The Inverse Voronoi Problem in Graphs II: Trees

Édouard Bonnet[*] Sergio Cabello[†] Bojan Mohar[‡] Hebert Pérez-Rosés[§]

May 1, 2020

Abstract

We consider the inverse Voronoi diagram problem in trees: given a tree T with positive edge-lengths and a collection $\mathbb{U}$ of subsets of vertices of $V(T)$, decide whether $\mathbb{U}$ is a Voronoi diagram in T with respect to the shortest-path metric. We show that the problem can be solved in $O(N + n\log^2 n)$ time, where n is the number of vertices in T and $N = n + \sum_{U\in\mathbb{U}} |U|$ is the size of the description of the input. We also provide a lower bound of $\Omega(n \log n)$ time for trees with n vertices.

Keywords: Voronoi diagram in graphs, inverse Voronoi problem, trees, applications of binary search trees, dynamic programming in trees, lower bounds.

1 Introduction

Let T be a tree with n vertices and abstract, positive edge-lengths $\lambda\colon E(T) \to \mathbb{R}_{>0}$. The length of a path in T is the sum of the edge-lengths along the path. The (shortest-path) ***distance*** between two vertices x and y of T, denoted by $d_T(x,y)$, is the length of the unique path in T from x to y.

Let Σ be a subset of $V(T)$. We refer to each element of Σ as a ***site***, to distinguish it from an arbitrary vertex of T. The ***Voronoi cell*** of each site $s \in \Sigma$ is then defined by

$$\mathrm{cell}_T(s,\Sigma) \;=\; \{x \in V(T) \mid \forall s' \in \Sigma : d_T(s,x) \le d_T(s',x)\}.$$

The ***Voronoi diagram*** of Σ in T is

$$\mathbb{V}_T(\Sigma) \;=\; \{\mathrm{cell}_T(s,\Sigma) \mid s \in \Sigma\}.$$

When the tree is clear from the context, we remove the subindex and thus just talk about $d(\,,\,)$, $\mathrm{cell}(s,\Sigma)$ and $\mathbb{V}(\Sigma)$. It is easy to see that, for each set Σ of sites, each vertex of T belongs to some Voronoi cell. Therefore, the sets in $\mathbb{V}_T(\Sigma)$ cover all vertices of T. On the other hand, the Voronoi cells do not need to be pairwise disjoint. In particular, when some vertex of T is closest to two sites, then it is in both Voronoi cells.

---

[*] Univ Lyon, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP UMR5668, France. Email address: `edouard.bonnet@ens-lyon.fr`. Supported by the LABEX MILYON (ANR-10- LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

[†] **Corresponding author**. Faculty of Mathematics and Physics, University of Ljubljana, and IMFM, Slovenia. Supported by the Slovenian Research Agency, program P1-0297 and projects J1-8130, J1-8155, J1-9109, J1-1693. Email address: `sergio.cabello@fmf.uni-lj.si`.

[‡] Department of Mathematics, Simon Fraser University, Burnaby, BC, Canada. Email address: `mohar@sfu.ca`. On leave from IMFM & FMF, Department of Mathematics, University of Ljubljana. Supported in part by the NSERC Discovery Grant R611450 (Canada), by the Canada Research Chairs program, and by the Research Project J1-8130 of ARRS (Slovenia).

[§] Departament d'Enginyeria Informàtica i Matemàtiques, Universitat Rovira i Virgili, Tarragona, Spain. Partially supported by Grant MTM2017-86767-R from the Spanish Ministry of Economy, Industry and Competitiveness.

In this paper we consider computational aspects of the *inverse* Voronoi problem in trees. This means that we are given a collection of candidate Voronoi cells in a tree, and we would like to decide whether they form indeed a Voronoi diagram. Let us describe the problem more formally.

> GRAPHIC INVERSE VORONOI IN TREES
> Input: $(T,\mathbb{U})$, where T is a tree with positive edge-lengths and $\mathbb{U}=(U_1,\dots,U_k)$ is a sequence of subsets of vertices of T that cover $V(T)$.
> Question: Are there sites $s_1,\dots,s_k\in V(T)$ such that $\text{cell}_T(s_i,\{s_1,\dots,s_k\})=U_i$ for each $i\in\{1,\dots,k\}$? When the answer is positive, provide a solution: sites $s_1,\dots,s_k\in V(T)$ that certify the positive answer.

The *inverse Voronoi* problem can be considered also in arbitrary graphs and metric spaces. In the accompanying paper [1], we provide NP-hardness and W[1]-hardness for several different scenarios. The problem is related to questions in pattern recognition; we refer to the discussion therein. Most notably for our work, we use the framework of parameterized complexity to show that, assuming the Exponential Time Hypothesis (ETH), the inverse Voronoi problem cannot be solved for graphs G of pathwidth $p(G)$ in time $f(p(G))|V(G)|^{o(p(G))}$, for any computable function f. This result justifies considering trees as a special case.

Our results. One has to be careful with the size of the description of the input because the size of the Voronoi diagram may be quadratic in the size of the tree. For example, in a star with $2n$ leaves and sites in n of the leaves, each Voronoi cell has size $\Theta(n)$, and thus an explicit description of the Voronoi diagram has size $\Theta(n^2)$. Motivated by this, we define the ***description size*** of an instance $I=(T,(U_1,\dots,U_k))$ for the GRAPHIC INVERSE VORONOI IN TREES to be $N=N(I)=|V(T)|+\sum_i|U_i|$. We use n for the number of vertices in the tree T, which is potentially smaller than N.

We show that the problem GRAPHIC INVERSE VORONOI IN TREES can be solved in $O(N+n\log^2 n)$ time for arbitrary trees. We also show a lower bound of $\Omega(n\log n)$ in the algebraic computation tree model.

One may be tempted to think that the problem is easy for trees. Our near-linear time algorithm for arbitrary trees is far from trivial. Of course we cannot exclude the existence of a simpler algorithm running in near-linear time, but we do think that the problem is more complex than it may seem at first glance. Figure 1 may help understanding that the interaction between different Voronoi cells may be more complex than it seems.

To obtain our algorithm, we consider the following more general problem, where the input also specifies, for each Voronoi cell, a subset of vertices where the site has to be placed.

> GENERALIZED GRAPHIC INVERSE VORONOI IN TREES
> Input: $(T,\mathbb{U})$, where T is a tree with positive edge-lengths and $\mathbb{U}=\big((U_1,S_1),\dots,(U_k,S_k)\big)$ is a sequence of pairs of subsets of vertices of G.
> Question: are there sites $s_1,\dots,s_k\in V(T)$ such that $s_i\in S_i$ and $U_i=\text{cell}_T(s_i,\{s_1,\dots,s_k\})$ for each $i\in\{1,\dots,k\}$? When the answer is positive, provide a solution: sites $s_1,\dots,s_k\in V(T)$ that certify the positive answer.

Following the analogy with GRAPHIC INVERSE VORONOI IN TREES, we define the ***description size*** of an instance $I=(T,((U_1,S_1),\dots,(U_k,S_k)))$ to be $N(I)=|V(T)|+\sum_i|U_i|+\sum_i|S_i|$.

Clearly, the problem GRAPHIC INVERSE VORONOI IN TREES can be reduced to the problem GENERALIZED GRAPHIC INVERSE VORONOI IN TREES by taking $S_i=U_i$ for all $i\in\{1,\dots,k\}$. This transformation can be done in linear time (in the size of the instance). Thus, for the rest of the paper our algorithms will be for the problem GENERALIZED GRAPHIC INVERSE VORONOI IN TREES. (The lower bound holds for the original problem.)

In our solution we first make a reduction to the same problem in which Voronoi cells are disjoint, and then we make another transformation to an instance having maximum degree 3.

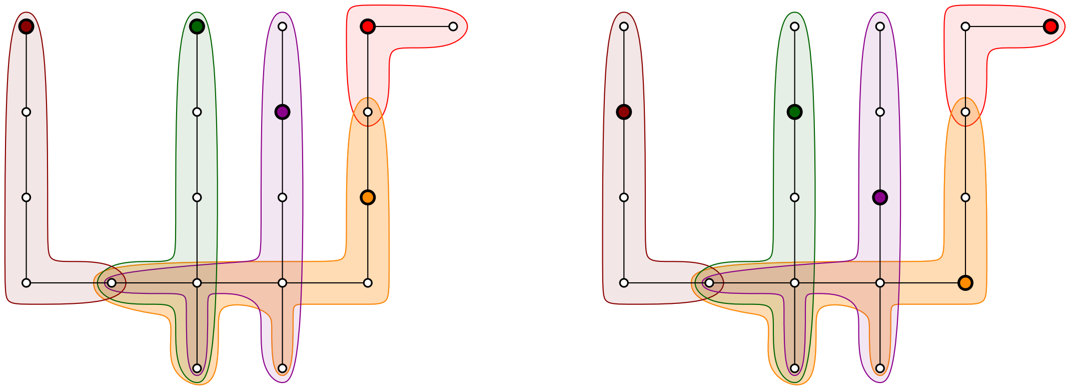


Figure 1: An instance with two solutions. The edges have unit length and the larger, filled dots represent the sites.

Finally, we employ a bottom-up dynamic programming procedure that, to achieve near-linear time, merges the information from the subproblems in time almost proportional to the smallest of the subproblems. For this, we employ dynamic binary search trees to manipulate sets of intervals.

Roadmap. In Section 2 we provide some basic tools. In Section 3 we show how to reduce the problem to a special instance where the candidate Voronoi cells are disjoint and the tree has maximum degree 3. In Section 4 we describe how to solve the problem, after the transformation, using dynamic programming. In Section 5 we provide a lower bound.

2 Basics

For a positive integer k we use the notation $[k] = \{1, \dots, k\}$.

In the following results we use T as the ground tree that defines the metric. Note that in the following claims it is important that T has positive edge-lengths. An alternative way to define cells is using strict inequalities. More precisely, for a set of sites Σ, the ***open Voronoi cell*** of each site $s \in \Sigma$ is then defined by

$$\mathrm{cell}^{<}(s,\Sigma) \;=\; \{x \in X \mid \forall s' \in \Sigma \setminus \{s\} : d(s,x) < d(s',x)\}.$$

In this case, the cells are disjoint but they do not necessarily form a partition of $V(T)$. The following two lemmas are straightforward folklore and we omit their proofs.

Lemma 1. *For each set Σ of sites and each site $s \in \Sigma$ we have $s \in \mathrm{cell}^{<}(s,\Sigma)$ and*

$$\mathrm{cell}^{<}(s,\Sigma) = \mathrm{cell}(s,\Sigma) \setminus \left(\bigcup_{s' \neq s} \mathrm{cell}(s',\Sigma) \right).$$

Lemma 2. *For each set Σ of sites, each site $s \in \Sigma$, and each vertex $v \in \mathrm{cell}(s,\Sigma)$, the path in T from s to v is contained in $T[\mathrm{cell}(s,\Sigma)]$, the subgraph of T induced by $\mathrm{cell}(s,\Sigma)$. The same statement is true for $\mathrm{cell}^{<}(s,\Sigma)$.*

A consequence of this Lemma is that the shortest path from s to $v \in \mathrm{cell}(s,\Sigma) \setminus \mathrm{cell}^{<}(s,\Sigma)$ has a part with vertices inside $\mathrm{cell}^{<}(s,\Sigma)$ followed by a part with vertices of $\mathrm{cell}(s,\Sigma) \setminus \mathrm{cell}^{<}(s,\Sigma)$.

Lemma 3. *Given an instance for the problem* GRAPHIC INVERSE VORONOI IN TREES *or the* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES, *and a candidate solution $s_1, \dots, s_k$, we can check in $O(N)$ time whether $s_1, \dots, s_k$ is indeed a solution.*

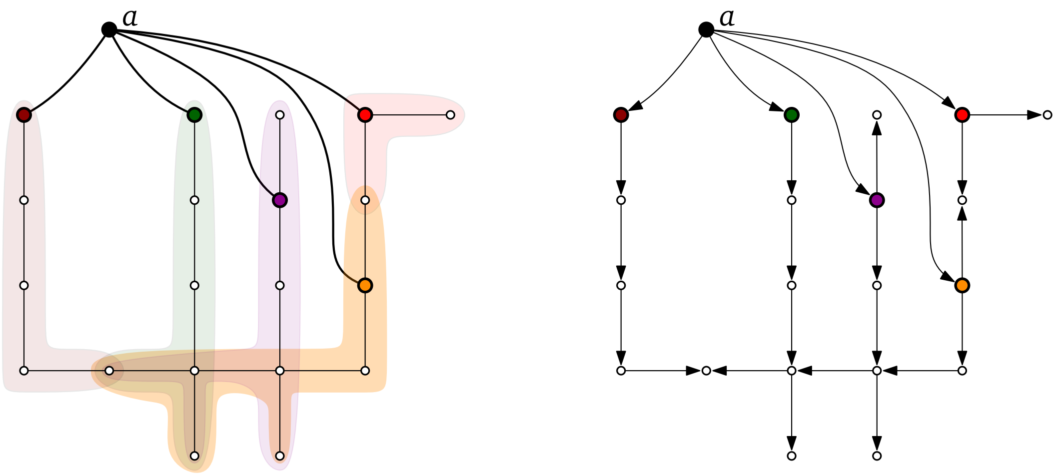


Figure 2: Construction of T_a (left) and the directed acyclic graph D_a (right).

Proof. Let T be the underlying tree defining the instance. We add a new vertex a (called the *apex*) to T and connect it to each candidate site $s_1, \dots, s_k$ with edges of the same positive length. See the left drawing in Figure 2. The resulting graph, denoted by T_a, has treewidth 2, and thus we can compute shortest paths from a to all vertices in linear time [3]. Let $d_a[v]$ be the distance in T_a from a to v.

Next we build a digraph D_a describing the shortest paths from a to all other vertices. The vertex set of D_a is $V(T) \cup \{a\} = V(T_a)$. For each arc $u \to v$, where $uv \in E(T_a)$, we add $u \to v$ to D_a if and only if $d_a[v] = d_a[u] + \lambda(uv)$. With this we obtain a directed acyclic graph D_a that contains *all* shortest paths from a to every $v \in V(T)$ and, moreover, each directed path in D_a is indeed a shortest path in T_a. See Figure 2 right.

Now we label each vertex v with the indices i of those sites s_i, whose Voronoi cells contain v, as follows. We start setting $L(s_i) = \{i\}$ for each site s_i. Then we consider the vertices $v \in V(T)$ in topological order with respect to D_a. For each vertex v, we set $L(v)$ to be the union of $L(u)$, where u iterates over the vertices of $V(T)$ with arcs in D pointing to v. It is easy to see by induction that $L(v) = \{i \in [k] \mid v \in \mathrm{cell}_T(s_i, \{s_1, \dots, s_k\})\}$. During the process we keep a counter for $\sum_v |L(v)|$, and if at some moment we detect that the counter exceeds N, we stop and report that $s_1, \dots, s_k$ is not a solution. Otherwise, we finish the process when we computed the sets $L(v)$.

Now we compute the sets $V_i = \{v \in V(T) \mid s_i \in L(v)\}$ for $i = 1, \dots k$. This is done iterating over the vertices $v \in V(T)$ and adding v to each site of $L(v)$. This takes $O(N + \sum_v |L(v)|) = O(N)$ time. Note that $V_i = \mathrm{cell}_T(s_i, \{s_1, \dots, s_k\})$. It remains to check that $U_i = V_i$ for all $i \in [k]$. For this we add flags to $V(T)$ that are initially set to false. Then, for each $i \in [k]$, we do the following: check that $|U_i| = |V_i|$, iterate over the vertices of U_i setting the flags to true, iterate over the vertices of V_i checking that the flags are true, iterate over the vertices of U_i setting the flags back to false. The procedure takes $O(N + \sum_v |L(v)|) = O(N)$ time and, if all the checks were correct, we have $U_i = V_i = \mathrm{cell}_T(s_i, \{s_1, \dots, s_k\})$ for all $i \in [k]$. ☐

3 Arbitrary trees – Transforming to nicer instances

In this section we provide a transformation to reduce the problem GENERALIZED GRAPHIC INVERSE VORONOI IN TREES to instances where the tree has maximum degree 3 and the candidate Voronoi regions are disjoint. First we show how to transform it into disjoint Voronoi regions, and then we handle the degree. In our description, we first discuss the transformation without paying attention to its efficiency. At the end of the section we discuss how the transformation can be done in linear time.

3.1 Transforming to disjoint cells

In this section we explain how to decrease the overlap between different Voronoi regions. The procedure is iterative: we consider one edge of the tree at a time and transform the instance. When there are no edges to process, we can conclude that the original instance has no solution or we can find a solution to the original instance.

Consider an instance $I = (T,((U_1,S_1),\dots,(U_k,S_k)))$ for the problem Generalized Graphic Inverse Voronoi in Trees. See Figure 1 for an example of such an instance.

For each index $i \in [k]$ we define

$$\begin{aligned} W_i &= U_i \setminus \bigcup_{j\neq i} U_j, \\ E_i &= \{uv \in E(T) \mid u \in W_i,\ v \in U_i \setminus W_i\}. \end{aligned}$$

The intuition is that each W_i should be the open Voronoi cell defined by the (unknown) site s_i, that is, the vertices of T with s_i as unique closest site; see Lemma 1. Each E_i is then the set of edges with one vertex in W_i and another vertex in $U_i \cap U_j$ for some $j \neq i$. The following result is easy to prove using Lemma 2.

Lemma 4. *Supposing that there is a solution to* Generalized Graphic Inverse Voronoi in Trees *with input I, the following hold.*

(a) *Each set U_i ($i \in [k]$) and each set W_i ($i \in [k]$) induces a connected subgraph of T.*

(a) *If two sets U_i and U_j ($i \neq j$) intersect, then $E_i \neq \emptyset$ and $E_j \neq \emptyset$.*

Proof. Consider a solution $s_1,\dots,s_k$ to Generalized Graphic Inverse Voronoi in Trees with input I, and define $\Sigma = \{s_1,\dots,s_k\}$. This means that, for each $i \in [k]$, we have $s_i \in S_i$ and $U_i = \mathrm{cell}_T(s_i,\Sigma)$. Note that because of Lemma 1, we have

$$\forall i \in [k]: \quad W_i \;=\; U_i \setminus \bigcup_{j\neq i} U_j \;=\; \mathrm{cell}_T(s_i,\Sigma) \setminus \bigcup_{j\neq i} \mathrm{cell}_T(s_j,\Sigma) \;=\; \mathrm{cell}^{<}_T(s_i,\Sigma).$$

If there are distinct indices $i,j \in [k]$ such that U_i and U_j intersect, then $W_i \subsetneq U_i$. Because of Lemma 1, we have $s_i \in W_i$, and therefore W_i is nonempty. Because of Lemma 2, the sets U_i and W_i induce subtrees of T. Since $W_i \subsetneq U_i$, it follows that T has some edge from W_i to $U_i \setminus W_i$, and therefore E_i is nonempty. □

As a preprocessing step, we replace S_i by $S_i \cap W_i$ for each $i \in [k]$. Since a site cannot belong to two Voronoi regions, this replacement does not reduce the set of feasible solutions for I. To simplify notation, we keep using I for the new instance. We check that, for each $i \in [k]$, the set S_i is nonempty and the sets U_i and W_i induce a connected subgraph of T. If any of those checks fail, we correctly report that there is no solution to I.

If the sets $U_1,\dots,U_k$ are pairwise disjoint, we do not need to do anything. If at least two of them overlap but the sets $E_1,\dots,E_k$ are empty, then Lemma 4 implies that there is no solution. In the remaining case some E_i is nonempty, and we transform the instance as follows.

In the transformations we will need "short" edges. To quantify this, we introduce the ***resolution*** $\mathrm{res}(I)$ of an instance I, defined by

$$\mathrm{res}(I) \;=\; \min\bigl(\mathbb{R}_{>0} \cap \{d_T(s_i,u) - d_T(s_j,u) \mid u \in U_i \cap U_j,\ s_i \in S_i,\ s_j \in S_j,\ i,j \in [k]\}\bigr).$$

Here we take the convention that $\min(\emptyset) = +\infty$. From the definition we have the following property:

$$\begin{aligned} &\forall i,j \in [k],\ u \in U_i \cap U_j,\ s_i \in S_i,\ s_j \in S_j: \\ &\qquad\qquad |d_T(s_i,u) - d_T(s_j,u)| < \mathrm{res}(I) \implies d_T(s_i,u) = d_T(s_j,u). \end{aligned} \tag{1}$$

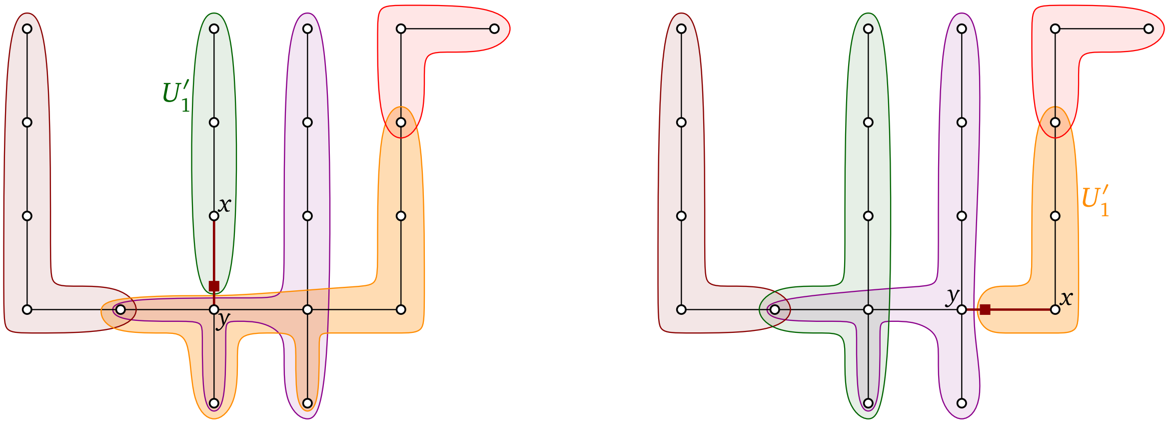


Figure 3: The transformation from the instance I in Figure 1 to I' for two different choices of the set U_1 and $xy \in E_1$. The new vertex y' appearing because of the subdivision is marked with a square. The "shorter" edges in the drawing have length ε; all other edges have unit length.

Consider any value $\varepsilon > 0$. Fix any index $i \in [k]$ such that $E_i \neq \emptyset$ and consider an edge $xy \in E_i$ with $x \in W_i$ and $y \in U_i \setminus W_i$. By renaming the sets, if needed, we assume henceforth that $i = 1$, that is, $E_1 \neq \emptyset$, $x \in W_1$ and $y \in U_1 \setminus W_1$. We build a tree T' with edge-lengths λ' and a new set U_1' as follows. We obtain T' from T by subdividing xy with a new vertex y'. We define U_1' to be the subset of vertices of U_1 that belong to the component of $T - y$ that contains x, and then we also add y' into U_1'. Note that $u \in U_1$ belongs to U_1' if and only if $d_T(u,x) < d_T(u,y)$. In particular, $y \notin U_1'$. Finally, we set the edge-lengths $\lambda'(xy') = \lambda(xy)$ and $\lambda'(yy') = \varepsilon$, and the remaining edges have the same length as in T. This completes the description of the transformation. Note that T' is just a subdivision of T and, effectively, the edge xy became a 2-edge path $xy'y$ that is longer by ε. All distances in T' are larger or equal than in T, and the difference is at most ε.

Let I' be the new instance, where we use T', λ' and U_1', instead of T, λ and U_1, respectively. (We leave U_i unchanged for each $i \in [k] \setminus \{1\}$ and S_i unchanged for each $i \in [k]$.) See Figure 3 for two examples of this transformation and Figure 4 for a schematic view. We call I' the instance obtained from I by ***expanding the edge*** xy ***from*** E_1 ***by*** ε. Note that y' is not a valid placement for a site in I', since $y' \notin S_1$.

Our definition of $\text{res}(I)$ is carefully chosen so that it does not decrease with the expansion of an edge. That is, $\text{res}(I') \geq \text{res}(I)$. (This property is exploited in the proof of Lemma 8.) This is an important but subtle point needed to achieve efficiency. It will permit that all the short edges that are introduced during the transformations have the same small length ε, and we will be able to treat ε symbolically.

The next two lemmas show the relation between solutions to the instances I and I'.

Lemma 5. *Suppose that $\varepsilon > 0$. If Σ is a solution to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input I, then Σ is also a solution to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input I'.*

Proof. We first introduce some notation. Let V_x be the vertex set of the component of $T' - y'$ that contains x and let V_y be the vertex set of the component of $T' - y'$ that contains y. See Figure 4. Note that $x \in V_x$ and $y \in V_y$, while y' is neither in V_x nor in V_y. From the definition of U_1', we have $U_1' = \{y'\} \cup (V_x \cap U_1)$ and $U_1 \setminus U_1' = V_y \cap U_1$.

We have the following easy relations between distances in T and T'; we will use them often

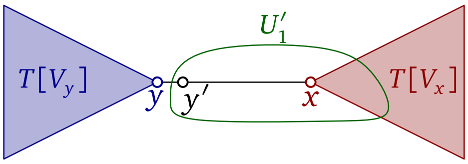


Figure 4: Notation in the proof of Lemma 5.

without explicit reference.

$$\begin{aligned}
\forall u,v \in V_x :&\quad d_{T'}(u,v) = d_T(u,v)\\
\forall u,v \in V_y :&\quad d_{T'}(u,v) = d_T(u,v)\\
\forall u \in V_x,\ v \in V_y :&\quad d_{T'}(u,v) = d_T(u,v) + \varepsilon\\
\forall u \in V_x :&\quad d_{T'}(u,y') = d_T(u,y)\\
\forall u \in V_y :&\quad d_{T'}(u,y') = d_T(u,y) + \varepsilon.
\end{aligned}$$

Consider a solution $s_1,\dots,s_k$ to GENERALIZED GRAPHIC INVERSE VORONOI IN TREES with input I, and define $\Sigma = \{s_1,\dots,s_k\}$. This means that, for all $i \in [k]$, we have $s_i \in S_i$ and $U_i = \mathrm{cell}_T(s_i,\Sigma)$. Our objective is to show that $U_1' = \mathrm{cell}_{T'}(s_1,\Sigma)$ and $U_i = \mathrm{cell}_{T'}(s_i,\Sigma)$ for all $i \in [k]\setminus\{1\}$.

Since $U_i = \mathrm{cell}_T(s_i,\Sigma)$ for all $i \in [k]$, Lemma 1 implies that $W_1 = \mathrm{cell}^{<}_T(s_1,\Sigma)$ and $s_1 \in W_1$. Since $x \in W_1$, $y \notin W_1$, and W_1 induces a connected subgraph of T because of Lemma 4(a), the set W_1 is contained in V_x. Since $W_1 \subseteq V_x$ and $W_1 \subseteq U_1$, we have $W_1 \subseteq V_x \cap U_1$ and we conclude that $W_1 \subseteq U_1'$. Furthermore, because $s_1 \in \mathrm{cell}^{<}_T(s_1,\Sigma) = W_1$ and $W_1 \subseteq V_x$, we obtain that $s_1 \in V_x$.

For each $i \in [k]\setminus\{1\}$, we have $x \notin U_i$ because $x \in W_1$, and Lemma 4(a) implies that the set $U_i = \mathrm{cell}_T(s_i,\Sigma)$ is fully contained either in V_x or in V_y.

Consider any index $\ell \in [k]\setminus\{1\}$ with the property that $y \in U_1 \cap U_\ell$. Since U_ℓ contains y, it cannot be that $U_\ell \subseteq V_x$, and therefore $U_\ell \subseteq V_y$. In particular, $s_\ell \in V_y$.

We first note that the sets $U_1', U_2, \dots, U_k$ cover $V(T')$. Indeed, since $y \in U_1 \cap U_\ell$, the sites s_1 and s_ℓ are closest sites to y in T, and using that $s_1 \in V_x$ and $s_\ell \in V_y$, we obtain that $U_1 \setminus U_1'$ is contained in U_ℓ. Further, since $U_1,\dots,U_k$ cover $V(T)$, $y' \in U_1'$ by construction, and $V(T') = V(T) \cup \{y'\}$, we conclude that indeed $U_1', U_2, \dots, U_k$ cover $V(T')$.

Next, we make the following two claims.

Claim 5.1. *$y' \in \mathrm{cell}_{T'}(s_1,\Sigma)$ and $y' \notin \mathrm{cell}_{T'}(s_i,\Sigma)$ for any $i \in [k]\setminus\{1\}$.*

Proof. Fix any index $i \in [k]\setminus\{1\}$. Consider first the case when $s_i \in V_x$. In this case the path from s_i to y' passes through x, which is a vertex in $\mathrm{cell}^{<}_T(s_1,\Sigma)$. It follows that $d_T(s_1,x) < d_T(s_i,x)$, which implies

$$d_{T'}(s_1,y') = d_T(s_1,y) < d_T(s_i,y) = d_{T'}(s_i,y').$$

Consider now the case when $s_i \in V_y$. Because $y \in U_1 = \mathrm{cell}(s_1,\Sigma)$, we have $d_T(s_1,y) \le d_T(s_i,y)$ and we conclude that

$$d_{T'}(s_i,y') = d_T(s_i,y) + \varepsilon \ge d_T(s_1,y) + \varepsilon = d_{T'}(s_1,y') + \varepsilon > d_{T'}(s_1,y').$$

In each case we get $d_{T'}(s_1,y') < d_{T'}(s_i,y')$, and the claim follows. □

Claim 5.2. *$y \notin \mathrm{cell}_{T'}(s_1,\Sigma)$.*

Proof. Since y belongs to $U_1 \cap U_\ell$, we have $d_T(s_1,y) = d_T(s_\ell,y)$. Using that U_ℓ is contained in V_y, and thus $s_\ell \in V_y$, we have

$$d_{T'}(s_\ell,y) = d_T(s_\ell,y) = d_T(s_1,y) = d_{T'}(s_1,y) - \varepsilon < d_{T'}(s_1,y).$$

We conclude that y is not an element of $\mathrm{cell}_{T'}(s_1,\Sigma)$. □

Claims 5.1 and 5.2 imply that y' belongs *only* to the Voronoi region $\mathrm{cell}_{T'}(s_1,\Sigma)$ and y does not belong to $\mathrm{cell}_{T'}(s_1,\Sigma)$. This means that each vertex of V_x belongs only to some regions $\mathrm{cell}_{T'}(s_i,\Sigma)$ with $s_i \in V_x$ and each vertex of V_y belongs to some regions $\mathrm{cell}_{T'}(s_i,\Sigma)$ with $s_i \in V_y$. That is, it cannot be that some vertex $u \in V_x$ belongs to $\mathrm{cell}_{T'}(s_i,\Sigma)$ with $s_i \in V_y$ and it cannot be that some vertex $u \in V_y$ belongs to $\mathrm{cell}_{T'}(s_i,\Sigma)$ with $s_i \in V_x$. Effectively, this means that y' splits the Voronoi diagram $\mathbb{V}_{T'}(\Sigma)$ into the part within $T'[V_x]$ and the part within $T'[V_y]$, with the gluing property that $y' \in \mathrm{cell}_{T'}(s_1,\Sigma)$. Since $U'_1 \setminus \{y'\} = U_1 \cap V_x$ and the distances within $T'[V_x]$ and within $T'[V_y]$ are the same as in T, the result follows. ☐

The converse property is more complicated. We need ε to be small enough and we also have to assume that I has a solution. It is this tiny technicality that makes the reduction nontrivial.

Lemma 6. *Suppose that* $0 < \varepsilon < \mathrm{res}(I)$ *and the answer to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input* I *is "yes". If* Σ' *is a solution to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input* I'*, then* Σ' *is also a solution to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input* I.

Proof. When the instance I has *some* solution, then the properties discussed in Lemmas 4 and 5 hold. We keep using the notation and the properties established earlier. In particular, each set U_i ($i \in [k] \setminus \{1\}$) is contained either in V_x or in V_y, and we have $W_1 \subseteq U'_1 \subseteq V_x \cup \{y'\}$.

Consider a solution $s_1,\dots,s_k$ to GENERALIZED GRAPHIC INVERSE VORONOI IN TREES with input I', and set $\Sigma = \{s_1,\dots,s_k\}$. This means that $U'_1 = \mathrm{cell}_{T'}(s_1,\Sigma)$ and, for all $i \in [k] \setminus \{1\}$, we have $U_i = \mathrm{cell}_{T'}(s_i,\Sigma)$. We have to show that, for all $i \in [k]$, we have $U_i = \mathrm{cell}_T(s_i,\Sigma)$, which implies that Σ is a solution to input I.

Like before, we split the proof into claims that show that Σ is a solution to GENERALIZED GRAPHIC INVERSE VORONOI IN TREES with input I. We start with an auxiliary property that plays a key role.

Claim 6.1. *For each* $i \in [k]$*, we have* $y \in U_i$ *if and only if* $y \in \mathrm{cell}_T(s_i,\Sigma)$*.*

Proof. Suppose first that $y \in U_i$ and $i \neq 1$. Then $U_i \subseteq V_y$. Since $y \in U_i = \mathrm{cell}_{T'}(s_i,\Sigma)$ and $y \notin U'_1 = \mathrm{cell}_{T'}(s_1,\Sigma)$, we have

$$d_T(s_i,y) = d_{T'}(s_i,y) < d_{T'}(s_1,y) = d_T(s_1,y) + \varepsilon. \tag{2}$$

Since $y' \notin U_i = \mathrm{cell}_{T'}(s_i,\Sigma)$ and $y' \in U'_1 = \mathrm{cell}_{T'}(s_1,\Sigma)$, we have

$$d_T(s_1,y) = d_{T'}(s_1,y') < d_{T'}(s_i,y') = d_T(s_i,y) + \varepsilon. \tag{3}$$

Combining (2) and (3) we get

$$|d_T(s_i,y) - d_T(s_1,y)| < \varepsilon < \mathrm{res}(I).$$

From property (1) and since $y \in U_1 \cap U_i$, we conclude that $d_T(s_1,y) = d_T(s_i,y)$. For each $s_j \in V_y$ we use that $y \in U_i = \mathrm{cell}_{T'}(s_i,\Sigma)$ to obtain

$$d_T(s_j,y) = d_{T'}(s_j,y) \geq d_{T'}(s_i,y) = d_T(s_i,y).$$

For each $s_j \in V_x$ we use that the path from s_j to y goes through $x \in U'_1 = \mathrm{cell}_{T'}(s_1,\Sigma)$ to obtain

$$d_T(s_j,y) \geq d_T(s_1,y) = d_T(s_i,y).$$

We conclude that for each $j \in [k]$ we have $d_T(s_j,y) \geq d_T(s_i,y)$, and therefore $y \in \mathrm{cell}_T(s_i,\Sigma)$.

Since $d_T(s_1,y) = d_T(s_i,y)$ whenever $y \in U_1 \cap U_i$, and $y \in U_\ell$ for some $\ell \in [k] \setminus \{1\}$, we also obtain $y \in \mathrm{cell}_T(s_1,\Sigma)$. With this we have shown one direction of the implication.

To show the other implication, consider some index $i \in [k]$ such that $y \in \mathrm{cell}_T(s_i, \Sigma)$. If $i = 1$, then $y \in U_1$ by construction, and the implication holds. So we consider the case when $i \neq 1$. First we show that it cannot be that $s_i \in V_x$. Assume, for the sake of reaching a contradiction, that $s_i \in V_x$. Because of the implication left-to-right that we showed, we have $y \in \mathrm{cell}_T(s_1, \Sigma)$. Since we have $y \in \mathrm{cell}_T(s_i, T)$ and $y \in \mathrm{cell}_T(s_1, \Sigma)$, we obtain $d_T(s_i, y) = d_T(s_1, y)$. Because $s_1, s_i \in V_x$, we obtain $d_T(s_i, x) = d_T(s_1, x)$ and therefore $d_{T'}(s_i, x) = d_{T'}(s_1, x)$. Further, since $x \in U'_1 = \mathrm{cell}'_T(s_1, \Sigma)$, we get $x \in \mathrm{cell}_{T'}(s_i, \Sigma) = U_i$, which implies $x \notin W_1$. We conclude that it must be $s_i \notin V_x$, and thus $s_i \in V_y$.

Take an index $\ell \in [k] \setminus \{1\}$ such that $y \in U_\ell$. Such an index exists because $y \notin W_1$. We have $U_\ell \subseteq V_y$ and thus $s_\ell \in V_y$. Because of the implication left-to-right that we showed, we have $y \in \mathrm{cell}_T(s_\ell, \Sigma)$. Since we have $y \in \mathrm{cell}_T(s_i, T)$ and $y \in \mathrm{cell}_T(s_\ell, \Sigma)$, we obtain $d_T(s_i, y) = d_T(s_\ell, y)$. Because $s_i, s_\ell \in V_y$ we then have

$$d_{T'}(s_i, y) = d_T(s_i, y) = d_T(s_\ell, y) = d_{T'}(s_\ell, y).$$

Since $d_{T'}(s_i, y) = d_{T'}(s_\ell, y)$ and $y \in U_\ell = \mathrm{cell}_{T'}(s_\ell, \Sigma)$, we conclude that $y \in \mathrm{cell}_{T'}(s_i, \Sigma) = U_i$. ☐

Claim 6.2. $x \in \mathrm{cell}_T(s_1, \Sigma)$ *and* $x \notin \mathrm{cell}_T(s_i, \Sigma)$ *for any* $i \in [k] \setminus \{1\}$.

Proof. Since $x \in U'_1 = \mathrm{cell}_{T'}(s_1, \Sigma)$ and $x \notin U_i = \mathrm{cell}_{T'}(s_i, \Sigma)$ for any $i \in [k] \setminus \{1\}$, we have

$$\forall i \in [k] \setminus \{1\} : \quad d_{T'}(s_1, x) < d_{T'}(s_i, x).$$

We then have

$$\forall s_i \in V_x,\ s_i \neq s_1 : \quad d_T(s_1, x) = d_{T'}(s_1, x) < d_{T'}(s_i, x) = d_T(s_i, x). \tag{4}$$

For each $s_i \in V_y$, note that the path from s_i to x passes through y, and $y \in \mathrm{cell}_T(s_1, \Sigma)$ because of Claim 6.1. Using that $s_1 \in V_x$, we have

$$\forall s_i \in V_y : \quad d_T(s_1, x) < d_T(s_i, x). \tag{5}$$

Combining (4) and (5), the claim follows. ☐

Claim 6.3. *For each* $u \in V_y$*, we have* $u \in U_1$ *if and only if* $u \in \mathrm{cell}_T(s_1, \Sigma)$*.*

Proof. Consider *some* solution $s^*_1, \dots, s^*_k$ to GENERALIZED GRAPHIC INVERSE VORONOI IN TREES with input I, and set $\Sigma^* = \{s^*_1, \dots, s^*_k\}$. This means that $U_i = \mathrm{cell}_T(s^*_i, \Sigma^*)$ for each $i \in [k]$. We also fix an index $\ell \in [k] \setminus \{1\}$ such that $y \in U_\ell \cap U_1$. Recall that $U_\ell \subseteq V_y$ because $x \notin U_\ell$, and $W_1 \subseteq V_x$ because $x \in W_1$ and $y \notin W_1$. Using Claim 6.1 and using that Σ^* is a solution to I we have

$$d_T(s_1, y) = d_T(s_\ell, y) \quad \text{and} \quad d_T(s^*_1, y) = d_T(s^*_\ell, y). \tag{6}$$

Consider some $u \in U_1 \cap V_y$. We will show that $u \in \mathrm{cell}_T(s_1, \Sigma)$. Consider the subtree $\tilde{T}$ defined by the paths connecting the vertices $s_1, s^*_1, s_\ell, s^*_\ell, u$. See Figure 5. The path from u to s^*_1 attaches to the path from s^*_ℓ to y at the vertex y. Indeed, if it attaches at another vertex $a \neq y$, then we would have $d_T(s^*_\ell, a) < d_T(s^*_1, a)$ because of (6), which would imply $d_T(s^*_\ell, u) < d_T(s^*_1, u)$, contradicting the assumption that $u \in \mathrm{cell}_T(s^*_1, \Sigma^*) = U_1$. Because W_ℓ does not contain y and W_ℓ is a connected subgraph of T (applying Lemma 2), W_ℓ is contained in a connected component of $T - y$. Further since W_ℓ contains s_ℓ and s^*_ℓ, and we replaced S_ℓ with $S_\ell \cap W_\ell$ in the preprocessing step [1], s_ℓ and s^*_ℓ are in the same component of $T - y$. Therefore, the (u, s_1)-path attaches to the (s_ℓ, y)-path at the vertex y.

---

[1] Without the replacement S_ℓ with $S_\ell \cap W_\ell$, the lemma is actually not true because it can happen that $s_\ell \in U_1 \cap U_\ell$. Indeed, we could have $s_\ell \in S_\ell \cap U_\ell \cap U_1$, which is not a valid placement in I but would be a valid placement in I'.

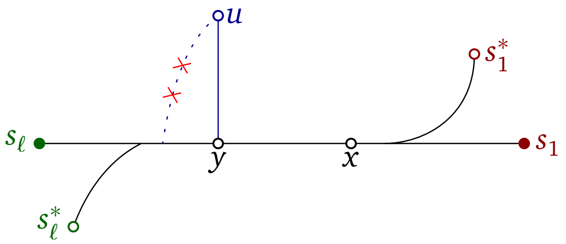


Figure 5: Situation in the proof of Claim 6.3.

Since each path from s_1, s_1^*, s_ℓ and s_ℓ^* to u passes through y, from (6) we get

$$d_T(s_1,u) = d_T(s_\ell,u) \quad \text{and} \quad d_T(s_1^*,u) = d_T(s_\ell^*,u). \tag{7}$$

Together with $u \in U_1 = \mathrm{cell}_T(s_1^*,\Sigma^*)$ we conclude that $u \in \mathrm{cell}_T(s_\ell^*,\Sigma^*) = U_\ell$. Since $u \in U_\ell = \mathrm{cell}_{T'}(s_\ell,\Sigma)$ we have

$$\forall s_j \in V_y: \quad d_T(s_1,u) = d_T(s_\ell,u) \le d_T(s_j,u).$$

Together with the fact that each $s_j \in V_x$ is no closer to u than s_1 because $x \in \mathrm{cell}_T(s_1,\Sigma)$, we conclude that $u \in \mathrm{cell}_T(s_1,\Sigma)$. This finishes the left-to-right direction of the implication.

Consider now a vertex $u \in V_y \cap \mathrm{cell}_T(s_1,\Sigma)$. Since y is on the path from s_1 to u, we obtain from (6) that $d_T(s_\ell,u) \le d_T(s_1,u)$, and therefore $u \in \mathrm{cell}_T(s_\ell,\Sigma)$. Because $u \in V_y$, $d_{T'}(s_\ell,u) = d_T(s_\ell,u)$, and distances in T' can only be larger than in T, we have $u \in \mathrm{cell}_{T'}(s_\ell,\Sigma) = U_\ell = \mathrm{cell}_T(s_\ell^*,\Sigma^*)$. This means that

$$\forall i \in [k]: \quad d_T(s_\ell^*,u) \le d_T(s_i^*,u). \tag{8}$$

Since $u \in \mathrm{cell}_T(s_1,\Sigma)$, $u \in \mathrm{cell}_T(s_\ell,\Sigma)$ and $d_T(s_1,y) = d_T(s_\ell,y)$, the vertex y is on the path from s_ℓ to u. Note that the vertices s_ℓ and s_ℓ^* must be contained in the same component of $T - y$ because the (s_ℓ,s_ℓ^*)-path must be contained W_ℓ (Lemma 2 and footnote 1), but $y \notin W_\ell$. This implies that y is also on the path from s_ℓ^* to u. Since y is also on the path from s_1^* to u, we get from (6) and (8) that

$$\forall i \in [k]: \quad d_T(s_1^*,u) = d_T(s_\ell^*,u) \le d_T(s_i^*,u).$$

It follows that $u \in \mathrm{cell}_T(s_1^*,\Sigma^*) = U_1$. This finishes the proof of the claim. □

We are now ready to prove Lemma 6: for all $i \in [k]$ we have $U_i = \mathrm{cell}_T(s_i,\Sigma)$. Consider first the case $i = 1$. Because of Claim 6.3 we have $V_y \cap U_1 = V_y \cap \mathrm{cell}_T(s_1,\Sigma)$. It remains to show that $U_1' = V_x \cap U_1 = V_x \cap \mathrm{cell}_T(s_1,\Sigma)$. Consider any vertex $u \in U_1'$. Because of Claim 6.2 we have $x \in \mathrm{cell}_T^<(s_1,\Sigma)$, and therefore $u \in V_x$ implies

$$\forall s_j \in V_y: \quad d_T(s_1,u) < d_T(s_j,u).$$

On the other hand, since $u \in U_1' = \mathrm{cell}_{T'}(s_1,\Sigma)$ we have

$$\forall s_j \in V_x: \quad d_T(s_1,u) = d_{T'}(s_1,u) \le d_{T'}(s_j,u) = d_T(s_j,u).$$

We conclude that $d_T(s_1,u) \le d_T(s_j,u)$ for all $s_j \in \Sigma$, and therefore $u \in \mathrm{cell}_T(s_1,\Sigma)$. This shows that $U_1' \subseteq V_x \cap \mathrm{cell}_T(s_1,\Sigma)$. To show the inclusion in the other direction, consider any $u \in V_x \cap \mathrm{cell}_T(s_1,\Sigma)$. We then have

$$\forall s_j \in \Sigma: \quad d_{T'}(s_1,u) = d_T(s_1,u) \le d_T(s_j,u) \le d_{T'}(s_j,u),$$

which implies $u \in \mathrm{cell}_{T'}(s_1,\Sigma) = U_1'$. This finishes the proof of $U_1 = \mathrm{cell}_T(s_1,\Sigma)$, that is, the case $i = 1$.

Figure 6: A similar transformation for arbitrary graphs does not work. On the right side we have the transformed instance with a feasible solution that does not correspond to a solution in the original setting.

Consider now the indices $i \in [k] \setminus \{1\}$ with $s_i \in V_y$. Recall that we have $U_i = \text{cell}_{T'}(s_i, \Sigma)$ and $U_i \subseteq V_y$. Fix an index $\ell \in [k] \setminus \{1\}$ such that $y \in U_\ell \cap U_1$. Such an index exists because $y \notin W_1$. We must have $U_\ell \subseteq V_y$ because $x \notin U_\ell$, and thus $s_\ell \in V_y$. Because of Claim 6.1 we have $d_T(s_1, y) = d_T(s_\ell, y)$, and using that $x \in \text{cell}^{<}_T(s_1, \Sigma)$, implied by Claim 6.2, we get

$$\forall u \in V_y,\ s_j \in V_x: \quad d_T(s_\ell, u) \le d_T(s_1, u) \le d_T(s_j, u).$$

This implies that in T each vertex of V_y has at least one closest site (from Σ) that belongs to V_y. Therefore, for each $s_i \in V_y$, we have

$$\text{cell}_T(s_i, \Sigma) = \text{cell}_{T[V_y]}(s_i, \Sigma \cap V_y).$$

A similar argument can be used for T': no site in V_x is the closest site to any vertex of V_y and the closest site to y' is s_1. Therefore, for each $s_i \in V_y$, we have

$$\text{cell}_{T'}(s_i, \Sigma) = \text{cell}_{T'[V_y]}(s_i, \Sigma \cap V_y).$$

Noting that $T[V_y] = T'[V_y]$ we obtain, for each $s_i \in V_y$,

$$U_i = \text{cell}_{T'}(s_i, \Sigma) = \text{cell}_{T'[V_y]}(s_i, \Sigma \cap V_y) = \text{cell}_{T[V_y]}(s_i, \Sigma \cap V_y) = \text{cell}_T(s_i, \Sigma).$$

It remains to consider the indices $i \in [k] \setminus \{1\}$ with $s_i \in V_x$. The approach is similar, and actually simpler because $x \in \text{cell}^{<}_T(s_1, T)$ implies that there is no influences from the sites V_y. (No care is needed for y' because it belongs to $\text{cell}^{<}_{T'}(s_1, \Sigma)$. Therefore, for each $s_i \in V_x \setminus \{s_1\}$,

$$U_i = \text{cell}_{T'}(s_i, \Sigma) = \text{cell}_{T'[V_x]}(s_i, \Sigma \cap V_x) = \text{cell}_{T[V_x]}(s_i, \Sigma \cap V_x) = \text{cell}_T(s_i, \Sigma).$$

We have covered all the cases: $s_i = s_1$, $s_i \in V_y$, and $s_i \in V_x \setminus \{s_1\}$. This finishes the proof of the Lemma. ☐

It is important to note that the transformation described above only works for trees. A similar transformation for arbitrary graphs may have feasible solutions that do not correspond to solutions in the original problem. See Figure 6 for a simple example.

Another important point is that we need the assumption that I had a solution. This means that, any solution Σ' we obtain after making a sequence of expansions, has to be tested in the original instance. However, if Σ' is not a valid solution in I, then I has no solution.

We are going to make a sequence of edge expansions. The replacement of S_i with $S_i \cap W_i$ (for $i \in [k]$) needs to be made only at the preprocessing step and it is important for correctness (see footnote 1). It is not needed later on because with each edge expansion the sets W_i (for $i \in [k]$) can only increase.

Consider an instance $I = (T, ((U_1, S_1), \dots, (U_k, S_k)))$. Set $I_0 = I$ and define, for $t \ge 1$, the instance I_t by transforming I_{t-1} using an expansion of some edge. For all expansions we use the same parameter ε. We finish the sequence when we obtain the first instance

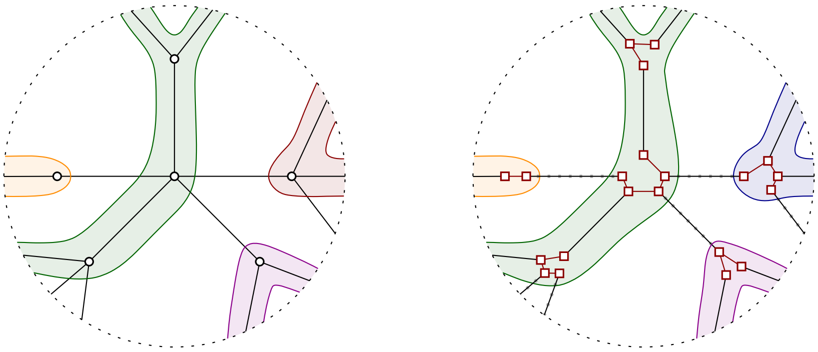


Figure 7: The behavior of the reduction to obtain maximum degree 3. Left: part of an instance with a tree of arbitrary degrees. Right: result after the reduction for the left instance. The edges between different candidate Voronoi cells are shortened by δ.

$\tilde{I} = (\tilde{T}, ((\tilde{U}_1, \tilde{S}_1), \ldots, (\tilde{U}_k, \tilde{S}_k)))$ such that the sets $\tilde{U}_1, \ldots, \tilde{U}_\ell$ are pairwise disjoint. Note that this procedure stops because the number of pairs (i, j) with $U_i \cap U_j \neq \emptyset$ decreases with each expansion. This implies that the number of steps is at most $\binom{k}{2}$. In fact, the number of steps is even smaller.

Lemma 7. *$\tilde{I}$ is reached after at most $k-1$ edge expansions.*

Proof. We prove this by induction on k. There is nothing to show if $k = 1$. Otherwise, note that the sets U_i in V_x and those in V_y (respectively) give rise to two independent subproblems with k_x and k_y sites (respectively), where $k_x + k_y = k$. By induction, the number of edge expansions is at most $1 + (k_x - 1) + (k_y - 1) = k - 1$. □

The next lemma shows that using the same parameter ε for all edge expansions is a correct choice. This is due to our careful definition of resolution $\operatorname{res}(\cdot)$.

Lemma 8. *Assume that $0 < \varepsilon < \operatorname{res}(I)$ and the answer to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input I is "yes". Then Σ is a solution to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input I if and only if Σ is also a solution to* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *with input $\tilde{I}$.*

Proof. Note that, by construction, $\operatorname{res}(I_{t-1}) \leq \operatorname{res}(I_t)$ for all $t \geq 1$. Indeed, when we expand the edge xy inserting y', then there is no set U_i that is on both sides of $T' - y'$. This means that for all the parameters s_i, s_j, u_i, u_j considered in the definition of $\operatorname{res}(I_t)$ we have $d_{T'}(s_i, u) - d_{T'}(s_j, u) = d_T(s_i, u) - d_T(s_j, u)$. Therefore, $\varepsilon < \operatorname{res}(I_t)$ for all t. The claim now follows easily from Lemmas 5 and 6 by induction on t. □

3.2 Transforming to maximum degree 3

Consider an instance $I = (T, ((U_1, S_1), \ldots, (U_k, S_k)))$ for the problem GENERALIZED GRAPHIC INVERSE VORONOI IN TREES, where T is a tree and the sets $U_1, \ldots, U_k$ are pairwise disjoint. We assume that each U_i induces a connected subgraph in T. See Figure 7 for an example of such an instance viewed around a vertex of degree > 3. We want to transform it into another instance $I' = (T', ((U'_1, S'_1), \ldots, (U'_k, S'_k)))$ where the maximum degree of T' is 3, the sets $U'_1, \ldots, U'_k$ are pairwise disjoint, and a solution to I' corresponds to a solution of I.

In the transformations we will need "short" edges again and we will shorten some edges. We need *another* version of the resolution:

$$\operatorname{res}'(I) = \min\left(\mathbb{R}_{>0} \cap \{d_T(v, u) - d_T(v', u) \mid v, v', u \in V(T)\}\right).$$

In particular, $\operatorname{res}'(I) \le \lambda(uv)$ for all edges uv of T. From the definition we have the following property:

$$\forall v, v', u \in V(T): \quad d_T(v,u) < d_T(v',u) \implies d_T(v,u) + \frac{\operatorname{res}'(I)}{2} < d_T(v',u). \tag{9}$$

We explain how to transform the instance into one where all vertices have maximum degree 3. We will use T' and λ' for the new graph and its edge-lengths. The construction uses two values δ and δ', where

$$0 < \delta < \frac{\operatorname{res}'(I)}{6n} \quad \text{and} \quad \delta' = \frac{\delta}{4n}.$$

The intuition is that edges connecting different candidate Voronoi cells are shorten by δ, and then we split the vertices of degree larger than three using short edges of length δ', where $0 < \delta' \ll \delta \ll \operatorname{res}'(I)$.

For each edge uv of T we place two vertices $a_{u,v}$ and $a_{v,u}$ in T', and connect them with an edge. If u and v belong to the same set U_i, then the length λ' of such an edge $a_{u,v}a_{v,u}$ is set to $\lambda(uv)$. If $u \in U_i$ and $v \in U_j$ with $i \neq j$, then the length λ' of such an edge $a_{u,v}a_{v,u}$ is set to $\lambda(uv) - \delta > 0$. For each vertex u of T, we connect the vertices $\{a_{u,v} \mid uv \in E(T)\}$ with a path. The length λ' of the edges on these $|V(T)|$ paths is set to δ'. Finally, for each $i \in [k]$ we define the sets

$$\begin{aligned} U'_i &= \{a_{u,v} \mid u \in U_i, uv \in E(T)\}, \\ S'_i &= \{a_{u,v} \mid u \in S_i, uv \in E(T)\}. \end{aligned}$$

Note that the sets $U'_1, \dots, U'_k$ are pairwise disjoint. For an example of the whole process see Figure 7.

To recover the solutions, we define the projection map $\pi(a_{u,v}) = u$. Thus, π sends each vertex of T' to the corresponding vertex of T that was used to create it. Note that for each $i \in [k]$ we have $\pi(S'_i) = S_i$ and $\pi(U'_i) = U_i$.

The distances in T' and T are closely related. Using that the tree T' has fewer than $2n$ new short edges of length δ' and the path connecting any two vertices of U'_i is contained in $T'[U'_i]$ we get

$$\begin{aligned} \forall i \in [k],\ u, v \in U'_i: \quad d_T(\pi(u), \pi(v)) \le d_{T'}(u,v) &< d_T(\pi(u), \pi(v)) + 2n\delta' \\ &< d_T(\pi(u), \pi(v)) + \delta. \end{aligned} \tag{10}$$

Using that the path between two vertices in different sets U_i and U_j, $i \neq j$, uses at least one edge and at most $n-1$ edges that have been shortened by δ, we get

$$\begin{aligned} \forall i \neq j \in [k],\ u \in U'_i,\ v \in U'_j: \quad d_T(\pi(u), \pi(v)) - n\delta < d_{T'}(u,v) &< d_T(\pi(u), \pi(v)) - \delta + 2n\delta' \\ &< d_T(\pi(u), \pi(v)). \end{aligned} \tag{11}$$

Lemma 9. *Suppose that* $0 < \delta < \operatorname{res}'(I)/6n$ *and the sets* $U_1, \dots, U_k$ *are pairwise disjoint subsets of* $V(T)$ *that induce connected subtrees of* T. *The answer to* $(T, ((U_1, S_1), \dots, (U_k, S_k)))$ *is "yes" if and only if the answer to* $(T', ((U'_1, S'_1), \dots, (U'_k, S'_k)))$ *is "yes".*

Proof. We first show the "if" part. Suppose that the answer to I' is "yes". Then, there exist $s'_1, \dots, s'_k$ with $s'_i \in S'_i$ and $U'_i = \operatorname{cell}_{T'}(s'_i, \{s'_1, \dots, s'_k\})$ for each $i \in [k]$. Set $s_i = \pi(s'_i)$ for all $i \in [k]$, $\Sigma' = \{s'_1, \dots, s'_k\}$ and $\Sigma = \{s_1, \dots, s_k\}$.

Consider any fixed $i \in [k]$ and any vertex $u \in U_i$. There exists some vertex $u' \in U'_i$ such that $u = \pi(u')$. Since $u' \in U'_i = \operatorname{cell}_{T'}(s'_i, \Sigma')$ and $u' \notin U'_j = \operatorname{cell}_{T'}(s'_j, \Sigma')$ for all $j \neq i$, we have

$$\forall j \in [k] \setminus \{i\}: \quad d_{T'}(s'_i, u') < d_{T'}(s'_j, u').$$

For $j \neq i$, since $u, s_i \in U_i$ and $s_j \notin U_i$ we use the relations (10) and (11) to get

$$\forall j \in [k] \setminus \{i\}: \quad d_T(s_i,u) \leq d_{T'}(s'_i,u') < d_{T'}(s'_j,u') < d_T(s_j,u).$$

We conclude that $u \in \text{cell}_T(s_i,\Sigma)$ and $u \notin \text{cell}_T(s_j,\Sigma)$ for all $j \in [k] \setminus \{i\}$. It follows that $U_i = \text{cell}_T(s_i,\Sigma)$ for all $i \in [k]$, and the answer to the instance I "yes".

Now we turn to the "only if" part. Then, there exist $s_1,\dots,s_k$ with $s_i \in S_i$ and $U_i = \text{cell}_T(s_i,\{s_1,\dots,s_k\})$ for each $i \in [k]$. Take a vertex $s'_i \in \pi^{-1}(s_i)$ for each $i \in [k]$, $\Sigma = \{s_1,\dots,s_k\}$ and $\Sigma' = \{s'_1,\dots,s'_k\}$.

Consider any fixed index $i \in [k]$ and any vertex $u' \in U'_i$. Set $u = \pi(u') \in U_i$. Since $u \in U_i = \text{cell}_T(s_i,\Sigma)$ and $u \notin U_j = \text{cell}_T(s_j,\Sigma)$ for all $j \neq i$, we have

$$\forall j \in [k] \setminus \{i\}: \quad d_T(s_i,u) < d_T(s_j,u).$$

Because of property (9) we have

$$\forall j \in [k] \setminus \{i\}: \quad d_T(s_i,u) + \frac{\text{res}'(I)}{2} < d_T(s_j,u),$$

and thus

$$\forall j \in [k] \setminus \{i\}: \quad d_T(s_i,u) + 3n \cdot \delta < d_T(s_j,u).$$

Using the relations (10) and (11) we get

$$\forall j \in [k] \setminus \{i\}: \quad d_{T'}(s'_i,u') < d_T(s_i,u) + 2n\delta < d_T(s_j,u) - n\delta < d_{T'}(s'_j,u').$$

This implies that $u' \in \text{cell}_{T'}(s'_i,\Sigma')$ and $u' \notin \text{cell}_{T'}(s'_j,\Sigma')$ for all $j \in [k] \setminus \{i\}$. It follows that $U'_i = \text{cell}_{T'}(s'_i,\Sigma')$. Since this holds for all $i \in [k]$, it follows that the answer to the instance I' "yes". ☐

3.3 Algorithm to transform

We are now ready to explain algorithmic details of the whole transformation and explain its efficient implementation.

Suppose that we have an instance $I = (T,(U_1,\dots,U_k))$ for the problem GRAPHIC INVERSE VORONOI IN TREES. Let us use n for the number of vertices in T and $N = N(I) = |V(T)| + \sum_i |U_i|$ for the description size of I. As mentioned earlier, we can convert in $O(N)$ time this to an equivalent instance $(T,((U_1,S_1),\dots,(U_k,S_k)))$ for the problem GENERALIZED GRAPHIC INVERSE VORONOI IN TREES. Let I' be this new instance and note that its description size is $O(N)$.

First, we root the tree T at an arbitrary vertex r and store for each vertex v of T its parent node $\text{pa}(v)$. (The parent of r is set to NULL.) We add to each vertex a flag to indicate whether it belongs to a subset of vertices under consideration. Initially all flags are set to false. This takes $O(|V(T)|) = O(N)$ time.

With this representation of T we can check whether any given subset U of vertices of T induces a connected subgraph in $O(|U|)$ time. The key observation is that the subgraph $T[U]$ induced by U is connected if and only if there is exactly one vertex in U whose parent does not belong to U. (Here we use the convention that for the root $\text{pa}(r) = \text{NULL} \notin U$.) To check this condition, we set the flag of the vertices of U to true, count how many vertices $v \in U$ have the property that $\text{pa}(v) \notin U$, decide the connectivity of $T[U]$ depending on the counter, and at the end set the flags of vertices of U back to false.

For each vertex $v \in V(T)$ we make a list $L(v)$ that contains the indices $i \in [k]$ with $v \in U_i$. The lists $L(v)$, for all $v \in V(T)$, can be computed in $O(N)$ time by scanning the sets $U_1,\dots,U_k$: for each $v \in U_i$ we add i to $L(v)$. Note that a vertex $v \in V(T)$ belongs to W_i if and only if i is the only index in the list $L(v)$. Thus, for any given $v \in U_i$, we can decide in $O(1)$ time whether $v \in W_i$. With this we can compute the sets $W_1, \dots, W_k$ in $O(\sum_i |U_i|) = O(N)$ time. Scanning the sets $S_1, \dots, S_k$, we can replace each set S_i with the set $S_i \cap W_i$. Together we have spent $O(N)$ time and we have made the preprocessing step described after Lemma 4.

During the algorithm, as we make the edge expansions, we maintain the lists $L(v)$ for each vertex v and the rooted representation of the tree.

Now we explain how to make the expansions of the edges in *batches*: we iterate over the indices $i \in [k]$ and, for each fixed i, we identify E_i and make all the edge expansions for E_i in $O(|U_i|)$ time. Assume for the time being that ε is already known. We will discuss its choice below.

Consider any fixed index $i \in [k]$. We compute W_i in $O(|U_i|)$ time using the lists $L(v)$ for $v \in U_i$. (The set W_i may have changed because of expansions for E_j, $j \neq i$, and thus has to be computed again.) We also check in $O(|U_i|)$ time that U_i and W_i induce connected subgraphs of T using the representation of T. (If any of them fails the test, then we correctly report that there is no solution.) We construct the induced tree $T[U_i]$ explicitly and store it using adjacency lists: for each vertex $v \in U_i$ we can find its neighbors in $T[U_i]$ in time proportional to the number of neighbors. From this point, we will use the representation of $T[U_i]$.

Next, we compute E_i, for the fixed index $i \in [k]$, in the obvious way. For each edge xy of $T[U_i]$, we check whether $x \in W_i$ and $y \notin W_i$ or whether $y \in W_i$ and $x \notin W_i$ to decide whether $xy \in E_i$. This procedure to compute E_i takes $O(|U_i|)$ time.

We keep considering the fixed index $i \in [k]$. Now we make the expansion for each edge of E_i. Here it is important that the expansion of different edges of E_i are independent: each expansion affects to U_i in a different connected component of $T - W_i$. We make the expansion of an edge $xy \in E_i$ with $x \in W_i$ and $y \in U_i \setminus W_i$ as follows: edit T by inserting y', set the new edge-lengths for the edges yy' and xy', remove from U_i the subset R_{xy} of elements of U_i that are closer to y than to x, and insert y' in U_i. The set R_{xy} of elements to be removed from U_i is obtained using the representation of $T[U_i]$ in $O(|R_{xy}|)$ time. We correct the lists $L(v)$ by removing i from $L(v)$ for each for each $v \in R_{xy}$. (We do not need to update $T[U_i]$ because the sets R_{xy} are pairwise disjoint for all $xy \in E_i$.) We conclude that expanding an edge $xy \in E_i$ takes $O(|R_{xy}|)$. Since each element of U_i can be deleted at most once from U_i, and the elements y' we insert cannot be deleted because they belong only to (the new) U_i, the expansions for the edges in E_i takes $O(|U_i|)$ time all together. This finishes the description of the work carried out for a fixed $i \in [k]$.

We iterate over all $i \in [k]$ making the expansions for (the current) edges in E_i. Since for each $i \in [k]$ we spend $O(|U_i|)$ time, all the expansions required for Lemma 8 are carried out in $O(N)$ time. All this was assuming that the value ε is available, which remains to be discussed. Let $\tilde{I}$ be the resulting instance with the disjoint sets.

Now we can make the transformation from $\tilde{I}$ to an instance with maximum degree 3. Assume for the time being that we have the parameter δ available. Then the transformation described in Section 3.2 can be easily carried out in linear time. Thus, in $O(N)$ time we obtain the final instance with pairwise disjoint sets $U_1, \dots, U_k$ and the tree T of maximum degree 3.

It remains to discuss how to choose the values of ε and δ for the transformations. It is unclear whether ε or δ can be computed in $O(N)$ time when the edges have arbitrary lengths. (If, for example, all edges have integral lengths, then we could take $\varepsilon = 1/4$, $\delta = 1/10n$ and $\delta' = 1/40n^2$.) We will handle this using composite lengths. The length of each edge e is going to be described by a triple (a, b, c) that represents the number $a + b\varepsilon + c\delta'$ for infinitesimals $\delta' \ll \varepsilon$. (Recall that $4n\delta' = \delta$.) Thus the length encoded by (a, b, c) is smaller than the length encoded by (a', b', c') if and only if (a, b, c) is lexicographically smaller than (a', b', c'). In the original graph we replace the length of each edge e by $(\lambda(e), 0, 0)$. In the expansion, the new edges yy' get length $(0, 1, 0)$, and in converting the tree to maximum degree 3 we introduce new edges of length $(0, 0, 1) \equiv \delta'$ and we replace some edges of length $(a, b, 0)$ by $(a, b, -4n)$. The length of a path becomes a triple (a, b, c) that is obtained as the vector sum of the triples over its edges. Each comparison and addition of edge-lengths costs $O(1)$ time. We summarize.

Theorem 10. *Suppose that we are given an instance I for the problem* GRAPHIC INVERSE VORONOI IN TREES *or for the problem* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *of description size $N = N(I)$ over a tree T with n vertices. In $O(N)$ time we can either detect that I has no solutions, or construct another instance I' for the problem* GENERALIZED GRAPHIC INVERSE VORONOI IN TREES *over a tree T' with the following properties:*

- *the tree T' in the instance I' has maximum degree* 3*,*
- *the sets in the instance I' are pairwise disjoint,*
- *the description size of I' and the number of vertices in T' is $O(n)$,*
- *if the answer to I is "yes", then any solution to I' is also a solution to I.*

Proof. It remains only to bound the size of T' and the description size of I'. If $k > n$, then the instance I has no solution and we report it. Otherwise, Lemma 7 implies that we are making $k-1$ expansions, which means that the resulting tree T' has $n+k-1 = O(n)$ vertices. The size of the instance I' is $O(n)$ because the sets in the instance are pairwise disjoint and there are $O(n)$ vertices in total. ☐

4 Algorithm for subcubic trees with disjoint Voronoi cells

In this section we consider the problem GENERALIZED GRAPHIC INVERSE VORONOI IN TREES for an input $(T, \mathbb{U})$, with the following properties:

- T is a tree of maximum degree 3
- $\mathbb{U}$ is a sequence of pairs $(U_1, S_1), \dots, (U_k, S_k)$ where the sets $U_1, \dots, U_k$ are pairwise disjoint.

Our task is to find sites $s_1, \dots, s_k$ such that, for each $i \in [k]$, we have $U_i = \mathrm{cell}_T(s_i, \{s_1, \dots, s_k\})$ and $s_i \in S_i$. We may assume that $V(T) = \bigcup_{i \in [k]} U_i$, that $T[U_i]$ is connected for each $i \in [k]$, and that $S_i \subseteq U_i$ for each $i \in [k]$, as otherwise it is clear that there is no solution. These conditions can easily be checked in linear time.

First, we describe an approach to decide whether there is a solution without paying much attention to the running time. Then, we describe its efficient implementation taking time $O(N \log^2 N)$, where N is the description size of the instance.

4.1 Characterization

For each vertex v, let $i(v)$ be the unique index such that $v \in U_{i(v)}$. We choose a leaf r of T as a root and henceforth consider the tree T rooted at r. We do this so that each vertex of T has at most two children. For each vertex v of T, let $T(v)$ be the subtree of T rooted at v, and define also

$$J(v) \;=\; \{j \in [k] \mid U_j \cap T(v) \neq \emptyset\}.$$

Note that $i(v) \in J(v)$. Since each U_j defines a connected subset of $T(v)$, for each $j \in J(v)$, $j \neq i(v)$, we have $U_j \subseteq T(v)$ and therefore it must be that $s_j \in T(v)$.

Consider a fixed vertex v of T and the corresponding subtree $T(v)$. We want to parameterize possible distances from v to the site $s_{i(v)}$, that is, the site whose cell contains the vertex v, that provide the desired Voronoi diagram restricted to $T(v)$. A more careful description is below. We distinguish possible placements of $s_{i(v)}$ within $T(v)$, which we refer as "below" (or on) v and for

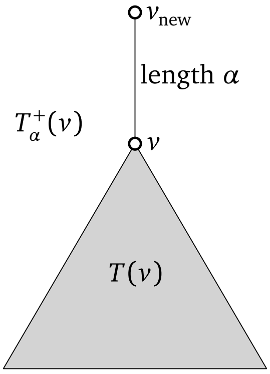


Figure 8: The tree $T^+_\alpha(v)$ used to define $A(v)$.

which we use the notation $B(v)$, and possible placements outside $T(v)$, which we refer as "above" and for which we use the notation $A(v)$.

First we deal with the placements where $s_{i(v)}$ is "below" v. In this case we start defining $X(v)$ as the set of tuples $(s_j)_{j\in J(v)}$ that satisfy the following two conditions:

$$\forall j \in J(v):\ s_j \in S_j,$$
$$\forall j \in J(v):\ \text{cell}_{T(v)}(s_j, \{s_t \mid t \in J(v)\}) \cap T(v) = U_j \cap T(v).$$

Note that $X(v) \subseteq \prod_{j\in J(v)} S_j$. Finally, we define

$$B(v) \ = \ \left\{ d_T(s_{i(v)}, v) \mid (s_j)_{j\in J(v)} \in X(v) \right\}.$$

The set $B(v)$ represents the valid distances at which we can place $s_{i(v)}$ inside $T(v)$ such that $s_{i(v)}$ is the closest site to v, and still complete the rest of the placements of the sites to get the correct portion of $\mathbb{U}$ inside $T(v)$.

Now we deal with the placements "above" v. For $\alpha > 0$, let $T^+_\alpha(v)$ be the tree obtained from $T(v)$ by adding an edge vv_{new}, where v_{new} is a new vertex, and setting the length of vv_{new} to α. The role of v_{new} is the placement of the site closest to v, when it is outside $T(v)$. See Figure 8 for an illustration. In the following discussion we also use Voronoi diagrams with respect to $T^+_\alpha(v)$. Let $Y_\alpha(v)$ be the set of tuples $(s_j)_{j\in J(v)}$ that satisfy all of the following conditions:

$$s_{i(v)} = v_{\text{new}},$$
$$\forall j \in J(v)\setminus\{i(v)\}:\ s_j \in S_j,$$
$$\forall j \in J(v):\ \text{cell}_{T^+_\alpha(v)}(s_j, \{s_t \mid t \in J(v)\}) \cap T(v) = U_j \cap T(v).$$

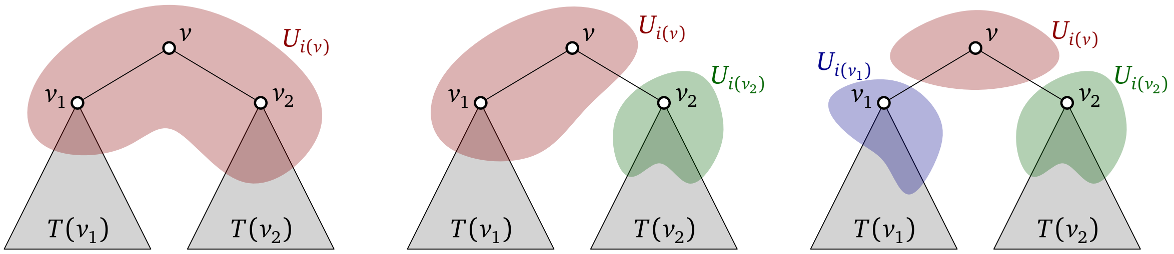
Finally we define

$$A(v) \ = \ \{\alpha \in \mathbb{R}_{>0} \mid Y_\alpha(v) \neq \emptyset\}.$$

We are interested in deciding whether $B(r)$ is nonempty. Indeed, for the root r we have $J(r) = [k]$ and $T(r) = T$ by construction. The definition of $X(v)$ implies that $B(r)$ is nonempty if and only if there is some tuple $(s_1,\dots,s_k) \in S_1 \times \cdots \times S_k$ such that

$$\forall i \in J(r) = [k]:\ \ \text{cell}_T(s_i, \{s_1,\dots,s_k\}) = \text{cell}_{T(r)}(s_i, \{s_1,\dots,s_k\}) = U_i \cap T(r) = U_i.$$

This is precisely the condition we have to check to solve GENERALIZED GRAPHIC INVERSE VORONOI IN TREES.



$i(v) = i(v_1) = i(v_2)$ $i(v) = i(v_1) \neq i(v_2)$ $i(v_1) \neq i(v) \neq i(v_2)$

Figure 9: Different cases in the computation of $A(v)$ and $B(v)$ when v has children v_1 and v_2. (The case $i(v) = i(v_2) \neq i(v_1)$ is symmetric to the case $i(v) = i(v_1) \neq i(v_2)$.)

We are going to compute $A(v)$ and $B(v)$ in a bottom-up fashion along the tree T. If v is leaf of T, then $J(v) = \{i(v)\}$ and clearly we have

$$A(v) = \mathbb{R}_{>0} \quad \text{and} \quad B(v) = \begin{cases} \{0\} & \text{if } v \in S_{i(v)}, \\ \emptyset & \text{if } v \notin S_{i(v)}. \end{cases}$$

Consider now a vertex v of T that has two children v_1 and v_2. Assume that we already have $A(v_j)$ and $B(v_j)$ for $j = 1, 2$. For $j = 1, 2$ define the sets

$$\begin{aligned} A'(v_j) &= \{x - \lambda(vv_j) \mid x \in A(v_j)\}, \\ B'(v_j) &= \{x + \lambda(vv_j) \mid x \in B(v_j)\}, \\ C'(v_j) &= \{\alpha \mid \exists x \in B(v_j) \text{ such that } x - \lambda(vv_j) < \alpha < x + \lambda(vv_j)\}. \end{aligned}$$

This is the offset we obtain when we take into account the length of the edge vv_j. The set $C'(v_j)$ will be relevant for the case when $i(v) \neq i(v_j)$. The following lemmas show how to compute $A(v)$ and $B(v)$ from its children. Figure 9 is useful to understand the different cases.

Lemma 11. *If the vertex v has two children v_1 and v_2, then*

$$A(v) = \mathbb{R}_{>0} \cap \begin{cases} A'(v_1) \cap A'(v_2) & \text{if } i(v) = i(v_1) = i(v_2), \\ A'(v_1) \cap C'(v_2) & \text{if } i(v) = i(v_1) \neq i(v_2), \\ A'(v_2) \cap C'(v_1) & \text{if } i(v) = i(v_2) \neq i(v_1), \\ C'(v_1) \cap C'(v_2) & \text{if } i(v) \neq i(v_1) \text{ and } i(v) \neq i(v_2). \end{cases}$$

Proof. This is a standard proof in dynamic programming. We only point out the main insight showing the role of $A'(v_j)$ and $C'(v_j)$ for $j \in \{1, 2\}$.

When $i(v) = i(v_j)$, placing $s_{i(v)}$ at v_{new} of the tree $T^+_\alpha(v)$ is the same as placing it at v_{new} of $T^+_{\alpha+\lambda(vv_j)}(v_j)$. The valid values α for $T^+_{\alpha+\lambda(vv_j)}(v_j)$ are described by $A'(v_j)$, a shifted version of $A(v_j)$.

When $i(v) \neq i(v_j)$, there has to be a compatible placement of $s_{i(v_j)}$ inside $T(v_j)$ such that v is closer to $s_{i(v)} = v_{\text{new}}$ than to $s_{i(v_j)}$, while v_j is closer to $s_{i(v_j)}$ than to $s_{i(v)}$. That is, we must have

$$d_T(v_{\text{new}}, v) < d_T(s_{i(v_j)}, v) \quad \text{and} \quad d_T(s_{i(v_j)}, v_j) < d_T(v_{\text{new}}, v_j),$$

or equivalently, α must satisfy

$$\alpha < d_T(s_{i(v_j)}, v_j) + \lambda(vv_j) \quad \text{and} \quad d_T(s_{i(v_j)}, v_j) < \alpha + \lambda(vv_j).$$

Thus, each possible value x of $d_T(s_{i(v_j)}, v_j)$, that is, each $x \in B(v_j)$, gives the interval $\big(x - \lambda(vv_j), x + \lambda(vv_j)\big)$ of possible values for α. The union of these intervals over $x \in B(v_j)$ is precisely $C'(v_j)$. ☐

To construct $B(v)$ it is useful to have a function that tells whether v is a valid placement for $s_{i(v)}$. For this matter we define the following function:

$$\chi(v) \;=\; \begin{cases} \{0\} & \text{if } i(v)=i(v_1)=i(v_2),\ v\in S_{i(v)},\ 0\in A'(v_1) \text{ and } 0\in A'(v_2),\\ \{0\} & \text{if } i(v)=i(v_1)\neq i(v_2),\ v\in S_{i(v)},\ 0\in A'(v_1) \text{ and } 0\in C'(v_2),\\ \{0\} & \text{if } i(v)=i(v_2)\neq i(v_1),\ v\in S_{i(v)},\ 0\in A'(v_2) \text{ and } 0\in C'(v_1),\\ \{0\} & \text{if } i(v)\neq i(v_1),\ i(v)\neq i(v_2),\ v\in S_{i(v)},\ 0\in C'(v_1) \text{ and } 0\in C'(v_2),\\ \emptyset & \text{otherwise.} \end{cases}$$

Lemma 12. *If the vertex v has two children v_1 and v_2, then*

$$B(v) \;=\; \chi(v)\cup \begin{cases} (B'(v_1)\cap A'(v_2))\cup(B'(v_2)\cap A'(v_1)) & \textit{if } i(v)=i(v_1)=i(v_2),\\ B'(v_1)\cap C'(v_2) & \textit{if } i(v)=i(v_1)\neq i(v_2),\\ B'(v_2)\cap C'(v_1) & \textit{if } i(v)=i(v_2)\neq i(v_1),\\ \emptyset & \textit{if } i(v)\neq i(v_1) \textit{ and } i(v)\neq i(v_2). \end{cases}$$

Proof. First we note that $\chi(v)=\{0\}$ if and only if v is a valid placement for $s_{i(v)}$. Indeed, the formula is the same that was used for $A(v)$, but for the value $\alpha=0$, and it takes into account whether $v\in S_{i(v)}$.

The proof for the correctness of $B(v)$ is again based in standard dynamic programming. The case for $s_{i(v)}$ being placed at v is covered by $\chi(v)$. The main insight for the case when $s_{i(v)}$ is placed in $T(v_1)$ is that, from the perspective of the other child, v_2, the vertex is placed "above" v_2. That is, only the distance from $s_{i(v)}$ to v_2 is relevant. Thus, we have to combine $B(v_1)$ and $A(v_2)$, with the appropriate shifts. More precisely, for v_2 we have to use $A'(v_2)$ or $C'(v_2)$ depending on whether $i(v_2)=i(v)$ or $i(v_2)\neq i(v)$. ☐

When v has a unique child v', then the formulas are simpler and the argumentation is similar. We state them for the sake of completeness without discussing their proof.

$$A(v) \;=\; \mathbb{R}_{>0}\cap \begin{cases} A'(v') & \text{if } i(v)=i(v'),\\ C'(v') & \text{if } i(v)\neq i(v'). \end{cases}$$

$$B(v) \;=\; \begin{cases} B'(v')\cup\{0\} & \text{if } i(v)=i(v'),\ v\in S_{i(v)}, \text{ and } \lambda(vv')\in A(v'),\\ B'(v') & \text{if } i(v)=i(v') \text{ and } \big(v\notin S_{i(v)} \text{ or } \lambda(vv')\notin A(v')\big),\\ \{0\} & \text{if } i(v)\neq i(v'),\ v\in S_{i(v)} \text{ and } 0\in C'(v'),\\ \emptyset & \text{if } i(v)\neq i(v') \text{ and } (v\notin S_{i(v)} \text{ or } 0\notin C'(v')). \end{cases}$$

4.2 Efficient manipulation of monotonic intervals

The efficient algorithm that we will present is based on an efficient representation of the sets $A(v)$ and $B(v)$ using binary search trees. Here we discuss the representation that we will be using.

We first consider how to store a set X of real values under the following operations.

- Copy makes a copy of the data structure storing X;
- Report returns the elements of X sorted;
- Insert(y) adds a new element y in X;
- Delete(y) removes the element $y\in X$ from X;

- $\mathsf{Succ}(y)$ returns the successor of y in X, defined as the smallest number in X that is at least as large as y;

- $\mathsf{Pred}(y)$ returns the predecessor of y in X, defined as the largest number in X that is smaller or equal than y;

- $\mathsf{Split}(y)$ returns the representation for $X_{\leq} = \{x \in X \mid x \leq y\}$ and the representation for $X_{>} = \{x \in X \mid x > y\}$; the representation of X is destroyed in the process;

- $\mathsf{Join}(X_1, X_2)$ returns the representation of $X = X_1 \cup X_2$ if $\max(X_1) < \min(X_2)$, and otherwise it returns an error; the representations of X_1 and X_2 are destroyed in the process;

- $\mathsf{Shift}(\alpha)$ adds the given value α to all the elements of X.

These operations can be done efficiently using dynamic balanced binary search tree with so-called augmentation, that is, with some extra information attached to the nodes. Strictly speaking the following result is not needed, but understanding it will be useful to understand the more involved data structure we eventually employ.

Theorem 13. *There is an augmented dynamic binary search tree to store sets of m real values with the following time guarantees:*

- *the operations Copy and Report take $O(m)$ time;*

- *the operations Insert, Delete, Succ, Pred, Split, Join and Shift take $O(\log m)$ time. (For Join the value m is the size of the resulting set.)*

Proof. Let X be the set of values to store. We use a dynamic balanced binary search tree $\mathcal{T}$ where each node represents one element of X. For each node μ of $\mathcal{T}$, let $x(\mu)$ be the value represented by μ. The tree $\mathcal{T}$ is a binary search tree with respect to the values $x(\mu)$. However, we *do not* store $x(\mu)$ explicitly at μ, but we store it in so-called difference form. At each non-root node μ with parent μ', we store $\mathsf{diff\text{-}val}(\mu) := x(\mu) - x(\mu')$. At the root r we store $\mathsf{diff\text{-}val}(r) = x(r)$. (This choice is consistent with using $x(\text{NULL}) = 0$.) This is a standard technique already used by Tarjan [6]. Whenever we want to obtain $x(\mu)$ for a node μ, we have to add $\mathsf{diff\text{-}val}(\mu')$ for the nodes μ' along the root-to-μ path. Since operations on a tree are performed always locally, that is, accessing a node from a neighbour, we spend $O(\log m)$ time to compute the first value $x(\mu)$, and from there on each value $x(\cdot)$ is computed in $O(1)$ additional time from the value of its neighbor. Of course, the values $\mathsf{diff\text{-}val}(\mu)$ have to be updated through the changes in the tree, including rotations or other balancing operations.

With this representation it is trivial to perform the operation $\mathsf{Shift}(\alpha)$ in constant time: at the root r of $\mathcal{T}$, we just add α to $\mathsf{diff\text{-}val}(r)$.

For the rest of operations, the time needed to execute them is the same as for the dynamic balanced search trees we employ. Brass [2, Chapter 3] explains dynamic trees with the requested properties; see Section 3.11 of the book for the more complex operations of split and join. (The same time bounds with amortized time bounds, which are sufficient for our application, can be obtained using the classical splay trees [5].) □

Consider now a family $\mathbb{I}$ of closed intervals on the real line. The family $\mathbb{I}$ is ***monotonic*** if no interval contains another interval. In a monotonic family of intervals, the left endpoints have to be distinct and the right endpoints also have to be distinct. Also, for such a family, sorting the intervals by their left endpoints or their right endpoints gives the same result. Because of this, we can talk about the ordering of the intervals, and we can also talk about the rightmost or leftmost interval in $\mathbb{I}$ with a certain property.

We want to maintain a set $\mathbb{I}$ of monotonic intervals under the following operations.

- IntCopy makes a copy of the data structure storing $\mathbb{I}$.

- IntReport returns the elements of $\mathbb{I}$ sorted by their left endpoint.

- IntInsert(J) adds a new interval J in $\mathbb{I}$; it assumes that the resulting family keeps being monotonic.

- IntDelete(J) deletes the interval $J \in \mathbb{I}$.

- IntHitBy(J), for an interval J, returns whether J intersects some interval of $\mathbb{I}$.

- IntContaining(J), for an interval J, returns the representation for $\mathbb{I}' = \{I \in \mathbb{I} \mid J \subseteq I\}$ and the representation for $\mathbb{I}'' = \mathbb{I} \setminus \mathbb{I}'$. The representation of $\mathbb{I}$ is destroyed in the process.

- IntClip(J), for an interval $J = [x, y]$, returns the representation for the intervals $\mathbb{I}' = \{I \cap J \mid I \in \mathbb{I}\}$ and for the intervals $\mathbb{I}'' = \{I \cap (-\infty, x] \mid I \in \mathbb{I}\} \cup \{I \cap [y, +\infty) \mid I \in \mathbb{I}\}$. In both cases we remove empty intervals, and remove intervals contained in another one, so that we keep having monotonic families. The representation of $\mathbb{I}$ is destroyed in the process.

- IntJoin($\mathbb{I}_1, \mathbb{I}_2$) returns the representation of $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$ if $\mathbb{I}$ is a monotonic family and all the intervals of $\mathbb{I}_1$ are to the left of all the intervals of $\mathbb{I}_2$. Otherwise it returns an error. The representations of $\mathbb{I}_1$ and $\mathbb{I}_2$ are destroyed in the process.

- IntShift(α), for a given real value α, shifts all the intervals by α; this is, each interval $[a, b]$ in $\mathbb{I}$ is replaced by $[a + \alpha, b + \alpha]$.

- IntExtend(λ), for a given real value $\lambda > 0$, extends all the intervals by λ in both directions; this is, each interval $[a, b]$ in $\mathbb{I}$ is replaced by $[a - \lambda, b + \lambda]$.

Theorem 14. *There is a data structure to store monotonic families of m intervals with the following time guarantees:*

- *the operations IntCopy and IntReport take $O(m)$ time;*

- *the operations IntInsert, IntDelete, IntHitBy, IntContaining, IntClip, IntJoin, IntShift, IntExtend take $O(\log m)$ time. (For IntJoin the value m is the size of the resulting set $\mathbb{I}$.)*

Proof. Let $\mathbb{I}$ be the family of monotonic intervals to store. We use a dynamic balanced binary search tree $\mathcal{T}$ where each node represents one element of $\mathbb{I}$. For the node μ of $\mathcal{T}$ that represents the interval I, let $a(\mu)$, $b(\mu)$ and $\ell(\mu) = b(\mu) - a(\mu)$ be the left endpoint, the right endpoint, and the length of I, respectively. Thus, if μ represents $[a_i, b_i]$, we have $a_i = a(\mu)$ and $b_i = a(\mu) + \ell(\mu)$.

The tree $\mathcal{T}$ is a binary search tree with respect to the values $a(\mu)$. Because the family of intervals is monotonic, $\mathcal{T}$ is *also* a binary search tree with respect to the values $b(\mu)$. However, the values $a(\mu)$, $b(\mu)$ or $\ell(\mu)$ are *not stored explicitly*. Instead, the values are stored in difference form and implicitly. More precisely, at each node μ of $\mathcal{T}$ we store two values, diff-val(μ) and diff-len(μ), defined as follows. If μ is the root of the tree and represents the interval $[a, b]$, then diff-val(μ) $= a$ and diff-len(μ) $= b - a$. If μ is a non-root node of the tree representing $[a, b]$, and μ' is its parent, then diff-val(μ) $= a -$ diff-val(μ') and diff-len(μ) $= (b - a) -$ diff-len(μ').

This is an extension of the technique employed in the proof of Theorem 13. In fact, $\mathcal{T}$ is just the tree in the proof of Theorem 13 for the left endpoints of the intervals, where additionally each node stores information about the length of the interval, albeit this additional information is stored also in difference form.

Whenever we want to obtain $a(\mu)$ or $\ell(\mu)$ for a node μ, we have to add diff-val(μ') or diff-len(μ') for the nodes μ' along the root-to-μ path, respectively. The right endpoint $b(\mu)$ is obtained from $b(\mu) = a(\mu) + \ell(\mu)$. Since operations in a tree always go from a node to a neighbor, we can assume that the values $a(\mu)$, $b(\mu)$ and $\ell(\mu)$ are available at a cost of $O(1)$ time per node, after an initial cost of $O(\log m)$ time to compute the values at the first node. Of course, the values $\mathsf{diff\text{-}val}(\mu)$ and $\mathsf{diff\text{-}len}(\mu)$ have to be updated through the changes in the tree, including rotations or other balancing operations.

Since $\mathcal{T}$ is a binary search tree with respect to the values $a(\cdot)$ and also with respect to the values $b(\cdot)$, we can make the usual operations that can be performed in a binary search tree, such as predecessor or successor, with respect to any of those two keys. For example, we can get in $O(\log m)$ time the rightmost interval that contains a given value y, which amounts to a predecessor query with y for the values $a(\cdot)$, or we can get the leftmost interval that contains a given value y, which amounts to a successor query with y for the values $b(\cdot)$.

With this representation, it is trivial to perform the operations $\mathsf{IntShift}(\alpha)$ or $\mathsf{IntExtend}(\lambda)$ in $O(1)$ time. We just update $\mathsf{diff\text{-}val}$ or $\mathsf{diff\text{-}len}$ at the root.

The operations $\mathsf{IntCopy}$, $\mathsf{IntReport}$, $\mathsf{IntInsert}$ and $\mathsf{IntDelete}$ can be carried out as normal operations in a dynamic binary search tree. The operation $\mathsf{IntJoin}$ is also just the join operation for trees.

For the operation $\mathsf{IntHitBy}(J)$ with $J=[x,y]$ we make a predecessor and a successor query with x for the values $a(\cdot)$. This gives the two intervals $I_1, I_2 \in \mathbb{I}$ such that x is between the left endpoint of I_1 and I_2. We then check whether $I_1 \cup I_2$ intersect J, which requires constant time.

For the operation $\mathsf{IntContaining}(J)$ we proceed as follows. We find the rightmost interval $[a_\ell, b_\ell] \in \mathbb{I}$ with the left endpoint outside J. We find the rightmost interval $[a_r, b_r] \in \mathbb{I}$ with the right endpoint inside J. Because $\mathbb{I}$ is a monotonic family of intervals, the intervals contained in J are precisely those with the right endpoint in the half-open interval $(a_\ell, a_r]$. We use the operations $\mathsf{Split}(a_\ell)$ and $\mathsf{Split}(a_r)$ with respect to the values $a(\cdot)$ to obtain the representations of

$$\begin{aligned}
\mathbb{I}_1 &= \{[a,b] \in \mathbb{I} \mid a \le a_\ell\},\\
\mathbb{I}_2 &= \{[a,b] \in \mathbb{I} \mid a_\ell < a \le a_r\} \;=\; \{I \in \mathbb{I} \mid J \subseteq I\},\\
\mathbb{I}_3 &= \{[a,b] \in \mathbb{I} \mid a_r < a\}.
\end{aligned}$$

We then use the Join operation to merge the representations of $\mathbb{I}_1$ and $\mathbb{I}_3$.

For the operation $\mathsf{IntClip}(J)$ with the interval $J=[x,y]$ we proceed as follows. We use $\mathsf{IntContaining}([x,x])$, $\mathsf{IntContaining}([y,y])$, and $\mathsf{IntJoin}$ to separate $\mathbb{I}$ into the group $\mathbb{I}'$ of intervals pierced by x or y, and the rest, $\mathbb{I}''$. Then we use $\mathsf{Split}(x)$ (with respect to $a(\cdot)$) and $\mathsf{Split}(y)$ (with respect to $a(\cdot)$) to split $\mathbb{I}''$ into three groups: $\mathbb{I}_1$ containing intervals of $\mathbb{I}$ contained in $(-\infty,x]$, $\mathbb{I}_2$ containing intervals of $\mathbb{I}$ contained in $[x,y]$, and $\mathbb{I}_3$ containing intervals of $\mathbb{I}$ contained in $[y,+\infty)$. In $\mathbb{I}'$ we find the leftmost interval that contains x, clip it with $(-\infty,x]$, and add it to $\mathbb{I}_1$. Again in the same group, $\mathbb{I}'$, we find the rightmost interval that contains x, clip it with $[x,y]$, and add it to $\mathbb{I}_2$. We do a similar procedure for y: add to $\mathbb{I}_2$ the leftmost interval of $\mathbb{I}'$ that contains y, clipped with J, and add to $\mathbb{I}_3$ the rightmost interval of $\mathbb{I}'$ that contains y, clipped with $[y,+\infty)$. If the two intervals we added to $\mathbb{I}_2$ are the same, which means that they both are $[x,y]$, we only add one of them. The procedure takes $O(\log m)$ time. $\square$

Consider a set $A \subseteq \mathbb{R}$. A ***representation*** of A if a family $\mathbb{I}$ of monotonic intervals such that $A = \bigcup_{I \in \mathbb{I}} I$. The intervals in $\mathbb{I}$ *may intersect* and the representation is not uniquely defined. See Figure 10 for an example. The *size* of the representation $\mathbb{I}$ is the number of (possibly non-disjoint) intervals in $\mathbb{I}$. This is potentially larger than the minimum number of intervals that is needed because the intervals in $\mathbb{I}$ can intersect.

Consider some set A and its representation $\mathbb{I}$. If we use the data structure of Theorem 14 to store $\mathbb{I}$, the operations reflect operations we do with A. For example, $\mathsf{IntHitBy}(J)$ tells whether J intersects A, while $\mathsf{IntClip}([x,y])$ returns a representation of $A \cap [x,y]$ and a representation of $A \cap (-\infty,x] \cup A \cap [y,+\infty)$. The operation $\mathsf{IntContaining}(J)$ will be used only when $\mathbb{I}$ is a set of zero-length intervals, and in that case it returns a representation of $A \cap J$. When $\mathbb{I}_1$ and $\mathbb{I}_2$ are

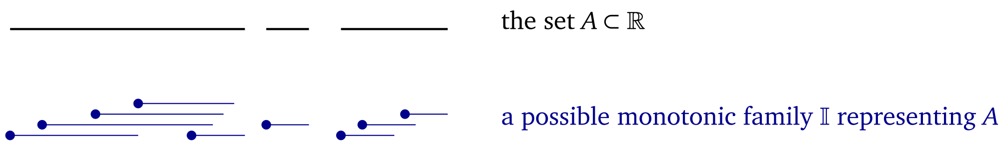


Figure 10: The set A at the top and one possible representation $\mathbb{I}$ of A. The size of this representation is 9.

representations of A_1 and A_2, then $\mathsf{IntJoin}(\mathbb{I}_1, \mathbb{I}_2)$ returns the representation of $A_1 \cup A_2$, assuming that $\max(A_1) < \min(A_2)$.

4.3 Algorithm

In this section we present an efficient algorithm based on the characterization of the previous section. We keep using the same notation. In particular, T keeps being a rooted tree and each vertex has at most two children. We use n for the number of vertices of T.

There are two main ideas used in our approach. The first one is that, for each vertex of the tree with two children, we want to spend time (roughly) proportional to the size of the smaller subtree of its children. The second idea is to use representations of $A(v)$ and $B(v)$ and manipulate them using the data structure of Theorem 14.

The following lemma, which is folklore, shows the advantage of the first idea. For each node v with two children, let v_1 and v_2 be its two children. If v has only one child, we denote it by v_1. For each node v, let $n(v)$ be the number of vertices in the subtree $T(v)$. (Thus $n(r) = n$.)

Lemma 15. *If V_2 denotes the vertices of T with two children, then*

$$\sum_{v \in V_2} \min\{n(v_1), n(v_2)\} = O(n \log n).$$

Proof. For each vertex u of T define

$$\sigma(u) \;=\; \sum_{v \in V_2 \cap V(T(u))} \min\{n(v_1), n(v_2)\}.$$

Thus, we want to bound $\sigma(r)$. We show by induction on $n(u)$ that

$$\sigma(u) \;\le\; n(u) \log_2 n(u).$$

For the base case note that, when $n(u) = 1$, the vertex u is a leaf and $\sigma(u) = 0$, so the statement holds.

If u has one child u_1, then we have $V_2 \cap T(u) = V_2 \cap T(u_1)$,

$$\sigma(u) \;=\; \sigma(u_1) \;\le\; n(u_1) \log_2 n(u_1) \;\le\; n(u) \log_2 n(u),$$

and the bound holds. If u has two children u_1 and u_2, then we can assume without loss of generality that $n(u_1) \le n(u_2)$, which implies that $n(u_1) < n(u)/2$. Using the induction hypothesis for $n(u_1)$ and $n(u_2)$, we obtain

$$\begin{aligned}
\sigma(u) &= \sum_{v\in V_2\cap V(T(u))} \min\{n(v_1), n(v_2)\}\\
&= \sigma(u_1)+\sigma(u_2)+n(u_1)\\
&\le n(u_1)\log_2 n(u_1)+n(u_2)\log_2 n(u_2)+n(u_1)\\
&< n(u_1)\log_2 (n(u)/2)+n(u_2)\log_2 n(u)+n(u_1)\\
&= n(u_1)\big(\log_2 n(u)-1\big)+n(u_2)\log_2 n(u)+n(u_1)\\
&= \big(n(u_1)+n(u_2)\big)\log_2 n(u)\\
&< n(u)\log_2 n(u). \qquad \square
\end{aligned}$$

We manipulate the sets $A(v)$ and $B(v)$ using representations $\mathbb{I}(A(v))$ and $\mathbb{I}(B(v))$, respectively. In the case of $B(v)$, since $B(v)$ is a finite set of values, the family $\mathbb{I}(B(v))$ consists of zero-length monotonic intervals. The reason for this artificial approach to treat $B(v)$, as opposed to using a set of values, is that in our algorithm sometimes we set the lengths of intervals defined by $B(v)$. Thus, there is no real difference between how we treat the representations of $A(\cdot)$ and $B(\cdot)$.

The families of intervals $\mathbb{I}(A(v))$ and $\mathbb{I}(B(v))$ are stored and manipulated using the data structure of Theorem 14. Thus, we are using the data structure described in Theorem 14 to represent $A(v)$ and $B(v)$ implicitly, as the union of monotonic intervals. The reason for this choice is technical and reflected in the proof of the next lemma.

For each vertex v of T, we use $m_A(v)$ and $m_B(v)$ to denote the sizes of $\mathbb{I}(A(v))$ and $\mathbb{I}(B(v))$, respectively. Although the value $m_A(v)$ actually depends on the family $\mathbb{I}(A(v))$ of intervals that is used, this relaxation of the notation will not lead to confusion.

It is clear that $B(v)$ has at most $n(v)$ values because each value corresponds to a vertex of $T(v)$. Thus, $m_B(v) \le n(v)$. A similar bound will hold for $m_A(v)$ by induction.

Lemma 16. *Consider a vertex v of T with two children v_1 and v_2, and assume that we have representations $\mathbb{I}(A(v_1))$, $\mathbb{I}(B(v_1))$, $\mathbb{I}(A(v_2))$ and $\mathbb{I}(B(v_2))$ of $A(v_1)$, $B(v_1)$, $A(v_2)$ and $B(v_2)$, respectively, each of them stored in the data structure of Theorem 14. Set $m_1 = m_A(v_1)+m_B(v_1)$ and $m_2 = m_A(v_2)+m_B(v_2)$, and assume that $m_1 \le m_2$. We can compute in $O(m_1 \log m_2)$ time families $\mathbb{I}(A(v))$ and $\mathbb{I}(B(v))$ that represent $A(v)$ and $B(v)$, respectively, each of them stored in the data structure of Theorem 14.*[2] *Moreover, the representation $\mathbb{I}(A(v))$ has size at most*

$$\max\{m_A(v_1)+m_A(v_2),\ m_A(v_1)+m_B(v_2),\ m_B(v_1)+m_A(v_2),\ m_B(v_1)+m_B(v_2)\}.$$

Proof. First we compute $\chi(v)$. To check whether $0\in A'(v_j)$, where $j\in\{1,2\}$, we perform the operation $\mathsf{IntHitBy}([\lambda(vv_j),\lambda(vv_j)])$ in the representation $\mathbb{I}(A(v_j))$. To check whether $0\in C'(v_j)$, where $j\in\{1,2\}$, we observe that $0\in C'(v_j)$ if and only if $[-\lambda(vv_j),+\lambda(vv_j)]$ contains some element of $B(v_j)$. This latter question is answered making the query $\mathsf{IntHitBy}([-\lambda(vv_j),+\lambda(vv_j)])$ in the representation $\mathbb{I}(B(v_j))$. We conclude, that $\chi(v)$ can be computed in $O(\log m_2)$ time without changing any of the representations.

Next, for each $j\in\{1,2\}$, we compute the representation $\mathbb{I}(A'(v_j))$ of $A'(v_j)$ applying the operation $\mathsf{IntShift}(-\lambda(vv_j))$ to $\mathbb{I}(A(v_j))$. Similarly, we can compute the representation $\mathbb{I}(B'(v_j))$ of $B'(v_j)$. This takes $O(\log m_1)+O(\log m_2)=O(\log m_2)$ time. More importantly, with an additional cost of $O(\log m_j)$ time we can use indistinctly the representation of $B(v_j)$ or $B'(v_j)$, whatever is more convenient.

Note that we cannot afford to make copies of the representations $\mathbb{I}(A'(v_2))$ or $\mathbb{I}(B'(v_2))$ because this would take $\Theta(m_2)$ time, which may be too much. On the other hand, we can manipulate and make explicit copies of $\mathbb{I}(A'(v_1))$ and $\mathbb{I}(B'(v_1))$ because it takes $O(m_1)$ time. Define the ***minimal***

---

[2] In the process we destroy the data structures for $\mathbb{I}(A(v_2))$ and $\mathbb{I}(B(v_2))$.

representation of a set $A \subset \mathbb{R}$ to be the maximal intervals (with respect to inclusion) in A. From $\mathbb{I}(A'(v_1))$ we can compute the minimal representation of $A'(v_1)$ in linear time, that is, $O(m_1)$ time. For this we use the operation $\mathsf{IntReport}$ in $\mathbb{I}(A'(v_1))$, which returns the intervals in $\mathbb{I}(A'(v_1))$ sorted by their left endpoints, and sequentially merge adjacent intervals that intersect. Similarly, we can find a minimal representation of $B'(v_1)$, which is a list of the values in $B'(v_1)$. Thus, after $O(m_1)$ time we have the minimal representation of $A'(v_1)$ as a list of (sorted) at intervals $J_1,\dots,J_s$ and $B'(v_1)$ as a sorted list of values $y_1,\dots,y_t$, where $k+t \le m_1$.

Now we distinguish cases depending on the relations between $i(v)$, $i(v_1)$ and $i(v_2)$.

Consider the case when $i(v) = i(v_1) = i(v_2)$. We have two parts.

1. First we compute the representation $\mathbb{I}(B(v))$ of $B(v)$. Because of Lemma 12, we have

$$B(v) = \chi(v) \cup (B'(v_1) \cap A'(v_2)) \cup (B'(v_2) \cap A'(v_1)).$$

   Recall that we have an explicit representation of $B'(v_1)$. For each element y in $B'(v_1)$, we query $\mathbb{I}(A'(v_2))$ using $\mathsf{IntHitBy}([y,y])$ to decide whether $y \in A'(v_2)$. Thus, we can compute an explicit representation of $B'(v_1) \cap A'(v_2)$ in $O(m_1 \log m_2)$ time.

   Recall that we also have an explicit minimal representation $J_1,\dots,J_s$ of $A'(v_1)$. For each interval J in that representation, we query $\mathbb{I}(B(v_2))$ with $\mathsf{IntClip}(J)$ to obtain the representation of $J \cap B'(v_2)$. Since the sets $J_1,\dots,J_s$ are pairwise disjoint, we indeed obtain representations of the sets $J_1 \cap B'(v_2),\dots,J_s \cap B'(v_2)$. We then merge them using $\mathsf{IntJoin}$. Since the intervals $J_1,\dots,J_t$ are pairwise disjoint, the operation $\mathsf{IntJoin}$ can be indeed performed. In total we have used $t \le m_1$ times the operations $\mathsf{IntClip}$ and $\mathsf{IntJoin}$, and thus we spent $O(m_1 \log m_2)$ time in total. Inserting in this representation the values (as zero-length intervals) of $B'(v_1) \cap A'(v_2)$, we finally obtain a representation of $(B'(v_1) \cap A'(v_2)) \cup (B'(v_2) \cap A'(v_1))$. If $\chi(v)$ is nonempty, we also insert the interval $[0,0]$ in the representation. The final result is a representation $\mathbb{I}(B(v))$ of $B(v)$ Note that in this computation we have destroyed the representation of $\mathbb{I}(B'(v_2))$ because of the operations $\mathsf{IntClip}$.

2. Next we compute the representation $\mathbb{I}(A(v))$ of $A(v)$. Because of Lemma 11 we have that $A(v) = \mathbb{R}_{>0} \cap A'(v_1) \cap A'(v_2)$. Recall that we have an explicit minimal representation $J_1,\dots,J_s$ of $A'(v_1)$. For each interval J_i in the minimal representation of $A'(v_1)$, we extract from $\mathbb{I}(A'(v_2))$ a representation of $J_i \cap A'(v_2)$ using $\mathsf{IntClip}(J_i)$. Then we compute a representation of $\bigcup_{i\in[s]} J_i \cap A'(v_2) = A'(v_1) \cap A'(v_2)$ using $s-1$ times the operation $\mathsf{IntJoin}$. In both cases it is important that the intervals $J_1,\dots,J_s$ are pairwise disjoint. This takes $O(s \log m_2) = O(m_1 \log m_2)$ time. To obtain $\mathbb{I}(A(v))$ we apply $\mathsf{IntClip}(\mathbb{R}_{>0})$. (Strictly speaking, in Theorem 14 we were assuming closed intervals, but this is not an important feature and we can maintain arbitrary intervals.) Note that in this computation we have destroyed the representation $\mathbb{I}(A'(v_2))$ of $A'(v_2)$, because of the $\mathsf{IntClip}$ operations, and therefore this step has to be made after the computation of $B(v)$, which is also using the representation $\mathbb{I}(A'(v_2))$, but not changing it.

Consider now the case when $i(v) = i(v_1) \neq i(v_2)$. We proceed as follows.

1. First we compute the representation $\mathbb{I}(B(v))$ of $B(v)$. Because of Lemma 12 we have $B(v) = \chi(v) \cup (B'(v_1) \cap C'(v_2))$. Note that, for each $y \in \mathbb{R}$, we have $y \in C'(v_2)$ if and only if the interval $[y - \lambda(vv_2), y + \lambda(vv_2)]$ contains some element of $B(v_2)$. Recall that we have an explicit description $y_1,\dots,y_t$ of $B'(v_1)$. Therefore, for each element $y \in B'(v_1)$, we use the operation $\mathsf{IntHitBy}([y - \lambda(vv_2), y + \lambda(vv_2)])$ in $\mathbb{I}(B(v_2))$ to detect whether $y \in C'(v_2)$. With this we computed $B'(v_1) \cap C'(v_2)$ explicitly in $O(m_1 \log m_2)$ time and we did not change the representation $\mathbb{I}(B(v_2))$. Finally, we build the data structure for the representation $\mathbb{I}(B(v))$ of $B(v)$ by inserting the intervals $[y,y]$ with $y \in B'(v_1) \cap C'(v_2)$ and, if $\chi(v)$ is nonempty, we also insert $[0,0]$ in the data structure.

2. Next we compute the representation $\mathbb{I}(A(v))$ of $A(v)$. Because of Lemma 11 we have $A(v) = \mathbb{R}_{>0} \cap A'(v_1) \cap C'(v_2)$. Note that we cannot compute $C'(v_2)$ explicitly, since that would take $\Theta(m_2)$ time. Recall that we have an explicit minimal representation $J_1, \dots, J_s$ of $A'(v_1)$. For each interval $J_i = [x_i, y_i]$ in the minimal representation of $A'(v_1)$, we use the operation $\mathsf{IntClip}([x_i - \lambda(vv_2), y_i + \lambda(vv_2)])$ in the representation $\mathbb{I}(B'(v_2))$. Note that the intervals $[x_i - \lambda(vv_2), y_i + \lambda(vv_2)]$ over $J_1, \dots, J_s$ may be intersecting, and therefore for index i we are actually obtaining the representation of

   $$B'(v_2) \cap \left([x_i - \lambda(vv_2), y_i + \lambda(vv_2)] \setminus \bigcup_{j<i} [x_j - \lambda(vv_2), y_j + \lambda(vv_2)] \right).$$

   Nevertheless, using $\mathsf{IntJoin}$ over the representations reported we obtain the representation of the set (of zero-length intervals)

   $$X := B'(v_2) \cap \bigcup_{i \in [s]} [x_i - \lambda(vv_2), y_i + \lambda(vv_2)].$$

   We then have

   $$A'(v_1) \cap C'(v_2) \ = \ \bigcup_{x \in X} [x - \lambda(vv_2), x + \lambda(vv_2)],$$

   which means that we obtain a representation of $A'(v_1) \cap C'(v_2)$ from the representation of X using the operation $\mathsf{IntExtend}(\lambda(vv_2))$. To obtain $\mathbb{I}(A(v))$ we apply $\mathsf{IntClip}(\mathbb{R}_{>0})$. Since we are making $O(m_1)$ operations, we spend $O(m_1 \log m_2)$ time. Note that in this computation we have destroyed the representation of $\mathbb{I}(B'(v_2))$, and thus this step has to be made after the computation of $\mathbb{I}(B(v))$, which is also using $\mathbb{I}(B'(v_2))$ (or the equivalent representation $\mathbb{I}(B(v_2))$).

Consider now the case when $i(v) = i(v_2) \neq i(v_1)$. We proceed as follows.

1. First we compute the representation $\mathbb{I}(B(v))$ of $B(v)$. Because of Lemma 12 we have $B(v) = \chi(v) \cup (B'(v_2) \cap C'(v_1))$. We compute explicitly the minimal representation of $C'(v_1)$. Then, for each interval I in that representation we query for the elements $I \cap B'(v_2)$ using $\mathsf{IntClip}(I)$ in $\mathbb{I}(B'(v_2))$ and join the answers using $\mathsf{IntJoin}$ over all intervals I. This takes $O(m_1 \log m_2)$ time and changes the data structure of the representation $\mathbb{I}(B'(v_2))$. Finally, if $\chi(v)$ is nonempty, we also insert $[0,0]$ in the result. The total time is $O(m_1 \log m_2)$.

2. Next we compute the representation $\mathbb{I}(A(v))$ of $A(v)$. Because of Lemma 11 we have $A(v) = \mathbb{R}_{>0} \cap A'(v_2) \cap C'(v_1)$. Again, we compute explicitly the minimal representation of $C'(v_1)$. For each interval I in the minimal representation of $C'(v_1)$ we use the operation $\mathsf{IntClip}(I)$ in $\mathbb{I}(A'(v_2))$ to obtain a representation of $I \cap A'(v_2)$, and then use $\mathsf{IntJoin}$ to join all the answers. With this we obtain a representation of $A'(v_2) \cap C'(v_1)$, to which we apply $\mathsf{IntClip}(\mathbb{R}_{>0})$. This procedure takes $O(m_1 \log m_2)$ time and changes the representation of $\mathbb{I}(A'(v_2))$.

Consider now the remaining case, when $i(v) \neq i(v_1)$ and $i(v) \neq i(v_2)$. We proceed as follows.

1. The computation of $B(v)$ is trivial, since $B(v) = \chi(v)$ by Lemma 12.

2. The computation of the representation of $A(v) = C'(v_1) \cap C'(v_2)$ is similar to the case when $i(v) = i(v_1) \neq i(v_2)$. We compute explicitly the minimal representation of $C'(v_1)$, and use it as it was done there (for $\mathbb{I}(A'(v_1))$). This takes $O(m_1 \log m_2)$ time.

In each case we spent $O(m_1 \log m_2)$ time, and the time bound follows. For the upper bound on the representation $\mathbb{I}(A(v))$ of $A(v)$, we note that each left endpoint of each interval in $\mathbb{I}(A(v))$ gives rise to at most one interval in the representation of $A(v)$. The four terms correspond to the four possible cases we considered for the indices $i(v)$, $i(v_1)$ and $i(v_2)$. □

Lemma 17. *The problem* Generalized Graphic Inverse Voronoi in Trees *for an input* $(T,\mathbb{U})$ *where* T *is an* n*-vertex tree of maximum degree* 3 *and the candidate Voronoi cells are pairwise disjoint, can be solved in* $O(n\log^2 n)$ *time.*

Proof. We root T at a leaf so that each node has at most two descendants. For each vertex v of T, we compute a representation $\mathbb{I}(A(v))$ and $\mathbb{I}(B(v))$ of the sets $A(v)$ and $B(v)$, respectively. The computation is bottom-up: we compute $\mathbb{I}(A(v))$ and $\mathbb{I}(B(v))$ when this has been computed for all the children of v. If v has two children, we use Lemma 16. If v has one child, then the computation can be done in $O(\log m_A(v) + \log m_B(v))$ time in a straightforward manner. When we arrive to the root r, we just have to check whether $B(r)$ is nonempty.

We can see by induction that, for each vertex v of T, $m_A(v) \le n(v)$. (We already mentioned earlier that $B(v)$ has at most $n(v)$ values, one per vertex of $T(v)$.) This is clear for the leaves because $A(\cdot)$ has only one interval. For the internal nodes v that have one child u it follows because the representation $\mathbb{I}(A(v))$ of $A(v)$ is obtained from the representation of $\mathbb{I}(A(u))$ by a shift. For the internal nodes v with two children v_1 and v_2, the bound on $m_A(v)$ follows by induction from the bound in Lemma 16. In particular, we have $O(\log m_A(v) + \log m_B(v)) = O(\log n)$ at each node v of T.

For each vertex with one child we spend $O(\log n)$ time. For each vertex v with two children v_1 and v_2 we spend $O(\min\{n(v_1), n(v_2)\}\log n)$ time. Thus, if V_1 and V_2 denote the vertices with one and two children, respectively, we spend

$$\begin{aligned}
O(n) + \sum_{v\in V_1} O(\log n) + \sum_{v\in V_2} O(\min\{n(v_1), n(v_2)\}\log n)& \\
= O(n\log n) + O(\log n)\sum_{v\in V_2} O(\min\{n(v_1), n(v_2)\})&
\end{aligned}$$

time. Using Lemma 15, this time is $O(n\log^2 n)$.

Standard (but non-trivial) adaptations can be used to recover an actual solution. One option is to use persistent data structures for the search trees that store families $\mathbb{I}$ of monotonic intervals. A persistent data structure allows to make queries to any version of the tree in the past. Thus, it stores implicitly copies of the trees that existed at any time. Sarnak and Tarjan [4] explain how to make red-black tree persistent (and how the Join and Split operations can also be done). Since we have access to the past versions of the tree, we can recover how the solution was obtained. Each operation in the past takes $O(\log m)$ time, where m is the sum of operations that were performed. In our case this is $O(\log(n\log^2 n)) = O(\log n)$ time per operation/query in the tree, and the running time is not modified. Another, conceptually simpler option is to store through the algorithm information on how to undo each operation. Then, at the end of the algorithm, we can run the whole algorithm backwards and recover the solutions. ☐

Theorem 18. *The problem* Generalized Graphic Inverse Voronoi in Trees *for instances* $I = (T, ((U_1,S_1),\dots,(U_k,S_k)))$ *can be solved in time* $O(N + n\log^2 n)$*, where* T *is a tree with* n *vertices and* $N = |V(T)| + \sum_i(|U_i| + |S_i|)$.

Proof. Because of Theorem 10, we can transform in $O(N)$ time the instance I to another instance $I' = (T', ((U'_1,S'_1),\dots,(U'_k,S'_k)))$, where T' has maximum degree 3, the sets $U'_1,\dots,U'_k$ are pairwise disjoint, and T' has $O(n)$ vertices. We can compute a solution to instance I' in $O(n\log^2 n)$ time using Lemma 17. Then, we have to check whether this solution is actually a solution for I. For this we use Lemma 3. ☐

Corollary 19. *The problem* Graphic Inverse Voronoi in Trees *for instances* $I = (T, (U_1,\dots,U_k))$*, can be solved in time* $O(N + n\log^2 n)$*, where* T *is a tree with* n *vertices and* $N = |V(T)| + \sum_i |U_i|$.

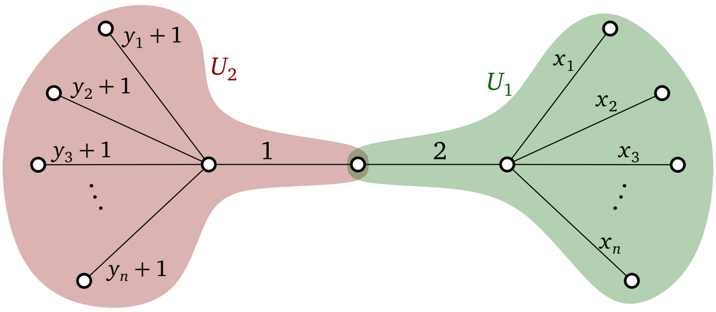


Figure 11: Construction to show the lower bound in Theorem 20.

5 Lower bound for trees

We can show the following lower bound on any algorithm based on algebraic operations on the lengths of the edges.

Theorem 20. *In the algebraic computation tree model, solving* GRAPHIC INVERSE VORONOI IN TREES *with n vertices takes $\Omega(n \log n)$ operations, even when the lengths are integers.*

Proof. Consider an instance X, Y for the decision problem SET INTERSECTION, where $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_n\}$ are sets of integers. The task is to decide whether $X \cap Y$ is nonempty. This problem has a lower bound of $\Omega(n \log n)$ in the algebraic computation tree model [7]. (In particular, this implies the same lower bound for the bounded-degree algebraic decision tree model.) Adding a common value to all the numbers, we may assume that X and Y contain only positive integers.

We construct an instance to the GRAPHIC INVERSE VORONOI IN TREES problem, as follows. See Figure 11. We construct a star S_X with $n+1$ leaves. The edges of S_X have lengths $x_1, \dots, x_n, 2$. We construct also a star S_Y with $n+1$ leaves whose edges have lengths $y_1+1, \dots, y_n+1, 1$. Finally, we identify the leaf of S_X incident to the edge of length 2 and the leaf of S_Y incident to the edge of length 1. Let T be the resulting tree. We take the sets U_1 and U_2 to be the vertex sets of S_X and S_Y, respectively. Note that T has $2n+3$ vertices. The reduction makes $O(n)$ operations.

Since placing the sites on the center of the stars does not produce a solution, it is straightforward to see that the answers to SET INTERSECTION(X, Y) and to GRAPHIC INVERSE VORONOI IN TREES$(T, (U_1, U_2))$ are the same. Thus, solving GRAPHIC INVERSE VORONOI IN TREES$(T, (U_1, U_2))$ in $o(n \log n)$ time would provide a solution to SET INTERSECTION(X, Y) in $o(n \log n)$ time, and contradict the lower bound. ☐

The lower bound also extends to the problem GENERALIZED GRAPHIC INVERSE VORONOI IN TREES with disjoint regions because we can apply the transformation to make the cells disjoint.

6 Conclusions

We have provided an algorithm for the inverse Voronoi problem in trees and a lower bound in a standard computation model. Since the upper bound of our algorithm and the lower bound differ, the main open question is closing this gap. Considering trees with unit edge lengths may also be interesting. Our lower bound does not apply for such instances.

Acknowledgments

We are very grateful to the anonymous reviewers for pointing out an error in the previous version of Section 3.2 and several other useful corrections.

Part of this work was done at the 21st Korean Workshop on Computational Geometry, held in Rogla, Slovenia, in June 2018. We thank all workshop participants for their helpful comments.

References

[1] É. Bonnet, S. Cabello, B. Mohar, and H. Pérez-Rosés. The inverse Voronoi problem in graphs I: hardness, 2018. Manuscript. Its content is included in `http://arxiv.org/abs/1811.12547`.

[2] P. Brass. *Advanced Data Structures*. Cambridge University Press, 2008, `https://doi.org/10.1017/CBO9780511800191`.

[3] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. part I: sequential algorithms. *Algorithmica* 27(3):212–226, 2000, `https://doi.org/10.1007/s004530010016`.

[4] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM* 29(7):669–679, 1986, `https://doi.org/10.1145/6138.6151`.

[5] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM* 32(3):652–686, 1985, `http://doi.acm.org/10.1145/3828.3835`.

[6] R. E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Math. Program.* 77:169–177, 1997, `https://doi.org/10.1007/BF02614369`.

[7] A. C. Yao. Lower bounds for algebraic computation trees with integer inputs. *SIAM J. Comput.* 20(4):655–668, 1991, `https://doi.org/10.1137/0220041`.