DOI 10.15622/ia.25.2.9

V. ZINOV

HOLES PROCESSING IN VORONOI DIAGRAM WITH RECTANGULAR SITES FOR THE COMPARATIVE ANALYSIS OF SLAB DEFORMATIONS

Zinov V. **Holes Processing in Voronoi Diagram with Rectangular Sites for the Comparative Analysis of Slab Deformations.**

Abstract. In this article, a new Voronoi diagram algorithm with rectangular sites and holes is proposed. The algorithm is based on the existing Voronoi diagram algorithm with the $L\infty$ distance metric by Papadopoulou E. and Lee D.-T. The new modifications of the Voronoi diagram model include the holes processing mechanisms. The algorithm handles the distortions in the diagram structure from the holes by using layers in the Voronoi front called shadows and a new type of bisectors that do not build any Voronoi edge in the diagram, but maintain the layers of the front. The algorithm defines new events for a sweep line, keeping the general processing in the same manner as the base algorithm. According to the results of time consumption comparison with the previous span determination algorithm, the proposed algorithm executes from 1.33 times faster for 75 supports up to 15.17 times faster for the largest number of supports tested, but is slower for fewer supports and more holes. The preliminary correlation analysis showed a significant correlation of 0.76 between the area of the Voronoi cell and the amount of reinforcement required, as well as strong and moderate correlation between other parameters of the cell and deformation metrics. The conclusion outlines the current limitations of the model and algorithm for future research.

Keywords: Voronoi diagram, deformation comparative analysis, rational support placement, correlation model, building design optimization.

1. Introduction. The support placement problem in the building design sphere is a complex combinatorial problem [1 – 2]. A wide variety of support plans and the occurring deformations in the floor slab above them are the main complexity factors. The most common approach for estimating the deformations in slabs is to use a functional approximation based on building physics dependencies. The best-known method is equivalent frame method [3, 1, 4 – 5], but Euler beam model [2], Young's modulus [6] and gradient-based optimization [7] can also be used for strength estimation. But these approaches process only columns with regular placement. There are some studies on using neural networks to estimate deformations [8], optimize the structures [9], or model the whole building [10].

In contrast, the comparative analysis of slab deformations uses heuristic additive functions that do not give the exact estimation of reinforcement needed. Instead, it builds a computationally simple and practically correct comparative process in order to range a large set of different support plans. This is one of the heuristic weight distribution methods, which partition the slab into small areas [11].

This paper introduces a new algorithm of slab partitioning via Voronoi diagrams. The previous algorithm of comparative deformation analysis [12] partitions the slab into span areas between supports. Then a multiobjective genetic algorithm applies estimation on spans to find Pareto-optimal plans [13]. However, this approach depends on heuristically selected algorithm parameters. In contrast, Voronoi diagrams are already used in the building design problems: to model crack distribution in concrete [14], design facades of a building [15], or rod structures [16].

The article defines different modifications to analyze deformation in the slab via Voronoi diagrams. A Voronoi diagram model with the rectangular sites represents support areas of the slab. New structures in the Voronoi diagram called "shadows" handle the holes inside the partitioning plane via layers in a height-balanced binary tree representing the sweep line wavefront.

The paper has the following sections. Section 2 defines the Voronoi diagram with rectangular sites and holes. Section 3 describes the core algorithm handling rectangular sites. Section 4 defines a new algorithm handling holes. Section 5 analyzes the time consumption of the proposed algorithm and potential usage in practice. Section 6 concludes the paper with the results. Table 1 contains the notation most used in this paper.

Table 1. Most used variables and functions

Symbol	Type	Description
$Slab$	Var.	Rectangular slab given by dimensions and set of holes
$Hole_k$	Var.	Rectangular hole by index $k = 1, \ldots, K$
T_i	Var.	Rectangular site-support by index $i = 1, \ldots, N$
T^u, $Hole^u$, $Slab^u$	Var.	Edge of T_i, $Hole_k$ or $Slab$ {R^s – south, R^n – north, R^w – west, R^e – east}
VC_i, VE_i	Set	Voronoi cells and edges of the support T_i
$\mathcal{L}$	Var.	The current abscissa of the sweep line
Q	Set	The priority queue of the sweep line events
$\mathcal{b}v$	Var.	General notation of bisector – locus of points equidistant to owners
bv	Var.	Bisector by rectangular sites or slab's boundary (real, border, turn)
βv	Var.	Bisector of virtual type created by rectangular holes (has subtypes)
$owners$	Fnc.	Defines the owners of the bisector $\mathcal{b}v$
ep	Fnc.	Defines current position of some bisector $\mathcal{b}v$ at the moment $\mathcal{L}$
Bv	Fnc.	Defines a new real bisector element by one or two edges
$\mathcal{B}v$	Fnc.	Defines a new virtual bisector element with corresponded type
$\mathcal{T}$	Var.	General notation of sweep line status node (AVL tree) by bisector
$\mathcal{T}^{\mathrm{SH}}$	Var.	Node of virtual bisector which maintains shadow (Shadow)
$\mathcal{T}^{\mathrm{DP}}$	Var.	Copy of shadow node inside the shadow (Duplicate)
$\mathcal{T}^{\mathrm{BL}}$	Var.	Node of real bisector at hole's west edge, maintains shadow (Blind)
$bisector$	Fnc.	Defines a bisector of sweep line status node $\mathcal{T}$
$basis$	Fnc.	Defines a root node of shadow from shadow node $\mathcal{T}^{\mathrm{SH}}$
end	Fnc.	Defines a paired shadow node of different shadow node $\mathcal{T}^{\mathrm{SH}}$
$dupl$	Fnc.	Defines a duplicate node in the shadow of shadow node $\mathcal{T}^{\mathrm{SH}}$
$main$	Fnc.	Defines a shadow node out of the shadow of duplicate node $\mathcal{T}^{\mathrm{DP}}$

2. Problem Formulation: Voronoi Diagram with Rectangular Sites and Holes. Given a rectangular slab $Slab = <L, W, Holes>$, where ($L \in \mathbb{N}$, $W \in \mathbb{N}$) are the length and width, $Holes = \{Hole_1, \ldots, Hole_K\}$ is the set of holes, $|Holes| = K$. A hole is a rectangle $Hole_k = <(x_k, y_k), (l_k, w_k)>$, where ($x_k \in \mathbb{Z}$, $y_k \in \mathbb{Z}$) is the left lower vertex, ($l_k \in \mathbb{N}$, $w_k \in \mathbb{N}$) are the dimensions. Point $p = (x \in \mathbb{Z}, y \in \mathbb{Z}) \in Slab$ if $0 \le x \le L$, $0 \le y \le W$, $\forall Hole_k \in Holes: p \notin Hole_k$.

Given a set of supports $T = \{T_1, \ldots, T_N\}$, $|T| = N$. A support is a rectangle $T_i = <(x_i, y_i), (l_i, w_i)>$, where ($x_i \in \mathbb{Z}$, $y_i \in \mathbb{Z}$) is the left lower vertex and ($l_i \in \mathbb{N}$, $w_i \in \mathbb{N}$) are the dimensions. A union of rectangles is $R' = Holes \cup T \cup \{Slab\}$, where $R \in R' = <(x_R, y_R), (l_R, w_R)>$ is a rectangle. The edges of any R are denoted $R^u \in \{R^s, R^n, R^w, R^e\}$, where $R^s = ((x_R, y_R), (x_R + l_R, y_R))$ is the south edge, $R^n = ((x_R, y_R + w_R), (x_R + l_R, y_R + w_R))$ is the north edge, $R^w = ((x_R, y_R), (x_R, y_R + w_R))$ is the west edge, $R^e = ((x_R + l_R, y_R), (x_R + l_R, y_R + w_R))$ is the east edge.

Integer constraints of position and dimension reflect the arrangement requirement for building structures, yet the coordinate system remains continuous. Current research sets the following geometric assumptions:

1. No intersection or overlapping between holes: $\forall Hole_1, Hole_2 \in Holes: Hole_1 \cap Hole_2 = \varnothing, Hole_1 \ne Hole_2$;
2. No intersection or overlapping between holes and supports: $\forall Hole_1 \in Holes, \forall T_2 \in T: Hole_1 \cap T_2 = \varnothing$;
3. Supports cannot overlap each other but can have an adjacent edge: $\forall T_1, T_2 \in T: T_1 \cap T_2 = \varnothing$ or $T_1 \cap T_2 = T^u: T^u = T^{u1} \in T_1 = T^{u2} \in T_2, T_1 \ne T_2$;
4. Slab restriction: $\forall R \in R': 0 \le x_R < x_R + l_R \le L, 0 \le y_R < y_R + w_R \le W$.

Let us denote the distance metric as $d(p_1, p_2) \in \mathbb{R}$, p_1 and $p_2 \in$ Slab. The distance between support and some point on the slab is calculated as:

$$d(p, T_i) = min\{d(p, p'), \forall p' \in T_i\}. \quad (1)$$

The distance between edge R^u and point p is defined as in (1):

$$d(p, R^u) = min\{d(p, p'), \forall p' \in R^u\}.$$

The distance between two edges R^{u1} and R^{u2} can be computed via corner-to-edge sampling and is determined by:

$$d(R^{u1}, R^{u2}) = min\{d(p_1, p_2), \forall p_1 \in R^{u1}, \forall p_2 \in R^{u2}\}.$$

Definition 1. The requirements for distance metric with holes: if $p_1, p_2 \in Slab$ and $\exists Hole_k \in Holes$: $(p_1, p_2) \cap Hole_k \neq \varnothing$, then d($p_1$, p_2) is defined as:

$$d(p_1, p_2) = min(d(p_1, v_{k1}) + d(v_{k1}, v_{k2}) + d(p_2, v_{k2})), \quad (2)$$

where v_{k1} and v_{k2} are vertices of $Hole_k$ from set $\{(x_k, y_k), (x_k + l_k, y_k), (x_k + l_k, y_k + w_k), (x_k, y_k + w_k)\}$ which satisfy following conditions:

- $(p_1, v_{k1}) \cap Hole_k = v_{k1}$, $(p_2, v_{k2}) \cap Hole_k = v_{k2}$;
- $v_{k1} = v_{k2}$ or $(v_{k1}, v_{k2}) = Hole_k^u$ – edge of $Hole_k$.

It is required to find a Voronoi diagram of *Slab* by *T*, defined as $VD(T) = \{VC_i \mid i = 1, \ldots, N\}$, where VC_i is a Voronoi cell of the support T_i:

$$VC_i = \{p \in Slab: d(p, T_i) \leq d(p, T_j), \forall T_j \in T, i \neq j\},$$

with distance metric requirement by equation (2). Voronoi edges of site *i* is:

$$VE_i = \{ve = (p_1, p_2) | \exists T_j \in T: d(p, T_i) = d(p, T_j), i \neq j, \forall p \in ve\}.$$

The next sections discuss algorithmic approaches applicable to this problem formulation and present a new algorithm based on it.

3. The Core Algorithm: Voronoi Diagram with Rectangular Sites. The most well-known algorithm for the Voronoi diagram is Fortune's algorithm [3]. Site points determine events of changes in the diagram structure. But the rational support placement problem requires rectangular sites and the mechanisms for processing holes. The current section explores existing algorithms for Voronoi diagrams with rectangular sites.

3.1. Existing algorithms with rectangular sites. One application area of Voronoi diagrams with polygonal sites is the path-finding problem [18 – 19]. The goal is to find a shorter path between polygonal obstacles. The polygonal obstacles define a set of point-sites on their edges with a certain step. Then Voronoi cells of sites with the same edges are united into one cell of the obstacle. On the one hand, the approach allows a high-detailed Voronoi structure. But because it is only the preliminary step of the optimization path finding algorithm and is executed only once, its main disadvantage is the time consumption, which increases as the decomposition step decreases and, consequently, as the diagram accuracy improves.

The previous research [13] has proposed a multiobjective genetic algorithm (MOGA) to find a Pareto-optimal set of rational support plans. The experiment of comparison with other MOGA set the iteration count to 375 and the number of plans to 350. Thus, it is critically important to have

as little time per iteration to build a diagram for a support plan as possible, even at the expense of diagram accuracy.

The different approach is implemented by E. Papadopoulou and D.T. Lee (EP&DTL) [20]. The algorithm handles the rectangular sites without additional division into points and uses the L_∞ distance metric. The sweep line algorithm processes events using each vertical edge of rectangles that allows constructing the Voronoi diagram with a larger number of rectangles and simplifies the adaptation of the algorithm for new events and structures. Thus, the present paper uses the EP&DTL algorithm as the core algorithm.

The EP&DTL algorithm finds the shortest critical areas on integrated circuits, such that which the plane of the diagram is connected everywhere. But the floor slabs can contain openings for different systems. Therefore, to correctly reflect the construction process, it is necessary to include a mechanism for processing rectangular holes in the Voronoi diagram.

3.2. Distance and bisector definitions. The EP&DTL algorithm uses the L_∞ distance instead of Euclidean, which simplifies Voronoi diagram calculation. Further, all the rules and conditions set in Section 2 use the L_∞ distance metric defined below:

$$d(p_1, p_2) = max(|x_1 - x_2|, |y_1 - y_2|), \forall p_1 = (x_1, y_1), p_2 = (x_2, y_2) \in Slab.$$

One of the main structures used in the EP&DTL algorithm is a bisector $\mathscr{b}v$. There are four general definitions that will be used further in the text. First of all, the main definition of the bisector as the locus of points at equal distance from two different edges:

$$\mathscr{b}v = Bv(R^{u1}, R^{u2}) = \{p: d(p, R^{u1}) = d(p, R^{u2})\}. \quad (3)$$

Secondly, the bisectors that represent edges of the elements:

$$\mathscr{b}v = Bv(R^u) = \{p: d(p, R^u) = 0\}. \quad (4)$$

Given a position of the vertical sweep line $\mathcal{L}$, the current position of bisectors $ep(\mathscr{b}v)$ (from equations (3), (4)) is equidistant from the sweep line:

$$ep(\mathscr{b}v) = p \in \mathscr{b}v: d(p, R^u) = \mathcal{L} - x_p, \forall R^u \in owners(\mathscr{b}v), \quad (5)$$

where $owners(\mathscr{b}v)$ are the edges given in the creation function of the bisector $\mathscr{b}v$.

The third type is a bisector that represents a single point:

$$\mathcal{bv} = Bv(R^{u1}, R^{u2}) = \{p: d(p, R^{u1}) = d(p, R^{u2}) = 0\}. \quad (6)$$

The current position of such bisector (6) is always the same point:

$$ep(\mathcal{bv}) = p \in \mathcal{bv}. \quad (7)$$

The last type is a bisector that depends on the edge of one element R^{u1} and lies on the edge of another element R^{u2}:

$$\mathcal{bv} = Bv(R^{u1}, R^{u2}) = \{p: d(p, R^{u2}) = 0\}. \quad (8)$$

The current position of this bisector from equation (8) is equidistant from the first owner and the sweep line position:

$$ep(\mathcal{bv}) = p \in \mathcal{bv}: d(p, R^{u1}) = \mathcal{L} - x_p, R^{u1} \in owners(\mathcal{bv}). \quad (9)$$

The creation function of the bisector types described further is set as one of the equations above or is defined additionally.

Conditions given in the bisector creation define the locus of points depending on the owners. Condition given by the current position defines a point depending on the owners and the sweep line position. The EP&DTL algorithm uses definitions of bisectors from (3) and (4) and the current position from (5), while others are used in the further modifications.

3.3. EP&DTL algorithm: the core algorithm definitions. Let us call bisectors induced by supports as real bisectors with denotation *bv*. The EP&DTL algorithm uses real bisectors equidistant from two support edges T_i^{u1} and T_i^{u2}, which can be defined by equation (3) as $bv = Bv(T_i^{u1}, T_i^{u2})$, and a support horizontal edge $T_i^u \in \{T_i^s, T_i^n\}$, which can be defined as a bisector by equation (4) as $bv = Bv(T_i^u)$.

A sweep line status $\mathcal{T}$ is a node of height-balanced binary tree consisting of a bisector:

$$\mathcal{T} = \langle bisector, left, right \rangle, \quad (10)$$

where *bisector*($\mathcal{T}$) is the bisector of $\mathcal{T}$; *left*($\mathcal{T}$), *right*($\mathcal{T}$) are the left and right nodes of $\mathcal{T}$.

The wavefront maintained by the tree contains segments between two consecutive nodes. If $bv = bisector(\mathcal{T})$, then the current position of the node $\mathcal{T}$ is denoted as $ep(\mathcal{T}) = ep(bv)$. The key of the node $\mathcal{T}$ in the tree equals the ordinate of its current position $key(\mathcal{T}) = y_{ep(bv)}$. Then *next*($\mathcal{T}$) and *prev*($\mathcal{T}$) are

the next and previous nodes of node $\mathcal{T}$ in the tree determined by *key*($\mathcal{T}$). This article implements a standard AVL tree for the sweep line status $\mathcal{T}$.

If owners are parallel in terms of coordinate, then the current position should be a point with the smaller maximum difference by another coordinate (Figure 1) that sets the unique current position. But there is an issue in the algorithm with west edge alignment; we discuss it later.

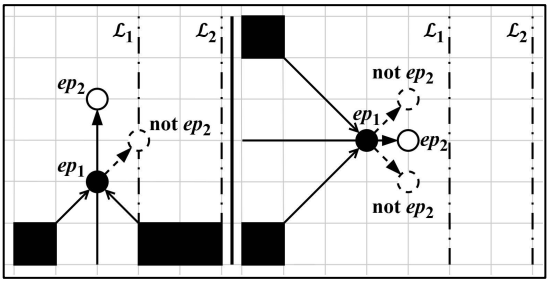

Fig. 1. Inexact current position with horizontal (left) and vertical (right) alignment

Finally, $\mathcal{Q}$ is a priority queue of diagram events. The EP&DTL algorithm defines three types of events: West Edge, East Edge and Spike. The type of some event $q \in \mathcal{Q}$ can be obtained by the function *type*(q):

$$type(q) = \{\text{West Edge, East Edge, Spike}\}. \tag{11}$$

West Edge and East Edge events occur as the sweep line reaches the edge of the site. Their priority equals the abscissa of the edge. West Edge event fixates part of the wavefront between intersection points, using bisectors induced from edge vertices, and creates two bisectors at these points and two bisectors at horizontal edges. East Edge event removes bisectors at horizontal edges and inserts two bisectors from edge vertices.

Spike event q_{Sp} occurs when two bisectors bv_1 and bv_2 intersect at a point $p_{\text{Sp}} = bv_1 \cap bv_2$. The priority of the Spike event q_{Sp} in the original paper [20] is the position of the sweep line $\mathcal{L}$ that is equidistant from p_{Sp} with abscissa x_{Sp} and the common owner of the bisectors $T_{\text{Sp}} = owners(bv_1) \cap owners(bv_2)$:

$$Priority(q_{\text{Sp}}) = x_{\text{Sp}} + d(p_{\text{Sp}}, T_{\text{Sp}}). \tag{12}$$

The Spike event removes bv_1 and bv_2 and creates a new bisector at p_{Sp}. Both bisectors must still be in $\mathcal{T}$ or the event should be ignored, which can be done by keeping a flag of removal state for each node $\mathcal{T}$.

The uniqueness of the current position from Figure 1 occurs differently for west and east vertical alignment. East alignment can be processed in the Spike event of east bisectors, with a new horizontal bisector. For the case of west edges, the condition is more complicated, especially with distortions by holes. One of the ways to handle the problem is to build the Voronoi diagram twice, flipping the abscissa of the slab elements, collecting edges from both diagrams and connecting disjoint Voronoi vertices. Section 5.2 considers this method and its limitations, while an applicable approach is a subject of future research. Figure 2 illustrates the problem of Voronoi edges' asymmetry.

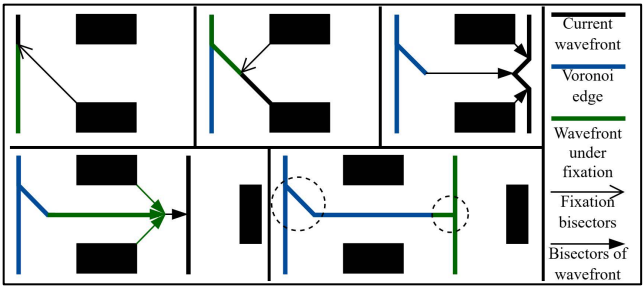

Fig. 2. Asymmetry in Voronoi diagram by edges alignment (shown by circles)

3.4. Slab boundary condition. Processing of slab boundaries during the sweep line algorithm in the L_∞ distance metric needs some modifications.

Definition 2. A bisector representing the slab horizontal edge $Slab^u \in \{Slab^s, Slab^n\}$ is called a border bisector and is defined by equation (4) as $bv_{\text{Brd}} = Bv(Slab^u)$. By equation (5) its current position follows the sweep line.

Definition 3. A bisector representing the slab west vertex of a horizontal edge $Slab^u \in \{Slab^s, Slab^n\}$ is called a turn bisector and is defined by equation (6) as $bv_{\text{Trn}} = Bv(Slab^w, Slab^u)$.

The real bisector bv induced by support edge T_i^{u1} and lying on the slab edge $Slab^{u2}$ is defined by equation (8) as $bv = Bv(T_i^{u1}, Slab^{u2})$. Then its current position is computed by equation (9) depending on T_i^{u1}.

Initially the sweep line status contains a pair of border bisectors for slab horizontal edges and a pair of turn bisectors at slab west vertices. Border bisectors always lie in the extreme positions in the AVL tree $\mathcal{T}$.

Let us denote a Turn event of intersection between a real bv and a turn bv_{Trn} as q_{Trn}, where x_{Trn} is the abscissa of the turn bisector position $ep(bv_{\text{Trn}})$. Then the priority of the Turn event is calculated as:

$$Priority(q_{\text{Trn}}) = x_{\text{Trn}} + d(ep(bv_{\text{Trn}}), owners(bv)).$$

The Turn event shown in Figure 3 removes *bv* and bv_{Trn} and inserts a real bisector on the slab horizontal edge $bv_{new} = Bv(T^{u1}, Slab^{u2})$, where $T^{u1} = owners(bv) \setminus owners(bv_{\text{Trn}})$, $Slab^{u2} = owners(bv_{\text{Trn}}) \setminus owners(bv)$.

Border and turn bisectors prevent violation of slab boundary. The accepted EP&DTL algorithm and proposed boundary process allow using Voronoi diagrams for simple building plans. The next section introduces holes processing in the Voronoi diagram to expand the scope of application.

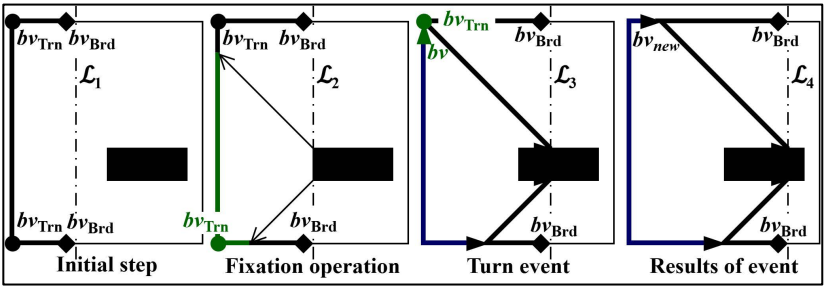

Fig. 3. Example of border and turn bisectors, Turn event (fixation shown in green)

4. Holes processing in Voronoi Diagram. Unlike supports, holes do not fixate wavefront segments but "shadow" them. Bisectors of a hole do not define edges in the Voronoi diagram, but instead determine shadow layers of other bisectors. Therefore, this section introduces new types of bisectors, a new structure of the wavefront with "shadows" and new types of events.

Let us define a real bisector *bv* with the owner T_i^{u1} on the hole edge $Hole^{u2}$ by equation (8) as $bv = Bv(T_i^{u1}, Hole^{u2})$.

Besides, let us define an additional real bisector *bv* with support edge owner T_i^u that is directed from the vertex v_k of $Hole_k$ by a set of conditions:

$$\begin{aligned} bv = Bv(T_i^u, v_k) = \{p: d(p, T_i^u) = d(v_k, T_i^u) + d(p, v_k); \\ \text{if } x_v = x_k \text{ then } x_p \neq x_v, y_p \neq y_v; \\ \text{else } sign(y_v - y_p) = sign(y_v - y_k - \frac{l_k}{2})\Big\}. \end{aligned} \tag{13}$$

The current position of the additional real bisector is calculated by equation (5) with T_i^u being the only owner. Such additional bisectors occur in some cases when a real bisector lying on the hole edge reaches its vertex, maintaining distractions from holes.

4.1. Virtual Bisectors in the Voronoi Wavefront. Let us call bisectors induced by holes virtual bisectors βv. Virtual bisectors determine the tree layer of other bisectors. The first virtual bisector owner is always a hole. There are four types of virtual bisectors with different behaviors.

Definition 4. A virtual bisector, which represents the hole horizontal edge $Hole_k^u \in \{Hole_k^s, Hole_k^n\}$, is called an edge virtual bisector and is defined by equation (4) as $\beta v_{\text{Edge}} = \mathcal{B}v_{\text{Edge}}(Hole_k^u) = Bv(Hole_k^u)$, where $\mathcal{B}v_{\text{Edge}}$ denotes the edge virtual bisector creation function.

Definition 5. A virtual bisector at hole vertex with horizontal edge $Hole_k^{u1} \in \{Hole_k^s, Hole_k^n\}$ and vertical edge $Hole_k^{u2} \in \{Hole_k^w, Hole_k^e\}$ is called a corner virtual bisector and is defined by equation (6) as $\beta v_{\text{Corner}} = \mathcal{B}v_{\text{Corner}}(Hole_k^{u1}, Hole_k^{u2}) = Bv(Hole_k^{u1}, Hole_k^{u2})$, where $\mathcal{B}v_{\text{Corner}}$ denotes the corner virtual bisector creation function.

Definition 6. A virtual bisector induced with hole adjacent edges starting from point $p_{\text{In}} \in Slab$ is called an inward virtual bisector:

$$\begin{gathered}\mathcal{B}v_{\text{In}}(Hole_k^{u1}, Hole^{u2}, R^{u3}, p_{\text{In}}) = \\ \{p: d(p, Hole_k^{u1}) = d(p, Hole^{u2})\},\end{gathered} \tag{14}$$

where $Hole_k^{u1} \in \{Hole_k^w, Hole_k^e\}$ is the hole vertical edge; if $Hole_k^{u1}$ is a west edge then $Hole^{u2} \in \{Hole_k^s, Hole_k^n\}$ is the hole horizontal edge; else $Hole^{u2} \in \{Hole^{+s}, Hole^{+n}\}$ is an extension of the horizontal edge from the vertex: $Hole^{+s} = ((x_k + l_k, y_k), (x_k + 2 \cdot l_k, y_k))$, $Hole^{+n} = ((x_k, y_k + w_k), ((x_k, y_k + 2 \cdot w_k))$; $R^{u3} \in R' \cup \{\varnothing\}$ is the owner of the wavefront segment in p_{In} if it exists. The owner R^{u3} affects only the definition of the wavefront segment owner, while p_{In} affects only the current position.

Inward virtual bisectors can be passive βv_{InP} or active βv_{InA} depending on the wavefront owner R^{u3} in p_{In}. For a passive inward virtual bisector βv_{InP} the third owner R^{u3} can be a slab, a hole edge or none. The current position of a passive inward virtual bisector βv_{InP} always equals p_{In}:

$$ep(\beta v_{\text{InA}}) = p_{\text{In}}. \tag{15}$$

Active inward virtual bisector βv_{InA} moves to the vertex of the hole and its third owner R^{u3} can be a support edge owner or none. The current position of an active inward virtual bisector βv_{InA} is obtained by a complex equality condition with an artificial sweep line $\mathcal{L}_{\text{Art}}$:

$$\begin{gathered}ep(\beta v_{\text{InA}}) = p \in \beta v_{\text{InA}}: \mathrm{d}(p, Hole^u) = \\ \mathcal{L}_{\text{Art}} - x_p, \forall Hole^u \in owners(\beta v_{\text{InA}}),\end{gathered} \tag{16}$$

where $\mathcal{L}_{\text{Art}} = \mathcal{L} - \mathcal{L}_{\text{In}} + 2 \cdot x_{\text{In}} - x_v$; $\mathcal{L}_{\text{In}}$ is the position of $\mathcal{L}$ at the moment of βv_{InA} creation; x_{In} is the abscissa of p_{In}; $x_v = Hole_k^{u1} \cap Hole^{u2}$ is the abscissa of the vertex.

Let us denote the creating function for a passive inward bisector as $\mathcal{B}v_{\text{InP}}$ and for an active one as $\mathcal{B}v_{\text{InA}}$; both are defined by equation (14), but create different types of inward virtual bisectors.

Definition 7. A virtual bisector lying parallel to the hole horizontal edge $Hole_k^{u1} \in \{Hole_k^s, Hole_k^n\}$ is called an outside virtual bisector. The behavior of an outside virtual bisector depends on its position: bounded by another hole or slab edge, or free. In the first case the outside virtual bisector is called limited and is defined by equation (8) as $\beta v_{\text{OutL}} = \mathcal{B}v_{\text{OutL}}(Hole_k^{u1}, R^{u2}) = Bv(Hole_k^{u1}, R^{u2})$, where $\mathcal{B}v_{\text{OutL}}$ is the denotation of the outside limited virtual bisector creation function, and R^{u2} is the edge of another hole or slab.

In the second case, when the position is free, the outside virtual bisector is called wide and is defined by equation (3) as $\beta v_{\text{OutW}} = \mathcal{B}v_{\text{OutW}}(Hole_k^{u1}, R^{u2}) = Bv(Hole_k^{u1}, R^{u2})$, where $\mathcal{B}v_{\text{OutW}}$ is the denotation of the outside wide virtual bisector creation function, R^{u2} is the edge that owns the wavefront segment in the beginning point p_{Out}, a support or hole, or an artificial rotated copy of $Hole_k^{u1}$ denoted as $Hole*$ if there are two owners of the wavefront segment:

$$\begin{gathered} Hole*(Hole_k^{u1}, p_{\text{Out}}) = ((x*, y*), (x*, y* + s \cdot l_k)), \\ x* = 2 \cdot x_{\text{Out}} - x_k; y* = 2 \cdot y_{\text{Out}} - y_k; s = sgn(y_{\text{Out}} - y_k). \end{gathered} \tag{17}$$

where $(x_{\text{Out}}, y_{\text{Out}}) = p_{\text{Out}}$; (x_k, y_k) are the coordinates of $Hole_k$, l_k is the abscissa dimension of $Hole_k$, that contains edge $Hole_k^{u1}$: $Hole_k^{u1} \in Hole_k$.

Table 2 summarizes the bisectors notation. Section 4.3 sets selection rules between specific creation functions. Once a virtual bisector is created it does not change its type. New inward or outside virtual bisector can be recreated with a new subtype and position.

Table 2. Notations for bisector creation and current position functions

Creation	Description	Position	Eq.
Real bisectors – induced by supports, set edges of the Voronoi diagram during fixation			
$Bv(T^{u1}, T^{u2})$	Bisector between edges of different supports	(5)	(3)
$Bv(T_i^{u1}, T_i^{u2})$	Bisector between edges of one support		
$Bv(T_i^u)$	Bisector along support horizontal edge		(4)
$Bv(T_i^u, p)$	Additional bisector from hole vertex		(13)
$Bv(T_i^{u1}, Slab^{u2})$	Bisector along slab edge	(9)	(8)
$Bv(T_i^{u1}, Hole^{u2})$	Bisector along hole edge		
Border and Turn bisectors – maintain slab boundary during intersection and fixation			
$Bv(Slab^u)$	Border bisector along slab horizontal edge	(5)	(4)
$Bv(Slab^w, Slab^{u2})$	Turn bisector at the slab west vertex	(7)	(6)

Creation	Description	Position	Eq.
Virtual bisectors – induced by holes, maintain shadows from the holes during intersection and fixation			
$\mathcal{B}v_{\text{Edge}}(Hole_k^{u})$	Edge bisector along hole horizontal edge	(5)	(4)
$\mathcal{B}v_{\text{Corner}}(Hole_k^{u1}, Hole_k^{u2})$	Corner bisector at the hole vertex	(7)	(6)
$\mathcal{B}v_{\text{InP}}(Hole_k^{u1}, Hole^{u2}, Hole^{u3}, p_{\text{In}})$	Passive inward bisector on some hole edge	(15)	(14)
$\mathcal{B}v_{\text{InP}}(Hole_k^{u1}, Hole^{u2}, Slab^{u3}, p_{\text{In}})$	Passive inward bisector on some slab edge		
$\mathcal{B}v_{\text{InA}}(Hole_k^{u1}, Hole^{u2}, T^{u3}, p_{\text{In}})$	Active inward bisector of support segment owner	(16)	
$\mathcal{B}v_{\text{In}}(Hole_k^{u1}, Hole^{u2}, \varnothing, p_{\text{In}})$	Passive or active inward bisector without segment owner depending on sweep line moment	(15)	
		(16)	
$\mathcal{B}v_{\text{OutL}}(Hole_k^{u1}, Hole^{u2})$	Outside limited bisector along hole edge	(9)	(8)
$\mathcal{B}v_{\text{OutL}}(Hole_k^{u1}, Slab^{u2})$	Outside limited bisector along slab edge		
$\mathcal{B}v_{\text{OutW}}(Hole_k^{u1}, T^{u2})$	Outside wide bisector with support segment owner	(5)	(3)
$\mathcal{B}v_{\text{OutW}}(Hole_k^{u1}, Hole^{u2})$	Outside wide bisector with hole segment owner		
$\mathcal{B}v_{\text{OutW}}(Hole_k^{u1}, Hole^{*})$	Outside wide bisector with two segment owners		

4.2. Shadows in the Voronoi Wavefront.

Definition 8. A shadow of the height-balanced binary tree is an inner equivalent tree between two nodes from the original tree called shadow nodes. Shadow node with virtual bisector βv is defined by:

$$\mathcal{T}^{\text{SH}} = \langle bisector, left, right, basis, end, dupl \rangle,$$

where $bisector(\mathcal{T}^{\text{SH}}) = \beta v$ is the virtual bisector of $\mathcal{T}^{\text{SH}}$; $left(\mathcal{T}^{\text{SH}})$, $right(\mathcal{T}^{\text{SH}})$ are the left and right nodes of $\mathcal{T}^{\text{SH}}$; $basis(\mathcal{T}^{\text{SH}})$ is the root node of the shadow; $end(\mathcal{T}^{\text{SH}})$ is the pair node on the opposite side of the shadow; $dupl(\mathcal{T}^{\text{SH}})$ is a node copy of $\mathcal{T}^{\text{SH}}$ inside the shadow which is called a duplicate node.

Definition 9. A duplicate node of some shadow node $\mathcal{T}^{\text{SH}}$ is:

$$\mathcal{T}^{\text{DP}} = \langle bisector, left, right, main \rangle,$$

where $main(\mathcal{T}^{\text{DP}}) = \mathcal{T}^{\text{SH}}$ is the shadow node of $\mathcal{T}^{\text{DP}}$; $bisector(\mathcal{T}^{\text{DP}}) = bisector(\mathcal{T}^{\text{SH}})$ is the virtual bisector of $\mathcal{T}^{\text{DP}}$, the same as $\mathcal{T}^{\text{SH}}$; $left(\mathcal{T}^{\text{DP}})$, $right(\mathcal{T}^{\text{DP}})$ are the left and right nodes of $\mathcal{T}^{\text{DP}}$, one of which is always $\varnothing$ because the duplicate node is the last node in the shadow.

Duplicate nodes ensure the exit of a bisector from the shadow through the intersection events.

The definitions of shadow and duplicate nodes extend the definition of real nodes from equation (10). Figure 4 shows an example of an AVL tree with a shadow layer in it.

Let us call nodes neither Shadow nor Duplicate Flat nodes $\mathcal{T}^{\text{F}}$. Shadow structure should provide a unique intersection between west-directed bisectors and the wavefront segments. Multiple intersection points

appear, because a hole should let the intersection be found on any of its edges or behind it. Therefore, a triple form of the hole shadow is proposed: upper and lower outer shadows and an inner shadow covered by them both, as shown in Figure 5. Inner shadow tracks the wavefront reaching the west edge of the hole. Outer shadows preserve the L_∞ structure of the wavefront and maintain a unique intersection.

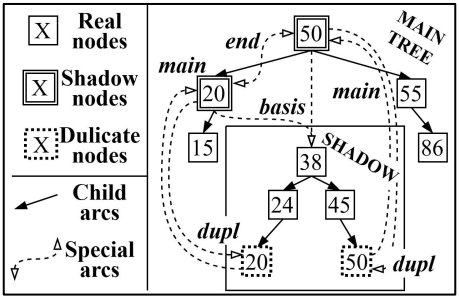

Fig. 4. An example of AVL tree with shadow

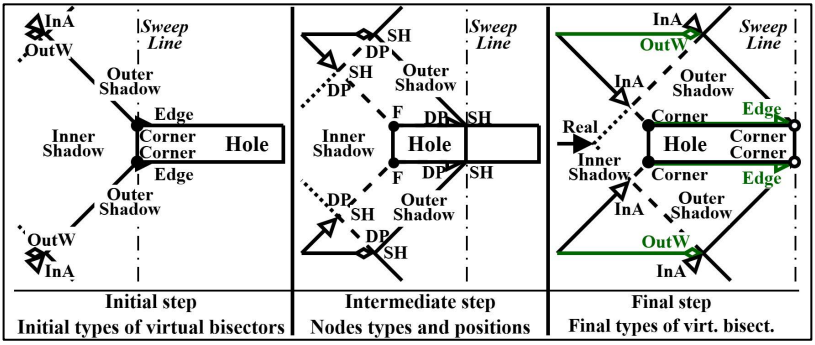

Fig. 5. Illustration of shadow structure; old shadow nodes are shown in green

Inward virtual bisectors enclose the inner shadow and, if active, move to the west vertices corner virtual bisectors. Outer shadow is active until the sweep line reaches the hole east edge, then it becomes inactive. In the active state, the outer shadow is enclosed between an outer virtual bisector and an edge virtual bisector. Inactivation replaces the outer virtual bisector with an inward virtual bisector directed to the hole east vertex where the corner virtual bisector lies.

The sweep line algorithm adds the inner shadow and the outer shadows when the sweep line reaches the west edge of a hole. To add a shadow, the algorithm should find the beginning positions of shadow nodes

and insert them, keeping nodes of the wavefront between them in the shadow using the basis node with duplicate nodes appended there. Shadows can contain each other entirely if both shadow nodes of another shadow are inside it, or partly if only one shadow node is covered.

Shadow nodes can be recreated separately retaining the same *end* and *basis*. The outer shadow is removed entirely when some real bisector reaches the east vertex of a hole. Inner shadow behaves differently: its shadow nodes can be removed, yet the shadow should still exist, which can be handled by a new type of nodes with real bisectors moving along the west edge of a hole.

Definition 10. A node with a real bisector *bv* lying on the west edge of the hole and being the last node in the inner shadow is called Blind Node:

$$\mathcal{T}^{\mathrm{BL}} = \langle bisector, left, right, basis, end \rangle,$$

where $bisector(\mathcal{T}^{\mathrm{BL}}) = bv$ is the real bisector of $\mathcal{T}^{\mathrm{BL}}$; $left(\mathcal{T}^{\mathrm{BL}})$, $right(\mathcal{T}^{\mathrm{BL}})$ are the left and right nodes of $\mathcal{T}^{\mathrm{BL}}$ one of which is always $\varnothing$ because the blind node is the last node of the shadow; $basis(\mathcal{T}^{\mathrm{BL}})$ is the root node of the shadow; $end(\mathcal{T}^{\mathrm{BL}})$ is the pair node on the opposite side of the shadow. Thus, the *end* function can set a paired shadow node outside of the shadow or a blind node inside it.

Figure 6 shows an example of the diagram and AVL tree structure when some inner shadow node is removed and a new blind node appears.

If the algorithm removes a blind node, it creates a new blind node at the same point or recreates the last node in the shadow as a new blind node with the same *end* and *basis*. When two blind nodes intersect, the algorithm removes the inner shadow entirely.

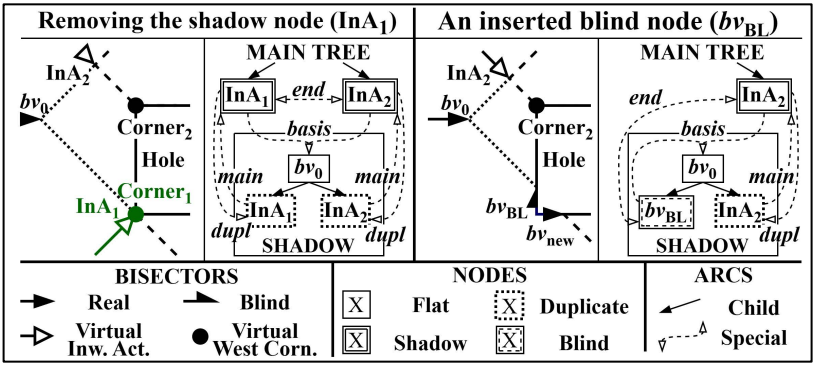

Fig. 6. Illustration of a blind node behavior in the diagram and the tree

4.3. Operations on the Wavefront. Let us determine the type operator of nodes such as $\mathcal{T}$ and any, real or virtual, bisectors such as $\mathscr{b}v$ by the function *type*:

1) Nodes: type($\mathcal{T}$) ∈ {F – flat, SH – shadow, DP – duplicate, BL – blind};

2) Bisectors: type($\mathscr{b}$v) ∈ {R – real, BD – border, TR – turn, VE – virtual edge, VC – virtual corner, VI – virtual inward, VO – virtual outside}.

There are some corollaries of the shadow structure and blind nodes:

- **if** $main(\mathcal{T}^{\mathrm{DP}}) = \mathcal{T}^{\mathrm{SH}}$ **then** $dupl(\mathcal{T}^{\mathrm{SH}}) = \mathcal{T}^{\mathrm{DP}}$,
- **if** $dupl(\mathcal{T}^{\mathrm{SH}}) = \mathcal{T}^{\mathrm{DP}}$ **then** $\mathcal{T}^{\mathrm{DP}} \in basis(\mathcal{T}^{\mathrm{SH}})$,
- **if** $type(\mathcal{T}_1) \in \{\mathrm{SH}, \mathrm{BL}\}$ **and** $type(\mathcal{T}_2) \in \{\mathrm{SH}, \mathrm{BL}\}$:
  - **if** $end(\mathcal{T}_2{}^{\mathrm{SH\ or\ BL}}) = \mathcal{T}_1$ **then** $end(\mathcal{T}_1{}^{\mathrm{SH\ or\ BL}}) = \mathcal{T}_2$,
  - **if** $end(\mathcal{T}_1{}^{\mathrm{SH\ or\ BL}}) = \mathcal{T}_2$ **then** $basis(\mathcal{T}_1{}^{\mathrm{SH\ or\ BL}}) = basis(\mathcal{T}_2{}^{\mathrm{SH\ or\ BL}})$,
  - **if** $end(\mathcal{T}_1{}^{\mathrm{SH\ or\ BL}}) = \mathcal{T}_2$ **and** $key(\mathcal{T}_1) < key(\mathcal{T}_2)$ **then**:

$$\forall \mathcal{T} \in basis(\mathcal{T}_1): key(\mathcal{T}_1{}^{\mathrm{DP\ or\ BL}}) \le key(\mathcal{T}) \le key(\mathcal{T}_2{}^{\mathrm{DP\ or\ BL}}).$$

Shadow and blind nodes should track changes on the basis. In practice, it is useful to have a reference to the basis with handlers of changing events. Besides, to avoid significant time consumption when computing distances with the holes condition (2) in practice, it is useful to keep a distance value of the bisector beginning point using the distance value of a former bisector or by a value itself.

Algorithm 1 defines the intersection of bisector $\mathscr{b}v'$ and node $\mathcal{T}$ with an inclusion flag $f \in \{-1, 1\}$ as *Intersect*($\mathscr{b}v'$, $\mathcal{T}$, f). If nodes lie on bisector $\mathscr{b}v'$, then the function finds the node with the lowest or highest abscissa, depending on $f = 1$ or $f = -1$. If all such nodes are at the same point, then the function returns the leftmost or rightmost node depending on f.

The function *Between* seeks the set of nodes in the tree $\mathcal{T}$ between two points p_{low} and $p_{\mathrm{up}} \in Slab$, such that $y_{\mathrm{up}} > y_{\mathrm{low}}$, as defined:

$$Between(\mathcal{T}, p_{\mathrm{up}}, p_{\mathrm{low}}) = \{\mathcal{T}' \in \mathcal{T} | y_{\mathrm{low}} \le key(\mathcal{T}') \le y_{\mathrm{up}}\}. \quad (18)$$

The operation searches for nodes whose keys are between the given points and goes through the tree using the *left* and *right* functions until conditions are met with O(log n + k), where n is the count of nodes in $\mathcal{T}$ and k is the output size.

Intersect($\mathcal{B}v'$, $\mathcal{T} \neq \varnothing$, $f \in \{-1, 1\}$)
1. **if** *ep*($\mathcal{T}$) lower $\mathcal{B}v'$ **and** *ep*(*next*($\mathcal{T}$)) lower $\mathcal{B}v'$ **then**
2. **return** ***Intersect***($\mathcal{B}v'$, *right*($\mathcal{T}$), *f*) ▷ seek intersection upper
3. **if** *ep*($\mathcal{T}$) upper $\mathcal{B}v'$ **and** *ep*(*next*($\mathcal{T}$)) upper $\mathcal{B}v'$ **then**
4. **return** ***Intersect***($\mathcal{B}v'$, *left*($\mathcal{T}$) , *f*) ▷ seek intersection lower
5. **if** *ep*($\mathcal{T}$) lower $\mathcal{B}v'$ **and** *ep*(*next*($\mathcal{T}$)) upper $\mathcal{B}v'$ **then**
6. **return** $\mathcal{B}v' \cap$ (*ep*($\mathcal{T}$), *ep*(*next*($\mathcal{T}$))) ▷ intersection point found
7. **if** *ep*($\mathcal{T}$) $\in \mathcal{B}v'$ **then** ▷ one or both lie on bisector
8. $\mathcal{T}_{\text{base}} \leftarrow \mathcal{T}$, $\mathcal{T}^{\text{on}} \leftarrow \{\mathcal{T}\}$, $p_{\text{base}} \leftarrow ep(\mathcal{T})$, $\mathcal{T}' \leftarrow prev(\mathcal{T})$
9. **else** $\mathcal{T}_{\text{base}} \leftarrow next(\mathcal{T})$, $\mathcal{T}^{\text{on}} \leftarrow \{next(\mathcal{T})\}$, $p_{\text{base}} \leftarrow ep(next(\mathcal{T}))$, $\mathcal{T}' \leftarrow \mathcal{T}$
10. add to $\mathcal{T}^{\text{on}}$ all nodes from left to right that lie on $\mathcal{B}v'$ by *left* and *right* from $\mathcal{T}_{\text{base}}$
11. $\mathcal{T}_{\text{Last}} \leftarrow \mathcal{T}^{\text{on}}$.Last(), $\mathcal{T}_{\text{First}} \leftarrow \mathcal{T}^{\text{on}}$.First(), $p_{\text{Last}} \leftarrow ep(\mathcal{T}_{\text{Last}})$, $p_{\text{First}} \leftarrow ep(\mathcal{T}_{\text{First}})$
12. **if** ($\exists \mathcal{T}^* \in \mathcal{T}^{\text{on}}$: $ep(\mathcal{T}^*) \neq p_{\text{base}}$ **and** sgn($x_{\text{Last}} - x_{\text{First}}$) = *f*) **or** ($x_{\text{base}} + 1$, $y_{\text{base}} - f$) $\in \mathcal{B}v'$ **then**
13. **return** $\mathcal{T}_{\text{Last}}$ ▷ return the rightest node
14. **else return** $\mathcal{T}_{\text{First}}$ ▷ return the most left node

Algorithm 1. Intersection operation

To create some types of bisectors we need to know an owner of the wavefront segment: real bisectors with two support owners; real bisectors on a hole or slab edge; additional real bisectors; outside virtual bisectors; inward virtual bisectors. The wavefront segment inside $\mathcal{T}$ at point p with a choice flag $f \in \{-1, 1\}$ is *Segment*($\mathcal{T}$, *p*, *f*):

$$Segment(\mathcal{T}, p) = \{(\mathcal{T}_1', \mathcal{T}_2') | key(\mathcal{T}_1') \le y_p \le key(\mathcal{T}_2'); \mathcal{T}_1', \mathcal{T}_2' \in \mathcal{T}, \mathcal{T}_2' = next(\mathcal{T}_1'), f = -1 \textbf{ and } ep(\mathcal{T}_1') \neq p \textbf{ or } f = 1 \textbf{ and } ep(\mathcal{T}_2') \neq p\}. \quad (19)$$

The function *Owner*($\mathcal{T}$, *p*, *f*) gets owner of the wavefront segment:

$$Owner(\mathcal{T}, p, f) = \{R^u | R^u \in owners(\mathcal{T}_1') \cap owners(\mathcal{T}_2'); \mathcal{T}_1', \mathcal{T}_2' \in \mathcal{T}, (\mathcal{T}_1', \mathcal{T}_2') = Segment(\mathcal{T}, p, f)\}. \quad (20)$$

The function *Owner* most of the time gives one owner, except for:

- if p lies between inward virtual bisectors inside the inner shadow, then the segment has two owners: the west edge of the hole and the same third owner of the inward bisectors;
- if p lies between an inward virtual bisector and a west corner virtual bisector of the same hole, then the segment has two owners: the west and horizontal edges of the hole;

- if p lies between a passive inward bisector without a third owner and a west corner virtual bisector of a different hole, then the segment has zero owners;
- if p lies between an inward virtual bisector and an outside virtual bisector of the same hole, then the segment has two owners: the horizontal edge of the hole and the third owner of the inward virtual bisector equal to the second owner of the outside bisector;
- if p lies between an active inward bisector with a support owner and a real bisector at the edge of the same hole, then the segment has two owners: the horizontal edge of the hole and the support edge owner.

In some other cases, there can be two or zero owners of the segment, yet such segments should never hold a new bisector and should not allow an intersection to pass through. The first case above can happen only for real bisector creation, and the second and third cases only for virtual creation.

Let us define the creation functions of bisectors at a point p on the wavefront with node $\mathcal{T}$ with direction flag $f \in \{-1, 1\}$ by rules for each bisector type. Because the function *Segment* has $O(\log n)$ time complexity, therefore the function *Owner* and creation functions have $O(\log n)$ time complexity as well.

With the function *Owner* returning one owner, except for the first case above, the function $Create_{\mathrm{R}}(\mathcal{T}, T_i^u, p, f)$ creates a real bisector:

- **if** $|Owner(\mathcal{T}, p, f)| = 1$ **then return** $Bv(T_i^u, Owner(\mathcal{T}, p, f))$;
- **else** $T_j^{u2} \in Owner(\mathcal{T}, p, f)$, $T_j \in T$: **return** $Bv(T_i^{u1}, T_j^{u2})$.

For the inward virtual bisector two owners are given. The function $Create_{\mathrm{In}}(\mathcal{T}, Hole_k^{u1}, Hole^{u2}, p_{\mathrm{In}}, f)$ is defined:

- **if** $|Owner(\mathcal{T}, p_{\mathrm{In}}, f)| = 1$ **and** $T_i^{u3} \in Owner(\mathcal{T}, p_{\mathrm{In}}, f)$: $T_i \in T$ **then return** $\mathcal{B}v_{\mathrm{InA}}(Hole_k^{u1}, Hole^{u2}, T_i^{u3}, p_{\mathrm{In}})$;
- **else if** $|Owner(\mathcal{T}, p_{\mathrm{In}}, f)| = 1$ **and** $R^u \in Owner(\mathcal{T}, p_{\mathrm{In}}, f)$: $p_{\mathrm{In}} \in R^u$ **then return** $\mathcal{B}v_{\mathrm{InP}}(Hole_k^{u1}, Hole^{u2}, R^u, p_{\mathrm{In}})$;
- **else if** $|Owner(\mathcal{T}, p_{\mathrm{In}}, f)| \neq 0$ **and** $Hole_k^u \in Owner(\mathcal{T}, p_{\mathrm{In}}, f)$: $\mathcal{L} \geq x_k + l_k$ **then return** $\mathcal{B}v_{\mathrm{InA}}(Hole_k^{u1}, Hole^{u2}, \varnothing, p_{\mathrm{In}})$;
- **return** $\mathcal{B}v_{\mathrm{InP}}(Hole_k^{u1}, Hole^{u2}, \varnothing, p_{\mathrm{In}})$.

For the outside virtual bisector, one owner is given, while the second is the wavefront segment owner. If there are two or zero owners, then we need to build an artificial edge owner by equation (17). The function $Create_{\mathrm{Out}}(\mathcal{T}, Hole_k^u, p, f)$ with the inducing owner $Hole_k^u$ is defined as:

- **if** $|Owner(\mathcal{T}, p, f)| \neq 1$ **then return** $\mathcal{B}v_{\mathrm{OutW}}(Hole_k^u, Hole^*(Hole_k^u, p))$;
- **else if** $R^u \in Owner(\mathcal{T}, p, f)$: $p \in R^u$ **then return** $\mathcal{B}v_{\mathrm{OutL}}(Hole_k^u, R^u)$;
- **else return** $\mathcal{B}v_{\mathrm{OutW}}(Hole_k^u, Owner(\mathcal{T}, p, f))$.

The function $Form(\mathcal{T}^{SH})$ returns a form of the shadow by a shadow node $\mathcal{T}^{SH}$: if $type(bisector(\mathcal{T}^{SH})) \in \{VE, VC\}$ or $type(bisector(end(\mathcal{T}^{SH}))) \in \{VE, VC\}$, then $Form(\mathcal{T}^{SH}) = Outer$; else $Form(\mathcal{T}^{SH}) = Inner$.

The function $Insert_t(\mathcal{T}, \mathcal{bv})$ adds bisector $\mathcal{bv}$ to the tree $\mathcal{T}$ with node type $t \in \{F, SH, DP, BL\}$. If the function inserts a shadow node, then it inserts a duplicate to the shadow of it as well. The function $Insert(\mathcal{T}, \beta v_1, \beta v_2)$ adds a shadow from the virtual bisectors βv_1 and βv_2 into tree $\mathcal{T}$ keeping nodes between them and their duplicates into the shadow. When inserting a shadow node $\mathcal{T}^{SH}$ of an inner shadow, we bond it with its pair using $Inner(Hole_k, f) = \mathcal{T}^{SH}$, $f = -1$ or 1 for the lower or upper node; if the wavefront reaches the west corners of $Hole_k$, then $Inner(Hole_k, f) = \varnothing$.

The function $Voronoi(T_i, (p_1, p_2))$: VE_i.Add((p_1, p_2)) fixates (p_1, p_2) for site T_i. Algorithm 2 describes the function $Voronoi(\mathcal{T}_{next}, \mathcal{T}_{prev}, T_{fix}, a, b)$ adding a Voronoi edge between nodes or a node and a point if a or $b \neq \varnothing$ by T_{fix}.

Voronoi($\mathcal{T}_{next}$, $\mathcal{T}_{prev}$: $\mathcal{T}_{next} = next(\mathcal{T}_{prev})$, $T_{fix} \in T$, $a \in Slab \cup \{\varnothing\}$, $b \in Slab \cup \{\varnothing\}$)
1. set p_{next} as a if it is given or as $ep(\mathcal{T}_{next})$ otherwise
2. set p_{prev} as b if it is given or as $ep(\mathcal{T}_{prev})$ otherwise
3. **if** $T_{mid}^{u} \in owners(\mathcal{T}_{prev}) \cap owners(\mathcal{T}_{next})$: $T_{mid} \in T$ **and** $T_{mid} \neq T_{fix}$ **then**
4. ***Voronoi***(T_{mid}, (p_{next}, p_{prev})), ***Voronoi***(T_{fix}, (p_{next}, p_{prev}))
5. **if** $R_{mid}^{u} \in owners(\mathcal{T}_{prev}) \cap owners(\mathcal{T}_{next})$: $p_{prev} \in R_{mid}^{u}$ **and** $p_{next} \in R_{mid}^{u}$ **and** $R_{mid} \notin T$ **then**
6. ***Voronoi***(T_{fix}, (p_{next}, p_{prev}))

Algorithm 2. Adding a Voronoi edge between two nodes

The function $Fixate(\mathcal{bv}')$ fixates a Voronoi edge by a real bisector $\mathcal{bv}'$ with two different owners. The function creates a Voronoi edge for every support owner from the beginning point to the current position of the bisector $ep(\mathcal{bv}')$. The function $Remove(\mathcal{T}, \mathcal{T}')$ deletes $\mathcal{T}'$ from $\mathcal{T}$ and fixates it: $\mathcal{T}$.Remove($\mathcal{T}'$), $Fixate(bisector(\mathcal{T}'))$.

The function $\mathcal{B}v_{insd}(\mathcal{T}^{SH})$ sets an intersection bisector of $\mathcal{T}^{SH}$, outside or inward. If $Hole_k \in Hole$ is the hole of $\mathcal{T}^{SH}$ and $end(\mathcal{T}^{SH})$, $p_{End} = ep(end(\mathcal{T}^{SH}))$, $type = type(bisector(\mathcal{T}^{SH})) \in \{VI, VO\}$, then:

- **if** $type = VI$ **then return** $\mathcal{B}v(owners(\mathcal{T}^{SH}))$ without third owner;
- **else return** $\mathcal{B}v(\{(p_{End}, (x_k + l_k, y_k)), (p_{End}, (x_{End}, y_k + w_k))\})$ – sets the bisector from the position of the paired shadow node.

The function $Front(\mathcal{T}_1)$ defines a root of the tree by searching for the leftmost node $\mathcal{T}_{min}$ from $\mathcal{T}_1$. If $type(\mathcal{T}_{min}) \in \{DP, BL\}$, then the function returns a basis from the shadow or blind node. Else it returns a root of the main tree $\mathcal{T}$.

Algorithm 3 describes the function $Relocate(\mathcal{T}, \mathcal{T}^{SH}, p)$ finding a new bisector for $\mathcal{T}^{SH}$ with an outside or inward bisector in position p of the tree $\mathcal{T}$.

Relocate($\mathcal{T} \neq \varnothing$, $\mathcal{T}^{SH}$: *type*(*bisector*($\mathcal{T}^{SH}$)) $\in$ {VI, VO}, $p \in$ *Slab*)
1. $f \leftarrow$ sgn(*key*($\mathcal{T}^{SH}$) – *key*(*end*($\mathcal{T}^{SH}$))), set $Hole_k^{u1}$ as the edge inducing the bisector of $\mathcal{T}^{SH}$
2. **if** *type*(*bisector*($\mathcal{T}^{SH}$)) = VI **then**
3. set $Hole^{u2}$ as the second edge inducing the bisector of $\mathcal{T}^{SH}$
4. **return** ***Create***$_{\mathbf{In}}$($\mathcal{T}$, $Hole_k^{u1}$, $Hole^{u2}$, p, f) ▷ return inward bisector
5. **else return** ***Create***$_{\mathbf{Out}}$($\mathcal{T}$, $Hole_k^{u1}$, p, f) ▷ return outside bisector

Algorithm 3. Relocation of outside or inward virtual bisector

The function *Inject*($\mathcal{T}$, $\mathcal{T}^{SH}$, p) inserts a relocated shadow node of the previous shadow node $\mathcal{T}^{SH}$ into the tree $\mathcal{T}$ at position p: $\mathcal{T}_{new}^{SH} = Insert_{SH}(\mathcal{T}, Relocate(\mathcal{T}, \mathcal{T}^{SH}, p))$, $end(\mathcal{T}_{new}^{SH}) = end(\mathcal{T}^{SH})$.

Algorithm 4 defines the function *Swap*($\mathcal{T}$, $\mathcal{T}_1^{DP}$, $\mathcal{T}_2^{SH}$) swapping shadow nodes *main*($\mathcal{T}_1^{DP}$) and $\mathcal{T}_2^{SH}$ with inward or outside bisectors in tree $\mathcal{T}$:

Swap($\mathcal{T} \neq \varnothing$, $\mathcal{T}_1^{DP}$, $\mathcal{T}_2^{SH} \in$ *basis*(*main*($\mathcal{T}_1^{DP}$)))
1. $\mathcal{T}_1^{SH} \leftarrow$ *main*($\mathcal{T}_1^{DP}$), $p_{1^*} \leftarrow$ ***Intersect***($\mathcal{B}v_{insd}$($\mathcal{T}_1^{SH}$), *basis*($\mathcal{T}_2^{SH}$), 1) ▷ new position of $\mathcal{T}_1^{SH}$
2. set (p_{Up}, p_{Low}) as (*ep*($\mathcal{T}_2^{SH}$), p_{1^*}) if *key*($\mathcal{T}_1^{SH}$) < *key*($\mathcal{T}_2^{SH}$) or vice versa otherwise
3. $\mathcal{T}^{btwn} \leftarrow$ ***Between***(*basis*($\mathcal{T}_2^{SH}$), p_{Up}, p_{Low})
4. $\beta v_{1^*} \leftarrow$ ***Relocate***(*basis*($\mathcal{T}_2^{SH}$), $\mathcal{T}_1^{SH}$, p_{1^*}),
5. $\beta v_{2^*} \leftarrow$ ***Relocate***($\mathcal{T}$, $\mathcal{T}_2^{SH}$, *ep*($\mathcal{T}_1^{DP}$))
6. ***Remove***(*basis*($\mathcal{T}_2^{SH}$), $\forall \mathcal{T}^* \in \mathcal{T}^{btwn}$)
7. ***Remove***(*basis*($\mathcal{T}_2^{SH}$), *dupl*($\mathcal{T}_2^{SH}$)), ***Remove***(*basis*($\mathcal{T}_1^{SH}$), $\mathcal{T}_2^{SH}$); ▷ remove $\mathcal{T}_2^{SH}$
8. ***Remove***(*basis*($\mathcal{T}_1^{SH}$), $\mathcal{T}_1^{DP}$), ***Remove***($\mathcal{T}$, $\mathcal{T}_1^{SH}$) ▷ remove $\mathcal{T}_1^{SH}$
9. $\mathcal{T}_{2^*}^{SH} \leftarrow$ ***Insert***$_{\mathbf{SH}}$($\mathcal{T}$, βv_{2^*}), *end*($\mathcal{T}_{2^*}^{SH}$) = *end*($\mathcal{T}_2^{SH}$) ▷ relocate $\mathcal{T}_2^{SH}$ to $\mathcal{T}$
10. $\mathcal{T}_{1^*}^{SH} \leftarrow$ ***Insert***$_{\mathbf{SH}}$(*basis*($\mathcal{T}_{2^*}^{SH}$), βv_{1^*}), *end*($\mathcal{T}_{1^*}^{SH}$) = *end*($\mathcal{T}_1^{SH}$) ▷ relocate $\mathcal{T}_1^{SH}$ into $\mathcal{T}_{2^*}^{SH}$
11. ***Insert***$_{\mathbf{type}(\mathcal{T}^*)}$(*basis*($\mathcal{T}_{1^*}^{SH}$), $\forall \mathcal{T}^* \in \mathcal{T}^{btwn}$) ▷ reinsert nodes of $\mathcal{T}^{btwn}$
12. **return** ($\mathcal{T}_{1^*}^{SH}$, $\mathcal{T}_{2^*}^{SH}$, $\mathcal{T}^{btwn}$) ▷ relocated nodes and $\mathcal{T}^{btwn}$

Algorithm 4. Swapping shadow nodes in tree

Algorithm 5 defines the function *OnShadow*($\mathcal{T}_{on}$, $f \in \{-1, 1\}$) giving a shadow node of the intersected shadow in the segment from node $\mathcal{T}_{on}$ in ordinate direction f and ascending abscissa. The time complexity of the function is $O(K)$, where K is the count of holes.

OnShadow($\mathcal{T}_{on} \neq \varnothing$: *type*(*bisector*($\mathcal{T}_{on}$)) $\in$ {VE, VC, VI ,VO}, $f \in \{-1, 1\}$)
1. $p_{on} \leftarrow$ *ep*($\mathcal{T}_{on}$), set $\mathcal{T}'$ as *next*($\mathcal{T}_{on}$) if f = 1 or as *prev*($\mathcal{T}_{on}$) otherwise, $p' \leftarrow$ *ep*($\mathcal{T}'$)
2. **while** ($x' > x_{on}$ **and** sgn($y' - y_{on}$) = f) **or** $p' = p_{on}$ **do** ▷ the same point or co-directional
3. check if *type*(*bisector*($\mathcal{T}_{on}$)) $\notin$ {VE, VC, VI ,VO} then **return** $\varnothing$ ▷ must be virtual
4. **if** *type*($\mathcal{T}'$) = SH **then** ▷ east corner of hole
5. **return** *end*($\mathcal{T}'^{SH}$);
6. **if** *type*(*bisector*($\mathcal{T}'$)) = VC **then** ▷ west corner of hole
7. set $Hole_k$ as hole owner of node $\mathcal{T}'$ with west corner bisector by *owners*($\mathcal{T}'$)
8. **return** ***Inner***($Hole_k$, f) ▷ inward bisector to the corner $\mathcal{T}'$
9. set $\mathcal{T}'$ as *main*($\mathcal{T}'^{DP}$) if *type*($\mathcal{T}'$) = DP ▷ get a shadow node if duplicate

10. set $\mathcal{T}'$ as *next*($\mathcal{T}'$) if $f = 1$ or as *prev*($\mathcal{T}'$) otherwise, $p' \leftarrow ep(\mathcal{T}')$
11. **return** ∅ ▷ no intersected shadow in segment

Algorithm 5. Checking existence of intersected shadow in wavefront segment

Algorithm 6 defines the function *Fixate*($\mathcal{T}$, bv_a' | a, bv_b' | b, $T_i \in T$, $f \in \{-1, 1\}$) fixating nodes between bisectors or points bv_a' or a and bv_b' or b in $\mathcal{T}$ by site T_i with an inclusion flag f. If $f = -1$ then the algorithm does not process shadow nodes at the intersection points; for the West Edge event, the initial state f equals –1. There are three cases processing shadows:

- if bv_a' or bv_b' intersects a pair of shadow nodes, and only one of the pair lies between, then we fixate part of their shadow and relocate the shadow node in the between set at a new position (Figure 7a);
- if there is a lower shadow node of an outer shadow between the intersection points and the upper shadow node is also in the between set, then the algorithm fixates the shadow of the shadow node entirely (Figure 7b);
- if there is a hole west vertex in the between set that is not on the slab edge then the algorithm fixates part of the inner shadow maintained by the inward shadow node directed to the related west vertex (Figure 7c).

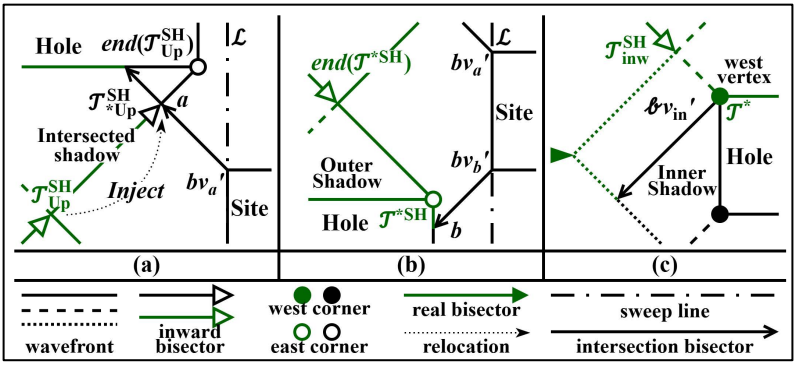

Fig. 7. Special cases of fixation: intersecting a shadow a) fixating an outer shadow, b) fixating an inner shadow, c) fixated and removed elements shown in green

Fixate($\mathcal{T} \neq \varnothing$, bv_a' **or** a, bv_b' **or** b, $T_i \in T$, $f \in \{-1, 1\}$)
1. $a \leftarrow$ ***Intersect***(bv_a', $\mathcal{T}$, f), $b \leftarrow$ ***Intersect***(bv_b', $\mathcal{T}$, f) ▷find intersection points if not given
2. $\mathcal{T}^{btwn} \leftarrow$ ***Between***($\mathcal{T}$, a, b), $\mathcal{T}_{First} \leftarrow \mathcal{T}^{btwn}$.First(), $\mathcal{T}_{Last} \leftarrow \mathcal{T}^{btwn}$.Last()
3. **if** $f = -1$ **then** ▷ is there shadow nodes at a or b
4. **while** $\mathcal{T}^{btwn} \neq \varnothing$ **and** *type*($\mathcal{T}_{First}$) = SH **do** ▷ shadow nodes in point b
5. $\mathcal{T}^{btwn}$.Remove($\mathcal{T}_{First}$), $\mathcal{T}_{First} \leftarrow \mathcal{T}^{btwn}$.First()
6. **while** $\mathcal{T}^{btwn} \neq \varnothing$ **and** *type*($\mathcal{T}_{Last}$) = SH **do** ▷ shadow nodes in point a
7. $\mathcal{T}^{btwn}$.Remove($\mathcal{T}_{Last}$), $\mathcal{T}_{Last} \leftarrow \mathcal{T}^{btwn}$.Last()

8. $\mathcal{T}_{\text{Low}}^{\text{SH}} \leftarrow$ ***OnShadow***$(\mathcal{T}_{\text{First}} \neq \varnothing, -1)$, $\mathcal{T}_{\text{Up}}^{\text{SH}} \leftarrow$ ***OnShadow***$(\mathcal{T}_{\text{Last}} \neq \varnothing, 1)$
9. **if** $\mathcal{T}_{\text{Up}}^{\text{SH}} \neq \varnothing$ **then** ▷ shadow intersected in point *a*
10. ***Fixate***$(basis(\mathcal{T}_{\text{Up}}^{\text{SH}}), bv_a', ep(\mathcal{T}_{\text{Up}}^{\text{SH}}), T_i, f)$ ▷ fixate under *a* (Fig. 7a);
11. **if** $\mathcal{T}^{\text{btwn}} = \varnothing$ **then** ▷ no nodes between *a* and *b* case
12. $(\mathcal{T}_{\text{First}}, \mathcal{T}_{\text{Last}}) \leftarrow$ ***Segment***$(\mathcal{T}, b, 1)$, ***Voronoi***$(\mathcal{T}_{\text{First}}, \mathcal{T}_{\text{Last}}, T_i, a, b)$
13. **else** ***Voronoi***$(\mathcal{T}_{\text{First}}, prev(\mathcal{T}_{\text{First}}), T_i, \varnothing, b)$, ***Voronoi***$(next(\mathcal{T}_{\text{Last}}), \mathcal{T}_{\text{Last}}, T_i, a, \varnothing)$
14. **for** $\mathcal{T}^* \in \mathcal{T}^{\text{btwn}}$ **do** ▷ nodes in $\mathcal{T}^{\text{btwn}}$ from upper to lower
15. $\mathscr{b}v^* \leftarrow bisector(\mathcal{T}^*)$, $p^* \leftarrow ep(\mathscr{b}v^*)$, $p_{\text{next}} \leftarrow ep(bisector(next(\mathcal{T}^*)))$
16. **if** $type(\mathcal{T}^*)$ = SH **and** ***Form***$(\mathcal{T}^{*\text{SH}})$ = Outer **and** $y^* < key(end(\mathcal{T}^{*\text{SH}}))$ **then** ▷ Fig. 7b
17. ***Fixate***$(basis(\mathcal{T}^{*\text{SH}}), p^*, ep(end(\mathcal{T}^{*\text{SH}})), T_i, -1)$)
18. **else if** $type(\mathscr{b}v^*)$ = VC **and** $type(\mathcal{T}^*)$ = F **and** $\nexists Slab^u$: $p^* \in Slab^u$ **then** ▷Fig. 7c
19. $(Hole_k^w, Hole_k^u) \leftarrow owners(\beta v^*)$ ▷ west and horizontal edge from βv^*
20. **if** $Hole_k^u = Hole_k^s$ **then** ▷ find extension of horizontal edge
21. $Hole^{+w} \leftarrow ((x_k, y_k), (x_k, y_k - w_k))$, $f_{\text{inw}} \leftarrow -1$
22. **else** $Hole^{+w} \leftarrow ((x_k, y_k + w_k), (x_k, y_k + 2 \cdot w_k))$, $f_{\text{inw}} \leftarrow 1$
23. $\mathcal{T}_{\text{inw}}^{\text{SH}} \leftarrow$ ***Inner***$(Hole_k, f_{\text{inw}})$, $p_{\text{inw}} \leftarrow ep(\mathcal{T}_{\text{inw}}^{\text{SH}})$, $\mathscr{b}v_{\text{in}}' \leftarrow$ ***Bv***$(Hole^{+w}, Hole_k^u)$
24. ***Fixate***$(basis(\mathcal{T}_{\text{inw}}^{\text{SH}}), p_{\text{inw}}, \mathscr{b}v_{\text{in}}', T_i, -1)$ ▷ fixate part of inner shadow
25. $\mathcal{T}_w^{\text{BL}} \leftarrow Insert_{\text{BL}}(basis(\mathcal{T}_{\text{inw}}^{\text{SH}}), Bv(T_i, Hole_k^w))$, $end(\mathcal{T}_w^{\text{BL}}) \leftarrow end(\mathcal{T}_{\text{inw}}^{\text{SH}})$
26. Insert in Q a new intersection event for $\mathcal{T}_w^{\text{BL}}$ if needed
27. add a Voronoi edge ***Voronoi***$(next(\mathcal{T}^*), \mathcal{T}^*, T_i, \varnothing, \varnothing)$ if $\mathcal{T}^* \neq \mathcal{T}_{\text{Last}}$
28. ***Remove***$(\mathcal{T}, \forall \mathcal{T}^* \in \mathcal{T}^{\text{btwn}})$
29. **if** $\mathcal{T}_{\text{Low}}^{\text{SH}} \neq \varnothing$ **then** ▷ shadow intersected in point *b*
30. ***Fixate***$(basis(\mathcal{T}_{\text{Low}}^{\text{SH}}), ep(\mathcal{T}_{\text{Low}}^{\text{SH}}), bv_b', T_i, f)$ ▷ fixate above *b* (Fig. 7a)
31. $\mathcal{T}_{\text{Low}^*}^{\text{SH}} \leftarrow$ ***Inject***$(\mathcal{T}, \mathcal{T}_{\text{Low}}^{\text{SH}}, b)$ ▷ relocate lower shadow node
32. **else** $\mathcal{T}_b \leftarrow$ ***Insert***$_{\text{F}}(\mathcal{T}$, ***Create***$_{\text{R}}(\mathcal{T}, T_i^s, b, -1))$ ▷ create bisector in lower point
33. **if** $\mathcal{T}_{\text{Up}}^{\text{SH}} = \varnothing$ **then**
34. $\mathcal{T}_a \leftarrow$ ***Insert***$_{\text{F}}(\mathcal{T}$, ***Create***$_{\text{R}}(\mathcal{T}, T_i^n, a, 1))$ ▷ create bisector in upper point
35. **else** $\mathcal{T}_{\text{Up}^*}^{\text{SH}} \leftarrow$ ***Inject***$(\mathcal{T}, \mathcal{T}_{\text{Up}}^{\text{SH}}, a)$ ▷ relocate upper shadow node
36. Insert in Q new intersection events for $\mathcal{T}_a$ and $\mathcal{T}_b$ or for $\mathcal{T}_{\text{Up}^*}^{\text{SH}}$ and $\mathcal{T}_{\text{Low}^*}^{\text{SH}}$ if needed

Algorithm 6. Fixation of the nodes between two points

The worst case of the count of *Fixate* executions follows. The left half of the holes lies lower than the other half in a descending order of abscissa and ordinate with upper outer shadows and upper shadow nodes of inner shadows enveloping each other, while the right half lies in the same order; therefore, their lower shadow nodes of inner shadows lie on the segment between the upper shadow node of inner shadow and the northwest corner of the uppermost hole from the left half. This is the worst possible case of a single *Fixate* execution, because the fixation from the northwest corner of every hole (step 18) from the right half can possibly interact with each upper outer shadow and upper parts of inner shadows of every hole from the left half. Then the *Fixate* operation will happen $K^2/2 + 5 \cdot K + 1$ times, where K is the count of holes. But the average number of occurrences for every outer shadow and twice for every inner shadow is $4 \cdot K + 1$ times.

The operation with the worst time complexity in each execution is step 28 removing nodes from the between set with $O(n \log n)$ in the worst case of each node removed (n is the count of nodes) and *OnShadow* execution with $O(K)$ in the worst case. Therefore, the time complexity of the *Fixate* operation is $O(K^2 \cdot n \cdot \log(n) + K^3)$.

Therefore, the proposed structure of the shadows in AVL trees handles the holes processing. Table 3 describes the wavefront functions and their time complexity, where n is the count of input nodes, k is the count of output nodes, and K is the count of holes. Further, the new sweep line events corresponding to the holes are described.

Table 3. Wavefront operations and functions

Function	Description	Time	Ref.
Intersect($\mathcal{B}v'$, $\mathcal{T}$, f)	Gets wavefront intersection by bisector	$O(\log n)$	Alg. 1
Between($\mathcal{T}$, p_{up}, p_{low})	Gets nodes in tree between points	$O(\log n + k)$	(18)
Segment($\mathcal{T}$, p, f)	Gets nodes of wavefront segment in point	$O(\log n)$	(19)
Owner($\mathcal{T}$, p, f)	Gets edge owner of wavefront in point	$O(\log n)$	(20)
Create$_{\text{R}}$($\mathcal{T}$, T_i^u, p, f)	Creates real bisector by owner in point	$O(\log n)$	–
Create$_{\text{In}}$($\mathcal{T}$, *Hole*$_k^{u1}$, *Hole*^{u2}, p_{In}, f)	Creates inward bisector by owners in point	$O(\log n)$	–
Create$_{\text{Out}}$($\mathcal{T}$, *Hole*$_k^u$, p, f)	Creates outside bisector by owner in point	$O(\log n)$	–
Form($\mathcal{T}^{\text{SH}}$)	Gets form of the shadow (Outer or Inner)	$O(1)$	–
Insert$_{type}$($\mathcal{T}$, $\mathcal{B}v$)	Creates a node of chosen type by bisector	$O(\log n)$	–
Insert($\mathcal{T}$, βv_1, βv_2)	Creates shadow nodes pair by bisectors	$O(\log n)$	–
Inner(*Hole*$_k$, f)	Gets a shadow node of inner shadow	$O(1)$	–
Voronoi(T_i, (p_1, p_2))	Sets Voronoi edge for support by segment	$O(1)$	–
Voronoi($\mathcal{T}_{\text{next}}$, $\mathcal{T}_{\text{prev}}$, T_i, a, b)	Sets Voronoi edge between nodes or points	$O(1)$	Alg. 2
Fixate($\mathcal{B}v'$)	Fixates Voronoi edge by bisector	$O(1)$	–
Remove($\mathcal{T}$, $\mathcal{T}'$)	Removes node from tree and fixates it	$O(\log n)$	–
$\mathcal{B}v_{\text{insd}}$($\mathcal{T}^{\text{SH}}$)	Gets intersection bisector by inward, outside	$O(1)$	–
Front($\mathcal{T}_1$)	Gets root node of the tree in which $\mathcal{T}_1$ is	$O(\log n)$	–
Relocate($\mathcal{T}$, $\mathcal{T}^{\text{SH}}$, p)	Relocates shadow node to point into tree	$O(\log n)$	Alg. 3
Inject($\mathcal{T}$, $\mathcal{T}^{\text{SH}}$, p)	Inserts a relocated shadow node	$O(\log n)$	–
Swap($\mathcal{T}$, $\mathcal{T}_1^{\text{DP}}$, $\mathcal{T}_2^{\text{SH}}$)	Swaps inward or outside shadow nodes	$O(n \log n)$	Alg. 4
OnShadow($\mathcal{T}_{\text{on}}$, f)	Checks if shadow under wavefront segment	$O(K)$	Alg. 5
Fixate($\mathcal{T}$, $bv_a' \mid a$, $bv_b' \mid b$, T_i, f)	Fixates nodes between bisectors or points from the wavefront	$O(K^2 \cdot n \cdot \log(n) + K^3$	Alg. 6

Let us extend set from (11) with new event types: Begin Hole and End Hole process holes; Blind Spike and Inner Wave describe the wavefront reaching the west edge of a hole; Virtual Spike, Outside Spike, Inward Spike and East Corner set the intersection with virtual bisectors. The priority of the first pair equals the abscissa of the edge; the priority of others follows the spike event priority as in (12). The ordering of the events with

the same priority: Blind Spike – Spike – East Edge – Outside Spike – End Hole – Inner Wave – Inward Spike – Virtual Spike – West Edge – East Corner – Begin Hole – Turn. The time complexity of the proposed algorithm is given in Section 4.6.

4.4. Hole Events of Voronoi Diagram. Algorithm 7 defines the Begin Hole event creating the hole shadow from Figure 5. The event has two special cases for intersecting shadows. Let us accept that the algorithm intersects only one shadow layer; then it is always an outer shadow. Then the shadow node with outside virtual bisector is removed from the between set and relocated to the main tree. If this node lies inside another shadow (the second special case), then its shadow node is relocated as well.

Algorithm 8 defines the End Hole event changing the outer shadows. The special case of the End Hole event occurs when the outside bisector belongs to the other active outer shadow. Then the outside bisector of this shadow should also be relocated inside the recreated shadow.

Input $\mathcal{T} \neq \varnothing$, $q \leftarrow Q.pop()$, $\mathcal{L} \leftarrow Priority(q)$, $type(q) \leftarrow$ Begin Hole, $Hole_q$ – hole of the event q
1. $a \leftarrow$ ***Intersect***$(\mathcal{B}v(Hole_q{}^w, Hole_q{}^n), \mathcal{T}, 1)$, $b \leftarrow$ ***Intersect***$(\mathcal{B}v(Hole_q{}^w, Hole_q{}^s), \mathcal{T}, 1)$
2. $\mathcal{T}^{btwn} \leftarrow$ ***Between***$(\mathcal{T}, a, b)$, $\mathcal{T}_{First} \leftarrow \mathcal{T}^{btwn}$.First(), $\mathcal{T}_{Last} \leftarrow \mathcal{T}^{btwn}$.Last()
3. $\mathcal{T}_a{}^{SH} \leftarrow$ ***OnShadow***$(\mathcal{T}_{First}, -1)$, $\mathcal{T}_b{}^{SH} \leftarrow$ ***OnShadow***$(\mathcal{T}_{Last}, 1)$ =
4. **if** $\mathcal{T}_b{}^{SH} \neq \varnothing$ **then** ▷ lower bisector intersect shadow
5. $a^* \leftarrow ep(\mathcal{T}_b{}^{SH})$, $b_{new} \leftarrow$ ***Intersect***$(\mathcal{B}v(Hole_q{}^w, Hole_q{}^s), basis(\mathcal{T}_b{}^{SH}), 1)$
6. $\mathcal{T}^{btwn} \leftarrow \{\mathcal{T}^* \mid \mathcal{T}^* \in$ ***Between***$(basis(\mathcal{T}_b{}^{SH}), a^*, b_{new})\} \cup \mathcal{T}^{btwn}$
7. $\beta v_{In}{}^s \leftarrow$ ***Create***$_{In}(basis(\mathcal{T}_b{}^{SH}), Hole_q{}^w, Hole_q{}^s, b_{new}, -1)$ ▷ lower inward bisector
8. $\beta v_{Out}{}^s \leftarrow$ ***Create***$_{Out}(basis(\mathcal{T}_b{}^{SH}), Hole_q{}^s, b_{new}, -1)$ ▷ lower outside bisector
9. **else** $\beta v_{In}{}^s \leftarrow$ ***Create***$_{In}(\mathcal{T}, Hole_q{}^w, Hole_q{}^s, b, -1)$, $\beta v_{Out}{}^s \leftarrow$ ***Create***$_{Out}(\mathcal{T}, Hole_q{}^s, b, -1)$
10. **if** $\mathcal{T}_a{}^{SH} \neq \varnothing$ **then** ▷ upper bisector intersect shadow
11. $a_{new} \leftarrow$ ***Intersect***$(\mathcal{B}v(Hole_q{}^w, Hole_q{}^n), basis(\mathcal{T}_a{}^{SH}), 1)$, $b^* \leftarrow ep(\mathcal{T}_a{}^{SH})$
12. $\mathcal{T}^{btwn} \leftarrow \mathcal{T}^{btwn} \cup \{\mathcal{T}^* \mid \mathcal{T}^* \in$ ***Between***$(basis(\mathcal{T}_a{}^{SH}), a_{new}, b^*)\}$
13. $\beta v_{In}{}^n \leftarrow$ ***Create***$_{In}(basis(\mathcal{T}_a{}^{SH}), Hole_q{}^w, Hole_q{}^n, a_{new}, 1)$ ▷ upper inward bisector
14 $\beta v_{Out}{}^n \leftarrow$ ***Create***$_{Out}(basis(\mathcal{T}_a{}^{SH}), Hole_q{}^n, a_{new}, 1)$ ▷ upper outside bisector
15. **else** $\beta v_{In}{}^n \leftarrow$ ***Create***$_{In}(\mathcal{T}, Hole_q{}^w, Hole_q{}^n, a, 1)$, $\beta v_{Out}{}^n \leftarrow$ ***Create***$_{Out}(\mathcal{T}, Hole_q{}^n, a, 1)$
16. **for** $\mathcal{T}_{inter}{}^{SH}$ in $\{\mathcal{T}_b{}^{SH} \neq \varnothing, \mathcal{T}_a{}^{SH} \neq \varnothing\}$ **do** ▷ first special case
17. set $\mathcal{T}_{neigh}$ as $prev(\mathcal{T}_{inter}{}^{SH})$ if $\mathcal{T}_{inter}{}^{SH} = \mathcal{T}_b{}^{SH}$ or as $next(\mathcal{T}_{inter}{}^{SH})$ otherwise
18. **if** $end(\mathcal{T}_{inter}{}^{SH}) \neq \mathcal{T}_{neigh}$ **then** ▷ second special case
19. $(\mathcal{T}_{neigh^*}{}^{SH}, \mathcal{T}_{inter}{}^{SH}, \mathcal{T}^{*btwn}) \leftarrow$ ***Swap***$(\mathcal{T}, dupl(\mathcal{T}_{neigh}{}^{SH}), \mathcal{T}_{inter}{}^{SH})$
20. $\mathcal{T}^{btwn} \leftarrow \mathcal{T}^{btwn} \setminus (\mathcal{T}^{*btwn} \cup \{\mathcal{T}_{neigh}{}^{SH}, \mathcal{T}_{inter}{}^{DP}\})$, $\mathcal{T}^{btwn} \leftarrow \{\mathcal{T}_{neigh^*}{}^{SH}\} \cup \mathcal{T}^{btwn}$
21. set $\mathcal{T}_a{}^{SH}$ as $\mathcal{T}_{neigh^*}{}^{SH}$ if $main(\mathcal{T}_{neigh}{}^{SH}) = \mathcal{T}_a{}^{SH}$ ▷ reference the relocated shadow
22. ***Remove***$(basis(\mathcal{T}_{inter}{}^{SH}), \mathcal{T}_{inter}{}^{DP})$, ***Remove***$(\mathcal{T}, \mathcal{T}_{inter}{}^{SH})$
23. $\mathcal{T}^{btwn} \leftarrow \mathcal{T}^{btwn} \setminus \{\mathcal{T}_{inter}{}^{SH}, \mathcal{T}_{inter}{}^{DP}\}$
24. ***Remove***$(basis(\mathcal{T}_{inter}{}^{SH}), \forall \mathcal{T}^* \in \mathcal{T}^{btwn})$
25. ***Insert***$(\mathcal{T}, \beta v_{In}{}^n, \beta v_{In}{}^s)$ ▷ inner shadow with $\mathcal{T}^{btwn}$ inside
26. ***Insert***$(\mathcal{T}, \mathcal{B}v_{Corner}(Hole_q{}^w, Hole_q{}^n))$, ***Insert***$(\mathcal{T}, \mathcal{B}v_{Corner}(Hole_q{}^w, Hole_q{}^s))$ ▷ west corners

27. ***Insert***($\mathcal{T}$, ***Bv*$_{\text{Edge}}$**($Hole_q^n$), $\beta v_{\text{Out}}{}^n$) ▷ upper shadow with north inward and corner inside
28. ***Insert***($\mathcal{T}$, ***Bv*$_{\text{Edge}}$**($Hole_q^s$), $\beta v_{\text{Out}}{}^s$) ▷ lower shadow with south inward and corner inside
29. set $\mathcal{T}_{b^*}{}^{\text{SH}}$ as ***Inject***($\mathcal{T}$, $\mathcal{T}_b{}^{\text{SH}}$, b) if $\mathcal{T}_b{}^{\text{SH}} \neq \varnothing$ ▷ relocate $\mathcal{T}_b{}^{\text{SH}}$ with $\beta v_{\text{Out}}{}^s$ inside
30. set $\mathcal{T}_{a^*}{}^{\text{SH}}$ as ***Inject***($\mathcal{T}$, $\mathcal{T}_a{}^{\text{SH}}$, a) if $\mathcal{T}_a{}^{\text{SH}} \neq \varnothing$ ▷ relocate $\mathcal{T}_a{}^{\text{SH}}$ with $\beta v_{\text{Out}}{}^n$ inside
31. Insert in Q new events for every inserted and relocated node, including the duplicates

Algorithm 7. Begin Hole event processing

Input $\mathcal{T} \neq \varnothing$, $q \leftarrow Q.pop()$, $\mathcal{L} \leftarrow Priority(q)$, $type(q) \leftarrow$ End Hole, $Hole_q$ – hole of the event q

1. set $\mathcal{T}_{\text{NEdge}}{}^{\text{SH}}$ as node of ***Bv*$_{\text{Edge}}$**($Hole_q^n$), set $\mathcal{T}_{\text{SEdge}}{}^{\text{SH}}$ as node of ***Bv*$_{\text{Edge}}$**($Hole_q^s$)
2. **for** $\mathcal{T}_{\text{Edge}}{}^{\text{SH}}$ in {$\mathcal{T}_{\text{NEdge}}{}^{\text{SH}}$, $\mathcal{T}_{\text{SEdge}}{}^{\text{SH}}$} **do** ▷ for outer shadows
3. $\mathcal{T}_{\text{Out}}{}^{\text{SH}} \leftarrow end(\mathcal{T}_{\text{Edge}}{}^{\text{SH}})$, ***Remove***($dupl(\mathcal{T}_{\text{Edge}}{}^{\text{SH}})$, $dupl(\mathcal{T}_{\text{Edge}}{}^{\text{SH}})$), ***Remove***($\mathcal{T}$, $\mathcal{T}_{\text{Edge}}{}^{\text{SH}}$)
4. **if** $\mathcal{T}_{\text{Edge}}{}^{\text{SH}} = \mathcal{T}_{\text{NEdge}}{}^{\text{SH}}$ **then**
5. $\mathcal{T}_{\text{neigh}} \leftarrow next(\mathcal{T}_{\text{Edge}}{}^{\text{SH}})$, $Hole_q{}^u \leftarrow Hole_q{}^n$, $f \leftarrow 1$
6. **else** $\mathcal{T}_{\text{neigh}} \leftarrow prev(\mathcal{T}_{\text{Edge}}{}^{\text{SH}})$, $Hole_q{}^u \leftarrow Hole_q{}^s$, $f \leftarrow -1$
7. $\mathcal{T}_{\text{Corner}}{}^{\text{SH}} \leftarrow$ ***Insert*$_{\text{SH}}$**($\mathcal{T}$, ***Bv*$_{\text{Corner}}$**($Hole_q{}^e$, $Hole_q{}^u$)), $end(\mathcal{T}_{\text{Corner}}{}^{\text{SH}}) \leftarrow \mathcal{T}_{\text{Out}}{}^{\text{SH}}$
8. **if** $\mathcal{T}_{\text{neigh}} \neq \mathcal{T}_{\text{Out}}{}^{\text{SH}}$ **then** ▷ the special case
9. ($\mathcal{T}_{\text{neigh}^*}{}^{\text{SH}}$, $\mathcal{T}_{\text{Out}}{}^{\text{SH}}$, $\mathcal{T}^{\text{btwn}}$) $\leftarrow$ ***Swap***($\mathcal{T}$, $dupl(\mathcal{T}_{\text{neigh}}{}^{\text{SH}})$, $\mathcal{T}_{\text{Out}}{}^{\text{SH}}$)
10. $\beta v_{\text{In}} \leftarrow$ ***Create*$_{\text{In}}$**($\mathcal{T}$, $Hole_q{}^e$, $ep(\mathcal{T}_{\text{Out}}{}^{\text{SH}})$, f)
11. ***Remove***($basis(\mathcal{T}_{\text{Out}}{}^{\text{SH}})$, $dupl(\mathcal{T}_{\text{Out}}{}^{\text{SH}})$), ***Remove***($\mathcal{T}$, $\mathcal{T}_{\text{Out}}{}^{\text{SH}}$)
12. $\mathcal{T}_{\text{In}}{}^{\text{SH}} \leftarrow$ ***Insert*$_{\text{SH}}$**($\mathcal{T}$, βv_{In}), $end(\mathcal{T}_{\text{In}}{}^{\text{SH}}) \leftarrow \mathcal{T}_{\text{Corner}}{}^{\text{SH}}$
13. Insert in Q new events for every inserted and relocated node, including the duplicates

Algorithm 8. End Hole event processing

Figure 8 illustrates the first and second special case of Begin Hole event and the special case of End Hole event.

Algorithm 9 defines the Blind Spike event handling intersection between a real node and a blind node at a point p_q. If both nodes are blind, then the algorithm only removes them; else it creates a new blind node in the same or in another position along the edge.

Algorithm 10 defines the Inner Wave event of the wavefront reaching the west edge of the hole. It occurs if some node with a real bisector bv_q reaches the west edge of $Hole_q$; then the wavefront segment with the bisector bv_q should be fixated partly or entirely on the west edge at the current moment of the sweep line. To track the Inner Wave event during the insertion of the new events for created flat nodes or relocated flat nodes from the between set in the Begin Hole event, we should check if they lie in the inner shadow using *Front* and if they have an intersection with the west edge of the hole. If p_q is the point of intersection, T_q is any owner of bv_q, then the priority of the Inner Wave event is calculated by equation (12): $x_q + \mathrm{d}(p_q, T_q)$.

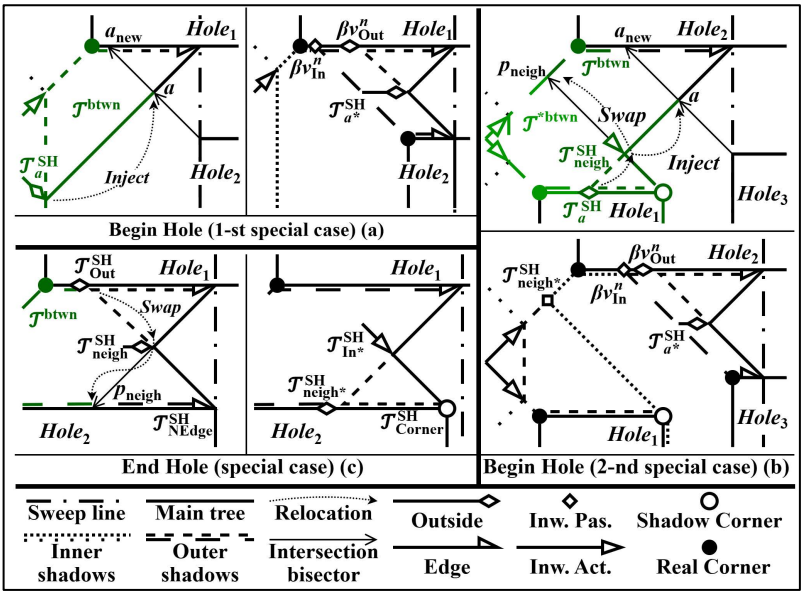

Fig. 8. Special cases of hole events: first a) and second, b) special cases of Begin Hole event, c) special case of End Hole; changed nodes shown in green

Input $q \leftarrow Q.pop()$, $\mathcal{L} \leftarrow Priority(q)$, $type(q) \leftarrow$ Blind Spike, $\mathcal{T}_{q1}$, $\mathcal{T}_{q2}$: $\mathcal{T}_{q2} \leftarrow next(\mathcal{T}_{q1})$, p_q
1. **if** $type(\mathcal{T}_{q1})$ = BL **and** $type(\mathcal{T}_{q2})$ = BL **then** ▷ both bisectors blind
2. ***Remove***$(basis(\mathcal{T}_{q1}^{BL}), \mathcal{T}_{q2})$, ***Remove***$(basis(\mathcal{T}_{q1}^{BL}), \mathcal{T}_{q1})$, **End** ▷ only remove them
3. **if** $type(\mathcal{T}_{q1})$ = BL **then**
4. $\mathcal{T}^{BL} \leftarrow \mathcal{T}_{q1}^{BL}$, $\mathcal{T}' \leftarrow \mathcal{T}_{q2}$, $\mathcal{T}_{neigh} \leftarrow next(\mathcal{T}')$, $p_{neigh} \leftarrow ep(\mathcal{T}_{neigh})$, $Hole^w \in owners(\mathcal{T}_{q1})$
5. **else** $\mathcal{T}^{BL} \leftarrow \mathcal{T}_{q2}^{BL}$, $\mathcal{T}' \leftarrow \mathcal{T}_{q1}$, $\mathcal{T}_{neigh} \leftarrow prev(\mathcal{T}')$, $p_{neigh} \leftarrow ep(\mathcal{T}_{neigh})$, $Hole^w \in owners(\mathcal{T}_{q2})$
6. **while** $\mathcal{T}_{neigh} \neq \varnothing$ **and** $x_{neigh} = x_q$ **do** ▷ while nodes with same abscissa
7. $T_i^{\mu} \leftarrow owners(\mathcal{T}') \cap owners(\mathcal{T}_{neigh})$, ***Voronoi***$(T_i, (ep(\mathcal{T}'), p_{neigh}))$
8. ***Remove***$(basis(\mathcal{T}^{BL}), \mathcal{T}')$, $\mathcal{T}' \leftarrow \mathcal{T}_{neigh}$
9. set $\mathcal{T}_{neigh}$ as $next(\mathcal{T}')$ if $type(\mathcal{T}_{q1})$ = BL or as $prev(\mathcal{T}')$ otherwise
10. **if** $type(\mathcal{T}') \notin$ {BL, DP} **then**
11. $T_i^{\mu} \leftarrow owners(\mathcal{T}') \cap owners(\mathcal{T}_{neigh})$, ***Remove***$(basis(\mathcal{T}^{BL}), \mathcal{T}')$
12. ***Remove***$(basis(end(\mathcal{T}^{BL})), \mathcal{T}^{BL})$, $\mathcal{T}_{new}^{BL} \leftarrow$ ***Insert***$_{BL}(basis(end(\mathcal{T}^{BL})),$ ***Bv***$(T_i^{\mu}, Hole^w))$
13. $end(\mathcal{T}_{new}^{BL}) \leftarrow end(\mathcal{T}^{BL})$
14. Insert in Q new spike events for $\mathcal{T}_{new}^{BL}$

Algorithm 9. Blind Spike event processing

Input $q \leftarrow Q.pop()$, $\mathcal{L} \leftarrow Priority(q)$, $type(q) \leftarrow$ Inner Wave, $\mathcal{T}_q$: $bisector(\mathcal{T}_q) \leftarrow bv_q$, $Hole_q^w$
1. check if $\mathcal{T}_q$ is removed then ignore this event
2. $\mathcal{T}' \leftarrow prev(\mathcal{T}_q)$, $p' \leftarrow ep(\mathcal{T}')$, $\mathcal{T}^{Wave} \leftarrow \{\mathcal{T}_q\}$, set x_q as abscissa of hole's west edge
3. add to $\mathcal{T}^{Wave}$ all nodes from left to right with current abscissa = x_q by *left* and *right* from $\mathcal{T}_q$

4. remove first node from $\mathcal{T}^{\text{Wave}}$ if it's duplicate or blind $\mathcal{T}^{\text{Wave}}$.Remove($\mathcal{T}^{\text{Wave}}$.First())
5. remove last node from $\mathcal{T}^{\text{Wave}}$ if it's duplicate or blind $\mathcal{T}^{\text{Wave}}$.Remove($\mathcal{T}^{\text{Wave}}$.Last())
6. $\mathcal{T}_{\text{Low}} \leftarrow \mathcal{T}^{\text{Wave}}$.First(), $\mathcal{T}_{\text{Up}} \leftarrow \mathcal{T}^{\text{Wave}}$.Last(), $\mathcal{T}_{\text{Front}} \leftarrow \mathit{Front}(\mathcal{T}_q)$
7. $T_{\text{Low}}{}^u \leftarrow \mathit{owners}(\mathcal{T}_{\text{Low}}) \cap \mathit{owners}(\mathit{next}(\mathcal{T}_{\text{Low}}))$, $T_{\text{Up}}{}^u \leftarrow \mathit{owners}(T_{\text{Up}}) \cap \mathit{owners}(\mathit{prev}(T_{\text{Up}}))$
8. **for** $\mathcal{T}^* \in \mathcal{T}^{\text{Wave}} \setminus \mathcal{T}^{\text{Wave}}$.Last() **do** ▷ fixate segments lying on edge of hole
9. $T_i{}^u \leftarrow \mathit{owners}(\mathcal{T}^*) \cap \mathit{owners}(\mathit{next}(\mathcal{T}^*))$, ***Voronoi***$(T_i, (\mathit{ep}(\mathcal{T}^*), \mathit{ep}(\mathit{next}(\mathcal{T}^*))))$
10. $\mathcal{T}_{\text{UpNew}} \leftarrow$ ***Insert***$_{\text{F}}(\mathcal{T}_{\text{Front}}, \mathit{Bv}(T_{\text{Up}}{}^u, \mathit{Hole}_q{}^w))$, $\mathcal{T}_{\text{LowNew}} \leftarrow$ ***Insert***$_{\text{F}}(\mathcal{T}_{\text{Front}}, \mathit{Bv}(T_{\text{Low}}{}^u, \mathit{Hole}_q{}^w))$
11. ***Remove***$(\mathcal{T}_{\text{Front}}, \forall \mathcal{T}^* \in \mathcal{T}^{\text{Wave}})$
12. Insert in $\mathcal{Q}$ new events for new flat nodes on hole west edge $\mathcal{T}_{\text{UpNew}}$ and $\mathcal{T}_{\text{LowNew}}$

Algorithm 10. Inner Wave event processing.

4.5. Virtual Spike Events of the Voronoi Diagram. The Virtual Spike event occurs when a bisector intersects a virtual bisector. If q_{VSp} is a Virtual Spike event, $\mathscr{b}v_1$ and $\mathscr{b}v_2$ are intersecting bisectors, $p_{\text{VSp}} = \mathscr{b}v_1 \cap \mathscr{b}v_2$ is the intersection point, then the priority of the Virtual Spike event is:

$$Priority(q_{\text{VSp}}) = x_{\text{VSp}} + d(p_{\text{VSp}}, R_{\text{VSp}}),$$

where R_{VSp} is a common support edge owner if the bisectors have such, or the only support edge owner if any bisector has such, or a hole edge owner which induces an inward or outside virtual bisector.

Figure 9 illustrates examples of every virtual intersection event types.

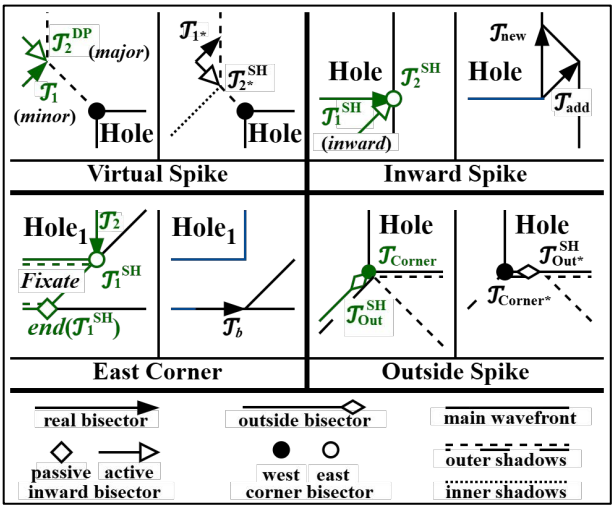

Fig. 9. Examples of intersection events with virtual bisectors

The intersection events are specified at the insertion moment. Figure 10 describes a specification of intersection events. The type of a virtual corner bisector can be defined by abscissa equality with the corresponding edge of the hole. Label "X" is shown for the pairs that should never have an intersection event.

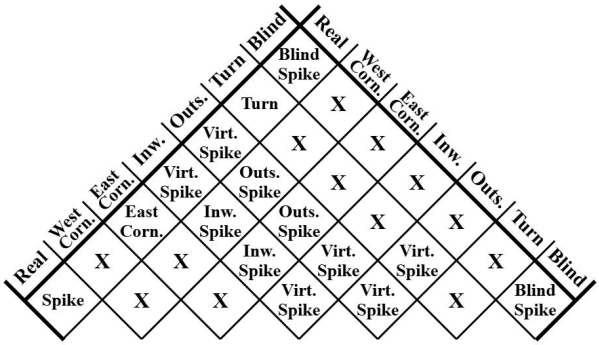

Fig. 10. Specification of intersection events between different types of bisectors

Algorithm 11 defines the Virtual Spike event. If the first bisector represents a duplicate node, then the second bisector exits the shadow, relocating both of them. Else one enters the shadow of the other. Both bisectors must not be removed or the event should be ignored.

Algorithm 12 defines the first special case with intersection between outside and corner virtual bisectors. If the outside bisector moves along the east edge, then the algorithm relocates it in the inward direction from the vertex including the corner and its paired inward bisector to the shadow. If the outside bisector reaches the east corner by the horizontal edge, then it exits the shadow, includes the corner and its paired shadow node into the shadow and relocates this corner pair in the inward direction. Else the outside bisector includes the corner node into the shadow and redirects itself along the edge. Let us call it an Outside Spike event.

Input $q = \mathcal{Q}.pop()$, $\mathcal{L} = Priority(q)$, $type(q)$ = Virtual Spike, $\mathcal{T}_1$, $\mathcal{T}_2$: $\mathcal{T}_2 = next(\mathcal{T}_1)$, p_{VSp}
1. check if $\mathcal{T}_1$ or $\mathcal{T}_2$ are removed then ignore this event
2. set ($\mathcal{T}_{major}$, $\mathcal{T}_{minor}$) as ($\mathcal{T}_2$, $\mathcal{T}_1$) if ($type(\mathcal{T}_2)$ = DP or $type(\mathcal{T}_1)$ = F) or vice versa otherwise
3. set all $\mathcal{T}_{Front}$, $\mathcal{T}_{from}$, $\mathcal{T}_{into}$ as ***Front***$(\mathcal{T}_{major})$, set all p_{minor^*}, p_{major^*} as p_{VSp}, $t \leftarrow type(\mathcal{T}_{minor})$
4. **if** $type(\mathcal{T}_{major})$ = DP **then** ▷ $\mathcal{T}_{minor}$ exits shadow $\mathcal{T}_{major}$
5. $\mathcal{T}_{major}^{SH} \leftarrow main(\mathcal{T}_{major}^{DP})$, $\mathcal{T}_{from} \leftarrow basis(\mathcal{T}_{major}^{SH})$, $p_{minor^*} \leftarrow p_{VSp}$, $p_{major^*} \leftarrow p_{VSp}$
6. **if** $\mathcal{T}_{major} = \mathcal{T}_2$ **then** ▷ points for $\mathcal{T}_{major}$ relocation
7. $p_{inside} \leftarrow ep(prev(\mathcal{T}_{major}^{SH}))$, $p_{neigh} \leftarrow ep(prev(\mathcal{T}_{minor}))$
8. **else** $p_{inside} \leftarrow ep(next(\mathcal{T}_{major}^{SH}))$, $p_{neigh} \leftarrow ep(next(\mathcal{T}_{minor}))$

9. set p_{major^*} as p_{inside} if $p_{inside} \in \mathcal{B}v_{insd}(\mathcal{T}_{major}^{SH})$
10. set p_{major^*} as p_{neigh} if ($p_{neigh} \in \mathcal{B}v_{insd}(\mathcal{T}_{major}^{SH})$ and $x_{neigh} < x_{major^*}$)
11. **if** t = SH **then** ▷ need to relocate $\mathcal{T}_{minor}$
12. **if** $\mathcal{T}_{major} = \mathcal{T}_2$ **then**
13. $p_{inside} \leftarrow ep(next(\mathcal{T}_{major}^{SH}))$, $p_{neigh} \leftarrow ep(next(dupl(\mathcal{T}_{minor}^{SH}))$
14. **else** $p_{inside} \leftarrow ep(prev(\mathcal{T}_{major}^{SH}))$, $p_{neigh} \leftarrow ep(prev(dupl(\mathcal{T}_{minor}^{SH}))$
15. set p_{minor^*} as p_{inside} if $p_{inside} \in \mathcal{B}v_{insd}(\mathcal{T}_{minor}^{SH})$
16. set p_{minor^*} as p_{neigh} if ($p_{neigh} \in \mathcal{B}v_{insd}(\mathcal{T}_{minor}^{SH})$ and $x_{neigh} < x_{minor^*}$)
17. **else** $\mathcal{T}_{into} \leftarrow basis(\mathcal{T}_{major}^{SH})$, $\mathcal{B}v_{major^*} \leftarrow$ ***Relocate***$(\mathcal{T}_{Front}, \mathcal{T}_{major}^{SH}, p_{VSp})$
18. **if** t = SH **then** ▷ find relocated $\mathcal{T}_{minor}$
19. $\mathcal{B}v_{minor^*} \leftarrow$ ***Relocate***$(\mathcal{T}_{into}, \mathcal{T}_{minor}^{SH}, p_{minor^*})$, ***Remove***$(basis(\mathcal{T}_{minor}^{SH}), dupl(\mathcal{T}_{minor}))$
20. **else** $\mathcal{B}v_{minor^*} \leftarrow$ ***Bv***$(owners(\mathcal{T}_{minor}))$
21. ***Remove***$(\mathcal{T}_{from}, \mathcal{T}_{minor})$, ***Remove***$(basis(\mathcal{T}_{major}^{SH}), \mathcal{T}_{major}^{DP})$, ***Remove***$(\mathcal{T}_{Front}, \mathcal{T}_{major}^{SH})$
22. $\mathcal{T}_{minor^*} \leftarrow$ ***Insert***$_t(\mathcal{T}_{into}, \mathcal{B}v_{minor^*})$
23. set $\mathcal{B}v_{major^*}$ as ***Relocate***$(\mathcal{T}_{Front}, \mathcal{T}_{major}^{SH}, p_{major^*})$ if $type(\mathcal{T}_{major})$ = DP
24. $\mathcal{T}_{major^*}^{SH} \leftarrow$ ***Insert***$_{SH}(\mathcal{T}_{from}, \mathcal{B}v_{major^*})$
25. Insert in $\mathcal{Q}$ new events for $\mathcal{T}_{major^*}^{SH}$ and its duplicate, $\mathcal{T}_{minor^*}$ and its duplicate if isn't flat

Algorithm 11. Virtual Spike event processing

Input $q = \mathcal{Q}.pop()$, $\mathcal{L} = Priority(q)$, $type(q)$ = Outside Spike, $\mathcal{T}_1$, $\mathcal{T}_2$: $\mathcal{T}_2 = next(\mathcal{T}_1)$, p_{VSp}
1. check if $\mathcal{T}_1$ or $\mathcal{T}_2$ are removed then ignore this event
2. **if** $type(\beta v_1)$ = C **then**
3. $\mathcal{T}_{Out}^{SH} \leftarrow \mathcal{T}_2^{SH}$, $\mathcal{T}_{Corner} \leftarrow \mathcal{T}_1$, $\beta v_{Out} \leftarrow bisector(\mathcal{T}_2)$, $\beta v_{Corner} \leftarrow bisector(\mathcal{T}_1)$, $f \leftarrow -1$
4. **else** $\mathcal{T}_{Out}^{SH} \leftarrow \mathcal{T}_1^{SH}$, $\mathcal{T}_{Corner} \leftarrow \mathcal{T}_2$, $\beta v_{Out} \leftarrow bisector(\mathcal{T}_1)$, $\beta v_{Corner} = bisector(\mathcal{T}_2)$, $f \leftarrow 1$
5. $Hole_{k1}^{u1} \in owners(\beta v_{Out})$: $Hole_{k1}^{u1} \in Hole_{k1}$ ▷ hole edge inducing outside bisector
6. **if** $type(\mathcal{T}_{Corner})$ = DP **then** ▷ east corner by horizontal edge case
7. $\mathcal{T}_{Corner}^{SH} \leftarrow main(\mathcal{T}_{Corner}^{DP})$, $\mathcal{T}_{End}^{SH} \leftarrow end(\mathcal{T}_{Corner}^{SH})$, $\mathcal{T}_{Front} \leftarrow$ ***Front***$(\mathcal{T}_1)$
8. $(\mathcal{T}_{End^*}^{SH}, \mathcal{T}_{Out}^{SH}, \mathcal{T}^{btwn}) \leftarrow$ ***Swap***$(\mathcal{T}_{Front}, dupl(\mathcal{T}_{End}^{SH}), \mathcal{T}_{Out}^{SH})$
9. $\beta v_{Out^*} \leftarrow$ ***Create***$_{Out}(\mathcal{T}_{Front}, Hole_{k1}^{u1}, ep(\mathcal{T}_{Corner}), f)$
10. ***Remove***$(basis(\mathcal{T}_{Corner}^{SH}), dupl(\mathcal{T}_{Corner}^{SH}))$, ***Remove***$(\mathcal{T}_{Front}, \mathcal{T}_{Corner}^{SH})$
11. $\mathcal{T}_{Corner^*}^{SH} \leftarrow$ ***Insert***$_{SH}(end(\mathcal{T}_{Out}^{SH}), \beta v_{Corner})$
12. ***Remove***$(basis(\mathcal{T}_{Out}^{SH}), dupl(\mathcal{T}_{Out}^{SH}))$, ***Remove***$(\mathcal{T}_{Front}, \mathcal{T}_{Out}^{SH})$
13. $\mathcal{T}_{Out^*}^{SH} \leftarrow$ ***Insert***$_{SH}(\mathcal{T}_{Front}, \beta v_{Out^*})$
14. **else if** $type(\mathcal{T}_{Corner}) \neq$ SH **then** ▷ west corner case
15. $\mathcal{T}_{Front} \leftarrow$ ***Front***$(\mathcal{T}_1)$, $\beta v_{Out^*} =$ ***Create***$_{Out}(\mathcal{T}_{Front}, Hole_{k1}^{u1}, ep(\mathcal{T}_{Corner}), f)$
16. ***Remove***$(\mathcal{T}_{Front}, \mathcal{T}_{Corner})$, $\mathcal{T}_{Corner^*} =$ ***Insert***$_F(end(\mathcal{T}_{Out}^{SH}), \beta v_{Corner})$
17. ***Remove***$(basis(\mathcal{T}_{Out}^{SH}), dupl(\mathcal{T}_{Out}^{SH}))$, ***Remove***$(\mathcal{T}_{Front}, \mathcal{T}_{Out}^{SH})$
18. $\mathcal{T}_{Out^*}^{SH} \leftarrow Insert_{SH}(\mathcal{T}_{Front}, \beta v_{Out^*})$
19. ▷ east corner by east edge case
20. **else** $\mathcal{T}_{End}^{SH} \leftarrow end(\mathcal{T}_{Corner}^{SH})$, $\mathcal{T}_{End}^{DP} \leftarrow dupl(\mathcal{T}_{End}^{SH})$, $\mathcal{T}_{Front} \leftarrow$ ***Front***$(\mathcal{T}_1)$
21. ***Remove***$(basis(\mathcal{T}_{Out}^{SH}), dupl(\mathcal{T}_{Out}^{SH}))$, ***Remove***$(\mathcal{T}_{Front}, \mathcal{T}_{Out}^{SH})$
22. set $\mathcal{T}_{neigh}$ as $next(\mathcal{T}_{End}^{DP})$ if $type(\beta v_1)$ = C or as $prev(\mathcal{T}_{End}^{DP})$ otherwise
23. ***Remove***$(basis(\mathcal{T}_{Corner}^{SH}), dupl(\mathcal{T}_{Corner}^{SH}))$, ***Remove***$(\mathcal{T}_{Front}, \mathcal{T}_{Corner}^{SH})$
24. $\mathcal{T}_{Corner^*}^{SH} \leftarrow$ ***Insert***$_{SH}(end(\mathcal{T}_{Out}^{SH}), \beta v_{Corner})$
25. **if** $type(\mathcal{T}_{neigh})$ = SH **then** ▷ if intersects another shadow

26. $(\mathcal{T}_{\text{End}}^{\text{SH}}, \mathcal{T}_{\text{neigh*}}^{\text{SH}}, \mathcal{T}^{\text{btwn}}) =$ ***Swap***$(\mathcal{T}, \mathcal{T}_{\text{End}}^{\text{DP}}, \mathcal{T}_{\text{neigh}}^{\text{SH}})$, $\mathcal{T}_{\text{Front}} = basis(\mathcal{T}_{\text{neigh*}}^{\text{SH}})$
27. $\mathcal{T}_{\text{Out*}}^{\text{SH}} =$ ***Inject***$(\mathcal{T}_{\text{Front}}, \mathcal{T}_{\text{Out}}^{\text{SH}}, ep(\mathcal{T}_{\text{End}}^{\text{SH}}))$
28. Insert in $\mathcal{Q}$ new events for all created and relocated nodes, including duplicates;

Algorithm 12. Outside Spike event processing

Algorithm 13 defines the second special case of inward and corner virtual bisectors. For the east corner, the algorithm removes the shadow and creates new real bisectors from the east vertex of the hole. Else the algorithm creates a real bisector along the horizontal edge and a blind node along the west edge. Both happen only if the corner virtual bisector does not lie on the slab edge. Let us call this special case an Inward Spike event.

Input $q = \mathcal{Q}.pop()$, $\mathcal{L} = Priority(q)$, $type(q) =$ Inward Spike, $\mathcal{T}_1, \mathcal{T}_2$: $\mathcal{T}_2 = next(\mathcal{T}_1)$, p_{VSp}
1. check if $\mathcal{T}_1$ or $\mathcal{T}_2$ are removed then ignore this event
2. **if** $type(bisector(\mathcal{T}_1)) =$ C **then**
3. $\quad \mathcal{T}_{\text{inw}}^{\text{SH}} \leftarrow \mathcal{T}_2^{\text{SH}}$, $\mathcal{T}_{\text{Corner}} \leftarrow \mathcal{T}_1$, $p_{\text{neigh}} \leftarrow ep(next(\mathcal{T}_2))$
4. **else** $\mathcal{T}_{\text{inw}}^{\text{SH}} \leftarrow \mathcal{T}_1^{\text{SH}}$, $\mathcal{T}_{\text{Corner}} \leftarrow \mathcal{T}_2$, $p_{\text{neigh}} \leftarrow ep(prev(\mathcal{T}_1))$
5. check if corner virtual bisector lies on the slab edge then ignore this event
6. $(Hole^{u1}, Hole^{u2}, T^{u3}) \leftarrow owners(\mathcal{T}_{\text{inw}}^{\text{SH}})$, $\mathcal{T}_{\text{Front}} \leftarrow$ ***Front***$(\mathcal{T}_1)$, ***Remove***$(\mathcal{T}_{\text{inw}}^{\text{SH}}, dupl(\mathcal{T}_{\text{inw}}^{\text{SH}}))$
7. set $Hole^{\text{ed}}$ as $Hole^{u2}$ if ***Form***$(\mathcal{T}_{\text{inw}}^{\text{SH}}) =$ Inner or as $Hole^{u1}$ otherwise
8. **if** $Form(\mathcal{T}_{\text{inw}}^{\text{SH}}) =$ Inner **then**
9. $\quad \mathcal{T}_w \leftarrow$ ***Insert***$_{\text{BL}}(end(\mathcal{T}_{\text{inw}}^{\text{SH}})$, ***Bv***$(T^{u3}, Hole^{u1}))$, $end(\mathcal{T}_w^{\text{BL}}) \leftarrow end(\mathcal{T}_{\text{inw}}^{\text{SH}})$
10. **else** remove flat node on the edge and both duplicates from $basis(\mathcal{T}_{\text{inw}}^{\text{SH}})$
11. ***Remove***$(\mathcal{T}_{\text{Front}}, \mathcal{T}_1)$, ***Remove***$(\mathcal{T}_{\text{Front}}, \mathcal{T}_2)$, $\mathcal{T}_{\text{new}} \leftarrow$ ***Insert***$_{\text{F}}(\mathcal{T}$, ***Bv***$(T^{u3}, Hole^{\text{ed}}))$
12. **if** (***Form***$(\mathcal{T}_{\text{inw}}^{\text{SH}}) =$ Inner **and** $x_{\text{neigh}} < x_{\text{VSp}}$) **or** (***Form***$(\mathcal{T}_{\text{inw}}^{\text{SH}}) =$ Outer **and** $x_{\text{neigh}} \le x_{\text{VSp}}$) **then**
13. $\quad \mathcal{T}_{\text{add}} \leftarrow$ ***Insert***$_{\text{F}}(\mathcal{T}_{\text{Front}}, Bv(T^{u3}, p_{\text{VSp}}))$ ▷ additional bisector as in (13)
14. Insert in $\mathcal{Q}$ new events for all created flat and blind nodes

Algorithm 13. Inward Spike event processing

Algorithm 14 defines the third special case with intersection of a real bisector along the hole's east edge and its corner. The algorithm fixates the shadow and creates a new flat node using the intersection of the bisector in the inward direction from the corner. Let us call this special case an East Corner event.

Input $q = \mathcal{Q}.pop()$, $\mathcal{L} = Priority(q)$, $type(q) =$ East Corner, $\mathcal{T}_1, \mathcal{T}_2$: $\mathcal{T}_2 = next(\mathcal{T}_1)$, p_{VSp}
1. check if $\mathcal{T}_1$ or $\mathcal{T}_2$ are removed then ignore this event
2. $\mathcal{T}_{\text{Front}} \leftarrow$ ***Front***$(\mathcal{T}_1)$, set $T_{\text{fix}} \in T$ such that $T_{\text{fix}}^u \in owners(\mathcal{T}_1) \cup owners(\mathcal{T}_2)$
3. **if** $type(\mathcal{T}_1) =$ SH **then**
4. $\quad$ ***Fixate***$(\mathcal{T}_{\text{Front}}, ep(\mathcal{T}_1),$ ***Bv***$_{\text{insd}}(end(\mathcal{T}_1^{\text{SH}})), T_{\text{fix}}, 1)$ ▷ south east corner case
5. **else** ***Fixate***$(\mathcal{T}_{\text{Front}},$ ***Bv***$_{\text{insd}}(end(\mathcal{T}_2^{\text{SH}})), ep(\mathcal{T}_2), T_{\text{fix}}, 1)$ ▷ north east corner case

Algorithm 14. East Corner event processing

4.6. Time complexity of the events and proposed algorithm. Each support creates 4 nodes and each hole creates 14 nodes. Therefore the total count of nodes is $n = 4 \cdot N + 14 \cdot K$; thus, the time complexity of the proposed algorithm $O(n)$ equals $O(N + K)$. Table 4 shows the time complexity of a single execution for each event type and the estimation for the count of events of each type happening in the proposed algorithm.

Table 4. Time complexity and number of events execution

Event	Purpose	Key operation	Single exec.	Count
Initial	Initialize the wavefront	Insert border, turn bisectors; initiate supports, holes events	$O((N+K)\cdot \log(N+K))$	$=1$
West Edge	Process a new support	Fixate nodes in wavefront; insert new real bisectors	$O(K^2 \cdot N \cdot \log(N+K) + K^3 \cdot \log(N+K)$	$=N$
East Corner	Intersection of a real bisector and hole's east vertex	Fixate nodes in wavefront; insert new real bisectors		$\leq 2 \cdot K$
Blind Spike	Intersection with a blind node	Remove nodes; recreate blind node along hole edge	$O((N+K)\cdot \log(N+K))$	$\leq 2 \cdot N + 2 \cdot K$
Begin Hole	Process a new hole	Create virtual bisectors; enclose wavefront nodes in new shadows; relocate intersected shadow nodes	$O((N+K)\cdot \log(N+K))$	$=K$
End Hole	End hole's processing	Transform shadow nodes of hole's outer shadows	$O((N+K)\cdot \log(N+K))$	$=K$
Outside Spike	Intersection of an outside bisector and hole's vertex	Relocate outside bisector, related shadow node of vertex	$(N+K)\cdot \log(N+K)$	$\leq 2 \cdot K$
Inner Wave	Process wavefront reaching hole's west edge	Fixate nodes reaching west edge; create new bisectors along the edge	$(N+K)\cdot \log(N+K)$	$<2 \cdot N \cdot K$
East Edge	End support's processing	Remove bisectors related to edges; create new from east vertices	$\log(N+K)$	$=N$
Spike	Intersection of real bisectors	Remove bisectors; create a new one	$\log(N+K)$	$\leq 2 \cdot N + 2 \cdot K$
Turn	Intersection of a real and a turn bisectors	Remove bisectors; create a new real bisector by slab's horizontal edge	$\log(N+K)$	≤ 2
Virtual Spike	Intersection with a virtual bisector	Remove bisectors; recreate both bisectors on needed shadow levels	$\log(N+K)$	$\leq 2 \cdot N + 2 \cdot K$
Inward Spike	Intersection of an inward bisector and its hole's vertex	Remove bisectors; create a real and an additional bisectors from vertex	$\log(N+K)$	$\leq 4 \cdot K$

Then the worst time complexity of the complete proposed algorithm is the time complexity of the fixation events $O((K^2 \cdot N^2 + K^3 \cdot N + K^4) \cdot \log(N+K))$, where N is the count of supports, K is the count of holes.

The proposed events cover most of the holes processing cases. The proposed shadow structure maintains holes in the EP&DTL algorithm without violating its integrity. The time consumption of the algorithm highly depends on the shadow hierarchy in the diagram and its concatenation and mutual relations. Therefore, the time complexity, which determines the practical applicability of the proposed algorithm will be analyzed in the computational experiment.

5. Computational Experiments. This section examines the time consumption of the proposed algorithm. Additionally, the last subsection contains a preliminary analysis of the correlation estimation between some heuristic parameters of the obtained Voronoi cells and evaluated deformation values and a discussion of the current limitations.

5.1. Time consumption experiment. The first experiment tests generated plans, grouped by the count of supports, for time consumption. Every group consists of 1000 generated plans. Table 5 describes the settings of the plans generator. The dimensions of supports and holes are selected to have a linear dependency on the count of supports and the slab area. The difference between maximum dimensions of holes correlates with the ratio of the potential maximum sum of areas of holes to the slab area. Algorithm 15 describes the generating process following the geometric assumptions.

Support dimensions depend on their direction, while the width is constant. Tests use C# .NET Framework 4.8 in Release mode and run on an Intel(R) Core(TM) i7-12650H @ 2.30GHz with 16 GB of RAM.

The experiment compares execution time for generated plans with different count of holes: 0%, 8%, 20%, 32%, 44% and 56% of the number of supports. The proposed algorithm involves the EP&DTL algorithm from [20] as a reimplementation in C#. Additionally, the experiment compares time consumption with the previous algorithm used for deformation comparative analysis, namely, span determination algorithm [12].

Table 5. Setting parameters for generating support-hole plans (in m)

Count of supports N	Slab dimensions $L\times W$	Dimensions of supports		Dimensions of holes					
		Width mT	Max length MT	Min mH	Max MH by sets with K as % of N				
					8%	20%	32%	44%	56%
25	20×20	0.25	5	1	9	5	3.5	3	2.25
50	30×30	0.25	5.75	1	9.5	5.5	4	3.5	2.75
75	40×40	0.25	6.5	1	10	6	4.5	4	3.25
100	50×50	0.25	7.25	1	10.5	6.5	5	4.5	3.75
125	60×60	0.25	8	1	11	7	5.5	5	4.25
150	70×70	0.25	8.75	1	11.5	7.5	6	5.5	4.75
175	80×80	0.25	9.5	1	12	8	6.5	6	5.25
200	90×90	0.25	10.25	1	12.5	8.5	7	6.5	5.75

Input *Slab*, *N*, *K*, (*MT*, *mT*) – support dimensions, (*MH*, *mH*) – hole dimensions, $Rect \leftarrow \{\varnothing\}$

1. **while** $|Rect| < N$ **do** ▷ supports generating
2. set $d_{new} \in [mT, MT]$ by random with step mT, set $f_{new} \in \{0, 1\}$ by random
3. set (l_{new}, w_{new}) as (d_{new}, mT) if $f_{new} = 0$ or (l_{new}, w_{new}) as (mT, d_{new}) otherwise
4. set $x_{new} \in [0, L - l_{new}]$ and $y_{new} \in [0, W - w_{new}]$ by random with step TW
5. $T_{new} \leftarrow <(x_{new}, y_{new}), (l_{new}, w_{new})>$, $T^* \leftarrow \{T_{new}\}$ ▷ new generated rectangle
6. check if any support from *Rect* contains T_{new} or is contained by T_{new} then **Continue**
7. **while** $\exists T_i \in Rect, \exists T' \in T^*: T_i \cap T' \neq \varnothing$ **and** $(\nexists T^u = T_i \cap T': T^u \in T_i, T^u \in T')$ **do**

split T_i by T' into set T_i^*, remove T_i from *Rect*, add rectangles from T_i^* to *Rect*
split T' by T_i without intersection $T_i \cap T'$ into set of rectangles T^{**}
remove T' from T^*, add all rectangles from T^{**} to T^*
add every rectangle of T^* to *Rect* as supports
remove a random support from *Rect* until $|Rect| > N$
while $|Rect| < N + K$ **do** ▷ holes generating
set $l_{hole} \in [mH, MH]$ and $w_{hole} \in [mH, MH]$ by random with step *TW*
set $x_{hole} \in [0, L - l_{hole}]$ and $y_{hole} \in [0, W - w_{hole}]$ by random with step *TW*
$Hole_{new} \leftarrow <(x_{hole}, y_{hole}), (l_{hole}, w_{hole})>$ ▷ new generated rectangle
check if any rectangle from *Rect* intersects with $Hole_{new}$ then **Continue**
return *Rect* ▷ set of supports and holes following geometric assumptions

Algorithm 15. Generator of experimental support plans with holes

The used version of the proposed algorithm does not consider the alignment problem mentioned in Section 3.2. The Voronoi diagrams built in the experiment violate the uniqueness restriction from Figure 1 of supports with west alignment. The approach of flipping the abscissa around the slab center is impractical due to the time consumptions and limitations.

The experiment runs each test three times per plan. Table 6 shows the average time and standard deviation for EP&DTL without holes, the proposed Voronoi diagram algorithm, and span determination algorithm from previous research [12] with minimum span area $min_S = 2$ m^2. Table 7 shows the ratio between average times for algorithms ("+" Voronoi faster, "–" Span faster). Table 8 shows the peak memory usage.

For 25 supports the proposed algorithm executes from 1.06 times faster without holes to 1.75 times slower with 56% holes than the span algorithm. For 50 supports the proposed algorithm executes from 1.04 times faster (for plans with 32% holes) to 2.36 times faster (for plans without holes) than the span algorithm, but to 1.07 times slower for plans with more holes. The proposed algorithm executes from 1.33 times faster (for 75 supports with 56% holes) to 15.17 times faster (for 200 supports without holes) than the span algorithm. The standard deviation of the time consumptions for the proposed algorithm is lower than for the span algorithm for every case. Holes accelerate the span algorithm to some extent, because the number of the decomposition is decreasing; holes slow down the Voronoi diagram algorithm as they create a more complex shadow structure. The memory usage has a near linear dependency on the number of supports and holes.

Table 6. Average time (in ms) and standard deviation of time consumptions

N	**25**		**50**		**75**		**100**	
K	**Voronoi**	Span	**Voronoi**	Span	**Voronoi**	Span	**Voronoi**	Span
0%	3.35	3.54	7.32	17.3	11.9	47.1	16.4	96.7
	.0002	.0010	.0003	.0037	.0004	.0086	.0005	.0164
8%	4.62	3.68	10.5	16.4	17.5	41.7	24.6	83.5
	.0004	.0011	.0008	.0036	.0009	.0073	.0023	.0137
20%	6.33	4.33	15.36	17.6	23.7	44.4	35.0	85.1
	.0005	.0012	.0009	.0035	.0016	.0073	.0024	.0121
32%	8.05	5.27	19.4	20.3	32.0	50.1	45.1	96.7
	.0005	.0014	.0011	.0037	.0022	.0072	.0031	.0123
44%	9.63	6.04	23.9	22.5	40.2	53.9	56.3	103
	.0007	.0014	.0014	.0039	.0027	.0074	.0034	.0125
56%	12.91	7.38	29.1	27.2	47.6	63.3	69.0	118
	.0008	.0016	.0020	.0045	.0033	.0084	.0045	.0134
N	**125**		**150**		**175**		**200**	
K	**Voronoi**	Span	**Voronoi**	Span	**Voronoi**	Span	**Voronoi**	Span
0%	20.8	169	26.5	267	31.1	400	37.2	565
	.0007	.0263	.0008	.0409	0.0008	.0617	.0009	.0812
8%	33.4	143	40.9	225	49.5	330	58.2	474
	.0026	.0224	.0032	.0330	.0034	.0495	.0039	.0702
20%	45.5	149	55.9	229	69.3	337	80.7	480
	.0031	.0205	.0034	.0302	.0042	.0438	.0047	.0617
32%	59.7	165	73.2	256	88.6	371	101	524
	.0033	.0209	0.0037	.0309	.0055	.0462	.0062	.0607
44%	73.8	173	90.7	263	106	401	123	552
	.0042	.0193	.0047	.0283	.0063	.0499	.0066	.0560
56%	88.3	197	106	300	126	448	147	616
	.0054	.0211	.0058	.0310	.0069	.0467	.0072	.0558

Table 7. Average time ratio comparison of the Voronoi and Span algorithms

	25	**50**	**75**	**100**	**125**	**150**	**175**	**200**
0%	+1.06	+2.36	+3.97	+5.88	+8.14	+10.1	+12.8	+15.2
8%	–1.26	+1.56	+2.38	+3.39	+4.29	+5.49	+6.66	+8.15
20%	–1.46	+1.15	+1.87	+2.43	+3.27	+4.11	+4.86	+5.94
32%	–1.53	+1.04	+1.57	+2.14	+2.77	+3.50	+4.19	+5.18
44%	–1.59	–1.06	+1.34	+1.83	+2.34	+2.90	+3.79	+4.50
56%	–1.75	–1.07	+1.33	+1.71	+2.23	+2.83	+3.56	+4.20

Table 8. Peak memory usage of the proposed algorithm for every set (in MB)

	25	**50**	**75**	**100**	**125**	**150**	**175**	**200**
0%	1,88	3,71	5,60	7,53	9,55	11,61	13,67	15,64
8%	2,49	4,71	7,26	9,60	12,24	14,82	17,33	20,02
20%	3,39	6,48	9,40	12,92	16,49	19,37	22,81	26,55
32%	3,92	7,88	12,07	15,98	20,35	24,41	28,65	33,05
44%	4,83	9,46	14,21	19,45	24,08	29,16	34,50	39,18
56%	5,57	11,13	16,64	22,50	28,31	34,43	39,88	46,09

The proposed algorithm can accelerate the comparative deformation process for a larger number of elements, while for the plans with fewer supports and more holes the span determination algorithm executes faster.

Figure 11 illustrates statistical values of the time consumption for the proposed algorithm. The gap of time consumption increases when the number of elements increases. The proposed holes processing always executes slower on average in comparison to without holes, even for the best case with holes and the worst case without them, except for the pair 25-0% and 25-8%. But the paired t-test for cases with and without holes showed that the difference in time execution cannot be explained only by the count of holes. Therefore, to understand the nature of the complexity brought by holes processing, several additional tests have taken place.

Figure 12 shows the average percent of plan processing time spent on event types. Spike event type also contains Blind Spike events. If some event type does not happen during the processing of some plan, the averaging function does not count it. Figure 13 shows the average time spent on one event processing. The plots do not show Inner Wave and Turn events, because they have small time consumptions: the Inner Wave event accounts for 0.96% and 0.024 ms, the Turn event accounts for 0.10% and 0.02 ms on average across all plans.

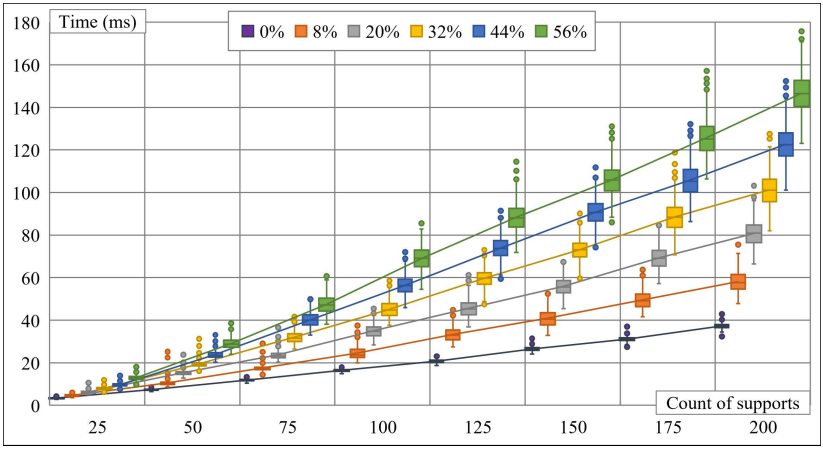

Fig. 11. Box plots for time consumption of the EP&DTL and proposed algorithms

The most time-consuming event by the average ratio to the whole process time is the West Edge event, taking from 25% to 50% of execution time. As the hole ratio grows, the ratio of hole events increases: from 17% for 8% plans to 52% for 56% plans. The most time-consuming hole event is the Begin Hole event. The average percentage of execution time for each event type does not vary by more than 1% with respect to the number of supports.

The average time for one execution of any type of event increases when the number of supports or holes increases. Among the average single execution times, the most time-consuming is the Begin Hole event. Four more events execute on average longer than 0.07 ms: West Edge, End Hole, East Corner and Outside Spike events. But East Corner and Outside Spike events take on average less than 4% and 3% of general time for any group. Therefore, let us focus on the research of the Begin Hole, West Edge and End Hole events for the further algorithm acceleration.

Figure 14 shows the average time spent on one function processing. Symbol "*" indicates the complex functions that include other simple functions. Function "Insert node" refers to the node insertion $Insert_{type}(\mathcal{T}, \not{b}v)$; function "Insert shadow" refers to the shadow insertion $Insert(\mathcal{T}, \beta v_1, \beta v_2)$; function "Fixate between" refers to the full execution of Algorithm 6. Functions with negligible execution times are not shown in the plot; the average single execution times for them across all plans are: Relocate is 0.0012 ms; Voronoi from Algorithm 2 is 0.0027 ms; Inserting a new event into $\mathcal{Q}$ is 0.0021 ms; Front is 0.0003 ms.

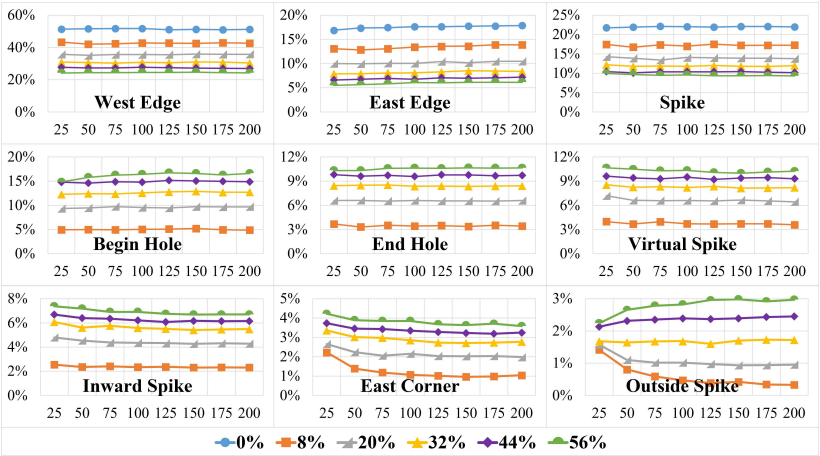

Fig. 12. Plots for processing rate of events; abscissa axis – count of supports; ordinate axis – average percent of processing time; series – ratio of holes number

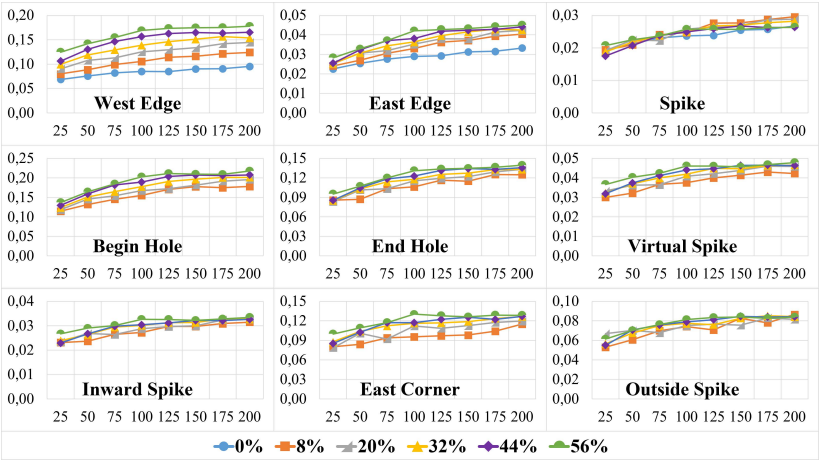

Fig. 13. Plots for time consumption of events; abscissa axis – count of supports; ordinate axis – average time of single execution in ms; series – ratio of holes number

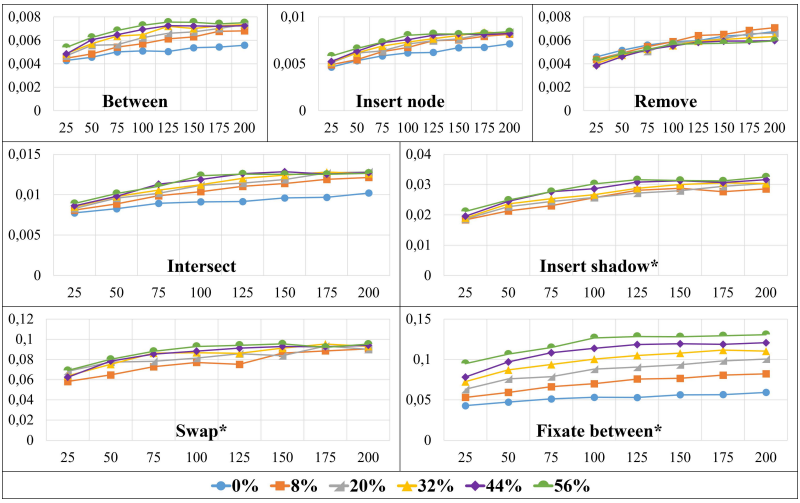

Fig. 14. Plots for time consumption of functions; abscissa axis – count of supports; ordinate axis – average time of single execution in ms; series – ratio of holes number

The most time-consuming function on average is the Fixate between from Algorithm 6. Function Swap executes slower for the 8% hole set for almost every number of supports. For other cases Fixate executes slower and has a clear correlation between the execution time and the number of elements, supports and holes; other functions do not have such an obvious

dependency. For all functions time consumption grows with the number of holes increasing, except Remove. Time consumption of Remove function on average decreases when the number of holes increases. This behavior is caused by the shadow structure: the number of simple removes from small shadow trees increases compared to removes from a single big tree.

Figure 15 shows the percentage of the time spent on operations in the execution of West Edge, Begin Hole and End Hole events. Operation "Voronoi" refers to the Algorithm 2, operation "Insert event" refers to inserting a new event into Q.

The most time-consuming operation in big events is Insert. Insert and Remove operations spend around 40% of West Edge, 45% of Begin Hole and 70% of End Hole events. The percentage of the Remove operation increases when the number of elements increases. For the Begin Hole and End Hole events the percentage of the Insert operation increases when the number of holes increases. For the West Edge event, the percentage of the Insert operation increases when the number of supports increases and the number of holes decreases, because not every shadow during fixation of the wavefront produces new nodes.

Though Insert and Remove operations account for most of the execution time in big events, they do not have much optimization potential, as they are already well-optimized operations of a standard AVL-tree.

Between and Relocate operations do not account for much execution time of big events. The Intersect operation in West Edge and Begin Hole events does not seem to be optimizable, as it uses a common mechanism of the AVL tree and may become less time-consuming only with another structure of the shadows. The Voronoi operation from Algorithm 2 accounts for 15% of West Edge event time, but it has a low average execution time of 0.0027 ms. The insertion of a new event into Q accounts for around 5% of West Edge, 17% of Begin Hole and 25% of End Hole events, though the average time of a single operation is very low (around 0.0021 ms). The main reason is the insertion of new events for reinserted nodes from the between sets during Begin Hole and End Hole events (steps 6, 12, 19, 20 in Algorithm 7; step 9 in Algorithm 8). Then a possible optimization can lie in searching for new events only for the first and last nodes in these sets, reassigning the existing events for every other node.

Besides, the experimental program implements the priority queue Q as a list with insertion complexity O(N), N is the number of events in Q, and extraction complexity O(1). The binary heap implementation with insertion and extraction complexity O(logN) can optimize this operation.

Thus, the main optimization to decrease the time consumption of the proposed algorithm lies in a better implementation of the event

creation operation, especially during the processing of the between sets. Other optimizations can lie in restructuring the shadow trees.

The experiment showed that the proposed algorithm is faster than the previous span determination algorithm for plans with more supports and fewer holes: 1.06 times faster for plans with 25 supports without holes and 1.75 times slower with 14 holes (56% of supports); 2.36 times faster with 50 supports without holes and 1.07 times slower with 28 holes; with 75 supports or more, the algorithm is from 1.33 times faster (for 75 supports and 42 holes) to 15.2 times faster (for 200 supports without holes). Thus, the proposed algorithm extends the scope of the deformation comparative analysis for plans with a large number of supports.

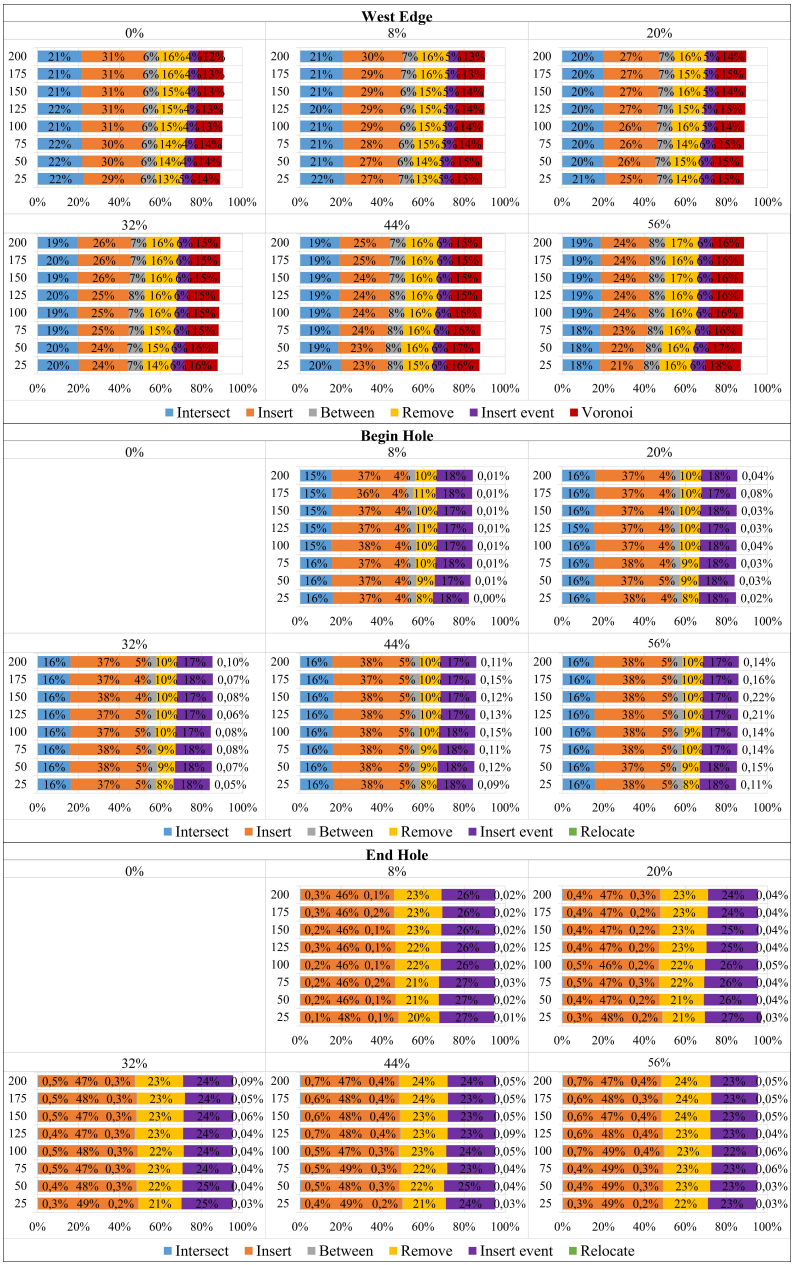

Fig. 15. Functions percentages in processing of the biggest events

5.2. Correlation analysis preliminary experiment. This section describes a preliminary test of the Voronoi diagram usage for deformation analysis. The experiment estimates correlations between the Voronoi cells and deformation, evaluated by the finite element (FE) method, in the slab. The aim of the experiment is to give a first check of the algorithm in the deformation comparative analysis and set a direction for future research.

Table 9 describes the sets of deformation metrics, which were chosen for the preliminary correlation analysis, Voronoi parameters and simple parameters of the supports for comparison.

Table 9. Voronoi cells parameters, simple support parameters and deformation metrics for preliminary correlation analysis

Deformation metrics	
Mx	The average value of the bending moment perpendicular to the X axis
My	The average value of the bending moment perpendicular to the Y axis
Z	The difference between extreme vertical deflections of vertices
Reinf	The amount of reinforcement needed in every finite element
Voronoi parameters	
Area	Area of the Voronoi cell
P	Perimeter of the Voronoi cell
CD	Distance between center of the Voronoi cell and center of the support
MD	Maximum distance between the support and edges of the Voronoi cell
AD	Average distance between the support and edges of the Voronoi cell
LX	The maximum difference in vertices abscissa of the Voronoi cell
LY	The maximum difference in vertices ordinate of the Voronoi cell
Simple support parameters	
SL	Length of the support
SDE	Distance from the support edge to the edge of the slab
SDS	Distance to the closest support
SDH	Distance to the closest hole
SLS	Density of supports in the local area around the support (relatively to local area)
SLH	Density of holes in the local area around the support (relatively to local area)

To handle the alignment problem mentioned in Section 3.2, the experiment applies the approach of flipping the abscissa approach with some secondary processing mechanisms. The following modifications define the approach:

1) To process simultaneous West Edge and East Corner events, we use a Fixation event instead; instead of the original events, we create Fixation events; each Fixation event has a flag indicating the original event type and executes as the original would; Fixation events have the same order position among events between East Corner and Begin Hole events;

2) The approach builds the Voronoi diagram twice: first with the abscissa of the elements flipped about the slab center, second with the original abscissa; after the second diagram is built, cells get edges from the

flipped copy, that do not appear in the original diagram; this yields the correct Voronoi edges according to the uniqueness rule from Section 3.2 for aligned supports;

3) A Fixation event does not produce Voronoi edges between nodes during Fixate operations (Algorithm 6: steps 12, 13 and 27) for aligned supports; thus, both diagrams do not have incorrect edges;

4) Every Voronoi edge should have the maximum possible length, meaning both original and flipped diagrams have only necessary vertices;

5) If there are supports with east alignment that have a Voronoi edge with supports that have west alignment on the east side of their cells, then disjoint vertices appear; each of them should be matched with the closest pair among other disjoint vertices in the cell with the same abscissa;

6) The Fixation operation can create bisectors with incorrect assignment of the owner for aligned supports in the processing of inner shadows (Fig. 7c, Algorithm 6: steps 24 and 25, T_i); therefore, there should be a check operation that will give the correct owner for these assignments in the case of support alignment during fixation; it can be done by creating polygonal zones for every valid Fixation event, such as $Zone = <\{p_1, p_2, \ldots, p_{NZ}\}, T_{Zone}>$ by set of points, NZ is the count of points in the polygon, and the support $T_{Zone} \in T$.

Algorithm 16 describes an operation to find zones of assignment before the first simultaneous Fixation event.

To clip a new zone by a simple convex polygon, the Sutherland–Hodgman algorithm can be used. To clip the previous zones by the new one, the new zone can be split into convex parts. During the fixation operation we need to find a zone that contains the beginning point for every new bisector (Algorithm 6: steps 25, 32 and 34), for example by the ray casting algorithm, and set the corresponding support as an owner in the *Create* function for it. Then the described approach will give the correct ownership for every created bisector, even for simultaneous Fixation events inside the inner shadows.

The flipping abscissa approach handles the alignment problem, but is time-consuming and should be used only for experimental purposes. A more convenient and practical approach is a subject of future research.

The FE mesh is overlaid onto the Voronoi diagram. If some finite element or vertex of the mesh lies inside several Voronoi cells, we apply decision rules. For vertices:

- find the Voronoi cell with the closest support to the vertex;
- if there are several such cells, then find the Voronoi cell with the closest centroid to the vertex;
- if there are several such cells, then the vertex is ignored.

For finite elements:

- find the Voronoi cell that contains the most vertices of the finite element;
- if there are several such cells, then find the Voronoi cell with the closest centroid to the center of the finite element;
- if there are several such cells, then the finite element is ignored.

Input $\mathcal{T} \neq \varnothing$, $q = Q.pop()$, $\mathcal{L} = Priority(q)$, $type(q) = $ Fixation
1. $Q^*_{Sim} \leftarrow \{q' \in Q \mid Priority(q') = \mathcal{L}, type(q') = \text{Fixation}\}$ ▷ simultaneous fixation events
2. order Q^*_{Sim} by ascending y, then by descending x
3. $Holes^* \leftarrow \{Hole_k \in Holes \mid x_k < \mathcal{L}\}$ ▷ holes on the left of sweep line
4. order $Holes^*$ by descending x, then by ascending y
5. $q_{Last} = \varnothing$, $Zones \leftarrow \{\varnothing\}$ ▷ polygonal zones of assignment
6. **for** $q_{New} \in Q^*_{Sim}$ **do**
7. set T_{New} as support of q_{New} event, set y_{New} as ordinate of T_{New}, set w_{New} as width of T_{New}
8. set $Zone_{Last}$ as $Zones$.Last() if $Zones \neq \varnothing$
9. **if** q_{New} is West Edge Fixation **then**
10. $Zone_{New} \leftarrow <\{(0, y_{New} + w_{New} + \mathcal{L}), (\mathcal{L}, y_{New} + w_{New}), (\mathcal{L}, y_{New}), (0, y_{New} - \mathcal{L})\}, T_{New}>$
11. **if** $q_{Last} \neq \varnothing$ **and** q_{Last} is West Edge Fixation **and** $Zone_{Last} \cap Zone_{New} \neq \varnothing$ **then**
12. ▷ find the rightest point of zones intersection
13. set $p_{Inter} \in Zone_{Last} \cap Zone_{New}$: $\nexists p' \in Zone_{Last} \cap Zone_{New}$, $p' \neq p_{Inter}$, $x' > x_{Inter}$
14. $Cut_{New} \leftarrow \{(0, -\mathcal{L}), (0, y_{Inter}), (\mathcal{L}, y_{Inter}), (\mathcal{L}, -\mathcal{L})\}$ ▷ cutting area for new zone
15. $Zone_{New} \leftarrow Zone_{New} \setminus Cut_{New}$ ▷ cutting zone by mid line
16. **else** set $Hole_k$ as hole of Q_{New} event, set $v_k{}^e$ as corresponded east corner of hole
17. set (f, w) as $(-1, 0)$ if $v_k{}^e$ is south corner or as $(1, w_k)$ otherwise
18. $Zone_{New} \leftarrow <\{v_k{}^e, (x_k, y_k + w), (0, y_k + w - f \cdot x_k), (0, y_k + w + f \cdot (x_k + l_k))\}, T_{New}>$
19. **for** $Hole' \in Holes^*$ **do**
20. check if west edge of $Hole'$ is passed by wavefront by $Inner = \varnothing$ then **Continue**
21. check if $Zone_{New} \cap Hole' = \varnothing$ then **Continue**
22. $v'^s \leftarrow (x' + l', y')$, $v'^n \leftarrow (x' + l', y' + w')$ ▷ east corners of $Hole'$
23. **if** $v'^s \notin Zone_{New}$ **and** $v'^n \notin Zone_{New}$ **then** ▷ zone is locked by hole
24. $Cut_{Hole} \leftarrow \{(0, -\mathcal{L}), (0, W + \mathcal{L}), (x' + l', W + \mathcal{L}), (x' + l', -\mathcal{L})\}$
25. $Zone_{New} \leftarrow Zone_{New} \setminus Cut_{Hole}$, **Break**
26. **else if** $v'^s \in Zone_{New}$ **and** $v'^n \in Zone_{New}$ **then** ▷ zone includes hole
27. find q'^s and q'^n in Q^*_{Sim} with East Corner flag related to v'^s and v'^n if exist
28. remove zones from $Zones$ created by q'^s and q'^n if exist
29. remove q'^s and q'^n from Q^*_{Sim} if exist
30. **else if** $v'^s \in Zone_{New}$ **then**
31. $Zone_{New} \leftarrow Zone_{New} \setminus \{v'^s, (x', y'), (0, y' + x'), (0, W + \mathcal{L}), (x' + l', W + \mathcal{L})\}$
32. find q'^s in Q^*_{Sim} with East Corner flag related to v'^s if exists
33. remove zone from $Zones$ created by q'^s, remove q'^s from Q^*_{Sim} if exists
34. **else** $Zone_{New} \leftarrow Zone_{New} \setminus \{v'^n, (x', y' + w'), (0, y' - x'), (0, -\mathcal{L}), (x' + l', -\mathcal{L})\}$
35. **if** $\exists Zone' \in Zones$: $v'^n \in Zone'$ **then** ▷ need to cut by mid of hole
36. $Zone_{New} \leftarrow Zone_{New} \setminus \{(x', y' + w'/2), (0, y' + w'/2), (0, -\mathcal{L}), (x', -\mathcal{L})\}$

find q'^n in Q^*_{Sim} with East Corner flag related to v'^n if exists
remove zone from *Zones* created by q'^n, remove q'^s from Q^*_{Sim} if exists
$\forall Zone' \in Zones: Zone' \leftarrow Zone' \setminus Zone_{\text{New}}$
return *Zones* ▷ assignment zones intersecting only by edges

Algorithm 16. Creation of assignment zones for inner shadows fixation

The deformations are evaluated using LIRA-FEM on a plane slab; the mesh step is 125 mm. The slab settings: width is 250 mm; elastic modulus is 3e+006 t/m^2; Poison's ratio is 0.2; specific gravity is 2.502 t/m^3 (B25). The reinforcement setting: A500 for longitudinal reinforcement and A240 for crosswise reinforcement. The slab is modeled by the plate finite elements. The slab contains uniformly distributed orthogonal loads equal to 0.6255 t/m^2. Supports are modeled as restraints along the axis orthogonal to the slab plane (Z axis). The technical setup is the same as in the previous experiment.

The experiment uses 250 plans with 25 supports and with hole ratios from 0% to 56% from the previous experiment, for a total of 1500 plans, computing correlations for a set of the constructed Voronoi cells. Thus, the sample size is 37500 Voronoi cells in total, with 6250 cells per hole percentage group. The local offset for SLS and SLH simple parameters equals 0.5 m; the local area is a rectangle with edges formed by offset from the support edges.

Table 10 and Table 11 show paired Pearson's correlation coefficients between Voronoi and support simple parameters and deformation metrics. The correlation values for SDH and SLH do not include plans without holes; therefore, their sample size is 31250 instead of 37500. To counteract the multiple comparisons problem, the experiment applies the Bonferroni correction with a significance level of 0.01 for 52 tests; the adjusted significance level is approximately 0.00019. The tables include p-value and confidence interval for coefficients using the corrected significance level.

The result of the experiment shows the existence of a correlation between the parameters of Voronoi diagram and deformation values. The bending moments (Mx, My) have moderate (>.3) correlation strength with the maximum distance to Voronoi vertex parameter (MD) and weak (>.1) with others, except the pairs LX-Average Mx and LY-Average My, that have negligible correlation coefficients (<0.1). This effect can be explained by the bending moment Mx acting around the x-axis; consequently, the larger the dimension of the slab in the orthogonal direction along the y-axis, the larger the bending moment Mx will be. The same reasoning applies to My. The range of the vertical deflection (Z) has strong (>.5) correlation coefficients with the area of the Voronoi cell (Area), distance between the center of the Voronoi cell and the center of the support (CD), maximum and average distance to Voronoi vertices from the support (MD, AD) and

moderate correlation coefficients with others. The required reinforcement amount (Reinf) has a strong correlation with all Voronoi parameters, the strongest being with Area: 0.763.

Table 10. Correlations between Voronoi parameters and deformation metrics

	Average Mx	**Average My**	**Max – min Z**	**Sum Reinf**
Area	*r* = .195 *p* = .00000 CI [.176 .213]	*r* = .231 *p* = .00000 CI [.213 .249]	*r* = .516 *p* = .00000 CI [.502 .530]	*r* = .763 *p* = .00000 CI [.755 .771]
P	*r* = .187 *p* = .00000 CI [.168 .205]	*r* = .211 *p* = .00000 CI [.193 .229]	*r* = .482 *p* = .00000 CI [.467 .496]	*r* = .652 *p* = .00000 CI [.641 .663]
CD	*r* = .241 *p* = .00000 CI [.223 .259]	*r* = .263 *p* = .00000 CI [.245 .281]	*r* = .581 *p* = .00000 CI [.568 .593]	*r* = .550 *p* = .00000 CI [.537 .563]
MD	*r* = .334 *p* = .00000 CI [.317 .351]	*r* = .346 *p* = .00000 CI [.329 .363]	*r* = .682 *p* = .00000 CI [.671 .692]	*r* = .717 *p* = .00000 CI [.707 .726]
AD	*r* = .285 *p* = .00000 CI [.267 .303]	*r* = .305 *p* = .00000 CI [.288 .323]	*r* = .613 *p* = .00000 CI [.601 .625]	*r* = .718 *p* = .00000 CI [.708 .727]
LX	*r* = .039 *p* = .00000 CI [.020 .058]	*r* = .294 *p* = .00000 CI [.277 .312]	*r* = .404 *p* = .00000 CI [.388 .420]	*r* = .565 *p* = .00000 CI [.552 .578]
LY	*r* = .292 *p* = .00000 CI [.275 .310]	*r* = .078 *p* = .00000 CI [.059 .097]	*r* = .435 *p* = .00000 CI [.419 .450]	*r* = .571 *p* = .00000 CI [.558 .584]

Table 11. Correlations between simple support parameters and deformation metrics

	Average Mx	**Average My**	**Max – min Z**	**Sum Reinf**
SL	*r* = –.003 *p* = .53049 CI [–.022 .016]	*r* = .002 *p* = .75768 CI [–.018 .021]	*r* = .096 *p* = .00000 CI [.077 .115]	*r* = .254 *p* = .00000 CI [.236 .272]
SDE	*r* = –.021 *p* = .00004 CI [–.041 –.002]	*r* = –.025 *p* = .00000 CI [–.044 –.006]	*r* = .015 *p* = .00424 CI [–.004 .034]	*r* = .010 *p* = .04674 CI [–.009 .030]
SDS	*r* = –.018 *p* = .00068 CI [–.037 .002]	*r* = .015 *p* = .00466 CI [–.005 .034]	*r* = .137 *p* = .00000 CI [.118 .156]	*r* = .273 *p* = .00000 CI [.255 .291]
SDH	*r* = –.031 *p* = .00000 CI [–.052 –.010]	*r* = –.009 *p* = .11890 CI [–.030 .012]	*r* = –.061 *p* = .00000 CI [–.082 –.040]	*r* = –.068 *p* = .00000 CI [–.089 –.047]
SLS	*r* = –.111 *p* = .00000 CI [–.130 –.092]	*r* = –.101 *p* = .00000 CI [–.120 –.082]	*r* = –.147 *p* = .00000 CI [–.165 –.128]	*r* = –.221 *p* = .00000 CI [–.239 –.203]
SLH	*r* = .056 *p* = .00000 CI [.035 .077]	*r* = .033 *p* = .00000 CI [.012 .054]	*r* = .098 *p* = .00000 CI [.077 .118]	*r* = .084 *p* = .00000 CI [.063 .105]

Only SLS and SLH simple support parameters have significant correlation coefficients with each deformation metric, while the others have some insignificant coefficients. The parameter of distance to the edge of the slab (SDE) has only one significant but negligible coefficient with Average

My and has insignificant coefficients with others; thus, it seems that this parameter does not bring any important information to the analysis. The parameters of distance to the closest hole (SDH) and the density of holes in the local area (SLH) have significant but negligible coefficients, except of the pair SDH-Average My, meaning these simple heuristics are not enough to describe deformations.

The length of the support parameter (SL) has a significant weak correlation with reinforcement needed, a significant but negligible correlation with deflection difference, and insignificant coefficients with other deformation metrics. The parameter of distance to the closest support (SDS) has a significant weak correlation coefficient with the difference of Z deflections and the needed reinforcement amount, and it has insignificant coefficients with the average bending moments. The parameter of the density of supports in the local area (SLS) has significant weak correlation coefficients with all deformation metrics.

The following conclusions are made based on the correlation analysis results:

1) Area and perimeter (P) of Voronoi cells have an obvious correlation with deformations in the slab: the bigger Voronoi cell leads to bigger spans with neighboring supports, which leads to greater deformation in the slab;

2) The distance between the center of the Voronoi cell and the center of the support (CD) describes the imbalance of the supporting area shape in the slab, leading to bigger bending moments and deflections;

3) The maximum and average distance to the Voronoi cell vertices (MD and AD) describe the length of the maximum and average span from the support around it; the larger the span length, the greater the deformations occur;

4) The bounding box dimensions in abscissa and ordinate of the Voronoi cell (LX and LY) should explain the bending moments, because the larger the extent of the supporting area, the larger the bending moments should be; but it seems that the distance to the Voronoi vertices from the support (MD, AD) describes deformations of the slab better;

5) Some simple parameters, such as length of the support (SL), distance to the closest support (SDS) and density of supports in the local area (SLS) have a significant weak correlation and can be used in deeper analysis, but for any deformation metric all of them have less correlation power than the Voronoi parameters.

In general, the Voronoi diagram seems to be a useful heuristic approach to evaluate slab deformation, because its nature reflected in the parameters of the Voronoi cells has a significant and valuable correlation

with the important deformation metrics. Future research will use the proposed algorithm and the obtained results of the preliminary correlation analysis as a starting point. Besides, the limitations of the described model, such as the single value for dimensions and thickness of the model slab, the single type of concrete and reinforcement tested, should be researched further as well to confirm the applicability of the Voronoi diagram to a wider range of floor slab and building design variations.

6. Conclusion. A new Voronoi diagram algorithm with rectangular sites and holes inside has been proposed. The new algorithm uses the algorithm of E. Papadopoulou and D.T. Lee [10] (EP&DTL) as the base and can process the rectangular sites fast enough by implementing the L_∞ distance metric and straightforward sweep line events. Therefore, the Voronoi diagram can be applied to the slab deformation analysis problem as part of the comparative deformation analysis research, as in [19] and [20].

The modification of the EP&DTL algorithm with rectangular holes inside the Voronoi diagram, which correspond to the holes in the floor slab, has been described. The new structure in the height-balanced binary tree representing the wavefront called "shadows" restricts the intersection operation during the sweep line algorithm. New "virtual" bisectors induced by holes maintain "shadow" structures, while the new events corresponding to them allow handling holes during the Voronoi algorithm.

The computational experiment of comparing time consumption against the previous span determination approach [19] has demonstrated that the proposed algorithm executes from 1.33 times faster for the plans with 75 supports and 42 holes to 15.17 times faster for the plans with 200 supports without holes. But the proposed algorithm executes slower for smaller cases, especially with more holes, up to 1.75 times slower for 25 supports and 14 holes. The extended analysis of the time consumption for events and functions has shown some potential optimizations.

The preliminary correlation analysis of the Voronoi cell parameters has shown a significant strong correlation between the reinforcement needed and the area of the cell (correlation coefficient 0.76) and the distance to the vertices (correlation coefficient 0.72). However, these findings should be interpreted with caution due to the simplified assumptions of the model, while the algorithm has a limitation in processing support alignment. Therefore, future research will propose a solution for the alignment problem and will use the obtained results to extend the analysis to build a practical slab deformation heuristic evaluation using the Voronoi diagram approach.

References

1. Hadi M.-H., Sharafi P., Teh L.-H. A new formulation for the geometric layout optimisation of flat slab floor systems. Proceeding of the Australasian Structural Engineering Conference: The Past, Present and Future of Structural Engineering. Perth, Western Australia: Engineers Australia. 2012. pp. 122–129. DOI: 10.3316/informit.026275045488938.
2. Petprakob W. Beam-slab floor optimization using genetic and particle swarm optimization algorithms. A Thesis for degree of master of science in engineering and technology. Thailand: Thammasat University, Sirindhorn International Institute of Technology. 2014. 90 p. DOI: 10.14457/TU.the.2014.490.
3. Beeby A.-W., Narayanan R.-S. Designers' Guide to EN 1992-1-1 and EN 1992-1-2. Eurocode 2: Design of Concrete Structures Design of Concrete Structures. General Rules and Rules for Buildings and Structural Fire Design. London: Thomas Telford Publishing, 2005. 230 p. DOI: 10.1680/dgte2docs.31050.
4. Sahab M.-G., Ashour A.-F., Toropov V.-V. Cost optimisation of reinforced concrete flat slab buildings. Engineering Structures. 2005. vol. 27. no. 3. pp. 313–322. DOI: 10.1016/j.engstruct.2004.1.002.
5. Sharafi P. Cost optimization of the preliminary design layout of reinforced concrete framed buildings. A Thesis for degree of Doctor of Philosophy in civil engineering. Australia: University of Wollongong, School of Civil, Mining and Environmental Engineering. 2013. 286 p.
6. Meng X., Lee T.-U., Xiong Y., Huang X., Xie Y.-M. Optimizing support locations in the roof – column structural system. Applied Science. 2021. vol. 11. no. 6. 2775 p. DOI: 10.3390/app11062775.
7. Zelickman Y., Amir O. Optimization of column layouts in buildings considering structural and architectural constraints. Engineering Archive (engrXiv). 2024. 45 p. DOI: 10.31224/2723.
8. Leshkevich O. [The use of artificial neural networks to evaluate the reinforcement of reinforced concrete floor slabs. Problems of modern concrete and reinforced concrete: Collection of scientific papers no. 11.]. Minsk: In-t BelNIIS, 2019. pp. 51–62. DOI: 10.35579/2076-6033-2019-11-04. (In Russ.)
9. Wang J., Chen K., Yang H., Zhang L. Ensemble deep learning enabled multi-condition generative design of aerial building machine considering uncertainties. Automation in Construction. 2024. vol. 157. DOI: 10.1016/j.autcon.2023.105134.
10. Liao W., Lu X., Fei Y., Gu Y., Huang Y. Generative AI design for building structures. Automation in Construction. 2024. vol. 157. DOI: 10.1016/j.autcon.2023.105187.
11. Steiner B., Mousavian E., Mehdizadeh F.-S., Wimmer M., Musialski P. Integrated structural-architectural design for interactive planning. Computer Graphics Forum. 2016. vol. 36. no. 8. pp. 1–13. DOI: 10.1111/cgf.12996.
12. Zinov V., Kartak V., Valiakhmetova Yu. [Algorithm for assessing the deformation of floor slabs based on the building's span-support schemes]. Sistemy analiza i obrabotki dannykh – Analysis and Data Processing Systems. 2023. vol. 92. no. 4. pp. 35–54. DOI: 1.17212/2782-2001-2023-4-35-54. (In Russ.).
13. Zinov V., Kartak V., Valiakhmetova Yu. [Solving a multi-criteria problem of rational placement of load-bearing walls using a genetic algorithm]. Informatika i avtomatizatsiya – Informatics and Automation. 2025. vol. 24. no. 2. pp. 464–491. DOI: 1.15622/ia.24.2.4. (In Russ.).
14. Jung C., Redenbach C. Crack modeling via minimum-weight surfaces in 3d Voronoi diagrams. Journal of Mathematics in Industry. 2023. vol. 13. no. 10. DOI: 10.1186/s13362-023-00138-1.
15. Polat H., Ilerisoy Z.-Y. A geometric method on facade form design with Voronoi Diagram. Modular. 2020. vol. 3. no. 2. pp. 179–194.

16. Rokicki W., Gawell E. Voronoi diagrams – architectural and structural rod structure research model optimization. Malowsze Studia Regoinalne. 2016. vol. 19. pp. 155–164. DOI: 10.21858/msr.19.10.
17. Fortune S. A sweepline algorithm for Voronoi diagrams. Algorithmica. 1987. vol. 2. pp. 153–174. DOI: 10.1007/BF01840357.
18. Bhattacharya P., Gavrilova M.-L., et al. Voronoi diagram in optimal path planning. Proceeding of the 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD). IEEE. 2007. pp. 38–47. DOI: 10.1109/ISVD.2007.43.
19. Ho Y.-J., Liu J.-S. Smoothing Voronoi-based obstacle-avoiding path by length-minimizing composite Bezier curve. Proceeding of the International Conference on Service and Interactive Robotics (SIRCon09). Institute of Information Science: Academia Sinica. 2009.
20. Papadopoulou E., Lee D.-T. Critical area computation via Voronoi Diagrams. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 1999. vol. 18. no. 4. pp. 463–474. DOI: 10.1109/43.752929.

Zinov Vladislav — Postgraduate student, Department of Technical Cybernetics, Federal State Budgetary Educational Institution of Higher Education «Ufa University of Science and Technology». Research interests: optimization in the field of building design, heuristic and metaheuristic methods, and decision support systems. The number of publications — 8. zinovvladislavufa@gmail.com; 32, Zaki Validi St., 450076, Ufa, Russia; office phone: +7(917)435-5560.

УДК 519.25, 624.042 **DOI** 10.15622/ia.25.2.9

В.И. Зинов

ОБРАБОТКА ОТВЕРСТИЙ В ДИАГРАММЕ ВОРОНОГО С ПРЯМОУГОЛЬНЫМИ ОСНОВАНИЯМИ ДЛЯ СРАВНИТЕЛЬНОГО АНАЛИЗА ДЕФОРМАЦИЙ В ПЛИТАХ

Зинов В.И. **Обработка отверстий в диаграмме Вороного с прямоугольными основаниями для сравнительного анализа деформаций в плитах.**

Аннотация. В данной статье предложен новый алгоритм построения диаграммы Вороного на прямоугольных основаниях с отверстиями. Алгоритм основан на алгоритме построения диаграммы Вороного с метрикой расстояния L_∞, разработанным Papadopoulou E. и Lee D.-T. Предложены модификации для обработки отверстий в модели диаграммы Вороного. Алгоритм обрабатывает искажения в структуре диаграммы, вызванные отверстиями, создавая слои береговой линии при построении диаграммы Вороного, названные тенями, и используя новый тип биссекторов, которые не задают ребер на диаграмме Вороного, но являются основой для структуры слоев береговой линии. Алгоритм задает новые события для заметающей прямой, сохраняя общие принципы обработки согласно базовому алгоритму. По результатам сравнения времени выполнения с предыдущим подходом определения пролетов в плите, предложенный алгоритм работает в 1.33 раза быстрее для плана с 75 опорами и в 15.17 раз быстрее для наибольшего протестированного количества опор, но медленнее для меньшего количества опор и большего числа отверстий. Предварительный корреляционный анализ показал наличие значимой линейной корреляции с коэффициентом 0.76 между площадью ячейки Вороного и требуемым количеством армирования, а также сильную и умеренную корреляцию между другими параметрами ячейки и метриками деформаций. В заключении указаны текущие ограничения модели и алгоритма, которые будут исследованы в дальнейшем.

Ключевые слова: диаграмма Вороного, сравнительная оценка деформаций, рациональное размещение опор, корреляционная модель, оптимизация проектирования зданий.

Литература

1. Hadi M.-H., Sharafi P., Teh L.-H. A new formulation for the geometric layout optimisation of flat slab floor systems // Proceeding of the Australasian Structural Engineering Conference: The Past, Present and Future of Structural Engineering. Perth, Western Australia: Engineers Australia. 2012. pp. 122–129. DOI: 10.3316/informit.026275045488938.
2. Petprakob W. Beam-slab floor optimization using genetic and particle swarm optimization algorithms // A Thesis for degree of master of science in engineering and technology. Thailand: Thammasat University, Sirindhorn International Institute of Technology. 2014. 90 p. DOI: 10.14457/TU.the.2014.490.
3. Beeby A.-W., Narayanan R.-S. Designers' Guide to EN 1992-1-1 and EN 1992-1-2. Eurocode 2: Design of Concrete Structures Design of Concrete Structures. General Rules and Rules for Buildings and Structural Fire Design // London: Thomas Telford Publishing, 2005. 230 p. DOI: 10.1680/dgte2docs.31050.
4. Sahab M.-G., Ashour A.-F., Toropov V.-V. Cost optimisation of reinforced concrete flat slab buildings // Engineering Structures. 2005. vol. 27. no. 3. pp. 313–322. DOI: 10.1016/j.engstruct.2004.1.002.

5. Sharafi P. Cost optimization of the preliminary design layout of reinforced concrete framed buildings // A Thesis for degree of Doctor of Philosophy in civil engineering. Australia: University of Wollongong, School of Civil, Mining and Environmental Engineering. 2013. 286 p.
6. Meng X., Lee T.-U., Xiong Y., Huang X., Xie Y.-M. Optimizing support locations in the roof – column structural system // Applied Science. 2021. vol. 11. no. 6. 2775 p. DOI: 10.3390/app11062775.
7. Zelickman Y., Amir O. Optimization of column layouts in buildings considering structural and architectural constraints // Engineering Archive (engrXiv). 2024. 45 p. DOI: 10.31224/2723.
8. Лешкевич О.Н. Использование искусственных нейронных сетей для оценки армирования железобетонных плит перекрытия. Проблемы современного бетона и железобетона: Сб. научн. тр. № 11. // Минск: Ин-т БелНИИС. 2019. С. 51–62. DOI: 10.35579/2076-6033-2019-11-04.
9. Wang J., Chen K., Yang H., Zhang L. Ensemble deep learning enabled multi-condition generative design of aerial building machine considering uncertainties // Automation in Construction. 2024. vol. 157. DOI: 10.1016/j.autcon.2023.105134.
10. Liao W., Lu X., Fei Y., Gu Y., Huang Y. Generative AI design for building structures // Automation in Construction. 2024. vol. 157. DOI: 10.1016/j.autcon.2023.105187.
11. Steiner B., Mousavian E., Mehdizadeh F.-S., Wimmer M., Musialski P. Integrated structural-architectural design for interactive planning // Computer Graphics Forum. 2016. vol. 36. no. 8. pp. 1–13. DOI: 10.1111/cgf.12996.
12. Зинов В.И., Картак В.М., Валиахметова Ю.И. Алгоритм оценки деформации плит перекрытий по пролетно-опорным схемам здания // Системы анализа и обработки данных. 2023. Т. 92. № 4. С. 35–54. DOI: 10.17212/2782-2001-2023-4-35-54.
13. Зинов В.И., Картак В.М., Валиахметова Ю.И. Решение многокритериальной задачи рационального размещения несущих стен с помощью генетического алгоритма // Информатика и автоматизация. 2025. Т. 24. № 2. С. 464–491. DOI: 10.15622/ia.24.2.4.
14. Jung C., Redenbach C. Crack modeling via minimum-weight surfaces in 3d Voronoi diagrams // Journal of Mathematics in Industry. 2023. vol. 13. no. 10. DOI: 10.1186/s13362-023-00138-1.
15. Polat H., Ilerisoy Z.-Y. A geometric method on facade form design with Voronoi Diagram // Modular. 2020. vol. 3. no. 2. pp. 179–194.
16. Rokicki W., Gawell E. Voronoi diagrams – architectural and structural rod structure research model optimization // Malowsze Studia Regoinalne. 2016. vol. 19. pp. 155–164. DOI: 10.21858/msr.19.10.
17. Fortune S. A sweepline algorithm for Voronoi diagrams // Algorithmica. 1987. vol. 2. pp. 153–174. DOI: 10.1007/BF01840357.
18. Bhattacharya P., Gavrilova M.-L., et al. Voronoi diagram in optimal path planning // Proceeding of the 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD). IEEE. 2007. pp. 38–47. DOI: 10.1109/ISVD.2007.43.
19. Ho Y.-J., Liu J.-S. Smoothing Voronoi-based obstacle-avoiding path by length-minimizing composite Bezier curve // Proceeding of the International Conference on Service and Interactive Robotics (SIRCon09). Institute of Information Science: Academia Sinica. 2009.
20. Papadopoulou E., Lee D.-T. Critical area computation via Voronoi Diagrams // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 1999. vol. 18. no. 4. pp. 463–474. DOI: 10.1109/43.752929.

Зинов Владислав Игоревич — аспирант, кафедра технической кибернетики, Федеральное государственное бюджетное образовательное учреждение высшего образования «Уфимский университет науки и технологий». Область научных интересов: оптимизация в сфере проектирования зданий, эвристические и метаэвристические методы, системы поддержки принятия решений. Число научных публикаций — 8. zinovvladislavufa@gmail.com; улица Заки Валиди, 32, 450076, Уфа, Россия; р.т.: +7(917)435-5560.