stack until v is found- the popped vertices will form the set of vertices in a pseudo-component. These vertices can be grouped and stored along with the cut-edges to form a weighted tree.

Part 2 [Calculations]: As in algorithm 4.3, DFS through the "new" tree and maintain a dp[] array that contains the number of vertices in a subtree- this accounts for equation (2). At each iteration of the DFS, apply the BFS algorithm on each vertex in the pseudo-section to calculate equation (1). The below algorithm uses a mark[] array to ensure that the BFS does not go out of bounds. In each iteration of the BFS, add the distance from the vertex to the vertices connecting adjacent cut-edges multiplied by their respective dp[] value. The parent cut-edge value is calculated from the expression $N - dp[v]$. The pairwise distances between vertices connecting adjacent cut-edges can similarly be calculated through a bfs at each of these vertices (3)

```
/* .f refers to first element and .s refers to second
   element in a pair of integers                          */
/* timer- current time in instance of dfs                 */
/* sec- current pseudo-section in initial dfs             */
/* low and cur as described in overview                   */
/* vis- vertex visited or not in dfs                      */
/* V[]- size of pseudo-section                            */
/* comp- current set of vertices remaining in stack       */
/* bridges- set of cut-edges                              */
/* adj2[][]- adjacency list for tree of connected
   components, contains edge and the vertex of this
   connected component that is the "window" of the cut-edge
   */
/* adj3[v][]- set of vertices in each pseudo-section v    */
/* dp[]- maintains the number of vertices in the sub tree at
   each stage in the tree dfs                             */
/* D[]- distance values for each BFS                      */
 /* mark[]- a value is set to one if it is part of the
   pseudo-section                                         */
/* parent[]- marks which pseudo-section a vertex is in    */
```

```
/* DFS function                                                    */
void Dfs(p, par):
    low[p]=cur[p]=timer++
    vis[p]=1
    comp.pushback(p)
    for k in adj[p] do
        if k = par then
            continue
        if !vis[k] then
            dfs(k, p)
        low[p]=min(low[p], low[k])
    end
    if low[p] = cur[p] then
        int size=1
        sec++
        parent[p]=sec
        while comp.back()! = p do
            parent[comp.back()]=sec
            comp.popback
            size++
        end
        comp.popback
        V[sec-N]=size
        if par! = −1 then
            bridges.pushback(p, par)
```

```
/* Bfs function                                                    */
void Bfs(s):
    queue q
    q.push(s)
    D[s]=0
    while q is not empty do
        v=q.front()
        q.pop()
        for k in adj[v] do
            if !mark[k] then
                continue
            if D[k] > D[v] + 1 then
                D[k] = D[v] + 1
                q.push(k)
        end
    end
void calcWiener(p, par):
    for k in adj3[p] do
        for k2 in adj3[p] do
            D[k2]=inf
        end
        Bfs(k)
        /* Term 3                                                  */
        for k2 in adj2[p] do
            if k2.f = par then
                res+=(N-dp[p])*(D[k2].s)
            else
                res+=dp[k2.f]*(D[k2].s)
            end
        end
        /* Term 1                                                  */
        dist=0
        for k2 in adj3[p] do dist+=D[k2]
        res+=dist/2
    end
```

```
void treeDfs(p, par):
    dp[p]=V[p]
    for k in adj2[p] do
        if k.f = par then
            continue
        treeDfs(k.f, p)
        dp[p]+=dp[k.f]
    end
    for k in adj3[p] do
        mark[k]=1
    end
    /* Term 2                                                    */
    res+=dp[p]*(N-dp[p])
    calcWiener(p, par)
    /* Term 4                                                    */
    dist=0
    for k in adj2[p] do
        for k2 in adj3[p] do D[k2]=inf
        Bfs(k.s)
        for k2 in adj2[p] do
            if k = k2 then continue
            n1, n2
            if k.f=par then n1=N-dp[p]
            else n1=dp[k.f]
            if k2.f=par then n2=N-dp[p]
            else n2=dp[k2.f]
            dist+=n1*n2*D[k2.s]
        end
    end
    res+=dist/2
    for k in adj3[p] do
        mark[k]=0
    end
void Main():
    sec=N
    /* set to N so DSU does not conflict with vertex numbers
       */
    dfs(1, -1)
    for i ← 1 to N do
        adj3[parent[i]].pushback(i)
    end
    for k in bridges do
        int a=parent[k.f], b=parent[k.f]
        adj2[a].pushback(b, k.first)
        adj2[b].pushback(a, k.first)
    end
treeDfs(1, -1)
```

5.3 Time complexity

Let C_i denote the set of vertices, M_i the set of cut-edges, and E_i the set of edges in pseudo-component i. The time complexity will be

$$\sum_{i=1}^{c} |C_i|(|E_i| + |M_i|) + |M_i|(|E_i| + |M_i|)$$

It is worth noting that if special care is taken to merge cut-edges from the same **cut vertex** in the calculation of (3), all occurrences of $|M_i|$ may be replaced with $|Z_i|$, where $|Z_i|$ denotes the set of cut-vertices in pseudo-component i.

If Dijkstra's is applied instead of a BFS to account for weighted graphs, the time complexity will be

$$\sum_{i=1}^{c}((|C_i|^2 + |M_i|^2)log|E_i| + (C_i + M_i)|E_i| + |M_i|^2 + |M_i||C_i|)$$

The following analysis is focused on unweighted graphs, although similar comparisons may be drawn to time complexities on weighted graphs.

In Figure 4, the run times of the CEPC Algorithm (Table 1) and the BFS Algorithm (Table 2) are compared at $N = 2000$. Each data point is averaged over 50 randomly generated trials. At each density, the same set of graphs is tested for both the CEPC Algorithm and the BFS Algorithm.

Table 1: CEPC Algorithm

Density (/N)	Time (ms)
2	16
2.02	125
2.04	225
2.06	264
2.08	335
2.1	388
2.2	584
2.4	880
2.6	1039
2.8	1462
3	1666
4	1928
5	1918
10	2472
15	3623
20	4153
30	5942
40	7382

Table 2: BFS Algorithm

Density (/N)	Time (ms)
2	1387
2.02	1250
2.04	1264
2.06	1115
2.08	1136
2.1	1151
2.2	1123
2.4	1179
2.6	1184
2.8	1490
3	1604
4	1704
5	1624
10	2085
15	2980
20	3417
30	4793
40	6047

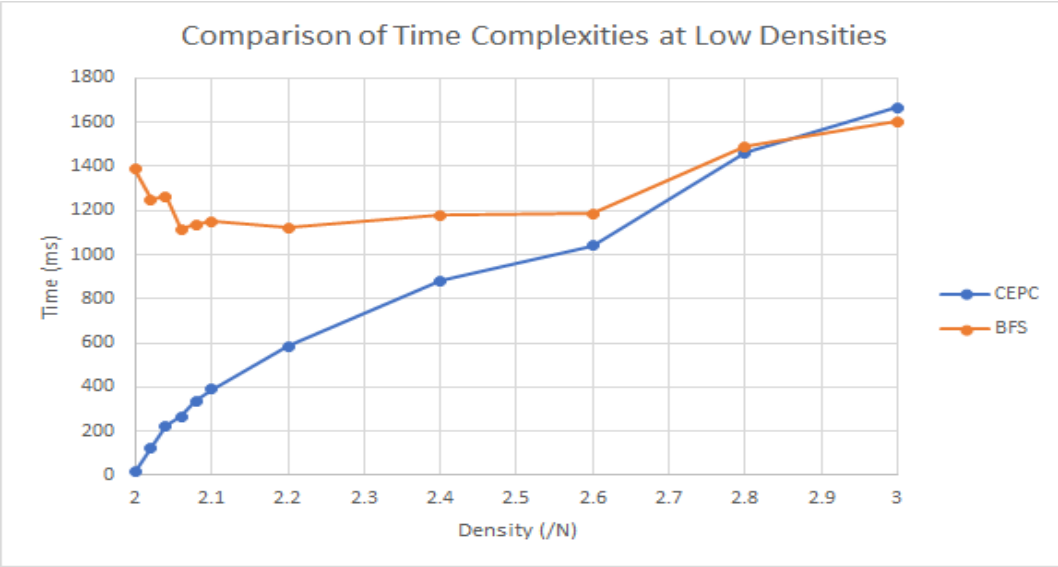

Figure 3: Graph of Time Complexities at Low Densities ($N = 2000$)

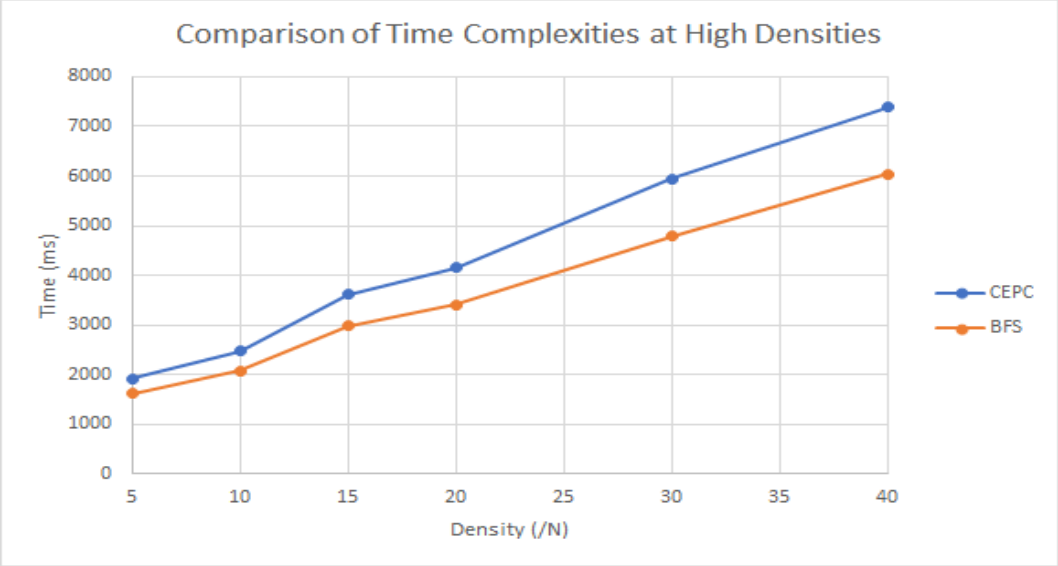

Figure 4: Graph of Time Complexities at High Densities ($N = 2000$)

Note that $\frac{2}{N}$ is the density of a tree. In such graphs, the CEPC Algorithm was faster by a factor of 87, or $\frac{N}{23}$. As the density of the graph increases, the number of pseudo-components decreases and the size of these pseudo-components increases, resulting in higher run times. This trend is seen in the graph as the time complexity of the CEPC Algorithm converges with the BFS algorithm at a density around $\frac{2.85}{N}$. As the density approaches 1 from this point, the ratio between the run times of the CEPC Algorithm and BFS remains around 1.2 as the difference between the number of pseudo-components becomes increasingly smaller. Importantly, the gap between the time complexities widens with larger N.

Testing on hexagon-shaped pseudo-components connected by cut-edges (simulating a chemical compound) yielded a run time close to $\frac{N}{20}$ times faster than the usual algorithm. For $N = 5001$, the CEPC algorithm had a run time of 17 ms compared to 6104 ms, and for $N = 50001$, the gap was even wider with 266 ms to 593319 ms

As expected, for a ring-shaped graph with 5000 nodes and the same number of edges which just one pseudo-component, the run time for both algorithms were the same.

6 Extremal Problems in Trees

In this section, we focus on proving, for various types of graphs, how to achieve the Minimum and Maximum Wiener Index.

6.1 Minimizing the Wiener Index

For a tree of N nodes, the minimum Wiener Index is achieved by having that tree be a star graph. In other words, a tree in which the diameter is 2 gives us the smallest Wiener Index among all trees. To prove this, simply observe the following:
1. In any tree with N nodes, there are $N-1$ edges. Therefore, there are $N-1$ paths that have weight 1.
2. Every other path must have weight more than 1. Therefore, the minimal possible Wiener Index would be achieved by a tree in which every path between non-adjacent nodes has weight 2.
3. Such a tree is also achievable, since a tree in which the longest path is of length 2 is, by definition, a star graph.

Furthermore, we can use this result to prove that the minimum Wiener Index is exactly $(N-1)^2$, as follows:

WI = total pairwise path length $= 1 \cdot$ (paths of len 1) $+ 2 \cdot$ (paths of len 2)
$= 1 \cdot (N-1) + 2 \cdot (\frac{(N-1)(N-2)}{2})$ ($N-1$ of len 1, every other path has len 2)
$= N - 1 + (N-1)(N-2) = (N-1)(N-1)$
$= (N-1)^2$.

6.2 Maximizing the Wiener Index

For a tree of N nodes, the maximum Wiener Index is achieved by a path; a tree in which the diameter is of length $N-1$ (basically, a graph that looks like a straight line). This can be proven by Proof by Contradiction, as shown by the following.

Consider a tree that is not a path. In any such tree, there is at least one edge, and therefore at least one component, that is not in the tree diameter. Select one of these components and call it component E. Furthermore, let Node A be the node at which component E intersects the diameter, and let components C and D be the collection of all nodes to the left and to the right of Node A, respectively.

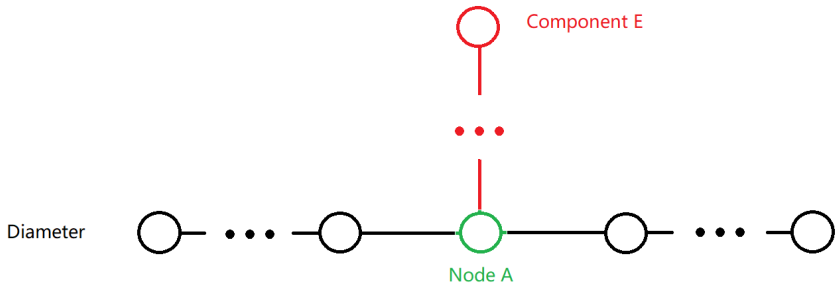

Figure 5: A tree that is not a path

Now, consider what happens if we were to shift all of Component E to the left or right by one node (in other words, changing Node A). If we shifted E left by one node, then all path lengths from nodes within E to nodes within C decrease by one, and all path lengths from nodes within E to nodes within D, as well as to node A, increase by one. No other path lengths change, which means that the new Wiener Index, WI_2 is equal to $WI_1 - |C| + |D| + 1$. Likewise, shifting to the right by one node results in a new Wiener Index of $WI_1 - |D| + |C| + 1$.

This by itself is enough to form a contradiction: If there exists a component E not on the diameter of the tree, we can always increase the Wiener Index by moving Component E towards the smaller component to its left or right. Specifically:

- if $|C| < |D|$, we will move towards D
- if $|D| > |C|$, we will move towards C
- if $|C| = |D|$, we can move in any direction

Applying this process repeatedly to E will eventually cause Node A to be at one of the endpoints of the diameter, at which point the diameter's length will increase. Doing this to every other component not on the tree diameter will cause every node to eventually be part of the diameter, meaning that the tree with the maximum Wiener Index must be a path.

7 More general extremal problems

7.1 Pseudo-star

In this section, we prove that for general graphs with given pseudo-components, the smallest Wiener Index is achieved by a particular pseudo-star. A pseudo-star is a graph where the underlying tree structure formed by its pseudo-components happen to form a star (a tree with diameter 2).

Assume that the pseudo-components are fixed. Then, we can use the expression for calculating the Wiener Index for our proof:

$$W(G) = \sum_{i=1}^{n} W(C_i) + VWW(T) + \sum_{i=1}^{n}\sum_{s=1}^{k} d_{C_i}(u_i^s) \cdot \left(N - n_{u_i^s}(u_i^s u_j^{s'})\right) + \sum_{i=1}^{n} \sum_{1 \le s < t \le k} d_{C_i}(u_i^s u_i^t) \cdot \left(N - n_{u_i^s}(u_i^s u_j^{s'})\right) \cdot \left(N - n_{u_i^t}(u_i^t u_\ell^{t'}))\right)$$

We will discuss the four terms in this expression separately:

- Since the pseudo-components are fixed, Term 1 is constant.
- Term 2: $VWW(T)$

  To prove that a pseudo-star minimizes this term, we use a Proof by Contradiction, highly similar to our previous proof that a path graph maximizes the Wiener Index of a tree. Consider a tree that is not a pseudo-star (note: the pseudo-components must form a tree), and say that it has the minimum $VWW(T)$. In any non-pseudo-star graph, there is a path of length at least 3. Consider the ends of any such path, then look at which end has the bigger total vertex weight. WLOG assume that the right end has the larger total vertex weight. In this case, we take the component attached to the left node and move the entire component closer to the right node. This means all nodes in the component move one further away from the left side and 1 closer to the rightside , so the VWW(T) will get smaller when we do this, based on our assumption that the right side has more nodes than the left side. This completes the Proof by Contradiction.
- Term 3 is minimized with the following strategy. Firstly, each pseudo-component should have exactly 1 "exit-node". Notice that every pseudo-component except the center of the star will have 1 exit-node by default, so we just have to arrange the edges such that the center also has 1 exit-node. This can be done by directly connecting all non-center pseudo-components to one chosen node in the center pseudo-component. As for which node is chosen, simply choose, for each pseudo-component, the exit vertex that has the minimum d_{C_i} (choose the exit vertex such the sum of all distances within its pseudo-component is the smallest). This minimizes Term 3.

To prove this, remember that Term 3 is $\sum_{i=1}^{n}\sum_{s=1}^{k} d_{C_i}(u_i^s)\cdot n_{u_j^{s'}}(u_i^s u_j^{s'})$. Firstly, fix i. For a given i, the sum of all $n_{u_j^{s'}}(u_i^s u_j^{s'})$ is a constant (it equals the number of nodes not inside C_i). But if that sum is a constant, then the remaining task is just to minimize $d_{C_i}(u_i^s)$ for every single s $(*)$. This obviously means that choosing one exit-node for each C_i that has the minimum d_{C_i} minimizes Term 3.

$(*)$ To prove this, consider numbers $a_1 < a_2 < a_3 < ...$ and $n_1 + n_2 + ...n_N = K$ (K is some constant), and say that we want to choose, for each n_i, some a_j such that $\sum_{i=1}^{n} a_j \cdot n_i$ is minimized. This is equivalent to minimizing Term 3, where a_i represents the different d_{C_i} and n_i represents the sizes of all other pseudo-components (which sum to a constant). Obviously, we minimize the expression by just choosing a_1 for every n_i, because for any case in which we do not choose a_1, we can always decrease the total sum by choosing a_1 (since a_1 is the smallest a).

- Term 4 is also minimized by a pseudo-star; it equals 0 because in a pseudo-star each pseudo-component will only have 1 node at its edge (assuming we select all cut-edges to connect to the same node in the center pseudo-component). This means that there will be no pairs of nodes at the edges of the pseudo-component at all, so this term will equal 0.

From the above discussion, we have the following theorem.

Theorem 7.1. *For a graph G with pseudo-components C_i ($1 \le i \le n$ and cut-edges e_j ($1 \le j \le n-1$), T as the underlying tree structure (where v_i corresponds to C_i) with the weight $w(v_i) = |V(C_i)|$, the Wiener index of G is minimized if and only if G is the unique pseudo-star where:*

- *T is a star with center v with the maximum weight in T;*
- *each pseudo-component C contains exactly one cut-vertex u that minimizes $d_C(u)$.*

7.2 Pseudo-path

In this section, we prove that for general graphs with given pseudo-components, the largest Wiener Index is achieved in some pseudo-path. A pseudo-path is a graph where the underlying tree structure formed by its pseudo-components happen to form a path.
Assume that the pseudo-components are fixed. Then, we can use the following expression for calculating the Wiener Index for our proof:

$$W(G) = \sum_{i=1}^{n} W(C_i) + VWW(T) + \sum_{i=1}^{n}\sum_{s=1}^{k} d_{C_i}(u_i^s)\cdot n_{u_j^{s'}}(u_i^s u_j^{s'}) + \sum_{i=1}^{n}\sum_{1\le s<t\le k} d_{C_i}(u_i^s u_i^t)\cdot n_{u_j^{s'}}(u_i^s u_j^{s'}))\cdot n_{u_\ell^{t'}}(u_i^t u_\ell^{t'}))$$

7.3 Algorithm to Calculate the Maximum Wiener Index of Weighted Path

Before describing the algorithm, it is necessary to establish the structure of the path. Let w be the array of the weight of each vertex sorted in non-increasing order, and let α be the desired permutation. The arrangement that maximizes the Wiener Index is formed by appending the weights beginning from the end of w either to the left or right of the initially empty array α.

More formally, there must exist an index i such that for $1 \leq k \leq i-1$, $\alpha[k] \geq \alpha[k+1]$ and for $i \leq k \leq n$, $\alpha[k] \leq \alpha[k+1]$.

Cela et al. [1] call this structure "V Shaped" and prove the correctness of this arrangement. They also describe a very similar algorithm although with different notation and in the context of the Quadratic Assignment Problem.

This observation prompts an immediate $O(2^n)$ brute-force algorithm that can be achieved by iterating through w and assigning values to α either to index $l+1$ or $r-1$ and adjusting these values accordingly (l and r are initially 1 and n respectively). The following is an optimization of this recursion.

Let $L = \sum_{i=1}^{l-1} \alpha[i]$ and $R = \sum_{i=r+1}^{n} \alpha[i]$. The second observation is that if two states are both on the same index i of w and share the same L and l values or the same R and r values, the arrangement that maximizes the Wiener Index from l to r is the same. This motivates the following dynamic programming approach

Let dp(k, l, L) denote the answer to the state currently deciding the position of $w[k]$ having already assigned $l-1$ values to the left and with the sum of these values being L. Note that this state only includes the contribution of edges in the range [l, r]. There are two transitions corresponding to whether $w[k]$ is placed at l or r:

$$T1 = dp(k+1, l+1, L+w[k]) + (L+w[k])(sum - (L+w[k])) \tag{5}$$

The second term represents the contribution that the edge from l to $l+1$ contributes, and sum is the sum of all the weights.

$$T2 = dp(k+1, l, L) + (R+w[k])(sum - (R+w[k])) \tag{6}$$

R and r can be stored as values in the recursion (as done in the algorithm below) or derived from k and l values along with preprocessing. The answer is given by dp(1, 0, 0).

Algorithm 6: Maximum Wiener Index of Weighted Path

```
/* uses 0-indexing                                                    */
Int solve(l, r, L, R):
    if l = r then
        return 0
    k = n − (r − l + 1)
    if dp[k][l][L] then
        return dp[k][l][L]
    T1 = solve(l+1, r, L+w[k], R) + (L+w[k])(sum-L-w[k])
    T2 = solve(l, r-1, L, R+w[k]) + (R+w[k])(sum-R-w[k])
    if T1 > T2 then
        to[k][l][L]=-1
    else
        to[k][l][L]=1
    end
    return dp[k][l][L] = max(T1, T2)
void Main():
    solve(0, n − 1, 0, 0)
    l = 0, r = n-1, lsum = 0
    while l ≤ r do
        k=n-(r-l+1)
        if l = r then
            ans[l] = w[k]
            break
        if to[k][l][lsum] = -1 then
            ans[l] = w[k]
            lsum += w[k]
            l++
        else
            ans[r] = w[k]
            r - -
        end
    end
```

The code above also finds an arrangement of the weights that maximizes the Wiener Index. Instead of storing the arrangement at each state, the information of whether the next element is placed to the left or right is stored in to[ ] and backtracking is used to recover the sequence.

This algorithm has a pseudo-polynomial time complexity of $O(n^2 \sum w_i)$ based off the number of states in the DP. There are n possible values for both k and l, and $\sum w_i$ possible values for the sum of the l values.

References

[1] E. Cela et al. “The Wiener maximum quadratic assignment problem”. In: *Discrete Optimization* (2011).

[2] J. Dearden. “In silico prediction of drug toxicity”. In: *Journal of Computer-aided Molecular Design* (2003).

[3] R. Diestel. “Extremal Graph Theory”. In: *Graph Theory* (2017).

[4] A. Dobrynin and A. Vesnin. “On the Wiener Complexity and the Wiener Index of Fullerene Graphs”. In: *Mathematics* 7.11 (2019). ISSN: 2227-7390. DOI: 10.3390/math7111071. URL: https://www.mdpi.com/2227-7390/7/11/1071.

[5] A. Manorama et al. “A Comparative QSAR Study Using Wiener, Szeged, and Molecular Connectivity Indices†”. In: *Journal of Chemical Information and Computer Sciences* (2000).

[6] S. Wang and X. Guo. “Trees with extremal Wiener indices”. In: *Communications in Mathematical and in Computer Chemistry* (2007).

[7] H. Wiener. “Structural determination of paraffin boiling points”. In: *Journal of the American Chemical Society* (1947).