

MASTER'S THESIS | LUND UNIVERSITY 2018

Orthogonal Range Searching and Graph Distances Parameterized by Treewidth

Måns Magnusson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-31



Orthogonal Range Searching and Graph Distances Parameterized by Treewidth

Måns Magnusson

`Mans.Magnusson.888@student.lu.se`

June 7, 2018

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Thore Husfeldt, `Thore.Husfeldt@cs.lth.se`

Examiner: Krzysztof Kuchcinski, `Krzysztof.Kuchcinski@cs.lth.se`

Abstract

In this master's thesis, we present an improved time complexity analysis of algorithms that calculate graph metrics: diameter, radius, eccentricities and Wiener index in weighted, undirected graphs. Our results show single-exponential dependence on the treewidth.

These metrics can be calculated in polynomial-time using Dijkstra's algorithm, and in linear time for trees. For graphs with bounded treewidth, i.e., graphs with small cycles, the linear algorithm can be generalized by finding points inside a d -dimensional box, so-called orthogonal range searching, which can be implemented using the data structure Range Trees.

Part of the work has been submitted to a peer-reviewed, specialist conference, as a joint paper with Bringmann and Husfeldt. In the paper, attached as an appendix, we present new upper bounds by improving the analysis of Range Trees.

This thesis provides more details to the background ideas and proofs covered in the paper, implementation, as well as empirical comparisons of running times.

Keywords: Graph diameter, Wiener Index, parameterized algorithms, orthogonal range searching, tree width, range trees, computational geometry

Acknowledgements

I want to thank my supervisor, Thore, for the many interesting discussions in his office, that were very focused, politically correct, and never ever went out of bounds. However, a word of advice, your shirts are not eccentric enough: the flowers have way too small diameter.

Thank you, Karl, for finding critical errors, and lifting the paper to a higher dimension!

Thank you, Maj, for proofreading, *proof* reading, and a never ending range of queries “What does this mean?”, which I’ve now cached the response for: “Oh, that’s an error.”¹

¹Also she is of the controversial opinion that the maximum of 5, 7 and 8 actually is 8, and not 7.

Contents

1	Introduction	7
1.1	Definitions	8
1.2	Related work	9
1.3	Contributions	10
2	Theoretical derivation	11
2.1	Diameter of an unweighted tree	11
2.1.1	Applying Centroid Decomposition	12
2.2	Generalization	14
2.2.1	Tree Decomposition	14
2.3	Eccentricities	16
2.4	Orthogonal Range Searching	20
2.4.1	Preliminaries	20
2.4.2	Range Trees	20
2.4.3	Remark on Range Trees	24
2.5	Asymptotics	25
2.6	Wiener Index	25
3	Experimental demonstration	27
3.1	Running Times	27
3.1.1	Generation of test data	27
3.1.2	Measurements	28
3.2	Discussion	28
4	Conclusions and Future Work	31
	Bibliography	33

Chapter 1

Introduction

Einstein was wrong, time is not relative, especially not in the world of algorithms. It matters how much time it takes for our algorithms to execute. Will it finish in the blink of an eye, in the time it takes to get coffee, in my lifetime or before the end of time? Exponential growth is scary. We analyze our algorithms by finding upper bounds for the number of operations as a function of the input size. Throughout this thesis you will encounter “algorithm x is bounded by $O(f(n))$ ” where $f(n)$ is some function of n . This means that the number of operations needed to execute x does not grow faster than f . Formally it means that there exists some constant c such that the number of operations is less than $cf(n)$ for every input of size $n \geq t$, where t is a constant threshold. This is a guarantee that should hold for all instances of the problem.

In this work we analyze different distance metrics of graphs. Graphs can vary in shape and size, by more than one parameter, the amount of vertices, amount of edges and what we are especially interested in, treewidth, which is a metric on how close a graph is to a tree, in the sense that low treewidth graphs only have small cycles, while trees have none.

To analyze these algorithms we use multivariate bounds, where we have two, or more parameters of the graph. One of the classic algorithms for graph traversal, breadth first search (BFS), has a running time bounded by $O(n + m)$, where n is the number of vertices, and m is the number of edges. With this we mean that the number of operations are bounded by $c_1n + c_2m$ for some constants c_1, c_2 , for all $n \geq t_1$ and $m \geq t_2$, where t_1 and t_2 are constant thresholds.

The graph metrics in focus are diameter, radius and Wiener index. These metrics are defined in Section 1.1. However this is not just an amusing intellectual exercise, these metrics have various applications (even though I had to look hard for some of them). Graphs can be used to model things in the real world, e.g., networks, like social networks, road networks or the internet, graph databases or molecules.

In networks, questions like “What are the two furthest vertices apart”, make sense, since we might want to modify the network to become more well connected. Or if we wish to decide where to have a fire station we would like the time it takes to reach any

intersection (vertex in a graph representation of the road network) to be minimal. If we efficiently can compute the eccentricity of every vertex in the graph, we place the fire station in the intersection where the eccentricity is minimal.

Perhaps more surprising are the chemical applications. Wiener showed that the Wiener index, in the original paper called path number, of an alkane molecule correlates strongly with its boiling temperature [15]. There exists very large molecules with thousands of atoms called macro molecules, where computing the Wiener Index is a heavy computation, and needs to improve. Note that molecules often have a tree-like structure, with possibly small cycles, which fits our approach perfectly.

1.1 Definitions

Given a connected, undirected graph $G(V, E)$ with non-negative edge weights, we define the following terms:

- Distance between two vertices $u, v \in V(G)$, denoted $d(u, v)$ is the minimum sum of edge weights along a shortest path from u to v . This can be found by the well known Dijkstra's algorithm. Let $w(a, b)$ denote the weight of an edge between vertices a and b . Formally the distance can be recursively defined as: for all $u \in V(G)$, $d(u, u) = 0$, $d(a, b) = \min_{(a,c) \in E(G)} (w(a, c) + d(c, b))$.
- Eccentricity of a node $u \in V(G)$, denoted $e(u) = \max_{v \in V(G)} d(u, v)$, the maximum distance between u and any other vertex v in the graph.
- Diameter of G , denoted $\text{diam}(G) = \max_{u \in V(G)} e(u)$, the largest distance in the graph.
- Radius of G , denoted $\text{rad}(G) = \min_{u \in V(G)} e(u)$, the smallest eccentricity of the graph.
- Wiener Index of G , denoted $\text{wien}(G) = \frac{1}{2} \sum_{u, v \in V(G)} d(u, v)$, the sum of all distances between all unordered pairs of vertices in the graph.
- Tree is a connected graph with exactly $n - 1$ edges, which means that there are no cycles in a tree, and exactly one path between every pair of vertices.
- Centroid of a tree is a vertex that if removed cuts the tree into disconnected sub-graphs, all with maximum size $n/2$.

In figure 1.1 an unweighted graph is shown, where the unique diameter is marked.

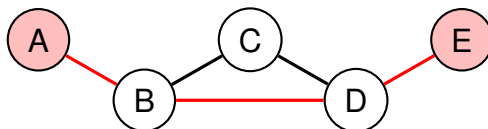


Figure 1.1: In this unweighted graph (all edges have weight 1) with 5 vertices, the marked diameter is 3, between the vertices A and E. The radius is 2 (the eccentricities of vertices B, C and D), and the Wiener index is 16, with 5 distances of length 1, 4 of length 2 and 1 of length 3.

1.2 Related work

Dijkstra presented His Algorithm to find the shortest path from the vertex u to the vertex v in a graph with n vertices and m edges in 1959 [8]. Dijkstra's original algorithm finds this distance in time $O(n^2)$, and just the distance between u and v , but it can easily be modified to find the distances from u to all other vertices in the graph. The time complexity was improved by Fredman and Tarjan in 1984 to $O(m + n \log n)$, by using a Fibonacci heap to keep the vertices sorted, that reduces a step in the algorithm from linear to logarithmic time [9]. Whenever Dijkstra's Algorithm is mentioned throughout this report it will refer to the algorithm presented by Fredman and Tarjan, and it will find the distance to all other vertices in the graph, from a single vertex. A naive approach for calculating the diameter, radius, eccentricities and Wiener index of a graph uses Dijkstra's Algorithm, which is similar to the described algorithm for Wiener index in unweighted graphs by Mohar [12]:

Algorithm D.

Given a graph $G(V, E)$ with $n = |V(G)|$ vertices and $m = |E(G)|$ edges, calculates the diameter, radius, the eccentricities and the Wiener index

D1 [Repeat] For each vertex $u \in V(G)$:

D1.2 [Dijkstra] Compute a vector of distances to all other vertices in G with Dijkstra's Algorithm. Save the vector in d_u .

D1.2 [Eccentricity] Save the eccentricity as $e(u) = \max d_u$.

D1.3 [Wiener index contribution] Save the Wiener index contribution as $s(u) = \sum d_u$.

D2 [Return] $\max_{u \in V(G)} e(u)$, $\min_{u \in V(G)} e(u)$, e , $\frac{1}{2} \sum_{u \in V(G)} s(u)$

Let D_{Diameter} return the diameter, D_{Radius} return the radius, $D_{\text{Eccentricities}}$ return the eccentricities, and $D_{\text{Wiener-index}}$ return the Wiener index.

Algorithm D runs in time $O(n(m + n \log n))$, which for graphs with treewidth k is $O(n^2(k + \log n))$, since the number of edges in a graph with treewidth k is at most nk , i.e., it has to be sparse. The argument for sparsity can be found in Proposition 3.3 by Rose [14], although make sure to read Section 2.2.1 about tree decompositions first.

It is hard to improve upon this idea, even though very similar things are calculated in each Dijkstra search from every vertex.

We focus our efforts on graphs with bounded treewidth. Cabello and Knauer show an algorithm for diameter that runs in time $O(n \log^k n)$ [5], where other dependence of k is discarded, since k is viewed as a constant. Abboud et al. raise the question how this grows with k , and if it is possible to separate the dependence of n and k into a multiplicative dependence, which is answered by a new analysis of Cabello and Knauer's algorithm, that runs in time $n^{1+\epsilon} \exp O(k \log k)$ [1].

Abboud et al. also prove that if an algorithm that computes the diameter is both sub-quadratic in the number of vertices and sub-exponential in the treewidth then that algorithm could solve some other algorithmic problems faster than the researching community thinks possible. In particular such an algorithm would refute a widely believed hypothesis called the Strong Exponential Time Hypothesis established by Impagliazzo [11].

Range Trees, invented by Bentley [2] and explained later in Section 2.4.2, is a data structure that finds which multi-dimensional points from a collection that satisfy restrictions given a criterion. The data structure is well studied in the field of computational geometry, for example by Chan [6] and Monier [13].

1.3 Contributions

The main contribution of this work consists of showing that d -dimensional Range Trees can be built in time $nd \binom{\lceil \log n \rceil + d}{d}$, and queried in time $2^d \binom{\lceil \log n \rceil + d}{d}$, which means that Wiener index, Eccentricities, Diameter and Radius can be computed in a graph with treewidth k in time $n^{1+\epsilon} \exp O(k)$, for all $\epsilon > 0$, in other words super-linear in n times single-exponential in k . This improves the bound presented by Abboud et al., in [1]. Also the multivariate analysis of Wiener index is new, since [1] did not cover that. However the argument for constant treewidth already exists in [5]. We can conclude that we answer the open question “Closing the small gap in the dependence on k between the $n^{1+\epsilon} \exp O(k \log k)$ upper bound and the $n^{2-\epsilon} \exp o(k)$ conditional lower bound is a very interesting open question” in [1]. We also reject Husfeldt’s conjecture that “However, we conjecture that the diameter of a graph with treewidth and diameter k cannot be computed in time $n^{2-\epsilon} \exp o(k \log k)$ ” [10].

We confirm the speculation in [1] that the Cabello and Knauer algorithm is practical by providing a complete implementation and compare it with the standard algorithm described by Mohar [12].

We will in Chapter 2 present background ideas including diameter on trees, centroid decomposition, tree decompositions and Range Trees, leading us to our solution that is a presentation of Algorithm E that calculates the eccentricity of every vertex, in time $n^{1+\epsilon} \exp O(k)$, for all $\epsilon > 0$, where n is the number of vertices in the graph, and k is the treewidth of the graph. One can think about a small treewidth as graphs with only small cycles.

Our method can be modified to compute the Wiener index as well, which there is a brief description of in the end.

In Chapter 3 experimental results of execution times of Algorithm D and Cabello and Knauer’s algorithm will be presented, followed by a discussion.

Chapter 2

Theoretical derivation

In this chapter we will present diameter algorithms for trees, definition of tree-decomposition, an introduction to range trees, the algorithm we are analyzing in the submitted paper, and notes on the analysis.

2.1 Diameter of an unweighted tree

Recall that a tree with n vertices is a connected graph with no cycles, it has exactly $n - 1$ edges, and all pairs of vertices are connected by exactly one path.

One endpoint of a diameter in a tree can be found by running a BFS from any vertex u . The vertex visited last of the BFS will be an endpoint in a diameter of the tree, however the proof for this is a bit intricate.

Lemma 1. *The last visited vertex w by a BFS in a tree from an arbitrary vertex u will be an endpoint of a diameter in the tree.*

Proof. Let us call the endpoints of a diameter x and y , and the path between x and y xPy . During the BFS we will find all vertices in the tree, since it is connected. Let us call the first vertex found by the BFS that lies on xPy , v . Since v is on the path between x and y , the vertex furthest away from v must be either x or y , otherwise x and y is not a diameter. Without loss of generality, let that be vertex x , meaning that $e(v) = d(v, x)$. We would like to show that $e(u) = d(u, x)$, and that all vertices at distance $e(u)$ from u is an endpoint of a diameter. Let the last visited vertex be called w . To derive a contradiction, suppose $d(u, w) > d(u, x)$, then $d(u, v) + d(v, w) \geq d(u, w) > d(u, x) = d(u, v) + d(v, x)$, giving $d(v, w) > d(v, x)$, which gives us a contradiction since $e(v) = d(v, x)$.

Furthermore if instead $d(u, w) = d(u, x)$, then $d(u, v) + d(v, w) \geq d(u, w) = d(u, x) = d(u, v) + d(v, x)$, we get that $d(v, w) \geq d(v, x)$, which means that $d(v, w) = d(v, x)$ since $e(v) = d(v, x)$. Also v is on the path from u to w , since $d(u, w) = d(u, v) + d(v, w)$ and there is only one path between any two vertices in a tree.

We investigate two cases, either w is in the same subtree as x if we root the tree in v , or it is not. If w is in the same subtree as x it creates a diameter with y , since $d(w, y) = d(w, v) + d(v, y) = d(x, v) + d(v, y) = d(x, y)$. Otherwise it creates a diameter with x since then $d(w, x) = d(w, v) + d(v, x) \geq d(y, v) + d(v, x) = d(y, x)$, where we with inequality get a contradiction of x, y being a diameter. This means that any vertex at distance $d(u, x)$ from u is an endpoint of a diameter.

When we have an endpoint of the diameter, we can just do a BFS again, and the diameter will be the distance to the last visited vertex.

However, this approach does not generalize at all. For a graph that is not a tree and has one small cycle, the proof does not work, since this cycle means that there are multiple paths between vertices. More specifically w does not have to form a diameter with x or y , since there might be a shorter path that does not go through v for both x and y .

Luckily there are other approaches.

2.1.1 Applying Centroid Decomposition

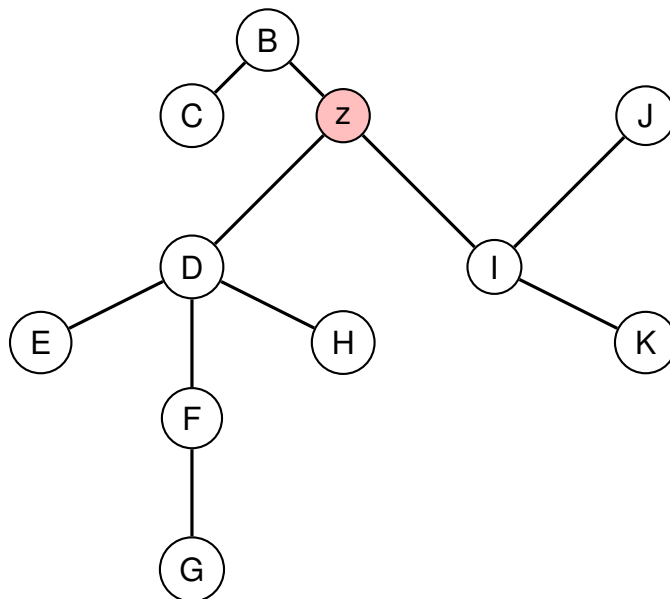


Figure 2.1: Algorithm CD will find vertex z as the centroid in the above tree. There are 3 subtrees connected to Z , $S_1 = \{B, C\}$, $S_2 = \{D, E, F, G, H\}$ and $S_3 = \{I, J, K\}$. The diameter is found between subtree S_2 and S_3 , and the diameter is $d(S_2) + d(S_3) = 3 + 2 = 5$. More specifically the diameter is found between G and J (G and K also creates a diameter).

Remember that a centroid is a vertex in a tree that separates the tree into subtrees, all with size at most $n/2$, if the size of the tree is n . We note that a diameter either has to pass through the centroid or not. If it does not pass through the centroid it is fully contained within an entire subtree, since there is only one path between two vertices in a tree. Figure 2.1 and 2.2 both display a tree with a centroid Z , and the two different cases

that can arise. Algorithm CD uses this to find the diameter.

Algorithm CD.

Given a tree T with n vertices, finds the diameter of T .

CD1 [Base case.] If $n \leq 2$, return $n - 1$.

CD2 [Find centroid.]

CD2.1 [Root the tree] In any vertex u .

CD2.2 [Find children below] Do a depth first search from u that finds the number of children below a given vertex in the rooted tree. Save this number as $below(v)$ for each vertex $v \in V(T)$.

CD2.3 [Walk] Start walking from u towards the child v with the largest $below(v)$, while $below(v) > n/2$.

CD2.4 [Save Centroid] Save the vertex that CD2.3 stopped at as the centroid, call it z .

CD3 [Group each vertex by subtree] Find the subtree each vertex belongs to with a depth first search from z . Label the subtrees as S_1, S_2, \dots, S_k .

CD4 [Distance from centroid] Find the distance from z to each vertex with a breadth first search.

CD5 [Distance to each subtree] Define the distance to each subtree S_i as the maximal distance to any vertex in the subtree, and label it $d(S_i)$.

CD6 [Find the two subtrees with largest distance] Let $d_{through} = \max_{i,j \in \{1 \dots k\}} d(S_i) + d(S_j)$

CD7 [Recurse on each subtree.] Disconnect each subtree from z , and find $D[i] = \text{CD}(S_i)$.

CD8 [Return.] Return $\max(D[1], \dots, D[k], d_{through})$.

To prove correctness we need to argue that no distance can be larger than the returned distance. All pairs of vertices are covered by inspecting pairs from the same subtree and pairs from different subtrees. If vertices x, y are in different subtrees, the path from x to y has to go through z , so the distance $d(x, y) = d(x, z) + d(z, y)$. $d(S_i)$ holds the largest distance to a vertex in S_i , the largest sum between two vertices is therefore $d_{through}$. By induction on tree-size, the largest distance for each subtree is found by Step CD7. \square

The running time of Step CD2 is dominated by CD2.2, which inspects the entire tree, which is done in time $O(n)$, because each edge and node is only inspected once. The walk in Step CD2.3 will visit different vertices each time, so the running time for CD2 is bounded by $O(n)$. Step CD3, CD4 and CD5 can all be calculated in linear time. Step CD6 can also be implemented in linear time by iterating over all subtrees, remembering the two largest so far found.

Let $t(n)$ denote the running time of Algorithm CD for a tree of size n . We establish the recurrence relation $t(n) \leq cn + \sum_{i \in \{1 \dots k\}} t(|S_i|)$. Clearly t is at least linear in the tree-size. This means that $t(x) + t(y) \leq t(x + y)$. We will prove that there exists a subset of

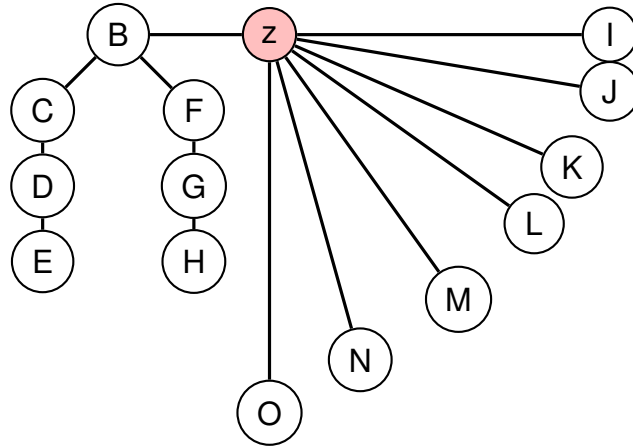


Figure 2.2: Algorithm CD will find vertex z as the centroid in the above tree. However the diameter is found within the large subtree containing $\{B, C, D, E, F, G, H\}$, between vertices E and H at distance 6, that is $CD(\{B, C, D, E, F, G, H\}) = 6$.

subtrees which size sums to s , such that $n/3 \leq s < 2n/3$. Since the largest subtree has size at most $n/2$, if any subtree has size at least $n/3$ that subtree on its own forms a subset satisfying the criteria. On the other hand, if all subtrees are of size less than $n/3$, we can add subtrees into one until the size is at least $n/3$, knowing that we never add more than $n/3$ each time, which means that we will get a subset of subtrees where the sum is within the range. Combining the rest of the subtrees creates another set, that also falls within the range, since that will have size $n - 1 - s$. Because of $t(x) + t(y) \leq t(x + y)$, we now know that $t(n) \leq cn + t(s) + t(n - 1 - s)$. Let \log denote \log_2 . We will show that $t(n) \leq 2cn \log n$ solves this equation, since

$$\begin{aligned}
 t(n) &\leq cn + 2cs \log s + 2c(n - 1 - s) \log(n - 1 - s) \leq \\
 &\leq cn + 2cs \log(2n/3) + 2c(n - 1 - s) \log(2n/3) \leq \\
 &\leq cn + 2c(n - 1)(\log n - \log 3 + 1) \leq \\
 &\leq cn + 2cn(\log n - \log 3 + 1) \leq \\
 &\leq cn + 2cn(\log n - 0.58) = \\
 &= cn - 1.16cn + 2cn \log n \leq \\
 &\leq 2cn \log n
 \end{aligned}$$

This concludes $t(n)$ is $O(n \log n)$.

2.2 Generalization

We will establish some more theory needed to generalize Algorithm CD.

2.2.1 Tree Decomposition

To use the ideas from centroid decomposition on graphs that are not trees, we use a *tree decomposition*. The tree decomposition can be thought of as a tree that covers the graph.

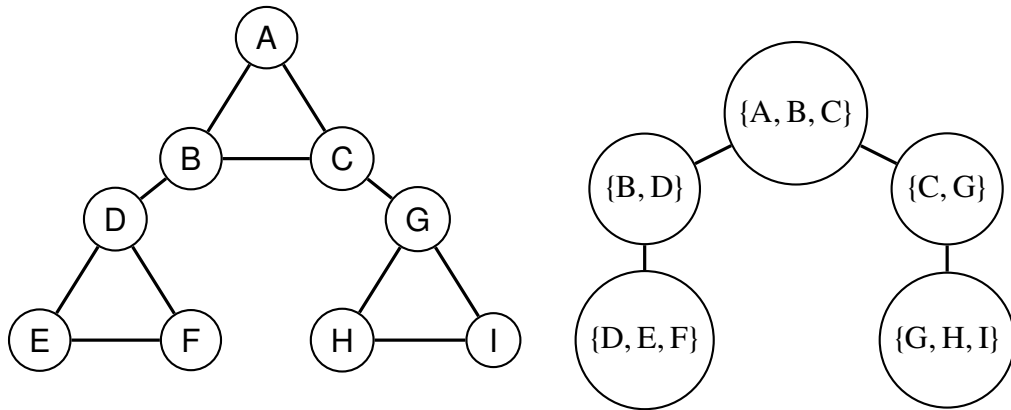


Figure 2.3: To the left there is a graph G of size 9. To the right is a tree decomposition of G , with multiple vertices inside each bag.

Inside each vertex, called bag, of a tree decomposition there exists multiple vertices from the original graph G .

Definition of Tree Decomposition

We repeat the definition of a Tree Decomposition found in [4]. Let $G = (V, E)$ be a graph. A tree-decomposition of G is a pair $(\{X_i | i \in I\}, T = (I, F))$, where $\{X_i | i \in I\}$ is a family¹ of subsets of $V(G)$. $T = (I, F)$ ² is a tree, with the following properties:

1. $\cup_{i \in I} X_i = V(G)$
2. For every edge $e = (v, w) \in E$, there is a subset X_i with $v, w \in X_i$.
3. For all $i, j, k \in I$, if j lies on the path in T from i to k , then $X_i \cap X_k \subseteq X_j$.

In “plain English”, a tree-decomposition is a tree T of vertices called bags, each containing a set of vertices from the original graph G , for an example see Figure 2.3. Also some conditions need to be fulfilled for T to be a tree decomposition of G .

- All vertices in G must belong to at least one bag in T .
- All edges in G must belong to a bag in T , i.e., both endpoints must be in some common bag.
- If there are two bags that share some vertices, all bags on the path in T must contain these vertices as well.

We want to use the following property of a tree-decomposition:

Lemma 2. Consider a path in G between vertices u and v and two bags B_u and B_v , where B_u contains u and B_v contains v , then every bag B on the path in T between B_u and B_v needs to contain at least one of the vertices on every path between u and v in G .

¹A family is a collection of sets.

² I are the bags, and F are the edges between bags.

Proof. Consider a path P between u, v in G , as a sequence of edges $P(u, v) = \{e_1, e_2, \dots, e_l\}$. Let T be a tree decomposition of G . Each edge in $P(u, v)$ needs to be inside a bag in T because of property 2. Call these bags B_1, B_2, \dots, B_l . Each pair of bags B_i, B_{i+1} share a common vertex, the i 'th vertex on P . Because of property 3, the i 'th vertex must be in every bag on the path from B_i to B_{i+1} in T . Therefore there exists a walk from bag B_1 to bag B_l , where all bags contains a vertex from some edge in $P(u, v)$.

Definition of treewidth

There can exist multiple tree decompositions of a graph. The treewidth of G , usually in this report referred to as k or $\text{tw}(G)$, is the size of the largest bag in the tree decomposition of G that has minimal largest bag, minus 1. Formally, let \mathcal{T} denote the set of tree-decompositions of G , then $\text{tw}(G) = \min_{(T,F) \in \mathcal{T}} (\max_{X_i | i \in I} |X_i|) - 1$. The reason for the -1 is because then all trees T has $\text{tw}(T) = 1$.

Also, the number of edges in a graph G with n vertices and treewidth k is less than nk . This is shown by Rose by a induction argument [14]. We will not cover that argument since it is not in the right direction of the thesis.

2.3 Eccentricities

In this section we present an algorithm calculating the eccentricities of a graph using the centroid bag of the tree decomposition. In the research paper, we used a more general approach, that instead uses k -separators. The usage of k -separators yields a slightly more general result, but would increase the complexity of the implementation.

Some problems arise when we try to use Algorithm CD on the tree decomposition. Especially, the only thing we know for sure is that paths between vertices in different subtrees pass through some vertex in the centroid bag. The question is through which one, or possibly which ones, do the shortest path(s) pass through?

To find a centroid in a tree decomposition, the DFS in Algorithm CD needs to be modified. In a tree the following holds: $\text{below}(u) = 1 + \sum_{c \in \text{children}(u)} \text{below}(c)$. Because vertices can reside in multiple bags, we can not add them indiscriminately anymore. Let $\text{below}_{id}(i)$ denote the number of vertices from G inside all the bags in the subtree rooted in bag i , then $\text{below}_{id}(i) = |X_i| + \sum_{j \in \text{children}(i)} (\text{below}_{id}(j) - |X_i \cap X_j|)$. Because of the third property of tree decompositions, every vertex inside both X_i and the subtree rooted at j needs to be in X_j , which means that the subtracted part $|X_i \cap X_j|$ is exactly the amount that would have been double counted.

In figure 2.4 we find that from the vertex x to vertices on the other side of the centroid bag Z , some paths to vertices in Y go through vertices in Z are not shortest.

For a general graph G with tree decomposition T , and a centroid bag Z , let $k = |Z|$. Let X be the vertices inside the bags of a set of subtrees in T rooted in Z . Let Y be the remaining vertices of G that are neither in X nor in Z .

Since all paths from $x \in X$ to $y \in Y$ has to go through some vertex in Z , we know that the shortest distance between x and a vertex $y \in Y$ has $z \in Z$ on its path, if for all $z_i \in Z$,

$$d(x, z) + d(z, y) \leq d(x, z_i) + d(z_i, y).$$

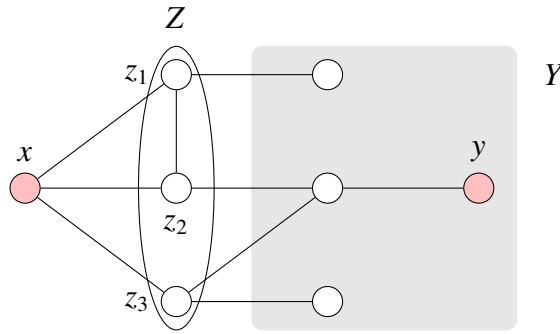


Figure 2.4: Example where the centroid bag Z contains $\{z_1, z_2, z_3\}$. The eccentricity of x to Y is 3, however $d(x, z_1) + d(z_1, y) = 4$ and forms no shortest path. The longest shortest path from x through z_1 is 2, z_2 and z_3 are both 3.

These inequalities can be rearranged as, for all $z_i \in Z$,

$$d(z, y) - d(z_i, y) \leq d(x, z_i) - d(x, z).$$

However $d(z, y) - d(z_i, y)$ is independent of x , and $d(x, z_i) - d(x, z)$ is independent of y . We can represent these inequalities by k -dimensional points. Let $p^{(x)} = [d(x, z_1) - d(x, z), d(x, z_2) - d(x, z), \dots, d(x, z_k) - d(x, z)]$, and $p^{(y)} = [d(z, y) - d(z_1, y), \dots, d(z, y) - d(z_k, y)]$. z is on a shortest path from x to y , if and only if $p^{(x)}$ is at least as large as $p^{(y)}$ on every coordinate.

For given $x \in X$ and $z \in Z$, we would like to efficiently find the y that maximizes $d(x, y)$, and has z on a shortest path from x to y . If there only were a data structure that could answer such questions! Luckily for us Bentley presented the data structure Range Tree almost 40 years ago [2].

Characteristics of a Range Tree

Given a set P of d -dimensional points, a Range Tree finds all the points inside a d -dimensional box, or some accumulative information about them. With a box B we mean two corner points lo and hi , representing an interval, and a point p is inside the box if for all $i \in \{1 \dots d\}$, $lo_i \leq p_i \leq hi_i$. If we associate each point with a value $f(p)$, the max-Range Tree answers $\max_{p \in P \cap B} f(p)$, for a box B , i.e., the maximum value of $f(p)$ of all points $p \in P$ inside the box B .

Algorithm E.

Given a graph G with n vertices, and tree decomposition $T(G)$, with treewidth k , calculates the eccentricities of G .

E1 [Base case] If $n < 2^k$, return $D_{\text{Eccentricities}}(G)$.

E2 [Find centroid in T .]

E2.1 [Root T] In any bag b .

E2.2 [Find children below] Do a depth first search from b that finds the number of vertices inside the union of all bags in the subtree rooted at c for every bag $c \in T$. Save this number as $below_{id}(c)$.

- E2.3** [Walk] Start walking from b towards the child c with the largest $below_{id}(c)$, while $below_{id}(c) > n/2$.
- E2.4** [Save Centroid] Save the bag that E2.3 stopped at as the centroid, call it Z .
- E3** Collect subtrees connected to Z in T , and label the union of the vertices from G inside the subtrees as X , such that $n/3 \leq |X| \leq 2n/3$. Label the union of the vertices inside the remaining subtrees as Y .
- E4** Run Dijkstra from every vertex in Z in G and save all the distances for each $z \in Z$ and $u \in V(G)$ into $d(z, u)$.
- E5** [Add shortcuts] for each pair of vertices $z_i, z_j \in Z$, add an edge G to have weight $d(z_i, z_j)$, if there was no edge, add it.
- E6** Allocate eccentricities, where $e(z) = \max_{u \in V(G)}(d(z, u))$, for all $z \in Z$, and $e(u) = 0$, for all $u \in V(G) \setminus Z$.
- E7** [Repeat] for each $z \in Z$:
- E7.1** [Build range tree] build a max-range-tree over the points $\{p^{(y)} | y \in Y\}$, where $p_i^{(y)} = d(z, y) - d(z_i, y)$, and $f(p^{(y)} = d(z, y)$.
- E7.2** [For each $x \in X$] Query the range tree with the box $lo = \{-\infty, \dots, -\infty\}$, $hi = \{d(x, z_1) - d(x, z), \dots, d(x, z_k) - d(x, z)\}$, and save the result in Q , which is $\max_{y \in Y \cup B} f(p^{(y)})$.
- E7.3** [Eccentricity] Set $e(x) = \max(e(x), Q + d(x, z))$.
- E8** [Flip] Do Step E7 with X and Y flipped.
- E9** [Recurse] on $E(G[X \cup Z], X \cup Z)$ and $E(G[Y \cup Z], Y \cup Z)$ and save the results in the vectors e_X and e_Y , where $G[X \cup Z]$ means that G is filtered to only contain vertices in $X \cup Z$.
- E10** [Update] For $x \in X$, update $e(x) = \max(e(x), e_X(x))$, for $y \in Y$, update $e(y) = \max(e(y), e_Y(y))$
- E11** Return the vector e .
-

Lemma 3. *Algorithm E correctly computes the eccentricity of every vertex of its input graph in time $n^{1+\epsilon}k^2(R(n, k) + k4^k)$, for every $\epsilon > 0$, if it takes $nR(n, k)$ time to create a k -dimensional range tree with n points, and it takes $R(n, k)$ time to query such range tree.*

Proof. To see correctness, we argue that each distance is inspected. Each $e(z)$, where $z \in Z$, is correct since these are found by Dijkstra's algorithm. For $e(x)$ either the eccentricity is inside $X \cup Z$, in that case it is found by the recursive call, by induction on the size of the graph. However this path can go through Z , into Y and then back to Z again, which is the reason for the shortcut edges. Otherwise its eccentricity is inside Y , and then the path to the eccentricity goes through some $z \in Z$. Step E7.2 and E7.3 calculates the maximal shortest path from x to Y , via z for each $z \in Z$, and updates $e(x)$ to the maximal one. The argument for each $y \in Y$ is similar.

Denote the running time of the algorithm as $T(n, k)$. The steps in the algorithm takes the following time:

- E1 $O(n^2 \log n)$ for Algorithm D, which is $O(k4^k)$, for $n < 2^k$.
- E2 $O(nk^2)$ for finding the centroid, since there are atmost one bag per edge in the graph, and each bag requiers k work.
- E3 $O(n)$, we might have very many bags leading out from the centroid bag, but at most n .
- E4 $O(kn \log n)$ Because of k Dijkstra computations.
- E5 $O(k^2)$ Adding edges between each pair of verticies of Z .
- E6 $O(nk)$ Max in k vectors of size n and allocating vector of size n .
- E7 $O(nkR(n, k))$ build k k -dimensional range trees, and query each one n times.
- E8 $O(nkR(n, k))$ same as E7.
- E9 $T(s, k) + T(n - s + k, k)$, two recursive calls.
- E10 $O(n)$ updates n values.
- E11 $O(1)$.

The dominating factor is $O(nkR(n, k))$. We get the following recurrence relation: Let $S(n, k) = O(kR(n, k) + k4^k)$. Let $T(n, k)$ denote the running time of Algorithm E, in that case:

$$T(n, k) \leq \begin{cases} O(k4^k), & \text{if } n < 2^k; \\ n \cdot S(n, k) + T(|X \cup Z|, k) + T(n - |X \cup Z| + k, k), & \text{otherwise.} \end{cases} \quad (2.1)$$

We will show

$$T(n, k) \leq S(n, k) \cdot 10n \log n, \quad (2.2)$$

which is true if $n < 2^k$, since $S(n, k)$ alone dominates $O(k4^k)$. Now do induction on n . Let $s = |X \cup Z|$, and $r = n - s + k$, and apply (2.2) to (2.1):

$$\begin{aligned} T(n, k) &\leq nS(n, k) + T(s, k) + T(r, k) \leq \\ &\leq nS(n, k) + 10(S(s, k) \cdot s \log s + S(r, k) \cdot r \log r) \leq \\ &\leq S(n, k) \cdot (n + 10s \log s + 10r \log r). \end{aligned} \quad (2.3)$$

We know that $n/3 \leq s < 2n/3$, which means that both s and r are bounded by $t = 2n/3 + k$. Applying these bounds of s and r to (2.3) gives us

$$\begin{aligned} n + 10(s \log s + r \log r) &\leq \\ &\leq n + 10(s \log t + (n - s + k) \log t) = \\ &= n + 10(n \log t + k \log t) \leq \\ &\leq n + 10(n \log t + k \log n). \end{aligned} \quad (2.4)$$

Step E1 ensures that $3k \leq n/2$, since $2^k/2 > 3k$ for $k \geq 5$, so we get:

$$t = \frac{2n}{3} + k \leq \frac{5n}{6}.$$

Then $\log t \leq \log \frac{5n}{6} = \log n - \log \frac{6}{5}$. Applying this to (2.4) gives us:

$$n + 10(n \log t + k \log n) \leq n + 10(n \log n - n \log \frac{6}{5} + k \log n) \leq 10n \log n.$$

Step E1 ensures $2^k \leq n$, taking the logarithm on both sides gives $k \leq \log n$, which is applied in the last step, $10k \log n + n \leq 10 \log^2 n + n \leq 10 \cdot n \cdot 0.18 \leq 10n \ln \frac{6}{5}$, for $n > 500$. Now substitute back $1 + s \log s + r \log r \leq n \log n$ into (2.3), and we get

$$T(n, k) \leq S(n, k) \cdot (n + 10(s \log s + r \log r)) \leq S(n, k) \cdot 10n \log n.$$

Since $10n \log n$ grows slower than $n^{1+\epsilon}$ for all $\epsilon > 0$,

$$T(n, k) \leq n^{1+\epsilon} S(n, k),$$

which proves Lemma 3.

2.4 Orthogonal Range Searching

In this section a formal definition of the data structure Range Tree is provided, together with analysis of running time of construction and query algorithms. This section is taken as is from the paper. Please think about \oplus as the max operator.

2.4.1 Preliminaries

Let P be a set of d -dimensional points. We will view $p \in P$ as a vector $p = (p_1, \dots, p_d)$.

A commutative *monoid* is a set M with an associative and commutative binary operator \oplus with identity. The reader is invited to think of M as the integers with $-\infty$ as identity and $a \oplus b = \max\{a, b\}$.

Let $f: P \rightarrow M$ be a function and define for each subset $Q \subseteq P$

$$f(Q) = \bigoplus \{f(q) : q \in Q\}$$

with the understanding that $f(\emptyset)$ is the identity in M .

2.4.2 Range Trees

Consider dimension $i \in \{1, \dots, d\}$ and enumerate the points in Q as $q^{(1)}, \dots, q^{(r)}$ such that $q_i^{(j)} \leq q_i^{(j+1)}$, for instance by ordering after the i th coordinate and breaking ties lexicographically. Define $\text{med}_i(Q)$ to be the *median* point $q^{(\lceil r/2 \rceil)}$, and similarly the $\text{min}_i(Q) = q^{(1)}$ and $\text{max}_i(Q) = q^{(r)}$. Set

$$Q_L = \{q^{(1)}, \dots, q^{(\lceil r/2 \rceil)}\}, \quad Q_R = \{q^{(1+\lceil r/2 \rceil)}, \dots, q^{(r)}\}. \quad (2.5)$$

For $i \in \{1, \dots, d\}$, the *range tree* $R_i(Q)$ for Q is a node x with the following attributes:

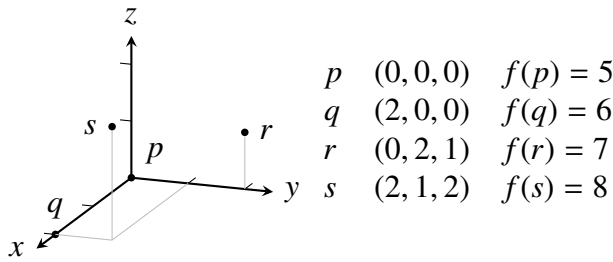


Figure 2.5: Four points in three dimensions. With the monoid (\mathbf{Z}, \max) we have $f(\{p, r, s\}) = 8$.

- $L[x]$, a reference to the range tree $T_i(Q_L)$, often called the *left child* of x .
- $R[x]$, a reference to the range tree $T_i(Q_R)$, often called the *right child* of x .
- $D[x]$, a reference to the range tree $T_{i+1}(Q)$, often called the *secondary, associate, or higher-dimensional* structure. This attribute only exists for $i < d$.
- $l[x] = \min_i(Q)$.
- $r[x] = \max_i(Q)$.
- $f[x] = f(Q)$. This attribute only exists for $i = d$.

In Figure 2.6 an example of a range tree is displayed, that contains the four points in Figure 2.5.

Construction Constructing a range tree for T is a straightforward recursive procedure:

► **Algorithm C** Given integer $i \in \{1, \dots, d\}$ and a list Q of points, this algorithm constructs the range tree $R_i(Q)$ with root x .

C1 [Base case $Q = \{q\}$.] Recursively construct $D[x] = T_{i+1}(Q)$ if $i < d$, otherwise set $f[x] = f(q)$. Set $l[x] = r[x] = q_i$. Return x .

C2 [Find median.] Determine $q = \text{med}_i Q$, $l[x] = \min_i(Q)$, $r[x] = \max_i(Q)$.

C3 [Split Q .] Let Q_L and Q_R as given by (2.5), note that both are nonempty.

C4 [Recurse.] Recursively construct $L[x] = R_i(Q_L)$ from Q_L . Recursively construct $R[x] = R_i(Q_R)$ from Q_R . If $i < d$ then recursively construct $D[x] = T_{i+1}(Q)$. If $i = d$ then set $f[x] = f[L[x]] \oplus f[R[x]]$.

The data structure can be viewed as a collection of binary trees whose nodes x represent various subsets P_x of the original point set P . In the interest of analysis, we now introduce a scheme for naming the individual nodes x , and thereby also the subsets P_x . Each node x is identified by a string of letters from $\{L, R, D\}$ as follows. Associate with x a set of points, often called the *canonical subset* of x , as follows. For the empty string ϵ we set $P_\epsilon = P$. In general, if $Q = P_x$ then $P_{xL} = Q_L$, $P_{xR} = Q_R$ and $P_{xD} = Q$. The strings over

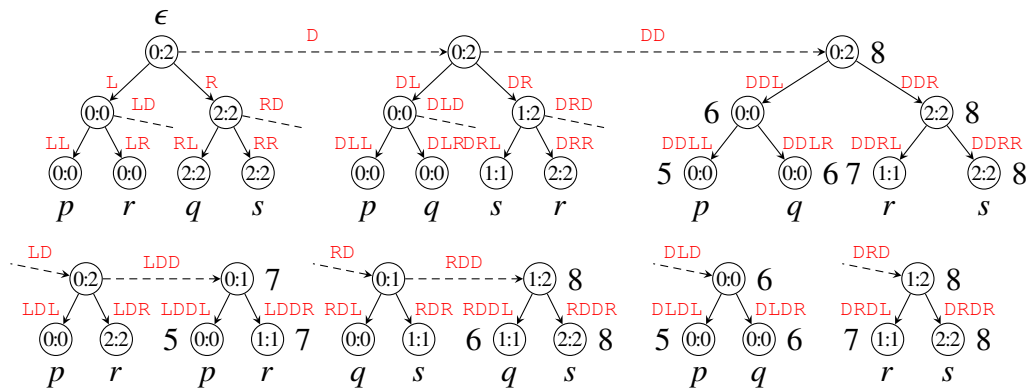


Figure 2.6: Part of the range tree for the points from Fig. 2.5. The label of node x appears in red on the arrow pointing to x . Nodes contain $l[x]:r[i]$. The references $L[x]$ and $R[x]$ appear as children in a binary tree using usual drawing conventions. The reference $D[x]$ appears as a dashed arrow (possibly interrupted); the placement on the page follows no other logic than economy of layout and readability. References $D[x]$ from leaf nodes, such as $D[LL]$ leading to node LLD , are not shown; this conceals 12 single-node trees. The ‘3rd-dimensional nodes,’ whose names contain two Ds, show the values $f[x]$ next to the node. To ease comprehension, leaf nodes are decorated with their canonical subset, which is a singleton from $\{p, q, r, s\}$. The reader can infer the canonical subset for an internal node as the union of leaves of the subtree; for instance, $P_{DR} = \{r, s\}$. However, note that these point sets are *not* explicitly stored in the data structure.

$\{L, R, D\}$ can be understood as uniquely describing a path through in the data structure; for instance, L means ‘go left, i.e., to the left subtree, the one stored at $L[x]$ ’ and D means ‘go to the next *dimension*, i.e., to the subtree stored at $D[x]$.’ The name of a node now describes the unique path that reaches it.

Lemma 4. *Let $n = |P|$. Algorithm C computes the d -dimensional range tree for P in time linear in $nd \cdot \binom{n+d}{d}$.*

Proof. We run Algorithm C on input P and $i = 1$.

Disregarding the recursive calls, the running time of algorithm C on input i and Q is dominated by Steps C2 and C3, i.e., splitting Q into two sets of equal size. It is known that this task can be performed in time linear in $|Q|$ [3]. Thus, the running time for constructing $R_i(Q)$ is linear in $|Q|$ plus the time spent in recursive calls.

This means that we can bound the running time for constructing $T_1(P)$ by bounding sizes of the sets P_x associated with every node x in the data structure. If for a moment X denotes the set of all these nodes then we want to bound

$$\sum_{x \in X} |P_x| = \sum_{x \in X} |\{p \in P: p \in P_x\}| = \sum_{p \in P} |\{x \in X: p \in P_x\}|.$$

Thus, we need to determine, for given $p \in P$, the number of subsets P_x in which p appears. By construction, there are fewer than d occurrences of D in x . Moreover, if x contains more than h occurrences of either L or R then P_x is empty. Thus, x has at most $h + d$ letters. For two different strings x and x' that agree on the positions of D, the sets P_x and $P_{x'}$ are

disjoint, so p appears in at most one of them. We conclude that the number of sets P_x such that $p \in P_x$ is bounded by the number of ways to arrange fewer than d many Ds and at most h non-Ds. Using the identity $\binom{a+0}{0} + \dots + \binom{a+b}{b} = \binom{a+b+1}{b}$ repeatedly, we compute

$$\begin{aligned} \sum_{i=0}^{d-1} \sum_{j=0}^h \binom{i+j}{j} &= \sum_{i=0}^{d-1} \binom{i+h+1}{h} = \sum_{i=0}^{d-1} \binom{i+h+1}{i+1} = \\ &(-1) + \sum_{i=0}^d \binom{i+h}{i} = \binom{h+d+1}{d} - 1 = \frac{h+d+1}{h+1} \binom{h+d}{d} - 1 \leq d \binom{d+h}{d}. \end{aligned}$$

The bound follows from aggregating this contribution over all $p \in P$.

Search. In this section, we fix two sequences of integers l_1, \dots, l_d and r_1, \dots, r_d describing the *query box* B given by

$$B = [l_1, r_1] \times \dots \times [l_d, r_d].$$

► **Algorithm Q** Given integer $i \in \{1, \dots, d\}$, a query box B as above and a range tree $R_i(Q)$ with root x for a set of points Q such that every point $q \in Q$ satisfies $l_j \leq q_j \leq r_j$ for $j \in \{1, \dots, i-1\}$. This algorithm returns $\bigoplus \{f(q) : q \in Q \cap B\}$.

Q1 [Empty?] If the data structure is empty, or $l_i > r[x]$, or $l[x] > r_i$, then return the identity in the underlying monoid M .

Q2 [Done?] If $i = d$ and $l_d \leq \min_d[x]$ and $\max_d[x] \leq r_d$ then return $f[x]$.

Q3 [Next dimension?] If $i < d$ and $l_i \leq l[x]$ and $r[x] \leq r_i$ then query the range tree at $D[x]$ for dimension $i+1$. Return the resulting value.

Q4 [Split.] Query the range tree $L[x]$ for dimension i ; the result is a value f_L . Query the range tree $R[x]$ for dimension i ; the result is a value f_R . Return $f_L \oplus f_R$. \square

To prove correctness, we show that this algorithm is correct for each point set $Q = P_x$.

Lemma 5. Let $i = D(x) + 1$, where $D(x)$ is the number of Ds in x . Assume that P_x is such that $l_j \leq p_j \leq r_j$ for all $j \in \{1, \dots, i-1\}$ for each $p \in P_x$. Then the query algorithm on input x and i returns $f(B \cap P_x)$.

Proof. Backwards induction in $|x|$.

If $|x| = h + d$ then P_x is the empty set, in which case the algorithm correctly returns the identity in M .

If the algorithm executes Step Q2 then B is satisfied for all $q \in P_x$, in which case the algorithm correctly returns $f[x] = f(P_x)$.

If the algorithm executes Step Q3 then B satisfies the condition in the lemma for $i+1$, and the number of Ds in P_{xD} is $i+1$, and $D[x]$ store the $(i+1)$ th range tree for P_{xD} . Thus, by induction the algorithm returns $f(P_{xD} \cap B)$, which equals $f(P_x \cap B)$ because $P_{xD} = P_x$.

Otherwise, by induction, $f_L = f(P_{xL} \cap B)$ and $f_R = f(P_{xR} \cap B)$. Since $P_{xL} \cup P_{xR} = P_x$, we have $f(P_x \cap B) = f((P_{xL} \cap B) \cup (P_{xR} \cap B)) = f_L \oplus f_R$.

Lemma 6. *If x is the root of the range tree for P then on input $i = 1$, x , and B , the query algorithm returns $f(P \cap B)$ in time linear in $2^d \binom{n+d}{d}$.*

Proof. Correctness follows from the previous lemma.

For the running time, we first observe that the query algorithm does constant work in each visited node. Thus it suffices to bound the number of visited nodes as

$$2^d \binom{h+d}{d} \quad (d \geq 1, h \geq 0). \quad (2.6)$$

We will show by induction in d that (2.6) holds for every call to a d -dimensional range tree for a point set P_x , where $h = \lceil \log |P_x| \rceil$. The two easy cases are Q1 and Q2, which incur no additional nodes to be visited, so the number of visited nodes is 1, which is bounded by (2.6). Step Q3 leads to a recursive call for a $(d-1)$ -dimensional range tree over the same point set $P_{xD} = P_x$, and we verify

$$1 + 2^{d-1} \binom{h+d-1}{d-1} \leq 2^d \binom{h+d}{d}.$$

The interesting case is Step Q4. We need to follow two paths from x to the leaves of the binary tree of x . Consider the leaves l and r in the subtree rooted at x associated with the points $\min_i(P_x)$ and $\max_i(P_x)$. We describe the situation of the path Y from l to x ; the other case is symmetrical. At each internal node $y \in Y$, the algorithm chooses Step Q4 (because $l_i \geq l[y]$). There are two cases for what happens at yL and yR . If $l_i \leq \text{med}_i(P_y)$ then P_{yR} satisfies $l_i \leq \min_i(P_{yR}) \leq r_i$, so the call to yR will choose Step Q3. By induction, this incurs $2^{d-1} \binom{d-1+i}{d-1}$ visits, where i is the height of y . In the other case, the call to yL will choose Step Q1, which incurs no extra visits. Thus, the number of nodes visited on the left path is at most

$$h + \sum_{i=0}^{h-1} 2^{d-1} \binom{d-1+i}{d-1},$$

and the total number of nodes visited is at most twice that:

$$2h + 2^d \sum_{i=0}^{h-1} \binom{d-1+i}{d-1} \leq 2^d \sum_{i=0}^h \binom{d-1+i}{d-1} = 2^d \binom{d+h}{d}.$$

2.4.3 Remark on Range Trees

The textbook analysis of range trees, and similar d -dimensional spatial algorithms and data structures sets up a recurrence relation like

$$r(n, d) = 2r(n/2, d) + r(n, d-1),$$

for the construction and

$$r(n, d) = \max\{r(n/2, d), r(n, d-1)\},$$

for the query time. One then observes that $n \log^d n$ and $\log^d n$ are the solutions to these recurrences. This analysis goes back to Bentley's original paper [2].

Along the lines of the previous section, one can show that the functions $n \cdot \binom{n+d}{d}$ and $\binom{n+d}{d}$ solve these recurrences as well. A detailed derivation can be found in [13], which also contains combinatorial arguments of how to interpret the binomial coefficients in the context of spatial data structures. A later paper of Chan [6] also takes the recurrences as a starting point, and observes an asymptotically improved solution for the related question of dominance queries.

2.5 Asymptotics

Combining the analysis of range trees into $n^{1+\epsilon}(k^2R(n, k) + k4^k)$ gives us $n^{1+\epsilon}(k^22^k\binom{k+h}{h} + k4^k)$ where $h = \lceil \log_2 n \rceil$.

To show that the complexity of Algorithm E is bounded by $n^{1+\epsilon} \exp O(k)$ all that remains is to show that $\binom{k+h}{k}$ is bounded by $n^\epsilon \exp O(k)$ for all n and k and $\epsilon > 0$.

We consider two cases. First assume $d < \epsilon h$ for all $\epsilon > 0$. From Stirling's formula (see property 2 inside Das's note on binomial coefficients [7]) we know $\binom{a}{b} \leq \left(\frac{ea}{b}\right)^k$, so

$$\binom{d+h}{d} < \binom{(1+\epsilon)h}{\epsilon h} \leq \left(\frac{e(1+\epsilon)h}{\epsilon h}\right)^{\epsilon h} < \left(\frac{e(1+\epsilon)}{\epsilon}\right)^{2\epsilon \log n} = n^{2\epsilon \log(e(1+\epsilon)\epsilon^{-1})} = n^{o(1)},$$

where the last expression uses that $\epsilon \mapsto 2\epsilon \log e(1+\epsilon)\epsilon^{-1}$ is a monotone increasing function in the interval $(0, \frac{1}{2}]$.

On the other hand, if $d \geq ch$ for some constant c , we have

$$\binom{d+h}{d} \leq \binom{(1+1/c)d}{d} < 2^{(1+1/c)d} = \exp O(d).$$

Which concludes that the diameter can be found in time $n^{1+\epsilon} \exp O(k)$, for all $\epsilon > 0$.

2.6 Wiener Index

To compute the Wiener index, some small modifications to Algorithm E are needed. We modify the range trees such that instead of returning the maximum $f(p)$ for points inside the box, it returns a tuple N, S , the number of points inside the box, and the sum of the points' values inside the box. For a given vertex $x \in X$, the contribution to the Wiener index of distances from x to all $y \in Y$ through z . Remember the definition of $P = \{p^{(y)}\}$ for each $y \in Y$ from Section 2.3. $P \cap B$ denoted the points inside the box B defined by x and z . Then,

$$\begin{aligned} \sum_{p^{(y)} \in P \cap B} d(x, y) &= \sum_{p^{(y)} \in P \cap B} (d(x, z) + d(z, y)) = \\ &= d(x, z) \cdot |\{p^{(y)} \in P \cap B\}| + \sum_{p^{(y)} \in P \cap B} d(z, y) = \\ &= d(x, z) \cdot N + S \end{aligned} \tag{2.7}$$

Each leaf corresponding to a vertex y in the Range Tree of vertex z stores $(1, d(z, y))$. Let the Range Tree operation \oplus on two such values be $(n_1, s_1) \oplus (n_2, s_2) = (n_1 + n_2, s_1 + s_2)$, with $(0, 0)$ as neutral element.

Two problems arise. First, the distances from x to vertices in Y may be double counted, since there may exist a shortest path from x to y going through both z_i and z_j , as shown in Figure 2.4. Secondly, the vertices in Z belong to both the left and the right subgraph, meaning that distances between vertices in Z are counted twice.

The first problem can be fixed by changing the queries to the Range Trees. Instead of only requiring that z is on the shortest path from x to y , we add the requirement that z also is the vertex with the smallest index within Z that is on a shortest path from x to y . By doing this we guarantee that the distance between x and y only is counted once. This is accomplished by, for z_j , changing some of the coordinates in the hi -point of the query in Step E7.3. Specifically hi_i is changed to $hi_i = d(x, z_i) - d(x, z_j) - 1$ if $i < j$. This implies that $d(x, z_j) + d(z_j, y) < d(x, z_i) + d(z_i, y)$, for $i < j$.

Secondly, calculating the Wiener index from the results is a matter of set theory. Let $X' = X - Z, Y' = Y - Z$. We will show that $wien(G) = wien(G[X \cup Z]) + wien(G[Y \cup Z]) + \sum_{x \in X', y \in Y'} d(x, y) - wien(G[Z])$. We introduce the new operator \otimes and define it as follows. Let $A \otimes B = \sum_{a \in A, b \in B} d(a, b)$. Then $wien(G[X]) = \frac{1}{2}X \otimes X$. Also note that $(A \cup C) \otimes B = A \otimes B + C \otimes B - (A \cap C) \otimes B$, and that $A \otimes B = B \otimes A$. The reader is welcome to think about \otimes as multiplication and \cup as addition, given that the sets the union is taken over are disjoint, i.e. has the empty set as intersection.

Using these calculation rules its possible to derive

$$\begin{aligned}
& 2\left(wien(G[X \cup Z]) + wien(G[Y \cup Z]) + \sum_{x \in X', y \in Y'} d(x, y) - wien(G[Z])\right) = \\
& (X \cup Z) \otimes (X \cup Z) + (Y \cup Z) \otimes (Y \cup Z) + 2X' \otimes Y' - Z \otimes Z = \\
& (X' \cup Z) \otimes (X' \cup Z) + (Y' \cup Z) \otimes (Y' \cup Z) + 2X' \otimes Y' - Z \otimes Z = \\
& X' \otimes X' + 2X' \otimes Z + Z \otimes Z + Y' \otimes Y' + 2Y' \otimes Z + Z \otimes Z + 2X' \otimes Y' - Z \otimes Z = \\
& X' \otimes X' + Y' \otimes Y' + Z \otimes Z + 2(X' \otimes Z + Y' \otimes Z + X' \otimes Y') = \\
& (X' \cup Y' \cup Z) \otimes (X' \cup Y' \cup Z) = \\
& 2 wien(G[X' \cup Y' \cup Z]) = 2 wien(G)
\end{aligned}$$

Both of these arguments closely follows Cabello and Knauer's arguments [5]. With these modifications to Algorithm E, the Wiener index is computed. Hereafter we will refer to this algorithm as Algorithm W.

Chapter 3

Experimental demonstration

To evaluate the theoretical results We have programmed the algorithm for Wiener index, referred to as Algorithm W, it is compared with the naive Algorithm D, that is modified to compute Wiener Index as well. The code lives inside the same repository as this report, which can be found at <https://gitlab.com/exoji2e/exjobb>.

3.1 Running Times

I have measured how the running times for Algorithm D and Algorithm W to compute Wiener index depend upon the graph size and treewidth. The language of choice has been Java, and I've generated test data myself. However I have not implemented the computation of tree-decomposition, instead I make sure to generate graphs and their tree-decomposition at the same time.

3.1.1 Generation of test data

I have generated randomized graphs and their tree-decompositoins with n vertices and treewidth k in the following way:

Algorithm G.

Generates a graph with n vertices and treewidth k .

G1 Generate a pre-tree T of size n/k . By selecting the parent of vertex i to be a random vertex of the vertices $\{0 \dots i - 1\}$.

G2 Create G by replacing each vertex in T with a fully connected subgraph (clique) of size k . For each pair of neighboring cliques, connect two random vertices, one from each clique.

G3 A tree-decomposition of size k can now be found by inserting each clique in a bag. Insert each edge between cliques inside one bag each, then connect these two to the clique bags their vertices belong to.

3.1.2 Measurements

We have in total done 4 test suites, with 4–14 data points in each, each data point is an average of 10 runs. In the first test suite, see figure 3.1 we fix the treewidth to 2, and let the graph size vary by a factor 2 from 16 to 131 072. In the second test suite, see figure 3.2 the only difference to the first suite is that the treewidth is fixed to 4, and the maximum graph size is 32 768. In the third test suite, see figure 3.3 we instead fix the graph size to 1000, and let the treewidth vary from 2 to 7 in steps of 1. In the fourth test suite, see figure 3.4 we increase the graph size to 10000, and lower the maximum treewidth to 5. We ran this on a laptop with an Intel i7 processor with 4 cores, and gave the Java process 8GB of RAM.

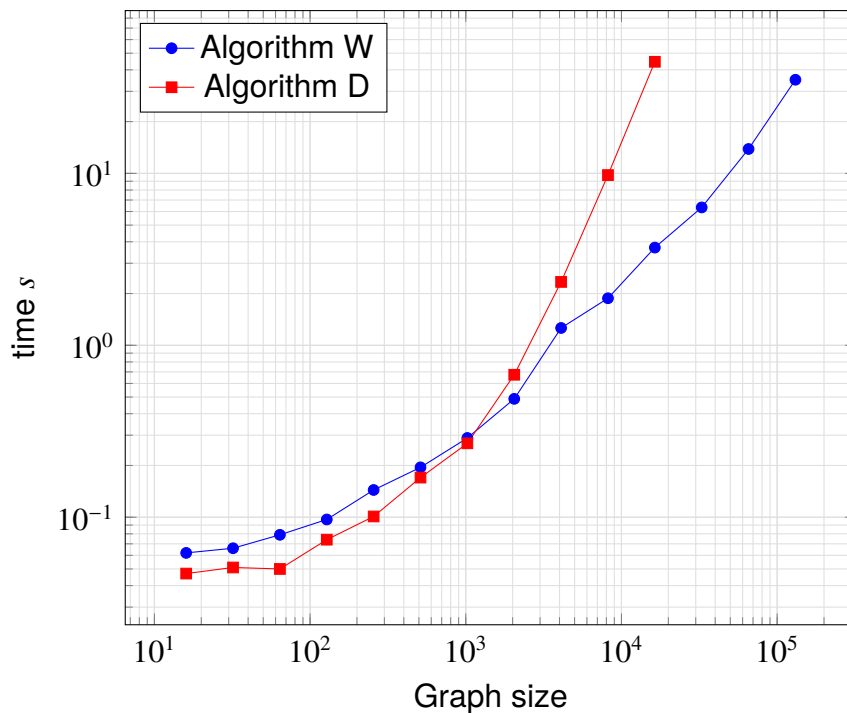


Figure 3.1: Running times of Algorithm D and Algorithm W for treewidth = 2 in a log-log diagram, dependent upon the graph size.

3.2 Discussion

In Figure 3.1 and 3.2 we see a slope very close to 2 for Algorithm D, which corresponds to quadratic dependence.

For $k = 2$, Algorithm W performs significantly better than Algorithm D. For $k = 4$, the log factors of Algorithm W become more pronounced, hence any gain in running time

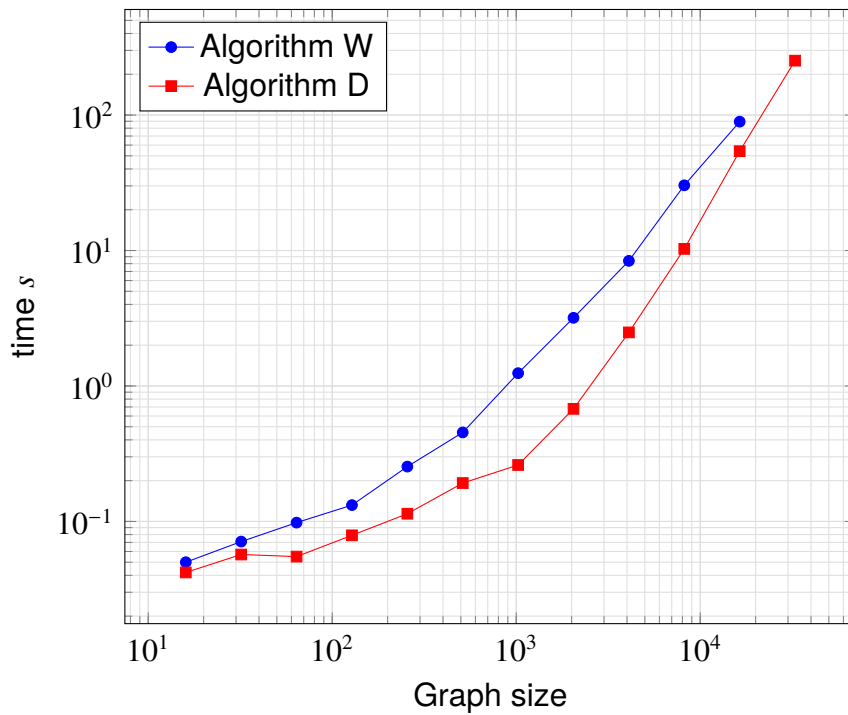


Figure 3.2: Running times of Algorithm D and Algorithm W for treewidth = 4 in a log-log diagram, dependent upon the graph size.

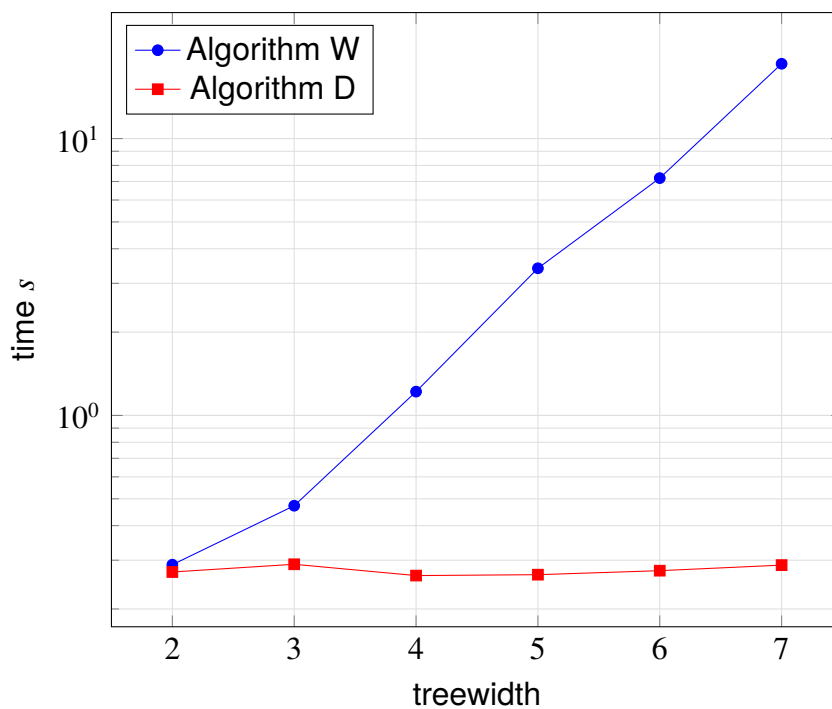


Figure 3.3: Running times of Algorithm D and Algorithm W on graphs of size 1000 in a lin-log diagram, dependent upon the treewidth.

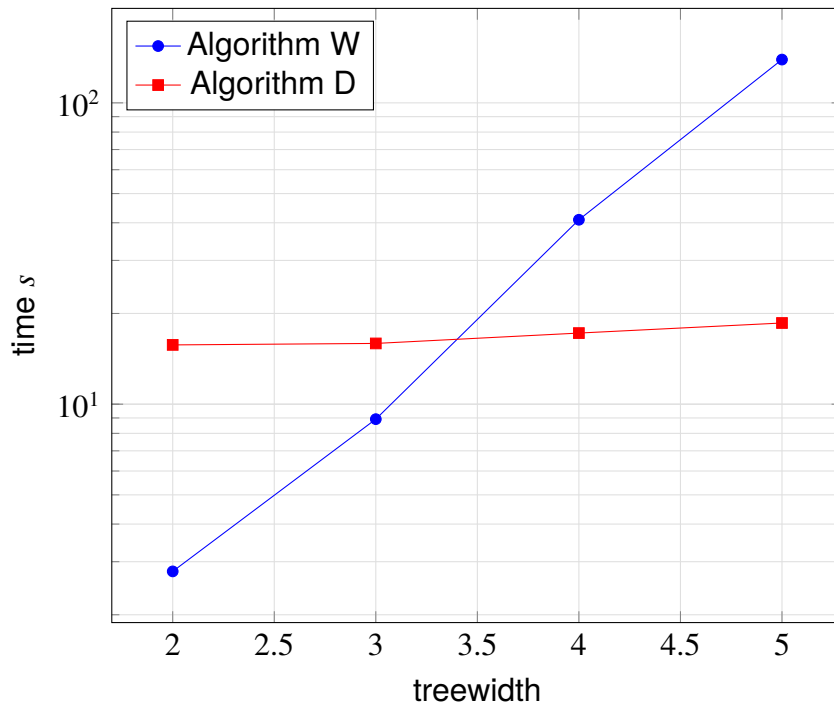


Figure 3.4: Running times of Algorithm D and Algorithm W on graphs of size 10 000 in a lin-log diagram, dependent upon the treewidth.

does not show when $n \leq 20\,000$. Algorithm W goes out of memory for some instances of size $k = 4, n = 2^{15} = 32\,768$, so this datapoint is excluded from the plot. However, for some instances it was faster than Algorithm D.

Figure 3.3, and especially 3.4 shows the exponential dependence on k : the plots are linear in the lin-log diagrams and linear growth in the plot corresponds to exponential growth.

These results are consistent with the theory presented in Chapter 2. Because of the exponential dependence on k , we can only increase n as far as needed for Algorithm W to overtake Algorithm D for small values of k . As expected, to benefit from Algorithm W as k grows, n has to grow very rapidly.

A word of caution, while the graphs generated have treewidth k , they might have smaller separators, that is, they are not necessarily the worst case instances for our algorithm. Worst case instances where the separators are as close to the treewidth as possible, are nontrivial to generate. However, this might be a cause for optimism, since unverified rumors indeed states that graphs in the wild have smaller separators than treewidth. Of course we will not fall into the trap of conjecturing anything about this.

It is now more accessible to implement¹ a treewidth-based diameter or Wiener index algorithm than it was before. We confirm the belief in Abboud, et al., [1] that those algorithms are competitive compared to the naive algorithms, however we reject the belief that they are easy to implement, since we skipped generating an approximate tree decomposition in lack of time, and they are very complicated compared to the naive Algorithm D.

¹In fact the code is open source, and available at <https://gitlab.com/exoji2e/exjobb>.

Chapter 4

Conclusions and Future Work

In this thesis we have shown that the problems of calculating the graph metrics diameter, radius, eccentricities and Wiener index can be computed in time $O(n^{1+\epsilon} \exp O(k))$, where k is the treewidth, closing an open gap in [1]. In the process, we rejected Husfeld's invalid and misguided conjecture [10]. We have presented both theoretical proofs and supporting experiments. Also it is possible to implement algorithms for these metrics with a low constant, meaning that they are fast for reasonably large graphs. The algorithm is competitive with the naive implementations, given enough specific graphs, i.e., extremely small treewidth.

It remains open whether it is possible to calculate diameter in time $O(n \exp d)$, or if it is impossible under current hypotheses.

Implementing this for a real application, for example the molecule modeling mentioned in the introduction, is out of scope for this thesis, and is left as future work.

Other interesting research directions include investigating the possibility of finding small separators without a tree decomposition, which should lower the constant significantly. Future work also includes extending the theory to work for directed graphs, or using another parameterization. Cabello and Knauer [5] mentions for example the Dilation number.

But we will save the future for another day, now I will go out and play with my small (motor)cycle.



Bibliography

- [1] Amir Abboud, Virginia Vassilevska Williams, and Joshua R. Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the Twenty-Seventh Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, Va, USA, January 10–12, 2016*, pages 377–391. SIAM, 2016.
- [2] Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.
- [3] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.
- [4] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In Timo Lepistö and Arto Salomaa, editors, *Automata, Languages and Programming*, pages 105–118, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.
- [5] Sergio Cabello and Christian Knauer. Algorithms for bounded treewidth with orthogonal range searching. *Comput. Geom.*, 42(9):815–824, 2009.
- [6] Timothy M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008.
- [7] Shagnik Das. A brief note on estimates of binomial coefficients. <http://page.mi.fu-berlin.de/shagnik/notes/binomials.pdf>. Fetched 7th of June 2018.
- [8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [9] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *25th Annual Symposium on Foundations of Computer Science, 1984.*, pages 338–346, Oct 1984.

- [10] Thore Husfeldt. Computing Graph Distances Parameterized by Treewidth and Diameter. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:11, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- [11] Russel Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.
- [12] Bojan Mohar and Tomaž Pisanski. How to compute the wiener index of a graph. *Journal of Mathematical Chemistry*, 2(3):267–277, Jul 1988.
- [13] Louis Monier. Combinatorial solutions of multidimensional divide-and-conquer recurrences. *J. Algorithms*, 1(1):60–74, 1980.
- [14] Donald J. Rose. On simple characterizations of k-trees. *Discrete Mathematics*, 7(3):317 – 322, 1974.
- [15] Harry Wiener. Structural determination of paraffin boiling points. *Journal of the American Chemical Society*, 69(1):17–20, 1947. PMID: 20291038.

Appendices


Multivariate Analysis of Orthogonal Range Searching and Graph Distances Parameterized by Treewidth

Karl Bringmann

Max-Planck-Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
kbringma@mpi-inf.mpg.de

Thore Husfeldt¹

BARC, IT University of Copenhagen, Denmark, and Lund University, Sweden.
thore@itu.dk

 <https://orcid.org/0000-0001-9078-4512>

Måns Magnusson

Department of Computer Science, Lund University, Sweden
mans.magnusson.888@student.lu.se

Abstract

We show that the eccentricities, diameter, radius, and Wiener index of an undirected n -vertex graph with nonnegative edge lengths can be computed in time $O(n \cdot \binom{k + \lceil \log n \rceil}{k} \cdot 2^k k^2 \log n)$, where k is the treewidth of the graph. For every $\epsilon > 0$, this bound is $n^{1+\epsilon} \exp O(k)$, which matches a hardness result of Abboud, Vassilevska Williams, and Wang (SODA 2015) and closes an open problem in the multivariate analysis of polynomial-time computation. To this end, we show that the analysis of an algorithm of Cabello and Knauer (Comp. Geom., 2009) in the regime of non-constant treewidth can be improved by revisiting the analysis of orthogonal range searching, improving bounds of the form $\log^d n$ to $\binom{d + \lceil \log n \rceil}{d}$, as originally observed by Monier (J. Alg. 1980).

We also investigate the parameterization by vertex cover number.

2012 ACM Subject Classification Theory of computation \rightarrow Shortest paths, Parameterized complexity and exact algorithms, Computational geometry. Mathematics of computing \rightarrow Paths and connectivity problems.

Keywords and phrases Diameter, radius, Wiener index, orthogonal range searching, treewidth, vertex cover number.

Acknowledgements We thank Amir Abboud and Rasmus Pagh for useful discussions.

1 Introduction

Pairwise distances in an undirected, unweighted graph can be computed by performing a graph exploration, such as breadth-first search, from every vertex. This straightforward procedure determines the diameter of a given graph with n vertices and m edges in time $O(nm)$. It is surprisingly difficult to improve upon this idea in general. In fact, Roditty and Vassilevska Williams [16] have shown that an algorithm that can distinguish between diameter 2 and 3 in an undirected sparse graph in subquadratic time refutes the Orthogonal Vectors conjecture.

¹ Swedish Research Council grant VR-2016-03855 and Villum Foundation grant 16582.

However, for very sparse graphs, the running time becomes linear. In particular, the diameter of a tree can be computed in linear time $O(n)$ by a folklore result that traverses the graph twice. In fact, an algorithm by Cabello and Knauer shows that for constant treewidth $k \geq 3$, the diameter (and other distance parameters) can be computed in time $O(n \log^{k-1} n)$, where the Landau symbol absorbs the dependency on k as well as the time required for computing a tree decomposition. The question raised in [1] is how the complexity of this problem grows with the treewidth of the graph. We show the following result:

► **Theorem 1.** *The eccentricities, diameter, radius, and Wiener index of a given undirected n -vertex graph G of treewidth $\text{tw}(G)$ and nonnegative edge lengths can be computed in time linear in*

$$n \cdot \binom{k + \lceil \log n \rceil}{k} \cdot 2^k k^2 \log n \quad (1)$$

where $k = 5 \text{tw}(G) + 4$.

For every $\epsilon > 0$, the bound (1) is $n^{1+\epsilon} \exp O(\text{tw}(G))$. This improves the dependency on the treewidth over the running time $n^{1+\epsilon} \exp O(\text{tw}(G) \log \text{tw}(G))$ of Abboud, Vassilevska Williams, and Wang [1]. Our improvement is tight in the following sense. Abboud *et al.* [1] also showed that under the Strong Exponential Time Hypothesis of Impagliazzo, Paturi, and Zane [12], there can be no algorithm that computes the diameter with running time

$$n^{2-\delta} \exp o(\text{tw}(G)) \quad \text{for any } \delta > 0. \quad (2)$$

In fact, this holds under the potentially weaker Orthogonal Vectors conjecture, see [18] for an introduction to these arguments. Thus, under this assumption, the dependency on $\text{tw}(G)$ in Theorem 1 cannot be significantly improved, even if the dependency on n is relaxed from just above linear to just below quadratic. Our analysis encompasses the Wiener index, an important structural graph parameter left unexplored by [1].

Perhaps surprisingly, the main insight needed to establish Theorem 1 has nothing to do with graph distances or treewidth. Instead, we make—or re-discover—the following observation about the running time of d -dimensional range trees:

► **Lemma 2** ([15]). *A d -dimensional range tree over n points supporting orthogonal range queries for the aggregate value over a commutative monoid has query time $O(2^d \cdot B(n, d))$ and can be built in time $O(nd \cdot B(n, d))$, where*

$$B(n, d) = \binom{d + \lceil \log n \rceil}{d}.$$

This is a more careful statement than the standard textbook analysis, which gives the query time as $O(\log^d n)$ and the construction time as $O(n \log^d n)$. For many values of d , the asymptotic complexities of these bounds agree—in particular, this is true for constant d and for very large d , which are the main regimes of interest to computational geometers. But crucially, $B(n, d)$ is always $n^\epsilon \exp O(d)$ for any $\epsilon > 0$, while $\log^d n$ is not.

After Lemma 2 is realised, Theorem 1 follows via divide-and-conquer in decomposable graphs, closely following the idea of Cabello and Knauer [6] and augmented with known arguments [1, 5]. We choose to give a careful presentation of the entire construction, as some of the analysis is quite fragile.

Using known reductions, this implies that the following multivariate lower bound on orthogonal range searching is tight:

► **Theorem 3** (Implicit in [1]). *A data structure for the orthogonal range query problem for the monoid (\mathbf{Z}, \max) with construction time $n \cdot q'(n, d)$ and query time $q'(n, d)$, where*

$$q'(n, d) = n^{1-\epsilon} \exp o(d)$$

for some $\epsilon > 0$, refutes the Strong Exponential Time hypothesis.

We also investigate the same problems parameterized by vertex cover number:

► **Theorem 4.** *The eccentricities, diameter, and radius of a given undirected, unweighted n -vertex graph G with vertex cover number k can be computed in time $O(nk + 2^k k^2)$. The Wiener index can be computed in time $O(nk2^k)$.*

Both of these bounds are $n \exp O(k)$. It follows from [1] that a lower bound of the form (2) holds for this parameter as well.

1.1 Related work

Abboud *et al.* [1] show that given a graph and an optimal tree decomposition, various graph distances can be computed in time $O(k^2 n \log^{k-1} n)$, where $k = \text{tw}(G)$. This bound is $n^{1+\epsilon} \exp O(k \log k)$ for any $\epsilon > 0$. This subsumes the running time for finding an approximate tree decomposition with $k = O(\text{tw}(G))$ from the input graph [5], which is $n \exp O(k)$.

If the diameter in the input graph is constant, the diameter can be computed in time $n \exp O(\text{tw}(G))$ [11]. This is tight in both parameters in the sense that [1] rules out the running time (2) even for distinguishing diameter 2 from 3, and every algorithm needs to inspect $\Omega(n)$ vertices even for treewidth 1. For non-constant diameter Δ , the bound from [11] deteriorates as $n \exp O(\text{tw}(G) \log \Delta)$. However, the construction cannot be used to compute the Wiener index.

The literature on algorithms for graph distance parameters such as diameter or Wiener index is very rich, and we refer to the introduction of [1] for an overview of results directly relating to the present work. A recent paper by Bentert and Nichterlein [2] gives a comprehensive overview of many other parameterisations.

Orthogonal range searching using a multidimensional range tree was first described by Bentley [3], Lueker [14], Willard [17], and Lee and Wong [13], who showed that this data structure supports query time $O(\log^d n)$ and construction time $O(n \log^{d-1} n)$. Several papers have improved this in various ways by factors logarithmic in n ; for instance, Chazelle's construction [8] achieves query time $O(\log^{d-1} n)$.

1.2 Discussion

In hindsight, the present result is a somewhat undramatic resolution of an open problem in that has been viewed as potentially fruitful by many people [1], including the second author [11]. In particular, the resolution has led neither to an exciting new technique for showing conditional lower bounds of the form $n^{2-\epsilon} \exp \omega(k)$, nor a clever new algorithm for graph diameter. Instead, our solution follows the ideas of Cabello and Knauer [6] for constant treewidth, much like in [1]. All that was needed was a better understanding of the asymptotics of bivariate functions, rediscovering a 40-year old analysis of spatial data structures [15] (see the discussion in Sec. 3.3), and using a recent algorithm for approximate tree decompositions [5].

Of course, we can derive some satisfaction from the presentation of asymptotically tight bounds for fundamental graph parameters under a well-studied parameterization. In particular, the surprisingly elegant reductions in [1] cannot be improved. However, as we show in the appendix, when we parameterize by vertex cover number instead of treewidth, we can establish even cleaner and tight bounds without much effort.

Instead, the conceptual value of the present work may be in applying the multivariate perspective on high-dimensional computational geometry, reviving an overlooked analysis for non-constant dimension. To see the difference in perspective, Chazelle's improvement [8] of d -dimensional range queries from $\log^d n$ to $\log^{d-1} n$ makes a lot of sense for small d , but from the multivariate point of view, both bounds are $n^\epsilon \exp \Omega(d \log d)$. The range of relationships between d and n where the multivariate perspective on range trees gives some new insight is when d is asymptotically just shy of $\log n$, see Sec. 2.1.

It remains open to find an algorithm for diameter with running time $n \exp O(\text{tw}(G))$, or an argument that such an algorithm is unlikely to exist under standard hypotheses. This requires better understanding of the regime $d = o(\log n)$.

2 Preliminaries

2.1 Asymptotics

We summarise the asymptotic relationships between various functions appearing in the present paper:

► **Lemma 5.**

$$B(n, d) = O(\log^d n). \quad (3)$$

For any $\epsilon > 0$,

$$B(n, d) = n^\epsilon \exp O(d), \quad (4)$$

$$\log^d n = n^\epsilon \exp \Omega(d \log d), \quad (5)$$

$$\log^d n = n^\epsilon \exp O(d \log d). \quad (6)$$

The first expression shows that $B(n, d)$ is always at least as informative as $O(\log^d n)$. The next two expressions show that from the perspective of parameterised complexity, the two bounds differ asymptotically: $B(n, d)$ depends single-exponentially on d (no matter how small $\epsilon > 0$ is chosen), while $\log^d n$ does not (no matter how *large* ϵ is chosen). Expression (6) just shows that (5) is maximally pessimistic.

Proof. Write $h = \lceil \log n \rceil$. To see (3), consider first the case where $d < h$. Using $\binom{a}{b} \leq a^b/b!$ we see that

$$\binom{d+h}{d} \leq \binom{2h}{d} \leq \frac{(2h)^d}{d!} = \frac{2^d}{d!} h^d = O(\log^d n). \quad (7)$$

Next, if $d \geq h$ then

$$\binom{d+h}{d} = \binom{d+h}{h} \leq \binom{2d}{h} = \frac{2^h}{h!} d^h \leq d^h,$$

provided $h \geq 4$. It remains to observe that $d^h \leq h^d = O(\log^d n)$. Ineed, since the function $\alpha \mapsto \alpha/\ln \alpha$ is increasing for $\alpha \geq e$, we have $h/\ln h \leq d/\ln d$, which implies $\exp(h \ln d) \leq \exp(d \ln h)$ as needed.

For (4), there are two cases. First assume $d < \epsilon h$ for all $\epsilon > 0$. From Stirling's formula we know $\binom{a}{b} \leq \left(\frac{ea}{b}\right)^k$, so

$$\binom{d+h}{d} < \binom{(1+\epsilon)h}{\epsilon h} < \left(\frac{e(1+\epsilon)h}{\epsilon h}\right)^{\epsilon h} < \left(\frac{e(1+\epsilon)}{\epsilon}\right)^{2\epsilon \log n} = n^{2\epsilon \log e(1+\epsilon)\epsilon^{-1}} = n^{o(1)},$$

where the last expression uses that $\epsilon \mapsto 2\epsilon \log e(1+\epsilon)\epsilon^{-1}$ is a monotone increasing function in the interval $(0, \frac{1}{2}]$.

On the other hand, if $d \geq ch$ for some constant c , we have

$$\binom{d+h}{d} \leq \binom{(1+1/c)d}{d} < 2^{(1+1/c)d} = \exp O(d).$$

We turn to (5). Assume that there is a function g such that

$$\log^d n = n^c g(d).$$

Then choose $b > 1$ and consider d such that $d = b^{-1} \log n$. Then

$$g(d) \geq \frac{\log^d n}{n^c} = 2^{d \log \log n - c \log n} = 2^{d \log (bd) - cbd} = \exp \Omega(d \log d).$$

Finally for (6), we repeat the argument from [1]. If $d \leq \epsilon \log n / \log \log n$ then $\log^d n = 2^{d \log \log n} \leq n^\epsilon$. In particular, if $d = o(\log n / \log \log n)$ then $\log^d n = n^{o(1)}$. Moreover, for $d \geq \log^{1/2} n$ we have $\log \log n \leq 2 \log d$ and thus $\log^d n = 2^{d \log \log n} \leq 4^{d \log d}$. ◀

These calculations also show the regimes in which these considerations are at all interesting. For $d = o(\log n / \log \log n)$ then both functions are bounded by $n^{o(1)}$, and the multivariate perspective gives no insight. For $d \geq \log n$, both bounds exceed n , and we are better off running n BFSs for computing diameters, or passing through the entire point set for range searching.

2.2 Model of computation

We operate in the word RAM, assuming constant-time arithmetic operations on coordinates and edge lengths, as well as constant-time operations in the monoid supported by our range queries. For ease of presentation, edge lengths are assumed to be nonnegative integers; we could work with abstract nonnegative weights instead [6].

3 Orthogonal Range Queries

3.1 Preliminaries

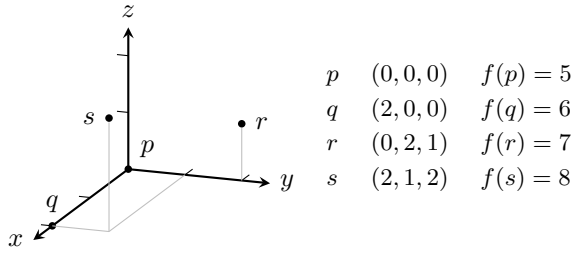
Let P be a set of d -dimensional points. We will view $p \in P$ as a vector $p = (p_1, \dots, p_d)$.

A commutative *monoid* is a set M with an associative and commutative binary operator \oplus with identity. The reader is invited to think of M as the integers with $-\infty$ as identity and $a \oplus b = \max\{a, b\}$.

Let $f: P \rightarrow M$ be a function and define for each subset $Q \subseteq P$

$$f(Q) = \bigoplus \{f(q) : q \in Q\}$$

with the understanding that $f(\emptyset)$ is the identity in M .



■ **Figure 1** Four points in three dimensions. With the monoid (\mathbf{Z}, \max) we have $f(\{p, r, s\}) = 7$.

3.2 Range Trees

Consider dimension $i \in \{1, \dots, d\}$ and enumerate the points in Q as $q^{(1)}, \dots, q^{(r)}$ such that $q_i^{(j)} \leq q_i^{(j+1)}$, for instance by ordering after the i th coordinate and breaking ties lexicographically. Define $\text{med}_i(Q)$ to be the *median* point $q^{(\lceil r/2 \rceil)}$, and similarly the $\text{min}_i(Q) = q^{(1)}$ and $\text{max}_i(Q) = q^{(r)}$. Set

$$Q_L = \{q^{(1)}, \dots, q^{(\lceil r/2 \rceil)}\}, \quad Q_R = \{q^{(1 + \lceil r/2 \rceil)}, \dots, q^{(r)}\}. \quad (8)$$

For $i \in \{1, \dots, d\}$, the *range tree* $R_i(Q)$ for Q is a node x with the following attributes:

- $L[x]$, a reference to the range tree $T_i(Q_L)$, often called the *left child* of x .
- $R[x]$, a reference to the range tree $T_i(Q_R)$, often called the *right child* of x .
- $D[x]$, a reference to the range tree $T_{i+1}(Q)$, often called the *secondary, associate, or higher-dimensional* structure. This attribute only exists for $i < d$.
- $l[x] = \text{min}_i(Q)$.
- $r[x] = \text{max}_i(Q)$.
- $f[x] = f(Q)$. This attribute only exists for $i = d$.

Construction

Constructing a range tree for T is a straightforward recursive procedure:

► **Algorithm C** (Construction). *Given integer $i \in \{1, \dots, d\}$ and a list Q of points, this algorithm constructs the range tree $R_i(Q)$ with root x .*

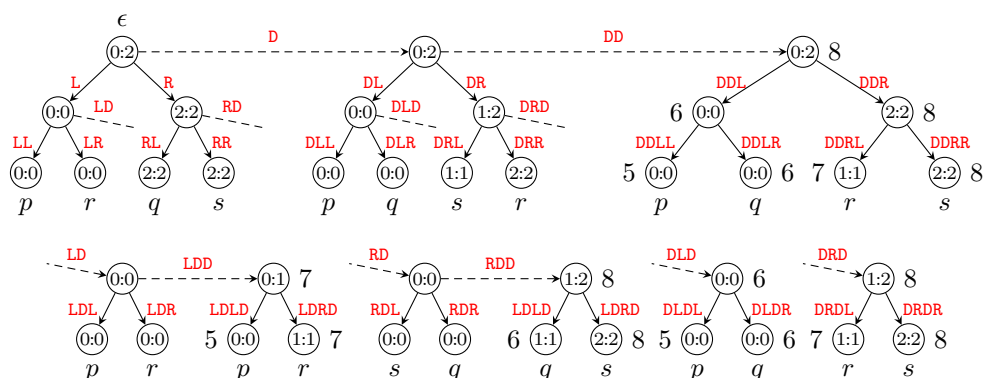
C1 [Base case $Q = \{q\}$.] Recursively construct $D[x] = T_{i+1}(Q)$ if $i < d$, otherwise set $f[x] = f(q)$. Set $l[x] = r[x] = q_i$. Return x .

C2 [Find median.] Determine $q = \text{med}_i Q$, $l[x] = \text{min}_i(Q)$, $r[x] = \text{max}_i(Q)$.

C3 [Split Q .] Let Q_L and Q_R as given by (8), note that both are nonempty.

C4 [Recurse.] Recursively construct $L[x] = R_i(Q_L)$ from Q_L . Recursively construct $R[x] = R_i(Q_R)$ from Q_R . If $i < d$ then recursively construct $D[x] = T_{i+1}(Q)$. If $i = d$ then set $f[x] = f[L[x]] \oplus f[R[x]]$.

The data structure can be viewed as a collection of binary trees whose nodes x represent various subsets P_x of the original point set P . In the interest of analysis, we now introduce a scheme for naming the individual nodes x , and thereby also the subsets P_x . Each node x is identified by a string of letters from $\{L, R, D\}$ as follows. Associate with x a set of points, often called the *canonical subset* of x , as follows. For the empty string ϵ we set $P_\epsilon = P$. In general, if $Q = P_x$ then $P_{xL} = Q_L$, $P_{xR} = Q_R$ and $P_{xD} = Q$. The strings over $\{L, R, D\}$ can be understood as uniquely describing a path through in the data structure; for instance, L



■ **Figure 2** Part of the range tree for the points from Fig. 1. The label of node x appears in red on the arrow pointing to x . Nodes contain $l[x]:r[i]$. The references $L[x]$ and $R[x]$ appear as children in a binary tree using usual drawing conventions. The reference $D[x]$ appears as a dashed arrow (possibly interrupted); the placement on the page follows no other logic than economy of layout and readability. References $D[x]$ from leaf nodes, such as $D[LL]$ leading to node LLD , are not shown; this conceals 12 single-node trees. The ‘3rd-dimensional nodes,’ whose names contain two Ds, show the values $f[x]$ next to the node. To ease comprehension, leaf nodes are decorated with their canonical subset, which is a singleton from $\{p, q, r, s\}$. The reader can infer the canonical subset for an internal node as the union of leaves of the subtree; for instance, $P_{DR} = \{r, s\}$. However, note that these point sets are *not* explicitly stored in the data structure.

means ‘go left, i.e., to the left subtree, the one stored at $L[x]$ ’ and D means ‘go to the next dimension, i.e., to the subtree stored at $D[x]$ ’. The name of a node now describes the unique path that reaches it.

► **Lemma 6.** *Let $n = |P|$. Algorithm C computes the d -dimensional range tree for P in time linear in $nd \cdot B(n, d)$.*

Proof. We run Algorithm C on input P and $i = 1$.

Disregarding the recursive calls, the running time of algorithm C on input i and Q is dominated by Steps C2 and C3, i.e., splitting Q into two sets of equal size. It is known that this task can be performed in time linear in $|Q|$ [4]. Thus, the running time for constructing $R_i(Q)$ is linear in $|Q|$ plus the time spent in recursive calls.

This means that we can bound the running time for constructing $T_1(P)$ by bounding sizes of the sets P_x associated with every node x in the data structure. If for a moment X denotes the set of all these nodes then we want to bound

$$\sum_{x \in X} |P_x| = \sum_{x \in X} |\{p \in P: p \in P_x\}| = \sum_{p \in P} |\{x \in X: p \in P_x\}|.$$

Thus, we need to determine, for given $p \in P$, the number of subsets P_x in which p appears. By construction, there are fewer than d occurrences of D in x . Moreover, if x contains more than h occurrences of either L or R then P_x is empty. Thus, x has at most $h + d$ letters. For two different strings x and x' that agree on the positions of D, the sets P_x and $P_{x'}$ are disjoint, so p appears in at most one of them. We conclude that the number of sets P_x such that $p \in P_x$ is bounded by the number of ways to arrange fewer than d many Ds and at most

h non-Ds. Using the identity $\binom{a+0}{0} + \dots + \binom{a+b}{b} = \binom{a+b+1}{b}$ repeatedly, we compute

$$\begin{aligned} \sum_{i=0}^{d-1} \sum_{j=0}^h \binom{i+j}{j} &= \sum_{i=0}^{d-1} \binom{i+h+1}{h} = \sum_{i=0}^{d-1} \binom{i+h+1}{i+1} = \\ &(-1) + \sum_{i=0}^d \binom{i+h}{i} = \binom{h+d+1}{d} - 1 = \frac{h+d+1}{h+1} \binom{h+d}{d} - 1 \leq d \binom{d+h}{d}. \end{aligned}$$

The bound follows from aggregating this contribution over all $p \in P$. \blacktriangleleft

Search.

In this section, we fix two sequences of integers l_1, \dots, l_d and r_1, \dots, r_d describing the *query box* B given by

$$B = [l_1, r_1] \times \dots \times [l_d, r_d].$$

► **Algorithm Q** (Query). *Given integer $i \in \{1, \dots, d\}$, a query box B as above and a range tree $R_i(Q)$ with root x for a set of points Q such that every point $q \in Q$ satisfies $l_j \leq q_j \leq r_j$ for $j \in \{1, \dots, i-1\}$. This algorithm returns $\bigoplus \{f(q) : q \in Q \cap B\}$.*

Q1 [Empty?] If the data structure is empty, or $l_i > r[x]$, or $l[x] > r_i$, then return the identity in the underlying monoid M .

Q2 [Done?] If $i = d$ and $l_d \leq \min_d[x]$ and $\max_d[x] \leq r_d$ then return $f[x]$.

Q3 [Next dimension?] If $i < d$ and $l_i \leq l[x]$ and $r[x] \leq r_i$ then query the range tree at $D[x]$ for dimension $i+1$. Return the resulting value.

Q4 [Split.] Query the range tree $L[x]$ for dimension i ; the result is a value f_L . Query the range tree $R[x]$ for dimension i ; the result is a value f_R . Return $f_L \oplus f_R$. \blacktriangleleft

To prove correctness, we show that this algorithm is correct for each point set $Q = P_x$.

► **Lemma 7.** *Let $i = D(x) + 1$, where $D(x)$ is the number of Ds in x . Assume that P_x is such that $l_j \leq p_j \leq r_j$ for all $j \in \{1, \dots, i-1\}$ for each $p \in P_x$. Then the query algorithm on input x and i returns $f(B \cap P_x)$.*

Proof. Backwards induction in $|x|$.

If $|x| = h + d$ then P_x is the empty set, in which case the algorithm correctly returns the identity in M .

If the algorithm executes Step Q2 then B is satisfied for all $q \in P_x$, in which case the algorithm correctly returns $f[x] = f(P_x)$.

If the algorithm executes Step Q3 then B satisfies the condition in the lemma for $i+1$, and the number of Ds in P_{xD} is $i+1$, and $D[x]$ store the $(i+1)$ th range tree for P_{xD} . Thus, by induction the algorithm returns $f(P_{xD} \cap B)$, which equals $f(P_x \cap B)$ because $P_{xD} = P_x$.

Otherwise, by induction, $f_L = f(P_{xL} \cap B)$ and $f_R = f(P_{xR} \cap B)$. Since $P_{xL} \cup P_{xR} = P_x$, we have $f(P_x \cap B) = f((P_{xL} \cap B) \cup (P_{xR} \cap B)) = f_L \oplus f_R$. \blacktriangleleft

► **Lemma 8.** *If x is the root of the range tree for P then on input $i = 1$, x , and B , the query algorithm returns $f(P \cap B)$ in time linear in $2^d B(n, d)$.*

Proof. Correctness follows from the previous lemma.

For the running time, we first observe that the query algorithm does constant work in each visited node. Thus it suffices to bound the number of visited nodes as

$$2^d \binom{h+d}{d} \quad (d \geq 1, h \geq 0). \quad (9)$$

We will show by induction in d that (9) holds for every call to a d -dimensional range tree for a point set P_x , where $h = \lceil \log |P_x| \rceil$. The two easy cases are Q1 and Q2, which incur no additional nodes to be visited, so the number of visited nodes is 1, which is bounded by (9). Step Q3 leads to a recursive call for a $(d-1)$ -dimensional range tree over the same point set $P_{xD} = P_x$, and we verify

$$1 + 2^{d-1} \binom{h+d-1}{d-1} \leq 2^d \binom{h+d}{d}.$$

The interesting case is Step Q4. We need to follow two paths from x to the leaves of the binary tree of x . Consider the leaves l and r in the subtree rooted at x associated with the points $\min_i(P_x)$ and $\max_i(P_x)$ as defined in Sec. 3.2. We describe the situation of the path Y from l to x ; the other case is symmetrical. At each internal node $y \in Y$, the algorithm chooses Step Q4 (because $l_i \geq l[y]$). There are two cases for what happens at yL and yR . If $l_i \leq \text{med}_i(P_y)$ then P_{yR} satisfies $l_i \leq \min_i(P_{yR}) \leq r_i$, so the call to yR will choose Step Q3. By induction, this incurs $2^{d-1} \binom{d-1+i}{d-1}$ visits, where i is the height of y . In the other case, the call to yL will choose Step Q1, which incurs no extra visits. Thus, the number of nodes visited on the left path is at most

$$h + \sum_{i=0}^{h-1} 2^{d-1} \binom{d-1+i}{d-1},$$

and the total number of nodes visited is at most twice that:

$$2h + 2^d \sum_{i=0}^{h-1} \binom{d-1+i}{d-1} \leq 2^d \sum_{i=0}^h \binom{d-1+i}{d-1} = 2^d \binom{d+h}{d}.$$

◀

3.3 Discussion

The textbook analysis of range trees, and similar d -dimensional spatial algorithms and data structures sets up a recurrence relation like

$$r(n, d) = 2r(n/2, d) + r(n, d-1),$$

for the construction and

$$r(n, d) = \max\{r(n/2, d), r(n, d-1)\},$$

for the query time. One then observes that $n \log^d n$ and $\log^d n$ are the solutions to these recurrences. This analysis goes back to Bentley's original paper [3].

Along the lines of the previous section, one can show that the functions $n \cdot B(n, d)$ and $B(n, d)$ solve these recurrences as well. A detailed derivation can be found in [15], which also contains combinatorial arguments of how to interpret the binomial coefficients in the context of spatial data structures. A later paper of Chan [7] also takes the recurrences as a starting point, and observes asymptotically improved solution for the related question of dominance queries.

4 Graph Distances

We present the algorithm for computing the diameter. The construction closely follows Cabello and Knauer [6], but uses the range tree bounds from Section 3. The analysis is extended to superconstant dimension as in Abboud *et al.* [1]. Using the approximate treewidth construction of Bodlaender *et al.* [5], we can pay more attention to the parameters of the recursive decomposition into small-size separators.

4.1 Preliminaries

We consider an undirected graph G with n vertices and m edges with nonnegative integer weights. The set of vertices is $V(G)$. For a vertex subset U we write $G[U]$ for the induced subgraph.

A path from u to v is called a u, v -path and denoted uPv . For $w \in V(uPv)$ we use the notation wPv for the subpath starting in w . The *length* of a path, denoted $l(uPv)$, is the sum of its edge lengths.

The *distance* from vertex u to vertex v , denoted $d(u, v)$, is the minimum length of shortest u, v -path. The *Wiener index* of G , denoted $wien(G)$ is $\sum_{u, v \in V(G)} d(u, v)$. The *eccentricity* of a vertex u , denoted $e(u)$ is given by $e(u) = \max\{d(u, v) : v \in V(G)\}$. The *diameter* of G , denoted $\text{diam}(G)$ is $\max\{e(u) : u \in V(G)\}$. The *radius* of G , denoted $\text{rad}(G)$ is $\min\{e(u) : u \in V(G)\}$.

4.2 Separation

A *skew k -separator tree* T of G is a binary tree such that each node t of T is associated with a vertex set $Z_t \subseteq V(G)$ such that

- $|Z_t| \leq k$,
- If $L_t R_t$ denote the vertices of G associated with the left and right subtrees of t , respectively, then Z_t separates L_t and R_t and

$$\frac{n}{k+1} \leq |L_t \cup Z_t| \leq \frac{nk}{k+1}, \quad (10)$$

- T remains a skew k -separator even if edges between vertices of Z_t are added.

It is known that such a tree can be found from a tree decomposition, and an approximate tree decomposition can be found in single-exponential time. We summarise these results in the following lemma:

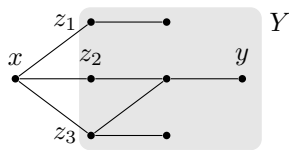
► **Lemma 9** ([6, Lemma 3] with [5, Theorem 1]). *For a given n -vertex input graph G , a skew $(5 \text{tw}(G) + 4)$ -separator tree can be computed in time $n \exp O(k)$.*

4.3 Algorithm

Given graph G , let \mathcal{S} denote the set of shortest paths. Let $e(x; W)$ denote the distance from x to any vertex in W . Formally,

$$e(x; W) = \max\{l(xPw) : xPw \in \mathcal{S}, w \in W\}.$$

The central idea of the algorithm, following [6], is the computation for $x \in X$, $z \in Z$ of z -visiting eccentricities $e(x, z; Y)$ defined as follows. Enumerate $Z = \{z_1, \dots, z_k\}$. Then define, for $x \in X$, $z_i \in Z$ the value $e(x, z_i; Y)$ as the maximum distance from z_i to y over all



■ **Figure 3** Example with $Z = \{z_1, z_2, z_3\}$. The eccentricity of x to Y is $e(x; Y) = 3$. Also, $e(x, z_1; Y) = 1$, $e(x, z_2; Y) = 2$. Note $e(x, z_3; Y) = 1$ despite the z_3, y -path.

$y \in Y$ such that some shortest x, y -path contains z_i but no shortest x, y -path contains any of $\{z_1, \dots, z_{i-1}\}$. Formally,

$$\begin{aligned}
 e(x, z_i; Y) &= \max l(zPy) \\
 &\text{such that } y \in Y, \\
 &\quad xPy \in \mathcal{S}, \\
 &\quad Z \cap V(xPy) \ni z_i, \\
 &\quad \{z_1, \dots, z_{i-1}\} \cap V(xQy) = \emptyset \text{ for all } zQy \in \mathcal{S}.
 \end{aligned}$$

See Figure 3 for a small example.

This definition ensures that in situations where x and y are connected by two shortest paths of the form xPz_jPy and xPz_iPy with $j \neq i$, then exactly one of them contributes to $e(x, z_j; Y)$ and $e(x, z_i; Y)$. This is important for avoiding over-counting in Section 4.5.

► **Lemma 10.** For $x \in X$, $e(x; Y) = \max\{d(x, z) + e(x, z; Y) : z \in Z\}$.

The proof is in Appendix B. The connection to orthogonal range queries is the following. Enumerate $Z = \{z_1, \dots, z_k\}$. A shortest path xPz_iPy attaining the distance $e(x; Y)$ maximises $d(z_i, y)$ over all $y \in Y$, where $z_i \in Z$ is such that for all $z_j \in Z$,

$$\begin{aligned}
 d(x, z_i) + d(z_i, y) &< d(x, z_j) + d(z_j, y), & (j < i), \\
 d(x, z_i) + d(z_i, y) &\leq d(x, z_j) + d(z_j, y), & (j \geq i),
 \end{aligned}$$

equivalently,

$$\begin{aligned}
 d(x, z_i) - d(x, z_j) &< d(z_j, y) - d(z_i, y), & (j < i), \\
 d(x, z_i) - d(x, z_j) &\leq d(z_j, y) - d(z_i, y), & (j \geq i).
 \end{aligned}$$

We are ready for the algorithm, which closely follows [6]:

► **Algorithm E (Eccentricities).** Given a graph G and a skew k -separator tree with root t , this algorithm computes the eccentricity $e(v)$ of every vertex $v \in V(G)$. We write $Z = Z_t$, $X = L_t \cup Z_t$, and $Y = R_t \cup Z_t$.

- E1** [Base case.] If $n/\ln n < 4k(k+1)$ then find all distances using Dijkstra's algorithm and terminate.
- E2** [Distances from separator.] Compute $d(z, v)$ for every $z \in Z, v \in V(G)$ using k applications of Dijkstra's algorithm.
- E3** [Add shortcuts.] For each pair $z, z' \in Z$, add the edge zz' to G , weighted by $d(z, z')$. Remove duplicate edges, retaining the shortest.
- E4.1** [Start iterating over $\{z_1, \dots, z_k\}$.] Let $i = 1$.

E4.2 [Build range tree for z_i .] Construct a k -dimensional range tree R of points $\{p(y) : y \in Y\}$ given by $p(y) = (p_1, \dots, p_k)$ where

$$p_j = d(z_i, y) - d(z_j, y) \quad j \in \{1, \dots, k\}$$

and $f(p(y)) = d(z_i, y)$ using the monoid (\mathbf{Z}, \max) .

E4.3 [Query range tree.] For each $x \in X$, query R with $l_1 = \dots = l_k = -\infty$ and

$$r_j = \begin{cases} d(x, z_i) - d(x, z_j) - 1, & (j < i); \\ d(x, z_i) - d(x, z_j), & (j \geq i). \end{cases}$$

The result is $e(x, z_i; Y)$.

E4.4 [Next z_i .] If $i < k$ then increase i and go to E4.1.

E5 [Recurse on $G[X]$.] Recursively compute the distances in $G[X]$ using the left subtree of t as a skew k -separator tree. The result are eccentricities $e(x; X)$ for each $x \in X$. For each $x \in X$, set $e(x; Y) = \max\{d(x, z_i) + e(x, z_i; Y) : i \in \{1, \dots, k\}\}$, then set $e(x) = \max\{e(x; X), e(x; Y)\}$. Set $e(z) = \max\{d(z, v) : v \in V(G)\}$ for $z \in Z$.

E6 [Flip.] Repeat Steps E4–5 with the roles of X and Y exchanged.

4.4 Running Time

► **Lemma 11.** *The running time of Algorithm E is $O(n \cdot B(n, k) \cdot 2^k k^2 \ln n)$.*

The proof is in Appendix C. We can now establish Theorem 1 for diameter and radius.

Proof of Thm. 1, distances. To compute all eccentricities for a given graph we find a k -skew separator for $k = 5 \text{tw}(G) + 4$ using Lemma 9 in time $n \exp O(\text{tw}(G))$. We then run Algorithm E, using Lemma 11 to bound the running time. From the eccentricities, the radius and diameter can be computed in linear time using their definition. ◀

4.5 Wiener Index

Algorithm E can be modified to compute the Wiener index, as described in [6, Sec. 4], completing the proof of Theorem 1. The main observation is that the sum of distances between all pair $u, v \in V(G)$ can be written as pairwise distances within X , within Y , and between X and Y , carefully subtracting contributions from these sums that were included twice.

The orthogonal range queries for vertex $x \in X$ now need to report the sum of distances to every $y \in Y$, rather than just the value of the maximum distance $e(x; Y)$. To this end, we use the monoid of positive integer tuples (d, r) with the operation

$$(d, r) \oplus (d', r') = (d + d', r + r')$$

with identity element $(0, 0)$. The value associated with vertex y in Step E4.2 is $f(p(y)) = (1, d(z_i, y))$.

We also observe the matching lower bound:

► **Theorem 12.** *An algorithm for computing the Wiener index in time $n^{2-\epsilon} \exp o(\text{tw}(G))$ time for any $\epsilon > 0$ refutes the Orthogonal Vector conjecture.*

Proof. The diameter of G is 2 if and only if $\text{wien}(G) = 2\binom{n}{2} - m$. Thus, an algorithm for Wiener index is able to distinguish input graphs of diameter 2 and 3. This problem was shown hard in [1]. ◀

References

- 1 Amir Abboud, Virginia Vassilevska Williams, and Joshua R. Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, Va, USA, January 10–12, 2016*, pages 377–391. SIAM, 2016. doi:10.1137/1.9781611974331.ch28.
- 2 Matthias Bentert and André Nichterlein. Parameterized complexity of diameter. *CoRR*, abs/1802.10048, 2018. arXiv:1802.10048.
- 3 Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980. doi:10.1145/358841.358850.
- 4 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.
- 5 Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshтанov, and Michał Pilipczuk. An $O(c^k n)$ 5-approximation algorithm for treewidth. *SIAM J. Comput.*, 45(2):317–378, 2016. doi:10.1137/130947374.
- 6 Sergio Cabello and Christian Knauer. Algorithms for bounded treewidth with orthogonal range searching. *Comput. Geom.*, 42(9):815–824, 2009. doi:10.1016/j.comgeo.2009.02.001.
- 7 Timothy M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008. doi:10.1007/s00453-007-9062-1.
- 8 Bernard Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *J. Assoc. Comput. Mach.*, 37(2):200–212, 1990. doi:10.1145/77600.77614.
- 9 Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *J. Algorithms*, 41:280–301, 2001. doi:10.1006/jagm.2001.1186.
- 10 Rodney G. Downey and Michael R. Fellows. *Parameterized complexity*. Springer-Verlag, New York, 1999.
- 11 Thore Husfeldt. Computing Graph Distances Parameterized by Treewidth and Diameter. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:11, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.IPEC.2016.16.
- 12 Russel Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.
- 13 Der-Tsai Lee and Chakkuen K. Wong. Quintary trees: A file structure for multidimensional database systems. *ACM Trans. Database Syst.*, 5(3):339–353, 1980. doi:10.1145/320613.320618.
- 14 George S. Lueker. A data structure for orthogonal range queries. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16–18 October 1978*, pages 28–34. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.1.
- 15 Louis Monier. Combinatorial solutions of multidimensional divide-and-conquer recurrences. *J. Algorithms*, 1(1):60–74, 1980. doi:10.1016/0196-6774(80)90005-X.
- 16 Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Symposium on Theory of Computing Conference, STOC’13, Palo Alto, CA, USA, June 1–4, 2013*, pages 515–524, 2013. doi:10.1145/2488608.2488673.
- 17 Dan E. Willard. New data structures for orthogonal range queries. *SIAM J. Comput.*, 14(1):232–253, 1985. doi:10.1137/0214019.

- 18 Virginia Vassilevska Williams. Hardness of easy problems: Basing hardness on popular conjectures such as the strong exponential time hypothesis (invited talk). In *10th International Symposium on Parameterized and Exact Computation, IPEC 2015, September 16-18, 2015, Patras, Greece*, pages 17–29, 2015. doi:10.4230/LIPIcs.IPEC.2015.17.

A Parameterization by Vertex Cover Number

We show Theorem 4.

A *vertex cover* is a vertex subset C of $V(G)$ such that every edge in G has at least one endpoint in C . The smallest k for which a vertex cover of size k exists is the *vertex cover number* of a graph, denoted $\text{vc}(G)$. The number of edges in a graph is at most $n \cdot \text{vc}(G)$.

A.1 Eccentricities and Wiener Index

A graph with vertex cover number 1 is a star, and its pairwise distances can be determined from the input size. It follows from [1] that the complexity of computing the diameter must depend exponentially on $\text{vc}(G)$, in the same way as for $\text{tw}(G)$. We observe here that algorithms that match this lower bound are quite immediate. The idea is that each $v \notin C$ has its entire neighbourhood $N(v)$, defined as

$$N(v) = \{u \in V(G) : uv \in E(G)\},$$

contained in C . Thus, all paths from v have their second vertex in C . In particular, two vertices v and w with $N(v) = N(w)$ have the same distances to the rest of the graph. Since $N(v) \subseteq C$ it suffices to consider all 2^k many subsets of C . The details are given in Algorithm V.

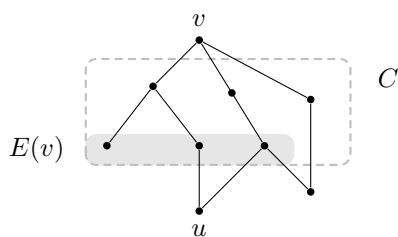
► **Algorithm V** (Eccentricities Parameterized by Vertex Cover). *Given a connected, unweighted, undirected graph G and a vertex cover C , this algorithm computes the eccentricity of each vertex and the Wiener index.*

- V1** [Initialise.] Set $\text{wien}(G) = 0$. Insert each $v \notin C$ into a dictionary D indexed by $N(v)$.
V2 [Distances from C .] For each $v \in C$, perform a breadth-first search from v in G , computing $d(v, u)$ for all $u \in V(G)$. Let $e(v) = \max_u d(v, u)$ and increase $\text{wien}(G)$ by $\frac{1}{2} \sum_u d(v, u)$.
V3 [Distances from $V(G) - C$.] Choose any $v \in D$. Perform a breadth-first search from v in G , computing $d(v, u)$ for all $u \in V(G)$. For each $w \in D$ with $N(w) = N(v)$ (including v itself), let $e(w) = \max_u d(v, u)$, increase $\text{wien}(G)$ by $\frac{1}{2} \sum_u d(v, u)$, and remove w from D . Repeat step V3 until D is empty.

► **Theorem 13.** *The eccentricities and Wiener index of an unweighted, undirected, connected graph with m edges and vertex cover number k can be computed in time $O(m2^k)$. Any algorithm with running time $m \exp o(k)$ would refute the Strong Exponential Time Hypothesis.*

Proof. It is well known that a minimum vertex cover can be computed in the given time bound [10].

For the running time of Algorithm V, we first observed that for each $v \notin C$ the neighbourhood $N(v)$ is entirely contained in C . Thus, there are only 2^k different neighbourhoods used as an index to D and we can bound the number of BFS computations in Step V3 by 2^k . (Step V2 incurs another k such computations.) Assuming constant-time dictionary operations, the running time of the algorithm is therefore bounded by $m \exp O(k)$.



■ **Figure 4** The distance from v to u is 3 and $N(u) \subseteq E(v)$.

To see correctness, we need to argue that the distances computed for $w \in D$ in Step 4 are correct. First, to argue $d(v, z) = d(w, z)$ for all $z \notin \{v, w\}$ consider shortest path vPz and wQz . Let v' and w' denote the second vertices on these paths (possibly $v' = z$ or $w' = z$). Then $v'Pz$ and $w'Qz$ are shortest paths of length $l(vPz) - 1$ and $l(wQz) - 1$, respectively. Since $N(v) = N(w)$, the path $wv'Pz$ exists and is a shortest w, z -path as well, of length $l(vPz)$. We conclude that $l(vPz) = l(wQz)$.

It is *not* true that $d(v, z) = d(w, z)$ for $z \in \{v, w\}$. Instead, we have $d(v, w) = d(w, v)$ (both equal 2) and $d(v, v) = d(w, w)$ (both equal 0). Thus, the contributions from v and w to W are the same, and the sets $d(v, \cdot)$ and $d(w, \cdot)$ have the same maxima.

For the hardness result, we merely need to observe that reduction in [1] has vertex cover number $k + 2$. ◀

A.2 Faster Eccentricities

Vertex cover number is an extremely well studied parameter, so the analysis need not stop here. The best current algorithm for *finding* a vertex cover runs in time $O(nk + 1.274^k)$ [9], so the bound in Theorem 13 is dominated by the distance computation. Thus it may make sense to look for distance computation algorithms with running times of the form $nk + g(k)$ rather than $m \cdot g(k)$.

We can find such an algorithm for eccentricities, but not for Wiener index. First, we observe that if C is a vertex cover then no path can contain consecutive vertices from $V(G) - C$. Thus, we can modify the graph by inserting length-2 shortcuts between nonadjacent vertices in C that share a neighbour without changing the pairwise distances in the graph. We can now run Dijkstra restricted to the subgraph $G[C \cup \{v\}]$, noting that the second layer of the Dijkstra tree consists of $N(v)$, which is contained in C . Thus the number of such computations that are different is bounded by 2^k , the number of neighbourhoods. The eccentricity $e(v)$ can be derived from this Dijkstra tree as follows. Let $E(v)$ denote the *eccentric* vertices from v in C , i.e., the vertices at maximum distance from v in C . Note that $E(v)$ contains exactly the vertices at the deepest layer of the Dijkstra tree from v in $G[C \cup \{v\}]$. The only vertices u in G that can be farther away from v than $E(v)$ must have their entire neighbourhood $N(u)$ contained in $E(v)$. See Figure 4.

The only confusion arises if the only such vertex is v itself. To handle these details we need to determine, for each cover subset $S \subseteq C$, if the number of u with $N(u) \subseteq S$ is 0, 1, or more. This can be solved by fast zeta transform in time $2^k k$, or more directly as follows. For

each $S \subseteq C$, let

$$h(S) = \begin{cases} \{w\}, & \text{if } N(w) \subseteq S \text{ for exactly one } w \notin C; \\ \emptyset, & \text{if there is no } w \notin C \text{ with } N(w) \subseteq S; \\ C, & \text{otherwise.} \end{cases}$$

(The third value is an arbitrary placeholder.) Then $h(S)$ can be computed for all $S \subseteq C$ in a bottom-up fashion.

The details are given in the following algorithm.

► **Algorithm F** (Faster Eccentricities Parameterized by Vertex Cover). *Given a connected, unweighted, undirected graph G and a vertex cover C , this algorithm computes the eccentricity of each vertex.*

- F1** [Initialise.] Insert each $v \in V(G)$ into a dictionary D indexed by $N(v)$. Set $h(S) = \emptyset$ for all $S \subseteq C$.
- F2** [Compute h .] For each $u \notin C$, set $h(N(u)) = \{u\}$ if $h(N(u)) = \emptyset$, otherwise $h(N(u)) = C$. For each nonempty subset $S \subseteq C$ in increasing order of size, compute $W = \bigcup_{w \in S} h(S - \{w\})$. If $|W| > 1$ then set $h(S) = C$. Else set $h(S) = W$.
- F3** [Shortcuts.] For each pair of covering vertices $u, v \in C$, if $uv \notin E(G)$ but u and v share a neighbour outside C , add the edge uv to $E(G)$ with length 2.
- F4** [Eccentricities from C .] For each $v \in C$, compute shortest distances in $G[C]$ from v . Set $d = \max_{u \in C} d(v, u)$ and let $E(v)$ denote the vertices in C at distance d . Let

$$e(v) = \begin{cases} d + 1, & \text{if } h(E(v)) - \{v\} \neq \emptyset, \text{ [equivalently, } E(v) \supseteq N(w) \text{ for some } w \neq v]; \\ d, & \text{otherwise.} \end{cases}$$

- F5** [Eccentricities from $V(G) - C$.] For each $v \in D$, compute shortest distances in $G[C \cup \{v\}]$ from v . Set $d = \max_{u \in C} d(v, u)$ and let $E(v)$ denote the vertices in C at distance d . For each $u \in D$ with $N(u) = N(v)$ (including v itself) let

$$e(u) = \begin{cases} d + 1, & \text{if } h(E(u)) - \{u\} \neq \emptyset, \text{ [equivalently, } E(u) \supseteq N(w) \text{ for some } w \neq u]; \\ d, & \text{otherwise.} \end{cases}$$

and remove u from D .

► **Theorem 14.** *The eccentricities an unweighted, undirected, connected graph with m edges and vertex cover number k can be computed in time $O(nk + 2^k k^2)$.*

Proof. Step F1 needs to visit every of the nk edges. There are 2^k subsets of C , bounding the running time of Step F2 to $O(2^k k)$. Step F3 can be performed in time $O(2^k k^2)$ (instead of the obvious $O(nk^2)$) by iterating over $w \in D$ and all pairs $u, v \in N(w)$. The shortest path computations in Steps F4 and F5 take time $O(k^2)$ each using Dijkstra's algorithm, for a total of $O(2^k k^2)$. The dictionary contains at most n values, so the total time of Step F4 and F5 is $O(n + 2^k k^2)$.

To see correctness, assume without loss of generality that we already performed the shortcut operation in Step F3.

We argue for correctness of Step F5, Step F4 is similar. Consider a shortest u, v -path uPv to an eccentric vertex v of u . If $v \in C$ then v belongs to $E(v)$. Moreover, there can be

no vertex w with $N(w) \subseteq E(v)$, because otherwise $uPvw$ is a shortest path and therefore v is not eccentric. Thus, Step F5 correctly sets $e(u)$ to $d(u, v)$.

Otherwise, assume all such paths have $v \notin C$. There are two cases. If uPv is just the edge uv then every vertex in G has distance at most 1 to u . If G is a star then $C = N(v) = \{u\}$ and $d = 0$. Moreover, $h(E(u)) \ni v$, so Step F5 correctly computes $e(u) = d + 1 = 1$. If G contains a triangle then $|C| > 1$ and the vertices in $E(u)$ are at distance 1. Moreover, there cannot exist $w \neq u$ with $N(w) \subseteq E(u)$ because then there would be a u, w -path of length 2. Thus, Step F5 correctly computes $e(u) = d = 1$.

The remaining case is when uPv contains at least 3 vertices. Let w denote the penultimate vertex, so the path is of the form $u \cdots wv$. Since $v \notin C$ we have $w \in C$. Moreover, we have $w \in E(u)$. (Otherwise there would be an eccentric path to another vertex $w' \in C$.) Let $d = d(u, w)$. Every neighbour x of v must belong to C , and by the shortcutting Step F3, we can assume it also belongs to $N(w) \cup \{w\}$. The distance $d(v, x)$ is at most $d + 1$ (because it is a neighbour of w , or w itself), but cannot be $d + 1$ (because then there would be an eccentric u, x -path for $x \in C$.) Thus, we have $d(u, x) = d$ and therefore $x \in E(u)$. We have established that $N(v) \subseteq E(u)$, so we can again conclude that Step F5 correctly computes $e(u) = d + 1$. ◀

B Proof of Lemma 10

Proof. Let xPy be a shortest path of length $e(x; Y)$. Since Z separates X from Y , any x, y -path must contain a vertex from Z . In particular, this is true of xPy , so we can choose $z_i \in Z \cap V(xPy)$ for some $i \in \{1, \dots, k\}$. Assume xPy was chosen so as to minimize i . Since xPz_i is a shortest path, we have $l(xPz_i) = d(x, z_i)$. Moreover, z_iPy is a shortest path, and there are no shortest x, y -paths through $\{z_1, \dots, z_{i-1}\}$, so $l(z_iPy) \leq e(x, z_i; Y)$. Thus $e(x; Y) \leq d(x, z_i) + e(x, z_i; Y)$ for some $z_i \in Z$.

For the opposite inequality, choose any $z \in Z$ and shortest paths xPz and xQy with $z \in V(xQy)$ such that $l(xPz) = d(x, z)$ and $l(zQy) = e(x, z; Y)$. Since xQz is a shortest path, we see that

$$d(xPz) + e(x, z; Y) = l(xPzQy) = l(xQzQy) = l(xQy),$$

which is the length of a shortest x, y -path, and therefore at most $e(x; Y)$. ◀

C Proof of Lemma 11

Proof. Assume $n \geq 8$. Let $T(n, d)$ denote the running time of Algorithm E.

Step E1 consists of n executions of Dijkstra's algorithm on a graph with n vertices and treewidth $O(k)$, and n bounded by $O(k^2 \log k)$. This takes time $O(k^5 \log^3 k)$. The range query operations in Steps E4.2-3 can be performed in time $O(n2^k \cdot B(n, k))$ according to Lemma 2. They are executed $2k$ times, twice for each $z_i \in Z$. Thus, adding the recursive calls in step E5 for both X and Y using $|Y| \leq n - |X| + k$, we arrive at the recurrence

$$T(n, k) = \begin{cases} O(k^5 \log^3 k), & \text{if } n/\ln n < 4k(k+1); \\ n \cdot S(n, k) + T(|X|, k) + T(n - |X| + k, k), & \text{otherwise.} \end{cases}$$

for some non-decreasing function S satisfying $S(n, k) = O(2^k k \cdot B(n, k))$.

We will show

$$T(n, k) \leq 4(k+1) \cdot S(n, k) \cdot n \ln n.$$

Write $s = |X|$ and $r = n - s + k$, and consider

$$\frac{T(s, k) + T(r, k)}{4(k+1)} = S(s, k) \cdot s \ln s + S(r, k) \cdot r \ln r \leq S(n, k) \cdot (s \ln s + r \ln r). \quad (11)$$

From the bounds (10) on s we have $s \leq nk/(k+1)$ and $r \leq n - n/(k+1) + k = (nk/(k+1)) + k$, so we can bound both r and s by t given as

$$t = \frac{nk}{k+1} + k.$$

Thus we can bound (11) by

$$\frac{T(s, k) + T(r, k)}{4(k+1) \cdot S(n, k)} \leq s \ln t + (n - s + k) \ln t = n \ln t + k \ln t \leq n \ln t + k \ln n. \quad (12)$$

Step E1 ensures $k(k+1) \leq n/(4 \ln n) \leq \frac{1}{2}n$, so we get

$$t \leq \frac{nk}{k+1} + k \leq \frac{nk}{k+1} + \frac{n/2}{k+1}.$$

Using the bound $\ln y \leq 1/(y-1)$ for $y \in (0, 1)$, we see $\ln((k + \frac{1}{2})/(k+1)) \leq -1/(2k+2)$, so

$$\ln t \leq \ln\left(\frac{n(k + \frac{1}{2})}{k+1}\right) \leq \ln n - \frac{1}{2k+2}, \quad (13)$$

Using this in (12), we have

$$\frac{T(s, k) + T(r, k)}{4(k+1) \cdot S(n, k)} \leq n \ln n - \frac{n}{2k+2} + k \ln n.$$

The last term satisfies $k \ln n \leq n/4(k+1)$ because of the guarantee in Step E1. Thus,

$$\frac{T(s, k) + T(r, k)}{4(k+1) \cdot S(n, k)} + \frac{n}{4(k+1)} \leq n \ln n,$$

so that T satisfies the recurrence. ◀

EXAMENSARBETE Orthogonal Range Searching and

Graph Distances Parameterized by TreeWidth

STUDENT Måns Magnusson**HANDLEDARE** Thore Husfeldt (LTH)**EXAMINATOR** Krzysztof Kuchcinski (LTH)

Avstånd i trädbreddsbegränsade grafer genom söking över ortogonala intervall

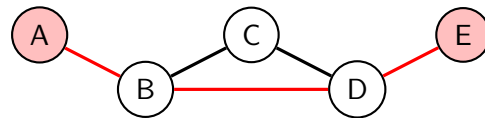
POPULÄRVETENSKAPLIG SAMMANFATTNING Måns Magnusson

Grafer, bestående av noder och bågar, kan användas för att modellera många olika saker, t.ex. infrastruktur som vägar och elledningar, men också molekyler och sociala nätverk. Detta arbete undersöker olika algoritmers beräkningskomplexitet och exekveringstider för att hitta grundläggande avståndsmått på grafer. Några av dessa är det största avståndet i grafen – diametern, samt summan av alla avstånd i grafen – wienerindexet.

För att minska tidsåtgången för att räkna ut dessa avstånd (se figur 1) i stora grafer vill vi ha effektiva algoritmer. Det visar sig att beroende på grafens utseende kan olika algoritmer användas. Det finns en generell algoritm som fungerar på alla grafer, men som utför mycket arbete, för varje nod inspekterar algoritmen hela grafen. För grafer med mer än 10^4 noder börjar denna algoritm gå väldigt långsamt. För 10^6 noder tar det i storleksordningen en timme för algoritmen att exekvera.

Det finns en effektiv algoritm för den specifika graftypen träd. Algoritmen inspekterar hela grafen endast två gånger, och hittar diametern av ett träd med 10^6 noder inom en sekund. Ett träd är en graf utan cykler, och det finns endast en väg mellan varje par av noder. Frågan är, går det att använda dessa trick för *trädlika* grafer?

Jag har använt idéer från trädalgoritmer för att ta fram en algoritm som är snabb på grafer med endast små cykler, dvs de är trädlika med liten trädbredd. Lösningen beräknar den maximala distansen genom att modellera avstånden i grafen som flerdimensionella punkter där de intressanta punkterna finns inom ortogonala intervall.



Figur 1: I denna graf med 5 noder och 5 bågar är diametern 3, mellan noderna A och E, och wienerindexet $5 \cdot 1 + 4 \cdot 2 + 1 \cdot 3 = 16$, summan av de kortaste avstånden mellan alla par av noder.

Anledningen till att dessa grafer är intressanta är att många grafer som modellerar en fysisk verklighet har en liten trädbredd, då de är kopplade till en fysisk geometri.

Wienerindexet, uppkallat efter kemisten Harry Wiener, används för att estimerar kemiska egenskaper av exempelvis alkaner. Bland annat kokpunkt korrelerar med wienerindexet och därför kan denna beräknas innan dyra experiment utförs.

I mitt examensarbete har jag dels bevisat teoretiskt hur exekveringstiden för min algoritm växer beroende av grafen, dels implementerat min algoritm och jämfört den mot den mer generella, men långsammare, algoritmen. Vi har därmed besvarat en öppen fråga inom algoritmforskningen, vilket har resulterat i en inskickad artikel till en algoritmkonferens.