Dynamic Well-Spaced Point Sets

Umut A. Acar[a], Andrew Cotter[b,*], Benoît Hudson[c], Duru Türkoğlu[d]

[a]*Carnegie Mellon University, Pittsburgh, PA, 15213, USA*
[b]*Toyota Technological Institute at Chicago, Chicago, IL 60637, USA*
[c]*Autodesk, Inc., Montreal, QC, Canada*
[d]*Department of Computer Science, University of Chicago, Chicago, IL 60637, USA*

Abstract

In a *well-spaced point set* the Voronoi cells all have bounded aspect ratio. Well-spaced point sets satisfy some important geometric properties and yield quality Voronoi or simplicial meshes that are important in scientific computations. In this paper, we consider the dynamic well-spaced point set problem, which requires constructing a well-spaced superset of a dynamically changing input set, e.g., as input points are inserted or deleted. We present a dynamic algorithm that allows inserting/deleting points into/from the input in $O(\log \Delta)$ time, where Δ is the geometric spread, a natural measure that yields an $O(\log n)$ bound when input points are represented by log-size words. We show that this algorithm is time-optimal by proving a lower bound of $\Omega(\log \Delta)$ for a dynamic update. We also show that this algorithm maintains size-optimal outputs: the well-spaced supersets are within a constant factor of the minimum possible size. The asymptotic bounds in our results work in any constant dimensional space. Experiments with a preliminary implementation indicate that dynamic changes may be performed with considerably greater efficiency than re-constructing a well-spaced point set from scratch. To the best of our knowledge, these are the first time- and size-optimal algorithms for dynamically maintaining well-spaced point sets.

Keywords: Well-spaced point set, clipped Voronoi cell, mesh refinement, dynamic stability, self-adjusting computation

*Corresponding author

Email addresses: umut@cs.cmu.edu (Umut A. Acar), cotter@ttic.edu (Andrew Cotter), benoit.hudson@autodesk.com (Benoît Hudson), duru@cs.uchicago.edu (Duru Türkoğlu)

1. Introduction

Given a hypercube B in $\mathbb{R}^d$, we call a set of points $\mathsf{M} \subset B$ *well-spaced* if for each point $p \in \mathsf{M}$ the ratio of the distance to the farthest point of B in the Voronoi cell of p divided by the distance to the nearest neighbor of p in M is small [32]. Well-spaced point sets are strongly related to meshing and triangulation for scientific computing, which require meshes to have certain qualities. In two dimensions, a well-spaced point set induces a Delaunay triangulation with no small angles, which is known to be a good mesh for the finite element method. In higher dimensions, well-spaced point sets can be post-processed to generate good simplicial meshes [8, 20]. The Voronoi diagram of a well-spaced point set is also immediately useful for the control volume method [22].

A well-spaced superset M of a point set N may be constructed by inserting so-called *Steiner* points, although one must take care to insert as few Steiner points as possible. We call the output and such an algorithm *size-optimal* if the size of the output, $|\mathsf{M}|$, is within a constant factor of the size of the smallest possible well-spaced superset of the input. This problem has been studied since the late 1980s (e.g., [6, 10, 26]), with several recent results obtaining fast runtime [14, 16, 31].

We are interested in the dynamic version of the problem, which requires maintaining a well-spaced output (M) while the input (N) changes dynamically due to insertion and deletion of points. Upon a modification to the input, the dynamic algorithm should efficiently update the output, preserving size-optimality with respect to the new input. There has been relatively little progress on solving the dynamic problem. Existing solutions either do not produce size-optimal outputs (e.g., [9, 25]) or they are asymptotically no faster than running a static algorithm from scratch [12, 21, 23].

In this paper, we present a dynamic algorithm for the well-spaced point set problem. Our algorithm always returns size-optimal outputs, and requires worst-case $O(\log \Delta)$ time for an input modification (an insertion or a deletion). Here, Δ is the *geometric spread*, a common measure, defined as the ratio of the diameter of the input set to the distance between the closest pair of points in the input. Our update runtime is optimal in the worst-case and our algorithms consume linear space in the size of the output. If the geometric spread is polynomially bounded in the size of the input, then $\log \Delta = O(\log n)$ (e.g., when the input is specified using $\log n$-bit numbers). For the purposes of our bounds, we assume the dimension of the space, d, to be an arbitrary constant.

To solve the dynamic problem, we first present an efficient construction algorithm for generating size-optimal, well-spaced supersets (algorithm in Sections 5

and 6, proofs in Sections 7 and 8). In addition to the output, the construction algorithm builds a *computation graph* that represents the operations performed during the execution and the dependencies between them. A key property of this algorithm is that it is *stable* in the sense that when run with similar inputs, e.g., that differ in only one point, it produces similar computation graphs and outputs. We make this property precise by describing a *distance* measure between the computation graphs of two executions and bounding this distance by $O(\log \Delta)$ when inputs differ by a single point (Section 9). Taking advantage of this bound, we design a change-propagation algorithm that performs dynamic updates in $O(\log \Delta)$ time by identifying the operations that are affected by the modification to the input and deleting/re-executing them as necessary (Section 10). For the lower bound, we show that there exist inputs and modifications that require $\Omega(\log \Delta)$ Steiner points to be inserted into/deleted from the output (Section 11).

The efficiency of our dynamic update algorithm directly depends on stability. In order to achieve stability, we use several techniques in the design of our construction algorithm. Generalizing the recently suggested choices of Steiner points [14, 18], we propose an approach for picking Steiner points by making local decisions only, using *clipped Voronoi cells*. Picking Steiner points locally makes it possible to structure the computation into $\Theta(\log \Delta)$ *ranks*, inductively ensuring that at the end of each rank the points up to that rank are well-spaced [31]. Processing points in rank order alone does not guarantee stability: we further partition points at a given rank into a constant number of *color* classes such that the points in each color class depend only on the points in the previous color classes. These techniques enable us to process each point only once and help isolate and limit the effects of a modification. Furthermore, our dynamic update algorithm returns an output and a computation graph that are isomorphic to those that would be obtained by re-executing the static algorithm with the modified input (Lemma 10.2). Consequently, the output remains both well-spaced and size-optimal with respect to the modified input (Theorem 10.3).

The approach of designing a stable construction algorithm and then providing a dynamic update algorithm based on change propagation is inspired by recent advances on *self-adjusting computation* (e.g., [2, 3, 13, 19]). In self-adjusting computation, programs can respond automatically to modifications to their data by invoking a change-propagation algorithm [1]. The data structures required by change propagation are constructed automatically. Our computation graphs are abstract representations of these data structures. Similarly our dynamic update algorithms are adaptations of the change-propagation algorithm for the problem of well-spaced point sets. Self-adjusting computation has been found to be ef-

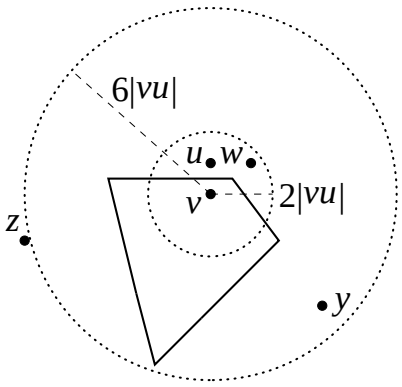


Figure 1: Let $\mathscr{M} = \{v, u, w, y, z\}$. The nearest neighbor distance of v, $\mathrm{NN}_{\mathscr{M}}(v)$, is $|vu|$. The polygon with solid boundary lines depicts the Voronoi cell of v, $\mathrm{Vor}_{\mathscr{M}}(v)$. Vertex v is 6-well-spaced, but not 2-well-spaced.

fective in kinetic motion simulation of three-dimensional convex hulls [3]. Although these initial findings are empirical, they have motivated the approach that we present in this paper. Since our approach takes advantage of the structure of a static algorithm to perform dynamic updates, it can be viewed as a dynamization technique, a technique which has been used effectively for a relatively broad range of algorithms (e.g., [7, 11, 24, 27]).

To assess the effectiveness of the proposed dynamic algorithm, we present a prototype implementation, and report the results of an experimental evaluation (Section 12). Our experimental results confirm our theoretical bounds, and demonstrate that dynamic updates to an existing well-spaced point set can be performed far more cheaply than re-computing from scratch. These results suggest that a well-optimized implementation can perform very well in practice.

This paper is the journal version of the following two abstracts: *An Efficient Query Structure for Mesh Refinement* published in the proceedings of the 20th Annual Canadian Conference on Computational Geometry [17], and *Dynamic Well-Spaced Point Sets* published in the proceedings of the 26th Annual Symposium on Computational Geometry [4].

2. Preliminaries

We present some definitions used throughout the paper, describe the technique that we use for selecting Steiner vertices, and present an overview of the point location data structure we use in our algorithms.

Given a set of points N, we define the *geometric spread* (Δ) to be the ratio of the diameter of N to the distance between the closest pair in N. We say that a d-dimensional hypercube B is a *bounding box* if $\mathsf{N} \subset B$ and each edge of B has length within a constant factor of the diameter of N. Without loss of generality,

we take the bounding box of N to be $B = [0,1]^d$. Given N as input, our algorithm constructs a well-spaced output $\mathsf{M} \subset B$ that is a superset of N. We use the term *point* to refer to any point in B and the term *vertex* to refer to the input and output points. Consider a vertex set $\mathscr{M} \subset B$. The *nearest-neighbor distance of v in $\mathscr{M}$*, written $\mathrm{NN}_{\mathscr{M}}(v)$, is the distance from v to the nearest other vertex in $\mathscr{M}$. The *Voronoi cell* of v in $\mathscr{M}$, written $\mathrm{Vor}_{\mathscr{M}}(v)$, consists of points $x \in B$ such that for all $u \in \mathscr{M}$, $|vx| \leq |ux|$. Following Talmor [32], a vertex v is *ρ-well-spaced* if the intersection of its Voronoi cell with B is contained in the ball of radius $\rho\,\mathrm{NN}_{\mathscr{M}}(v)$ centered at v; $\mathscr{M}$ is ρ-well-spaced if every vertex in $\mathscr{M}$ is ρ-well-spaced. Figure 1 illustrates these definitions.

In order to achieve size-optimality, we ensure that the output that our algorithm constructs is size-conforming [26]. A set of vertices $\mathscr{M} \supset \mathsf{N}$ is *size-conforming* if there exists a constant c independent of N such that for all vertices $v \in \mathscr{M}$, $\mathrm{NN}_{\mathscr{M}}(v) > c \cdot \mathrm{lfs}(v)$, where $\mathrm{lfs}(v)$, the *local feature size* of v, is the distance from v to the second-nearest vertex in N.

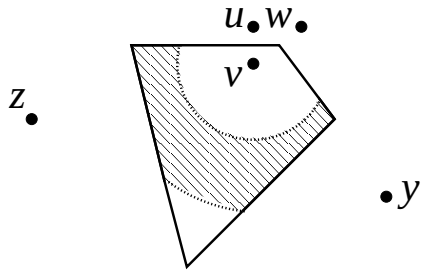
3. Clipped Voronoi Cells

Regardless of runtime considerations, the fundamental question in mesh refinement is about where to insert the Steiner vertices [29]. Traditional solutions place Steiner vertices as far from any other vertex as possible, namely, at the circumcenters of Delaunay simplices (equivalently, at the nodes of the Voronoi diagram). In two dimensions, Har-Peled and Üngör instead place Steiner vertices close to a vertex, but not too close: at the *off-centers* [14]. This local scheme allows them to build a data structure that can locate the off-centers in constant time. More recent work by Jampani and Üngör extends this idea to three dimensions using *core disks* [18]. In this paper, we generalize these two results, by describing a local picking region that gives us a variety of Steiner vertex choices including their proposals. The definition of our picking region straightforwardly extends to arbitrary dimensions. In order to form the basis for our picking region, we begin by defining a local neighborhood of a vertex.

Definition 3.1 (β-clipped Voronoi cell)**.** *Given a vertex $v \in \mathscr{M}$, the β-clipped Voronoi cell of v, written $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$, is the intersection of $\mathrm{Vor}_{\mathscr{M}}(v)$ with the ball of radius $\beta\,\mathrm{NN}_{\mathscr{M}}(v)$ centered at v.*

Selecting two parameters ρ and β, we define the picking region we use in our algorithms. As it will be clearer in Section 6, the parameters $\rho > 1$ and $\beta > \rho$ define the rate of geometric expansion and the degree of locality respectively.

Figure 2: This is the same example as Figure 1. The shaded region displays the $(2,4)$ picking region of v. Vertices y and z are 4-clipped but not 2-clipped Voronoi neighbors of v.



Definition 3.2 ((ρ,β) picking region)**.** *Given a vertex $v \in \mathscr{M}$, the (ρ,β) picking region of v, written $\mathrm{Vor}^{(\rho,\beta)}_{\mathscr{M}}(v)$, is $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v) \setminus \mathrm{Vor}^{\rho}_{\mathscr{M}}(v)$, the region of the Voronoi cell bounded by concentric balls of radius $\rho\,\mathrm{NN}_{\mathscr{M}}(v)$ and $\beta\,\mathrm{NN}_{\mathscr{M}}(v)$.*

The parameter ρ also defines the well-spacedness of the resulting set after the insertion of Steiner vertices: a vertex v is ρ-well-spaced if and only if the (ρ,β) picking region of v is empty, i.e., $\mathrm{Vor}^{\rho}_{\mathscr{M}}(v) = \mathrm{Vor}^{\beta}_{\mathscr{M}}(v) = \mathrm{Vor}_{\mathscr{M}}(v)$.

In order to correctly compute the β-clipped Voronoi cell $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$ of a vertex v, an algorithm must certify that v is the closest vertex to any point inside $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$ and that any point on the boundary of $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$ is either equidistant to v and another vertex or at the clipping distance $\beta\,\mathrm{NN}_{\mathscr{M}}(v)$ from v. In this regard, we introduce the following notion:

Definition 3.3 (β-clipped Voronoi neighbor)**.** *Given any two vertices v and u, u is called to be a β-clipped Voronoi neighbor of v if the β-clipped Voronoi cell of v contains a point equidistant from v and u.*

Using this definition, we reduce the problem of computing β-clipped Voronoi cell of v to computing the set of β-clipped Voronoi neighbors of v. Simply, given the set of β-clipped Voronoi neighbors of v, one can construct its β-clipped Voronoi cell efficiently. Figure 2 illustrates these definitions.

Correct computation of the β-clipped Voronoi neighbors thus the β-clipped Voronoi cell requires certificates for any point $x \in \mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$ that there is an empty ball centered at x containing v on its boundary. More formally, for any such point x, we define the *certificate ball* of x to be the the ball centered at x with radius $|vx|$ and we define the *certificate region* of $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$ as the union of the certificate balls of the points inside $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$. Figure 3 illustrates these definitions. Finally, we conclude that identifying the certificate region and certifying that it is empty of vertices is necessary and sufficient for computing clipped Voronoi cells.

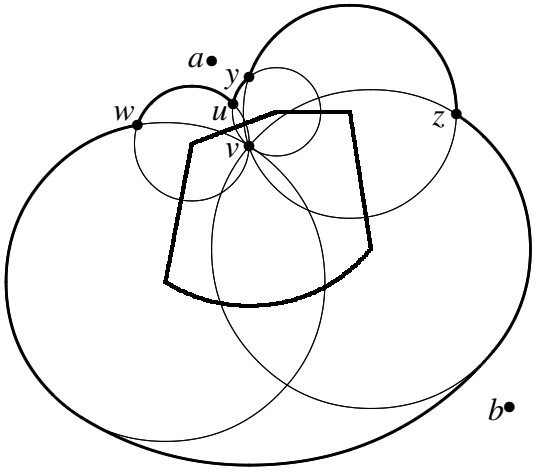


Figure 3: $\mathscr{M} = \{a, b, v, u, w, y, z\}$. $\mathrm{NN}_{\mathscr{M}}(v) = |vu|$. The thick boundary depicts the β-clipped Voronoi cell of v, $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$, for $\beta = 4$. The thin boundary depicts the certificate region of $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$. Vertices a and b are not β-clipped Voronoi neighbors of v since there is no empty certificate ball for a and the vertex b is too far away from v, at distance more than $\beta\,\mathrm{NN}_{\mathscr{M}}(v)$.

4. Dynamic Balanced Quadtrees

To permit the efficient calculation of nearest neighbors and clipped Voronoi cells, we use a point location data structure based on the balanced quadtrees of Bern, Eppstein, and Gilbert [6]. We extend this balanced quadtree to arbitrary (d) dimensions and dynamize it by describing an algorithm for inserting/deleting an input vertex in $O(\log \Delta)$ time.

We use the term *quadtree* to mean a 2^d-tree for which each *quadtree node* corresponds to a d-dimensional hypercube, which is itself partitioned into 2^d equally-sized hypercubes controlled by the node's children. Our well-spaced point set algorithms treat the quadtree data structure almost as a black box; they use only the leaves of the quadtree, which we refer to as *(quadtree) squares*. Our quadtrees are minimal among those that satisfy the *crowding* and the *grading* properties defined as follows:

- Crowding: every quadtree square (leaf node) contains at most one vertex, and if it does, none of its neighbors contains a vertex.
- Grading: all neighbors of any internal node must exist in the quadtree.

Here, we define the *neighbors* of a quadtree node to be the same size nodes in each of the $3^d - 1$ cardinal and diagonal directions. To support fast traversal and access, a quadtree node keeps pointers to its parent, children, and neighbors. Additionally, every square contains a pointer to an input vertex it may contain, as well as a list of all Steiner vertices contained within the square. We define two squares to be *adjacent* if their intersection contains at least one point. Given any two adjacent squares, the grading condition ensures that either they are neighbors or one of them is a neighbor of the parent of the other.

For efficiently locating and inserting Steiner vertices, our quadtree data structure supports the `QTInsertSteiner` function. Given a Steiner vertex w that is inside the (ρ, β) picking region of a vertex v, this function finds the quadtree square that contains w by a traversal starting from the square of v towards w. Since w is inside the β-clipped Voronoi cell of v, this function runs in $O(1)$ time.

For constructing the quadtree, we provide the function `QTBuild`. For inserting/deleting an input vertex $\hat{v}$ into/from a quadtree, we provide the functions `QTInsertInput` and `QTDeleteInput`. Given the original quadtree $\mathscr{Q}$ and the vertex $\hat{v}$, these two functions return the updated quadtree $\mathscr{Q}'$ and the set of obsolete squares of $\mathscr{Q}$. For insertions, the obsolete squares are those that become internal nodes in $\mathscr{Q}'$; for deletions, they are the deleted leaf nodes of $\mathscr{Q}$.

In the rest of this section, we briefly explain the construction and the dynamic modification functions. Also, we state the lemmas summarizing our results, some of which will be useful in the analysis of the dynamic well-spaced point set algorithms. The `QTBuild` function iteratively inserts each input vertex into an empty quadtree using `QTInsertInput`. Given an input vertex $\hat{v}$ to be inserted, `QTInsertInput` first determines the square that contains $\hat{v}$ by performing a top-down traversal of the quadtree. If this square already contains an input vertex, it then splits this square and descends into the child containing $\hat{v}$, repeating as necessary. Finally, it inserts $\hat{v}$ into the resulting (currently empty) square, and in order to restore the quadtree, it imposes the crowding and the grading conditions. We state the following easily-verified facts which characterize the squares that become obsolete as a result of inserting $\hat{v}$.

Fact 1. *During a call to `QTInsertInput` to insert $\hat{v}$, if a quadtree node is split due to crowding, then either that node or one of its neighbors contains $\hat{v}$.*

Fact 2. *During a call to `QTInsertInput`, if a quadtree node is split due to grading, then a descendant of one of its neighbors must have been split due to crowding.*

Lemma 4.1. *For any square $s \in \mathcal{Q}$ that is returned by* `QTInsertInput`$(\mathcal{Q}, \hat{v})$*, we have $|s\hat{v}| \in O(|s|)$, where $|s\hat{v}|$ is the minimum distance between $\hat{v}$ and s, and $|s|$ is the side-length of s.*

Proof. Facts 1 and 2 imply that every split square s is at most a neighbor's neighbor of the quadtree node containing $\hat{v}$ at the same depth as s. Hence, the distance between $\hat{v}$ and s satisfies $|s\hat{v}| \leq 2\sqrt{d}|s|$. □

The `QTDeleteInput` function performs essentially the same steps as the `QTInsertInput` function, in reverse. First, descending through the quadtree, it locates the quadtree square containing $\hat{v}$ and deletes it from the square. Next, motivated by Facts 1 and 2, it checks all ancestors of this square, their neighbors and neighbors' neighbors, all in a bottom-up fashion, merging them if they are no longer crowded, and do not need to be split due to grading. An analogue of Lemma 4.1 holds for `QTDeleteInput`, and the proof follows similarly.

Lemma 4.2. *For any square $s \in \mathcal{Q}$ that is returned by* `QTDeleteInput`$(\mathcal{Q}, \hat{v})$*, we have $|s\hat{v}| \in O(|s|)$.*

Proof. Every square returned by `QTDeleteInput` is a child of a square which would be split by `QTInsertInput`, were the deleted vertex to be re-inserted into the quadtree. The proof of Lemma 4.1 shows that all such squares satisfy the claim, so their children will also. □

By Facts 1 and 2, the functions `QTInsertInput` and `QTDeleteInput` perform a constant number of operations at each level of the quadtree. Additionally, the quadtree depth is bounded by $O(\log \Delta)$, permitting us to state the following theorem:

Theorem 4.3. *The functions* `QTInsertInput` *and* `QTDeleteInput` *insert or delete a single point, and update the quadtree in $O(\log \Delta)$ time. Furthermore,* `QTBuild` *requires $O(n \log \Delta)$ time to construct a quadtree on n input points.*

Proof. Because `QTInsertInput` and `QTDeleteInput` perform a constant number of operations at each of the $O(\log \Delta)$ levels of the quadtree, they require $O(\log \Delta)$ time. Because `QTBuild` consists of n calls to `QTInsertInput`, it requires $O(n \log \Delta)$ time. □

Finally, we prove a lemma demonstrating that the quadtree data structure we describe in this section can be used to approximate the local feature size (lfs) of the points up to a constant factor. By definition of lfs, this implies an approximation of the nearest-neighbor distances of the input vertices.

Lemma 4.4. *Given an input* N*, let* $\mathcal{Q}$ *be the minimum quadtree that represents* N *satisfying the crowding and the grading conditions, and let* $s \in \mathcal{Q}$ *be a square and* p *be a point in* s*. We have* $\mathrm{lfs}(p) \in \Theta(|s|)$*; also, if* $p \in \mathsf{N}$*, then* $\mathrm{lfs}(p) > |s|$.

Proof. If $p \in \mathsf{N}$ or there exists an input vertex in s, then by the crowding condition, the neighbors of s do not contain a vertex. This implies that there are no other input vertices in a ball of radius $|s|$ around p, i.e., $\mathrm{lfs}(p) > |s|$. Otherwise, let $v \in s' \neq s$ be the input vertex nearest p. If s' is not adjacent to s, since all the adjacent squares have size at least $|s|/2$ by the grading condition, we have $\mathrm{lfs}(p) \geq \mathrm{NN}_{\mathsf{N}}(p) > |s|/2$. If s' is adjacent to s, then by the Lipschitz condition $\mathrm{lfs}(p) + |pv| \geq \mathrm{lfs}(v)$, and $\mathrm{lfs}(v) > |s'|$ by the analysis above, and consequently $\mathrm{lfs}(v) > |s|/2$ by s' being adjacent to s. Therefore, if $|pv| > |s'|/2$ then $\mathrm{lfs}(p) \geq \mathrm{NN}_{\mathsf{N}}(p) = |pv| > |s|/4$; otherwise, the same lower bound still holds, $\mathrm{lfs}(p) \geq \mathrm{lfs}(v) - |pv| > |s'|/2 \geq |s|/4$.

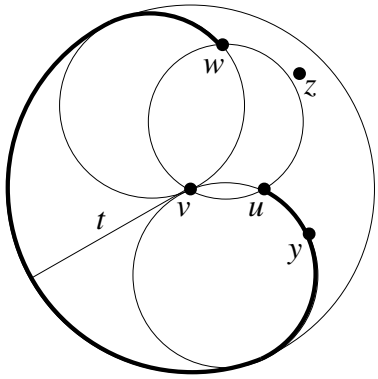
For the upper bound, we use the minimality of the quadtree, that $\tilde{s}$, the parent of s, must have been split because of one of the two conditions. If $\tilde{s}$ is split because of crowding then there exist two vertices within $2\sqrt{d}|\tilde{s}|$ distance of p. If $\tilde{s}$ is split due to grading, this split must have been caused by a crowding split due to vertices $v, u \neq p$. Facts 1 and 2 prove that $\tilde{s}$ is at most a neighbor's neighbor of the quadtree node containing v, also u. This implies that there exist two vertices within $3\sqrt{d}|\tilde{s}|$ distance of p. In either case, $\mathrm{lfs}(p) < O(|s|)$. □

5. Computing Clipped Voronoi Cells

For computing the clipped Voronoi cells, our quadtree data structure supports the function `QTClippedVoronoi`. Given a vertex v and a parameter $\beta > 1$, this function returns the set of β-clipped Voronoi neighbors of v. For an arbitrary vertex in an arbitrary quadtree, the routine can take time polynomial in the number of cells of the quadtree. However, in Section 8 we prove that given the access pattern of our algorithm, all the `QTClippedVoronoi` calls terminate in worst-case constant time.

To determine the β-clipped Voronoi neighbors, CV, we perform a scan starting at v and proceeding along a circular frontier that moves away from v up to a maximum distance $t_{\max} = \beta\,\mathrm{NN}_{\mathscr{M}}(v)$. At time t the frontier is the boundary of the ball of radius t. When the scan reaches a vertex u, we determine whether u is a Voronoi neighbor of v or not. If it is, we add u to CV, otherwise, we discard it. This way, throughout the scan, we maintain the set CV, which consists of the β-clipped Voronoi neighbors of v within the current scan distance. For efficiency, we need to ensure that the scan does not exceed the certificate region. Because if the scan goes

Figure 4: Illustration of the distance function δ^v. In this example, $CV_t = \{u, w\}$ and the thick curve is the set of points with distance $\delta^v_{CV_t}(x) = t$, e.g., y is at distance t (note that u and w are at distance $< t$). Since y is guaranteed to be a Voronoi neighbor, the algorithm inserts y into CV_t. There is no empty ball that touches both v and z, so $\delta^v_{CV_t}(z) = \infty$.



past the certificate region, we may not be able to bound the number of quadtree squares and thus the number of vertices we visit in this scan. Therefore, we use a distance function that differs from the Euclidean one. For any point x, we define the distance $\delta^v_{\mathscr{M}}(x)$ as the diameter of the smallest certificate ball that includes v and x on its boundary. If no such ball exists then we define $\delta^v_{\mathscr{M}}(x) = \infty$. By using this distance function, we eliminate the need to check whether a vertex that the scan reaches is a Voronoi neighbor or not, because the empty ball that defines its distance to v is a certificate ball for x. Figure 4 illustrates an example.

It is not clear how to compute the distance $\delta^v_{\mathscr{M}}(x)$ exactly without a prohibitive overhead. However, we can relax the requirement that the balls be empty of vertices in $\mathscr{M}$. Instead, using CV_t, which we define as the set of β-clipped Voronoi neighbors of v up to time t, we compute $\delta^v_{CV_t}(x)$, the diameter of the smallest ball with v and x on its boundary that includes no vertex of CV_t in its interior. As we advance time, this distance function does not decrease because adding new vertices into CV can only make it harder for a ball to be empty. Therefore, $\delta^v_{CV_t}(x) \leq \delta^v_{\mathscr{M}}(x)$ for all t. We also prove that at time t, when the scan reaches a vertex x, the distance to x is accurately computed, that $\delta^v_{CV_t}(x) = \delta^v_{\mathscr{M}}(x) = t$ (Lemma 5.1). A corollary of this result is that $CV_{t_{\max}} = CV$ and that the scan visits the certificate region of $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$ because all points in the certificate region have distance less than $t_{\max}$.

Lemma 5.1. *During the computation of* $\mathrm{Vor}^{\beta}_{\mathscr{M}}(v)$*, when the scan reaches time t, for any vertex u satisfying $\delta^v_{CV_t}(u) \leq t$, we have $\delta^v_{CV_t}(u) = \delta^v_{\mathscr{M}}(u)$.*

Proof. Pick any vertex u satisfying $\delta^v_{CV_t}(u) \leq t$, i.e., there exists a ball of diameter $\delta^v_{CV_t}(u)$ touching both v and u and containing no vertex of CV_t in its interior. By the monotonicity of the distance function, we know that $\delta^v_{CV_t}(u) \leq \delta^v_{\mathscr{M}}(u)$. We want to show that $\delta^v_{CV_t}(u) < \delta^v_{\mathscr{M}}(u)$ is not possible by proving that there is no vertex of

```
QTClippedVoronoi (v_(t), β) =
  t_max ← ∞,  E ← {square of v},  CV ← ∅
  while ∃p ∈ E such that δ^v_CV(p) < t_max
    p ← arg min_{p∈E} δ^v_CV(p)
    if p is a vertex then
      if CV = ∅ then t_max ← β|vp|
      CV ← CV ∪ {p}
    else (p is a square)
      CGInsertEdge(p ⟶ v_(t))
      E ← E ∪ {vertices of p}
      E ← E ∪ {squares adjacent to p}
  return CV
```

For computing the distance $\delta^v_{CV}(p)$

$$\begin{aligned} \text{minimize} \quad & |cv| \\ \text{subject to} \quad & |cv| = |cp| \\ & |cv| \leq |cu| \ \forall u \in CV \\ \text{distance } \delta^v_{CV}(p) &= 2|cv| \end{aligned}$$

Figure 5: Pseudo-code for computing clipped Voronoi cells.

$\mathscr{M} \setminus CV_t$ inside this ball. Towards a contradiction, assume that there exists one, say w. Then, there exists a smaller ball for w, i.e., $\delta^v_{CV_t}(w) < \delta^v_{CV_t}(u)$. Again, by monotonicity, we have $\delta^v_{CV_{t'}}(w) \leq \delta^v_{CV_t}(w)$ for all $t' < t$. Therefore, w must have been discovered at some time $t' < t$ and inserted into $CV_{t'} \subset CV_t$. This contradicts our assumption that w is a vertex in $\mathscr{M} \setminus CV_t$. □

Figure 5 shows the pseudo-code for the scan we describe above. The algorithm discretizes the scan using quadtree squares. Starting at the square of v, it explores outward from v using a queue E of events — reaching vertices and squares. Upon reaching a vertex, it updates CV, and upon reaching a square, it enqueues the vertices that the square contains and the unvisited squares it is adjacent to. Also, to facilitate dependency tracking in dynamic updates, the algorithm inserts a (read) dependency edge for each square scanned, using the `CGInsertEdge` function described in Section 6.

To compute $\delta^v_{CV_t}(p)$, the algorithm finds a point c that is the center of a certificate ball of minimum radius using the convex program of Figure 5 with $O(1)$ variables and $O(|CV|)$ constraints. For a quadtree square, the distance is the minimum distance to any point p in the square. This corresponds to letting p to be free variables in the above program and adding a box constraint ($2d$ linear constraints) on the coordinates of p. This leaves us with a quadratic program rather than a convex one, but it remains a program with $O(1)$ variables and $O(|CV|)$ constraints. As we prove in Section 8, in our algorithms, we make sure that the clipped Voronoi calls return a constant number of clipped Voronoi neighbors. Thus, for our purposes, computation of this distance function takes $O(1)$ time.

6. A Stable Construction Algorithm

Given an input set of vertices, we can construct a ρ-well-spaced superset by repeatedly choosing a vertex, and directly enforcing ρ-well-spacedness on this vertex through what we call a "fill" operation. For any $\beta > \rho > 1$, the fill operation on a vertex inserts Steiner vertices in the (ρ, β) picking region of that vertex until it becomes ρ-well-spaced. Although correct, this basic algorithm is not efficient because vertices may need to be filled multiple times. More specifically, a Steiner vertex inserted while filling a vertex may become the nearest neighbor of an already filled vertex, causing it to no longer be ρ-well-spaced, and requiring it to be filled again. This algorithm is not stable either; it can generate very different outputs when run on similar inputs, because the presence/absence of a single vertex can affect the choices of many subsequent Steiner vertices. To address these problems and achieve efficiency and stability, we refine the basic algorithm by specifying an order in which vertices are filled.

In order to ensure efficiency and avoid filling vertices multiple times, we fill vertices in increasing order according to their nearest-neighbor distances. Before we explain the details of this schedule, we discuss the intuition behind it by pointing out several facts about our Steiner vertex selection scheme. Given a vertex set $\mathscr{M}$, consider filling a vertex $v \in \mathscr{M}$ that is not ρ-well-spaced. Let w be a Steiner vertex inserted while filling v. The first fact is that w is in the (ρ, β) picking region of v.

Fact 3. *The Steiner vertex w is in $\mathrm{Vor}_{\mathscr{M}}^{(\rho,\beta)}(v)$. That is, $\forall u \in \mathscr{M}, |wv| \leq |wu|$ and $\rho\,\mathrm{NN}_{\mathscr{M}}(v) \leq |wv| < \beta\,\mathrm{NN}_{\mathscr{M}}(v)$.*

Since v is the nearest neighbor of w, i.e., $|wv| = \mathrm{NN}_{\mathscr{M}}(w)$, this fact implies that $\mathrm{NN}_{\mathscr{M}}(w) \geq \rho\,\mathrm{NN}_{\mathscr{M}}(v)$. Now, let us suppose that the vertices whose nearest neighbors are at distance less than α are all ρ-well-spaced. Since v is not ρ-well-spaced in $\mathscr{M}$, we have $\mathrm{NN}_{\mathscr{M}}(v) \geq \alpha$. Then, we infer the following fact.

Fact 4. *For any given $\alpha > 0$, if every vertex $u \in \mathscr{M}$ with $\mathrm{NN}_{\mathscr{M}}(u) < \alpha$ is ρ-well-spaced, then $\mathrm{NN}_{\mathscr{M}}(w) \geq \rho\,\alpha$.*

This fact implies that the Steiner vertices that will be inserted into $\mathscr{M}$ are all at least $\rho\,\alpha$ away from any vertex in $\mathscr{M}$. Therefore, inserting a Steiner vertex does not change the nearest neighbors and hence the well-spacedness of the vertices whose nearest neighbors are at distance less than $\rho\,\alpha$. Motivated by this property, we define the *rank* of a vertex $v \in \mathscr{M}$ as the logarithm in base ρ of its nearest

neighbor distance, i.e., $\left\lfloor \log_\rho \mathrm{NN}_{\mathscr{M}}(v) \right\rfloor$ and fill the vertices in a single pass using the rank order. With this partial ordering, for example, the vertices with nearest neighbor distances in $[\rho^r, \rho^{r+1})$ would be at rank r. Note that for any $\rho > 1$, this partial order has only $O(\log \Delta)$ ranks. As we prove in Lemma 8.5, filling vertices in rank order guarantees that filling each vertex takes $O(1)$ time, yielding an efficient construction algorithm. However, these refinements are not enough to ensure stability.

In order to achieve stability, we take advantage of the locality of our Steiner vertex selection scheme and geometrically partition the vertices at each rank into a constant number of *color* classes and fill them in color order so that vertices of the same color class can be filled independently. More specifically, we say that two vertices at the same rank are *independent* if at least one of them is not ρ-well-spaced and the certificate region of the β-clipped Voronoi cell of any of them does not intersect the (ρ, β) picking region of the other. Intuitively, two vertices are independent if the Steiner vertices inserted while filling one of them do not alter the picking region of the other. We identify independent vertices by using a *coloring scheme* that partitions the space based on a *coloring parameter* κ, and a real valued function $\ell(r)$ defined on ranks. At each rank r, we partition the space into d-dimensional hypercubes or *r-tiles* with side length $\ell(r)$. We color r-tiles such that they are colored periodically in each dimension with period κ, using κ^d colors in total. A vertex v has color $c \in \{0, 1, \ldots, \kappa^d - 1\}$ if it lies in a c colored r-tile. Figure 6 illustrates a coloring scheme. By choosing $\ell(r)$ small enough and κ large enough, we prove that two vertices at the same rank are independent if they have the same color (Lemma 9.1). Therefore, at a given rank, filling vertices in color order restricts the dependencies between vertices: filling a vertex may affect only the unprocessed vertices of different colors. During a dynamic update, this makes it possible to re-fill a vertex without affecting other independent vertices at the same rank and color.

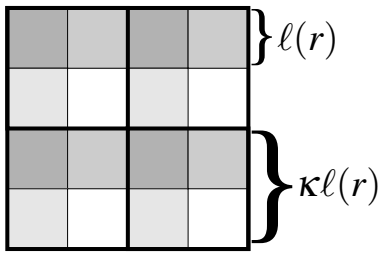


Figure 6: Illustration of a coloring scheme in 2D. The coloring parameter κ is 2 and there are 4 colors in total.

The efficiency and stability of our algorithm critically relies on filling vertices in rank and color order. In order to fill a vertex v that is currently at rank r, the algorithm schedules a *fill* operation acting on v at rank r. However, since the rank

of a vertex depends on its nearest neighbor and since that can change as Steiner vertices are inserted, we need to update the ranks of the vertices dynamically. In order to ensure that the ranks of the vertices are up-to-date, in our algorithm, we use another type of operation called *dispatch*. For each vertex v, our algorithm creates a single dispatch operation acting on v. This operation computes the rank of v, updates the ranks of β-clipped Voronoi neighbors of v, and schedules new fill operations as necessary. In order to ensure timely execution of dispatch operations, the algorithm assigns ranks to dispatch operations as well and executes both types of operations in rank order with dispatches having precedence over fills at the same rank. For an input vertex v, the algorithm assigns the logarithm in base ρ of the side length of the quadtree square that contains v as the rank of the dispatch operation acting on v. As we prove in Lemma 4.4 this rank is $O(1)$ ranks below the actual rank of v. For a Steiner vertex w, at the time of its insertion, since we know that the nearest neighbor of w is the vertex being filled, the fill operation that inserts w easily computes the current rank of w and schedules a dispatch operation acting on w at its current rank.

Using dispatch operations, the algorithm guarantees that for each vertex there exists a fill operation acting on it at its most up-to-date rank (Lemma 7.1). When executed, a fill operation makes the vertex it acts on ρ-well-spaced; subsequent fill operations terminate immediately without inserting Steiner vertices. Instead of creating a single fill operation per vertex and updating its rank as the ranks of the vertices change, we prefer the approach of recording and executing multiple fill operations acting on a single vertex because it simplifies the analysis by making the dependencies between the operations explicit.

Figure 7 shows the pseudo-code of our algorithm `StableWS`. The algorithm starts by constructing a quadtree $\mathcal{Q}$ and stores it for use in dynamic updates. It then constructs a ρ-well-spaced output by performing dispatch and fill operations that it enqueues in Ω. When enqueueing a dispatch or a fill operation acting on a vertex w, the algorithm computes the rank r_w of w by using its current nearest neighbor distance. (For dispatch operations acting on input vertices, it uses an approximation.) Then, computing the color c_w of w at this rank, it determines the *time* of this operation, which is comprised of the rank r_w, a flag indicating a dispatch (`D`) or a fill (`F`), and the color c_w only if the operation is a fill operation. In the pseudo-code, we represent this operation with the vertex w itself and its time t_w in the subscript ($w_{(t_w)}$). The algorithm executes the operations in Ω in time order, first by rank then by operation type and then by color order (for fills): it performs the dispatch operations before the fill operations, ordering fill operations at the same rank by color. For brevity, we define time $t = 0$ to be the beginning of time,

```
Dimension:  d
Parameters:  ρ, β, κ, ℓ(r)

StableWS (N) =
  Q ← QTBuild(N)
  for each v ∈ N
    apxnnv ← |square of v|
    Enqueue(v, D, apxnnv, v_(0), Ω)
  for r = min rank in Ω to ⌊log_ρ √d⌋
    for each v_(r,D) ∈ Ω
      Dispatch(v_(r,D), Ω)
    for c = 0 to κ^d − 1
      for each v_(r,F,c) ∈ Ω
        Fill(v_(r,F,c), Ω)
  return Q

Dispatch (v_(t), Ω) =
  CV ← QTClippedVoronoi(v_(t), β)
  nnv ← min{|vu| : u ∈ CV}
  Enqueue(v, F, nnv, v_(t), Ω)
  for each w ∈ CV
    Enqueue(w, F, |wv|, v_(t), Ω)

Fill (v_(t), Ω) =
  CV ← QTClippedVoronoi(v_(t), β)
  while v is not ρ-well-spaced
    Pick w ∈ Vor^(ρ,β)(v)
    QTInsertSteiner(v_(t), w)
    CGInsertEdge(v_(t) ⟶ square of w)
    Enqueue(w, D, |wv|, v_(t), Ω)
    CV ← CV ∪ {w}

Enqueue (w, flag, nnw, v_(t), Ω) =
  r_w ← ⌊log_ρ nnw⌋,  c_w ← Color(w, r_w, κ)
  if flag = D then t_w ← (r_w, D)
  else t_w ← (r_w, F, c_w)
  if t_w > t then
    if ∄ edge · ⟶ w_(t_w) then
      Ω ← Ω ∪ {w_(t_w)}
    CGInsertEdge(v_(t) ⟶ w_(t_w))

Color (v, r, κ) =
  for i = 1 to d do c_i ← ⌊v_i/ℓ(r)⌋ mod κ
  return c = c_1 c_2 ... c_d in base κ
```

Figure 7: The pseudo-code of the stable construction algorithm.

when the input vertices are enqueued for dispatch operations but before any of them are performed, and we define time $t = \infty$ to be the end of the algorithm. We write M_t to refer to the output at time t, e.g., M_0 is the input, N, and M_∞ is the output, M. For readability, we use t instead of M_t in the subscript, e.g., NN_t instead of $\mathrm{NN}_{\mathsf{M}_t}$.

To support efficient dynamic updates, the construction algorithm builds a computation graph. The *computation graph* $G = (V, E)$ consists of nodes, $V = \Sigma \cup \Omega$, comprised of the set of quadtree squares (Σ) and the set of operations (Ω), and directed edges representing various dependencies between operations and squares. One can view these dependencies as read, write, and execution flow dependencies. The construction algorithm uses the `CGInsertEdge` function to record each dependency by inserting an edge into the computation graph. Consider executing an operation represented by $v_{(t)}$. If the function `QTClippedVoronoi` executed by $v_{(t)}$ reads a square s, it records this (read) dependency by inserting the edge $s \longrightarrow v_{(t)}$ (Section 5). If $v_{(t)}$ calls `Enqueue` to schedule an operation op acting on w into Ω, the `Enqueue` function first computes the rank and color of w and determines the time t_w of $op = w_{(t_w)}$. If $t_w > t$, `Enqueue` records this (execu-

tion) dependency by inserting the edge $v_{(t)} \longrightarrow w_{(t_w)}$. In order to avoid duplicate operations in Ω, it schedules $w_{(t_w)}$ into Ω only if there is no edge from another operation towards $w_{(t_w)}$. Finally, to account for the (write) dependencies that arise by inserting Steiner vertices, for each Steiner vertex w that $v_{(t)}$ inserts, the algorithm inserts the edge $v_{(t)} \longrightarrow s$, where s is the square that contains w. For the purposes of facilitating analysis, we tag each edge with the time of the operation that creates it, in the examples above, this time is t.

7. Output Quality and Size

This section includes the proofs of the quality of the output of our algorithm, i.e., M is ρ-well-spaced and size-optimal. Lemma 7.3 proves size-optimality by showing that M is size-conforming. For ρ-well-spacedness, the first two lemmas prove that our algorithm fills vertices in an order such that after filling a vertex the key invariant is satisfied—the vertex becomes and remains ρ-well-spaced. Therefore, our algorithm incrementally progresses towards a ρ-well-spaced output. In these two lemmas, let $\mathscr{M}$ be the set of vertices in the output at the beginning of rank r.

Lemma 7.1. *At the beginning of rank r, assume that every vertex $u \in \mathscr{M}$ with $\mathrm{NN}_{\mathscr{M}}(u) < \rho^r$ is ρ-well-spaced. Then, for every vertex $w \in \mathscr{M}$ with $\mathrm{NN}_{\mathscr{M}}(w) \in [\rho^r, \rho^{r+1})$, there exists a fill operation that acts on w at rank r.*

Proof. Consider a vertex $w \in \mathscr{M}$, let u be its nearest neighbor in $\mathscr{M}$, and assume that $\rho^r \leq |wu| < \rho^{r+1}$. Let $w_{(r_w,\mathtt{D})}$ and $u_{(r_u,\mathtt{D})}$ be the dispatch operations that act on w and u respectively. If $r_w \leq r$ and u is in the output at the beginning of rank r_w then $w_{(r_w,\mathtt{D})}$ schedules a fill operation that acts on w at rank r. Alternatively, $u_{(r_u,\mathtt{D})}$ schedules such a fill operation if $r_u \leq r$ and w is a β-clipped Voronoi neighbor of u at the beginning of rank r_u. We prove that one of the two conditions holds.

Analyzing the vertices w and u, whether they are both input vertices or one of them is a Steiner vertex inserted when the other one was in the output, in two of the three cases, we prove that the first condition holds. In the first case, if both w and u are input vertices then by Lemma 4.4, $r_w \leq r$. In the second case, in which w is a Steiner vertex and u is in the output when w is being inserted, consider the vertex v that creates w. By Fact 3, we know that $|wv| \leq |wu|$, which implies that $r_w \leq r$.

We prove that the second condition holds in the remaining case, in which u is a Steiner vertex and w is already in the output when u is being inserted. Similar to the previous case, we deduce that $r_u \leq r$. Since u is the nearest neighbor of w

in $\mathscr{M}$, w is a Voronoi neighbor of u in $\mathscr{M}'$, where $\mathscr{M}' \subset \mathscr{M}$ is the output at the beginning of rank r_u. If u is ρ-well-spaced in $\mathscr{M}$ then $|wu| \leq 2\rho\,\mathrm{NN}_{\mathscr{M}}(u) < 2\beta\,\mathrm{NN}_{\mathscr{M}'}(u)$. Otherwise, the assumption of the lemma implies $\rho^r \leq \mathrm{NN}_{\mathscr{M}}(u)$. Since $|wu| < \rho^{r+1}$, we get $|wu| < \rho\,\mathrm{NN}_{\mathscr{M}}(u) < 2\beta\,\mathrm{NN}_{\mathscr{M}'}(u)$. Either way, w is a β-clipped Voronoi neighbor of u in $\mathscr{M}'$. □

Lemma 7.2 (Progress)**.** *At the beginning of rank r, every vertex $u \in \mathscr{M}$ with $\mathrm{NN}_{\mathscr{M}}(u) < \rho^r$ is ρ-well-spaced.*

Proof. We use induction. At the minimum rank, there are no vertices with smaller nearest-neighbor distance, so the claim is trivially true. Assume that the lemma holds up to rank r, that is, every vertex $u \in \mathscr{M}$ with $\mathrm{NN}_{\mathscr{M}}(u) < \rho^r$ is ρ-well-spaced. For rank $r+1$, let $\mathscr{M}' \supset \mathscr{M}$ be the set of vertices in the output at the beginning of rank $r+1$ and consider a vertex $w \in \mathscr{M}'$ with $\mathrm{NN}_{\mathscr{M}'}(w) < \rho^{r+1}$. We claim that $w \in \mathscr{M}$; towards a contradiction, assume that $w \in \mathscr{M}' \setminus \mathscr{M}$. Then, w is a Steiner vertex inserted at rank r. Repeatedly applying Fact 4 for each (Steiner) vertex in $\mathscr{M}' \setminus \mathscr{M}$, we see that the nearest neighbors of these Steiner vertices are at distance $\geq \rho^{r+1}$; in particular, $\mathrm{NN}_{\mathscr{M}'}(w) \geq \rho^{r+1}$. This is a contradiction to our criteria $\mathrm{NN}_{\mathscr{M}'}(w) < \rho^{r+1}$, thus, $w \in \mathscr{M}$. Furthermore, $\mathrm{NN}_{\mathscr{M}}(w) < \rho^{r+1}$ for similar reasons. If $\mathrm{NN}_{\mathscr{M}}(w) < \rho^r$ then by our induction hypothesis w is ρ-well-spaced. Otherwise, if $\rho^r \leq \mathrm{NN}_{\mathscr{M}}(w) < \rho^{r+1}$, by Lemma 7.1, there exists a fill operation that acts on w at rank r. After executing that operation, w becomes ρ-well-spaced. Finally, Fact 4 implies that w remains ρ-well-spaced. □

Lemma 7.3. *The output* M *is size-conforming and size-optimal with respect to* N.

Proof. We use induction over the order in which the algorithm inserts Steiner vertices and show that there exists a constant c such that for every $v \in \mathsf{M}$, $c\,\mathrm{NN}_{\mathsf{M}}(v) \geq \mathrm{lfs}(v)$, thereby proving that M is size-conforming. In the base case, every vertex is an input vertex and the nearest neighbor of an input vertex is exactly the local feature size. For the inductive case, assume that there exists a constant c such that, for every $v \in \mathscr{M}$, we have $c\,\mathrm{NN}_{\mathscr{M}}(v) \geq \mathrm{lfs}(v)$. Furthermore, assume that v inserts a Steiner vertex w and the new output is $\mathscr{M}' = \mathscr{M} \cup \{w\}$. We analyze the inductive claim for w and for any vertex $u \in \mathscr{M}$ separately. For w, by Fact 3 we know that $|wv| \geq \rho\,\mathrm{NN}_{\mathscr{M}}(v)$ and $\mathrm{NN}_{\mathscr{M}'}(w) = |wv|$. By the triangle inequality, lfs satisfies the Lipschitz condition: $\mathrm{lfs}(v) + |wv| \geq \mathrm{lfs}(w)$. By the inductive hypothesis, $c\,\mathrm{NN}_{\mathscr{M}}(v) \geq \mathrm{lfs}(v)$. Therefore, we have $(\frac{c}{\rho}+1)|wv| = (\frac{c}{\rho}+1)\,\mathrm{NN}_{\mathscr{M}'}(w) \geq \mathrm{lfs}(w)$.

For any vertex $u \in \mathscr{M}$, if $\mathrm{NN}_{\mathscr{M}}(u) = \mathrm{NN}_{\mathscr{M}'}(u)$ then the claim holds trivially. Otherwise, assume that $\mathrm{NN}_{\mathscr{M}}(u) > \mathrm{NN}_{\mathscr{M}'}(u) = |wu|$. By the Lipschitz

condition, we have $|wu| + \mathrm{lfs}(w) \geq \mathrm{lfs}(u)$ and by Fact 3 we know $|wu| \geq |wv|$. Combining these by the bound we obtained for $\mathrm{lfs}(w)$, we get $(\frac{c}{\rho}+2)|wu| = (\frac{c}{\rho}+2)\,\mathrm{NN}_{\mathscr{M}'}(u) \geq \mathrm{lfs}(u)$. Solving for $c \geq \frac{c}{\rho}+2$, we conclude that any $c \geq \frac{2\rho}{\rho-1}$ suffices to prove the inductive step. Therefore, M is size-conforming and hence size-optimal [26]. □

Theorem 7.4 (Correctness)**.** `StableWS` *constructs a size-optimal* ρ*-well-spaced superset* M *of its input* N.

Proof. The property that M is ρ-well-spaced follows from the Progress Lemma and the fact that `StableWS` iterates over all ranks. Lemma 7.3 proves the size bound. □

8. Runtime

We analyze the running time of our static algorithm and emphasize two lemmas that are useful in the analysis of our dynamic algorithm. The first lemma (Lemma 8.1) proves that throughout the algorithm, the nearest-neighbor distance of a vertex v changes only by a constant factor. The second lemma (Lemma 8.2) proves that all operations acting on v have rank $\lfloor \log_\rho \mathrm{NN}_\infty(v) \rfloor \pm O(1)$; none are scheduled too early nor too late.

Lemma 8.1. *Let t be the time at which v is created ($t = 0$ for input vertices). Then,* $\mathrm{NN}_t(v) \in \Theta(\mathrm{NN}_\infty(v))$.

Proof. As time progresses, more vertices are added, so the nearest neighbor distance can only shrink: $\mathrm{NN}_t(v) \geq \mathrm{NN}_\infty(v)$. For the upper bound, we analyze input vertices and Steiner vertices separately. By definition, an input vertex v has $\mathrm{lfs}(v) = \mathrm{NN}_0(v)$. The algorithm is size-conforming (Lemma 7.3), so $\mathrm{NN}_0(v) = \mathrm{lfs}(v) \in O(\mathrm{NN}_\infty(v))$. For a Steiner vertex w that is created at time $t = (r, \mathtt{F}, c)$, Fact 3 implies that $\rho^{r+1} \leq \mathrm{NN}_t(w) \leq \beta\rho^{r+1}$. For any other Steiner vertex u that is created later, the same fact implies that $\rho^{r+1} \leq |uw|$ which means $\rho^{r+1} \leq \mathrm{NN}_\infty(w)$. Therefore, $\mathrm{NN}_t(w) \leq \beta\rho^{r+1} \leq \beta\,\mathrm{NN}_\infty(w)$. □

Lemma 8.2. *If an operation at rank r acts on v then* $\mathrm{NN}_\infty(v) \in \Theta(\rho^r)$.

Proof. Consider an operation $v_{(t_v)}$ at rank r. Lemma 8.1 implies that it suffices to prove $\mathrm{NN}_t(v) \in \Theta(\rho^r)$, where t is the time $v_{(t_v)}$ is created. If $v_{(t_v)}$ is a dispatch operation and v is an input vertex, then $t = 0$, and our algorithm uses the size of the square that contains v to approximate $\mathrm{NN}_0(v)$. Hence, $\mathrm{NN}_0(v) \in \Theta(\rho^r)$

by Lemma 4.4. If $v_{(t_v)}$ is a dispatch operation and v is a Steiner vertex then we know that v is created at time t by a fill operation acting on a vertex u and that $r = \lfloor \log_\rho |vu| \rfloor$. Since v is picked from the Voronoi cell of u, $|vu| = \mathrm{NN}_t(v)$, thus $\mathrm{NN}_t(v) \in \Theta(\rho^r)$. If $v_{(t_v)}$ is a fill operation created by the dispatch operation acting on v, then we know that the rank is computed exactly, i.e., $r = \lfloor \log_\rho \mathrm{NN}_t(v) \rfloor$. The last case is that $v_{(t_v)}$ is a fill operation created by a dispatch operation op acting on another vertex u at rank r'. We know that op assigns the rank of v to be $r = \left\lfloor \log_\rho |vu| \right\rfloor$. Since $\mathrm{NN}_t(v) \leq |vu|$, we get $\mathrm{NN}_t(v) < \rho^{r+1}$, thus, the upper bound holds. For the lower bound, since $|vu| \geq \rho^r$, it suffices to show that $\mathrm{NN}_t(v) \in \Omega(|vu|)$. If $\mathrm{NN}_t(v) \geq \rho^{r'}$, we show that $\rho^{r'} \in \Omega(|vu|)$ by applying the result from above for op, that $\mathrm{NN}_t(u) \in \Theta(\rho^{r'})$, and by using the fact that v is a β-clipped Voronoi neighbor of u at time t, that $2\beta\,\mathrm{NN}_t(u) \geq |vu|$. Otherwise, if $\mathrm{NN}_t(v) < \rho^{r'}$ then by the Progress Lemma, v is ρ-well-spaced at time t. Since v and u are Voronoi neighbors at time t, this implies that u is a ρ-clipped Voronoi neighbor of v. Therefore, $2\rho\,\mathrm{NN}_t(v) \geq |vu|$ and we prove in all cases that $\mathrm{NN}_\infty \in \Theta(\rho^r)$. □

Lemma 8.3. *At the beginning of rank r, any point p inside an empty ball of radius* ρ^r *satisfies* $\mathrm{lfs}(p) \in \Omega(\rho^r)$.

Proof. Let c be the center of an empty ball of radius ρ^r, i.e., $\mathrm{NN}_t(c) \geq \rho^r$, where $t = (r, \mathtt{D})$. Given this ball, let p be a point inside it. For some constant ε whose value we will set later, if $\mathrm{NN}_t(p) \geq \varepsilon\rho^r$ then our proof is done. Otherwise, if $\mathrm{NN}_t(p) < \varepsilon\rho^r$, let q be the point at distance $\rho^r/2$ away from c on the ray from c to p, and let $u \in \mathsf{M}_t$ be the vertex nearest q. Then, we claim that $\mathrm{NN}_t(u) > \rho^r/2\rho$ and for some small enough ε that u is the vertex nearest p as well. Using the Lipschitz condition, we get $\mathrm{lfs}(p) + |pu| \geq \mathrm{lfs}(u) \geq \mathrm{NN}_t(u)$. Then, our claims imply $\mathrm{lfs}(p) > \rho^r/2\rho - |pu| > (1/2\rho - \varepsilon)\rho^r$ and consequently prove our lemma statement.

Our first claim is trivially true if $\mathrm{NN}_t(u) \geq \rho^r$; otherwise, u must be ρ-well-spaced by Lemma 7.2, which implies that q, being a point inside the Voronoi cell of u, is within $\rho\,\mathrm{NN}_t(u)$ distance of u. In other words, $\mathrm{NN}_t(u) > |qu|/\rho$. Since u is outside the empty ball, $|qu| \geq \rho^r/2$, therefore, we prove our first claim.

For proving the second claim, we first observe that u, the nearest neighbor of q, lies inside the ball of radius $(1/2 + \varepsilon)\rho^r$ centered at q because there is a vertex inside the ball of radius $\varepsilon\rho^r$ centered at p. Since the ball of radius ρ^r centered at c is empty, the crescent defined by this empty ball and the ball centered at q contains u. We will show that this region, shaded in Figure 8, is contained in a ball

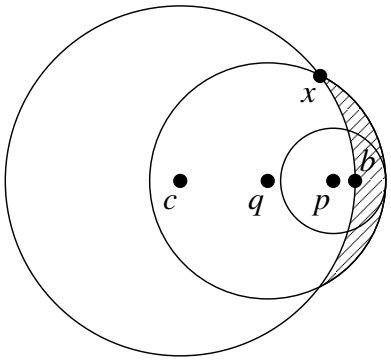


Figure 8: Illustration of the proof of Lemma 8.3. There is an empty ball centered at c of radius ρ^r and p is a point inside this ball. The nearest neighbor of p is within $\varepsilon\rho^r$ distance (the small ball). The point q, $\rho^r/2$ away from c on the ray from c to p, has its nearest neighbor within $(1/2+\varepsilon)\rho^r$ distance (the mid-size ball), inside the shaded region. The point x in the shaded region is one of the farthest away from b. The lemma is proven by showing that the shaded region can be made small enough.

of diameter $\rho^r/2\rho$. Then, using the fact that the nearest neighbor of u is at least $\rho^r/2\rho$ away from u, we prove that u is the only vertex in this region and therefore the vertex nearest p. In order to bound the diameter, for appropriate ε, we show that any point of the shaded region is within $\rho^r/4\rho$ distance of the point b, that is located ρ^r away from c on the ray from c to p: observe that any point x on the intersection of the ball centered at c of radius ρ^r and the ball centered at q of radius $(1/2+\varepsilon)\rho^r$ is the farthest away from b. Since xq is the median of the side cb of the triangle cbx, using Apollonius' theorem, one can get $|xb|^2 = 2\varepsilon(\varepsilon+1)\rho^{2r}$. Solving for $|xb| \leq \rho^r/4\rho$, we see that any $\varepsilon \leq \sqrt{1/4+1/32\rho^2} - 1/2$ suffices to prove our claim and therefore our lemma. □

Lemma 8.4. *Computing the β-clipped Voronoi cell of a vertex takes $O(1)$ time.*

Proof. When the `QTClippedVoronoi` function visits a quadtree square, that square either intersects the certificate region, or it is a neighbor of a square that intersects the certificate region. Let r be the rank at which the function is called and s be a square that intersects the certificate region. By Lemma 8.3, for any point $p \in s$, that is inside the certificate region, $\mathrm{lfs}(p) \in \Omega(\rho^r)$. Furthermore, by Lemma 4.4, $\mathrm{lfs}(p) \in \Theta(|s|)$, which implies that s covers a volume of $\Omega(\rho^r)$. By Lemmas 8.1 and 8.2, the certificate region of the β-clipped Voronoi cell of v is within a ball of radius $O(\rho^r)$. This implies that there are only $O(1)$ squares that intersect the certificate region. Thanks to the grading condition, the squares have bounded number of squares adjacent to them, therefore `QTClippedVoronoi` visits only $O(1)$ squares. The function also does work iterating over the vertices each square contains. By Lemma 7.3, a vertex u has a nearest neighbor no closer

than $\Omega(\mathrm{lfs}(u))$; meanwhile, again by Lemma 4.4, the quadtree square that contains u has side length $O(\mathrm{lfs}(u))$. Hence, each square contains only $O(1)$ vertices and the total work is $O(1)$. □

Lemma 8.5. *Dispatch and fill operations run in $O(1)$ time.*

Proof. The main costs of an operation $v_{(t)}$ are the β-clipped Voronoi cell computations and the loops. Lemma 8.4 shows that Steiner vertex insertions and the clipped Voronoi cell computations take constant time. This implies that there are a constant number of β-clipped Voronoi neighbors of v. If $v_{(t)}$ is a dispatch operation, it iterates over each of them, this takes constant time. If $v_{(t)}$ is a fill operation, it has a loop that inserts Steiner vertices until v is ρ-well-spaced. For each inserted Steiner vertex w, Fact 3 implies $\mathrm{NN}_t(w) \geq \rho\,\mathrm{NN}_t(v)$. Thus, we can associate non-overlapping empty balls of radius $\rho\,\mathrm{NN}_t(v)/2$ around every Steiner vertex. Since the Steiner vertices are in a ball of radius $\beta\,\mathrm{NN}_t(v)$ around v, a packing argument shows that $v_{(t)}$ inserts a constant number of Steiner vertices. This concludes that the operation represented by $v_{(t)}$ runs in constant time. □

Lemma 8.6. *For every vertex $v \in \mathsf{M}$, there are $O(1)$ operations that act on v.*

Proof. By Lemma 8.2, any operation acting on v has rank $\left\lfloor \log_\rho \mathrm{NN}_\infty(v) \right\rfloor \pm O(1)$. Therefore, if we can bound the number of operations acting on v at each rank by a constant, our claim will hold. There is only one dispatch operation for each vertex, so we only need to count the fill operations scheduled by other dispatch operations. Fix r and consider a dispatch operation acting on u at time $t' = (r', \mathtt{D})$ scheduling a fill operation acting on v at rank r. Then, v is β-clipped Voronoi neighbor of u, in other words, $|uv| \leq 2\beta\,\mathrm{NN}_{t'}(u)$. The fact that the fill operation is scheduled for rank r implies $\rho^r \leq |uv| < \rho^{r+1}$. Considering the dispatch operation, Lemmas 8.1 and 8.2 show that $\mathrm{NN}_{t'}(u) = O(\rho^{r'})$. These facts imply $\rho^r = O(\rho^{r'})$. Again by Lemma 8.2, we know that there exists an empty ball around u with radius $\Omega(\rho^{r'})$ which is $\Omega(\rho^r)$ by the previous assertion. We already know that $|uv| < \rho^{r+1}$, therefore, a packing argument proves our claim. □

Theorem 8.7 (Efficiency)**.** `StableWS` *runs in $O(n \log \Delta)$ time.*

Proof. As shown in Section 4, building the quadtree takes $O(n \log \Delta)$ time. By Lemmas 8.5 and 8.6, the rest of the algorithm takes $O(m)$ time, where $m = |\mathsf{M}|$. The total runtime is $O(n \log \Delta + m)$. That $m \in O(n \log \Delta)$ follows from our dynamic bounds. □

9. Dynamic Stability

We call two inputs N and N' *related* if they differ by one vertex, i.e., N' can be obtained from N by inserting or deleting a vertex. To analyze the stability of the algorithm `StableWS`, we define a notion of distance between two executions with related inputs. We prove that this distance is bounded by $O(\log \Delta)$ in the worst-case, where Δ is the larger geometric spread of the inputs N and N' (Lemma 9.5).

As described in Section 6, $\texttt{StableWS}(\mathsf{N})$ constructs a computation graph $G = (V, E)$ by building quadtree squares Σ and a set of operations Ω. The set of nodes V is $\Sigma \cup \Omega$; the edges E represent the dependencies in the computation. For another input set N' which is related to N, consider running $\texttt{StableWS}(\mathsf{N}')$ and creating $G' = (V', E')$, Σ', and Ω' similarly. We define a recursive *matching* between the nodes of the two executions: two operations $v_{(t)} \in \Omega$ and $v'_{(t')} \in \Omega'$ match if $v = v'$, $t = t'$, and either the times $t = t' = 0$ or there exist matching operations $w_{(\tau)} \in \Omega$ and $w'_{(\tau')} \in \Omega'$ such that the computation graphs G and G' include the edges $w_{(\tau)} \longrightarrow v_{(t)}$ and $w'_{(\tau')} \longrightarrow v'_{(t')}$ respectively; and, two squares $s \in \Sigma$ and $s' \in \Sigma'$ match if s and s' have the same corner points. We denote this matching by $\mu : V' \to V$, where $\mu = \{(v'_{(t')}, v_{(t)}) \mid v'_{(t')} \text{ and } v_{(t)} \text{ match}\}$. We denote the domain and the range of μ by $\mathsf{dom}(\mu)$ and $\mathsf{range}(\mu)$. Using this matching, we define $\mu' = \mu \cup \{(u,u) \mid u \in V' \setminus \mathsf{dom}(\mu)\}$ to be a total function defined on the nodes V' of G'. We combine the computation graphs in a *union graph* $G^{\cup} = (V \cup \mu'(V'), E \cup \mu'(E'))$, where $\mu'(E') = \{(\mu'(u), \mu'(v)) \mid (u,v) \in E'\}$. Intuitively, the union graph injects G' into G under the guidance of μ by extending G with the unmatched nodes of G', unifying the matched nodes, and adding the edges of G' while redirecting them to the matched nodes appropriately. In order to capture the dependencies between two operations, we define a path in the union graph to be a dependency path if the times of the edges on the path do not decrease. Lemma 9.1 allows us to refine this definition: a path $(x_0, x_1, \ldots, x_k)$ is a *dependency path* if the times of the edges $x_0 \longrightarrow x_1$, $x_1 \longrightarrow x_2, \cdots, x_{h-1} \longrightarrow x_h$ increase monotonically.

Lemma 9.1. *Set coloring parameters $\ell(r)$ and κ such that $\ell(r) < \rho^r/\sqrt{d}$ and $\kappa > 1 + 3\beta\rho^{r+1}/\ell(r)$. Then, any two fill operations at the same rank are independent if the vertices they act on have the same color.*

Proof. Consider two operations $v_{(t)}$ and $u_{(t)}$, where $t = (r, \mathtt{F}, c)$. Let $\mathscr{M}$ be the set of vertices in the output at the beginning of rank r. If both v and u are ρ-well-spaced in $\mathscr{M}$ then $v_{(t)}$ and $u_{(t)}$ are independent. Otherwise, if v is not ρ-well-spaced the Progress Lemma implies that $\mathrm{NN}_{\mathscr{M}}(v) \geq \rho^r$. Since $\ell(r) < \rho^r/\sqrt{d}$,

the diameter of an r-tile is less than ρ^r, and thus v and u cannot be in the same r-tile. Since v and u have the same color, v and u are far apart, more precisely, $|vu| \geq (\kappa - 1)\ell(r) > 3\beta\rho^{r+1}$. The fact that our construction algorithm creates these operations implies $\mathrm{NN}_{\mathscr{M}}(v), \mathrm{NN}_{\mathscr{M}}(u) < \rho^{r+1}$. Then, the (ρ, β) picking regions and the certificate regions of the β-clipped Voronoi cells of v and u are inside balls of radii $\beta\rho^{r+1}$ and $2\beta\rho^{r+1}$ around these vertices respectively. Using the triangle inequality, we know that that the (ρ, β) picking region of one of them does not intersect the certificate region of the β-clipped Voronoi cell of the other; therefore, $v_{(t)}$ and $u_{(t)}$ are independent. □

We partition the nodes of the union graph $G^{\cup} = (V^{\cup}, E^{\cup})$ into several categories. The nodes $V^- = V \setminus \mathsf{range}(\mu)$ are called *obsolete* (squares Σ^-, operations Ω^-); these are the nodes of G that have no matching pairs in G'. The nodes $V^+ = V' \setminus \mathsf{dom}(\mu)$ are called *fresh* (squares Σ^+, operations Ω^+); these are the nodes of G' that have no matching pairs in G. Furthermore, we call a square $s \in V^{\cup}$ *inconsistent* if it is fresh or obsolete, or if it contains the vertex $\hat{v}$ of the symmetric difference of N and N'. We define an operation $v_{(t)} \in \mathsf{range}(\mu)$ to be *inconsistent* if it is reachable from an inconsistent square via a dependency path. We represent inconsistent nodes with $V^{\times}$ (squares $\Sigma^{\times}$, operations $\Omega^{\times}$). We define the *distance* between the executions with related inputs N and N' to be the number of obsolete, fresh, or inconsistent operations of the union graph, i.e., $|\Omega^- \cup \Omega^+ \cup \Omega^{\times}|$.

Lemma 9.2. *For every operation in $\Omega^- \cup \Omega^+ \cup \Omega^{\times}$, there exists a dependency path from a square in $\Sigma^{\times}$.*

Proof. By definition, an inconsistent operation can be reachable via a dependency path from $\Sigma^{\times}$. For unmatched operations, assume towards a contradiction that there exist an operation in $\Omega^- \cup \Omega^+$ that is not reachable from $\Sigma^{\times}$. Let $v_{(t)}$ be the earliest of such operations. Let us assume that $v_{(t)}$ represents a dispatch operation, and that v is an input vertex. Since $v_{(t)}$ does not depend on an inconsistent square, it does not read one. Therefore, v is in $\mathsf{N} \cap \mathsf{N}'$ and lies in identical squares in both executions, which implies that its nearest neighbor approximation is the same in both executions. Hence, there exists an operation $v_{(t)}$ in the other execution as well. The definition of μ matches these operations because in both computation graphs contain the edge $v_{(0)} \longrightarrow v_{(t)}$. For the remaining cases, there exists an edge $w_{(\tau)} \longrightarrow v_{(t)}$ for some unmatched or inconsistent operation $w_{(\tau)}$ with $\tau < t$. By the minimality of $v_{(t)}$, $w_{(\tau)}$ can be reached via a dependency path from a square in $\Sigma^{\times}$. Extending that path to $v_{(t)}$ proves the contradiction. □

As proven in the previous section, the function `QTClippedVoronoi` satisfies the following locality property: for a given input N, a size-conforming set of vertices $\mathscr{M} \supset \mathsf{N}$, and a square s read by `QTClippedVoronoi`, for all $x \in s$, $|vx| \in O(\mathrm{NN}_{\mathscr{M}}(v))$. This property allows us to relate the operations on a dependency path geometrically.

Lemma 9.3. *Consider two operations $w_{(\tau)}$ and $v_{(t)}$ in $G^{\cup}$. If there exists a dependency path from $w_{(\tau)}$ to $v_{(t)}$ and rank of t is r, then $|vw| \in O(\rho^r)$.*

Proof. First, we show that for any edge in $G^{\cup}$, the distance between its nodes is short. We define the distance between a square and an operation to be the distance from the vertex of the operation to the farthest point in the square, and the distance between two operations to be the distance between their vertices. Consider an edge $e \in E$ with time t_e whose rank is r_e. The edge e consists of an operation $u_{(t_e)} \in \Omega$ and either a square s that $u_{(t_e)}$ accesses (reads/writes) or another operation $u'_{(t')}$ that it schedules. Using the locality result stated prior to this lemma, we bound the distance between $u_{(t_e)}$ and s by $O(\mathrm{NN}_{t_e}(u))$. Also, $u'_{(t')}$ is within the same distance. Lemmas 8.1 and 8.2 bound $\mathrm{NN}_{t_e}(u)$ by $O(\rho^{r_e})$; thus, the distance between the nodes of e is at most $\alpha\rho^{r_e}$, where α is a constant in the big-Oh notation. The same analysis applies for any edge $e' \in E^{\cup}$.

By the definition of dependency paths, the times of the edges on a dependency path from $w_{(\tau)}$ to $v_{(t)}$ monotonically increase. Assuming that the rank of τ is r', there can be at most κ^d edges for each rank between r' and r. Therefore, in the worst case, the distance between v and w is bounded by $\sum_{i=r'}^{r} \kappa^d \alpha\rho^i = \alpha\kappa^d \frac{\rho^{r+1}-\rho^{r'}}{\rho-1} < \alpha\kappa^d \frac{\rho^{r+1}}{\rho-1}$. Consequently, $|vw| \in O(\rho^r)$. □

In order to bound the distance between the executions with inputs N and N' which generate outputs M and M', we focus on the vertices rather than the operations. We define a vertex to be *affected* if there exists an obsolete, a fresh, or an inconsistent operation of it. Since there is a constant number of operations acting on a given vertex (Lemma 8.6), the number of affected vertices measures the distance asymptotically. We define the sets of affected vertices in both executions: $\widehat{\mathsf{M}} = \{v \mid v_{(t)} \in \Omega^{-} \cup \Omega^{\times}\}$ and $\widehat{\mathsf{M}}' = \{v \mid v_{(t)} \in \Omega^{+} \cup \Omega^{\times}\}$. The next two lemmas bound the number of affected vertices.

Lemma 9.4. *For any vertex $v \in \widehat{\mathsf{M}}$, $|v\hat{v}| \in O(\mathrm{NN}_{\mathsf{M}}(v))$ and for any $v \in \widehat{\mathsf{M}}'$, $|v\hat{v}| \in O(\mathrm{NN}_{\mathsf{M}'}(v))$.*

Proof. We prove the lemma for $v \in \widehat{\mathsf{M}}$; symmetric arguments apply for $\widehat{\mathsf{M}}'$. By definition of $\widehat{\mathsf{M}}$, there exists an operation $v_{(t_v)} \in \Omega^- \cup \Omega^\times$ at rank r. Lemma 9.2 suggests that there exists a dependency path from a square $s \in \Sigma^\times$ to $v_{(t_v)}$. Let $s \longrightarrow u_{(t_u)}$ be the first edge on this path, where the rank of t_u is r_u. By Lemma 9.3, we know that $|vu| \in O(\rho^r)$. By the fact that the operation that $u_{(t_u)}$ represents reads s, we know $|us|$ is in $O(\rho^{r_u})$ and by lemmas 4.1 and 4.2 the quadtree functions `QTInsertInput` and `QTDeleteInput` guarantee that $|s\hat{v}| \in O(|s|)$ which is in $O(\rho^{r_u})$ as well. Using the triangle inequality and the fact that $r_u \leq r$, we bound $|v\hat{v}|$ by $O(\rho^r)$. It only remains to prove that there is a ball around v of radius $\Omega(\rho^r)$ empty of vertices of M. Lemma 8.2 proves precisely this. □

Lemma 9.5 (Distance)**.** *The distance between two executions with related inputs is bounded by* $O(\log \Delta)$.

Proof. The distance is asymptotically bounded by the sizes of the affected sets of vertices $|\widehat{\mathsf{M}}|$ and $|\widehat{\mathsf{M}}'|$. Consider the vertices $v \in \widehat{\mathsf{M}}$ with $|v\hat{v}| \in [2^i, 2^{i+1})$. By Lemma 9.4, we can assign non-overlapping empty balls of radius $\Omega(2^i)$ to them. Therefore, there is a constant number of such vertices for any i. At most $O(\log \Delta)$ values of i cover $\widehat{\mathsf{M}}$, so $|\widehat{\mathsf{M}}| \in O(\log \Delta)$. Similar arguments apply to $\widehat{\mathsf{M}}'$. □

10. Dynamic Update Algorithm

We describe an algorithm for dynamically updating the output of `StableWS` when the input is modified by insertion/deletion of a vertex, prove it correct (Lemma 10.2) and efficient (Theorem 10.3).

Our dynamic update algorithm is a change-propagation algorithm. Given the input modification, the update algorithm re-executes the actions of the stable algorithm for the part of the computation affected by the modification and undoes the part of the computation that becomes obsolete. More precisely, the algorithm maintains distinct set of operations for removal $\Omega^\ominus$ (obsolete operations), for execution $\Omega^\oplus$ (fresh operations), and for re-execution $\Omega^\otimes$ (inconsistent operations), which contain operations representing the operations that become obsolete, that need to be executed, and that become inconsistent respectively. The inconsistent operations are updated by deleting their old versions and executing them again, which may now perform actions different than before. The algorithm removes and executes operations in the same order as the stable algorithm. It uses the `Undo` function to remove obsolete operations and the `Dispatch` and `Fill` functions of the stable algorithm for executing fresh operations.

```
Global queues:  Ω^⊖, Ω^⊕, Ω^⊗

PropagateWS (Σ^−, v̂) =
  for each s ∈ Σ^− ∪ {square of v̂}
    MarkReaders(s, 0)
    for each input vertex v ≠ v̂ ∈ s
      Undo(v_(0))
      apxnnv ← |square of v|
      Enqueue(v, D, apxnnv, v_(0), Ω^⊕)
  r_min ← min rank in Ω^⊖ ∪ Ω^⊕ ∪ Ω^⊗
  for r = r_min to ⌊log_ρ √d⌋
    for each v_(r,D) ∈ Ω^⊖ ∪ Ω^⊗
      Undo(v_(r,D))
    for each v_(r,D) ∈ Ω^⊕ ∪ Ω^⊗
      Dispatch(v_(r,D), Ω^⊕)
    for c = 0 to κ^d − 1
      for each v_(r,F,c) ∈ Ω^⊖ ∪ Ω^⊗
        Undo(v_(r,F,c))
      for each v_(r,F,c) ∈ Ω^⊕ ∪ Ω^⊗
        Fill(v_(r,F,c), Ω^⊕)
        for each Steiner w inserted by v
          MarkReaders(square of w, (r,F,c))

MarkReaders (s,t) =
  for each edge s ⟶ v_(t_v)
    if t_v > t then Ω^⊗ ← Ω^⊗ ∪ {v_(t_v)}

Insert (Q, v̂) =
  (Q′, Σ^−) ← QTInsertInput(Q, v̂)
  appnnv̂ ← |square of v̂|
  Enqueue(v̂, D, apxnnv̂, v̂_(0), Ω^⊕)
  PropagateWS(Σ^−, v̂)
  return Q′

Delete (Q, v̂) =
  (Q′, Σ^−) ← QTDeleteInput(Q, v̂)
  Undo(v̂_(0))
  PropagateWS(Σ^−, v̂)
  return Q′

Undo (v_(t)) =
  for each edge s ⟶ v_(t)
    CGDeleteEdge(s ⟶ v_(t))
  for each edge v_(t) ⟶ w_(t_w)
    CGDeleteEdge(v_(t) ⟶ w_(t_w))
    if ∄ edge · ⟶ w_(t_w) then
      Ω^⊖ ← Ω^⊖ ∪ {w_(t_w)}
    if t = (r,F,c) then
      s_w ← square of w
      MarkReaders(s_w, t)
      CGDeleteEdge(v_(t) ⟶ s_w)
  if v_(t) ∈ Ω^⊖ then
    Ω^⊗ ← Ω^⊗ \ {v_(t)}
```

Figure 9: The pseudo-code of the dynamic algorithm.

Figure 9 shows the pseudo-code for the `Insert` and `Delete` functions for inserting and deleting a vertex $\hat{v}$ into and from the input, and the `PropagateWS` function for dynamic updates. Given $\hat{v}$, `Insert`/`Delete` updates the quadtree, determines the set of inconsistent squares $\Sigma^{\otimes}$, and initializes the fresh/obsolete set by creating a dispatch operation or by marking the old dispatch operation acting on $\hat{v}$. Both functions then call `PropagateWS`.

The `PropagateWS` function starts by updating the operation sets by finding the input vertices that are contained in the inconsistent squares, deleting their dispatch operations, and creating new dispatch operations for them. It also initializes the inconsistent operation set, as `MarkReaders` marks inconsistent all operations that read an inconsistent square. The algorithm then proceeds in time order, first undoing the obsolete and inconsistent operations and then performing the fresh and inconsistent operations by calling `Dispatch` and `Fill` (Figure 7).

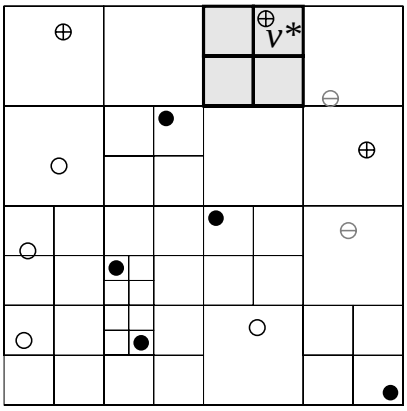


Figure 10: Dynamic update after insertion of $\hat{v}$. Solid vertices are input (N), vertices marked $+$ are inserted, vertices marked $-$ are deleted. Gray squares are inconsistent. The four smaller gray squares are fresh; they replace the bigger obsolete square.

The `Undo` function undoes the work of obsolete and fresh operations by removing the Steiner vertices they insert (if any) and by removing the edges incident to them (dependencies) from the computation graph. While removing these edges, it also marks for removal the operations that lose their last incoming edge from other operations—these operations would not be created in a fresh execution with the modified input. The `Undo` function also calls the `MarkReaders` function to expand the set of inconsistent operations as the set of vertices in a square changes due to the removal of Steiner vertices. After `Undo` finishes its work and as the algorithm executes fresh fill operations, it calls the `MarkReaders` function for a similar reason: to update the set of inconsistent operations due to the insertion of fresh Steiner vertices.

As their notation suggests, the obsolete, fresh, and inconsistent operations used by the algorithm are related to those defined in the stability analysis; the following lemma makes the relationship between them precise.

Lemma 10.1. *The set of operations processed in the dynamic update algorithm, $\Omega^{\ominus} \cup \Omega^{\oplus} \cup \Omega^{\otimes}$, is a subset of the set of obsolete, fresh, and inconsistent operations, $\Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}$.*

Proof. Let $A = \Omega^{\ominus} \cup \Omega^{\oplus} \cup \Omega^{\otimes}$ and $B = \Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}$. Towards a contradiction, assume that $A \not\subset B$ and let $v_{(t)}$ be the earliest operation in $A \setminus B$. If $v_{(t)} \in \Omega^{\ominus}$ then either $v_{(t)}$ is a dispatch operation acting on an input vertex or there is an edge from another operation $w_{(\tau)} \in \Omega^{\ominus} \cup \Omega^{\otimes}$ towards $v_{(t)}$. In the first case, $v_{(t)}$ depends on a square in $\Sigma^{\times}$, which implies $v_{(t)} \in B$. In the second case, by minimality of t, since $\tau < t$, $w_{(\tau)} \in B$. Lemma 9.2 implies that $w_{(\tau)}$ is reachable from a square in $\Sigma^{\times}$ by a dependency path, therefore, $v_{(t)}$ is also reachable using dependency paths. Then $v_{(t)}$ must be in B, either because it is inconsistent or because there it has

no matching operation. Similar arguments show that $v_{(t)} \in \Omega^{\oplus}$ implies $v_{(t)} \in B$. Therefore $v_{(t)}$ must be in $\Omega^{\otimes}$, i.e., $v_{(t)}$ reads a square s for which the algorithm calls the function `MarkReaders` (s,t') with a time $t' < t$. If $s \in \Sigma^{\otimes}$ then clearly $v_{(t)} \in B$; otherwise, there is another operation $v'_{(t')}$ that writes into s. Again, by minimality of $v_{(t)}$, $v'_{(t')} \in B$ and by Lemma 9.2 there exists a dependency path from $v'_{(t')}$ to $v_{(t)}$ which puts $v_{(t)}$ in B. Contradiction. □

When completed, `PropagateWS` updates the output to $\tilde{\mathsf{M}}$ and the computation graph to $\tilde{G}$ as if `StableWS` is run from-scratch with N' as input, computing M' and G'.

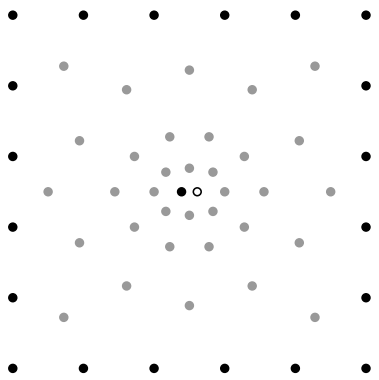
Lemma 10.2 (Isomorphism)**.** *The output sets $\tilde{\mathsf{M}}$ and M' are equal and there exists an isomorphism $\phi : \tilde{G} \to G'$ that preserves the vertex and time of each operation.*

Proof. We prove equality of the output and build ϕ by induction on time. Define the following sets of operations: $\Omega_t^{\ominus} = \{v_{(\tau)} \in \Omega^{\ominus} \mid v_{(\tau)} \text{ is created at time} < t\}$ based on their creation times. ($\Omega_0^{\ominus}$ is the set of dispatch operations acting on input vertices). Also, define a similar assemblage for the $^{\oplus}$, $^{\otimes}$, and $'$ sets. Let $\tilde{G}_t$ be the subgraph of $\tilde{G}$ induced by the nodes $\tilde{\Omega}_t \cup \tilde{\Sigma}$ excluding the edges with time $\geq t$; the excluded edges are related to the execution of operations at time $\geq t$. Define G'_t similarly and let $\tilde{\mathsf{M}}_t$ be the updated set of vertices obtained by removing and inserting vertices until time t, just before the executing operations at time t.

Initially, $\tilde{\mathsf{M}}_0 = \mathsf{M}'_0 = \mathsf{N}'$ and $\tilde{\Sigma} = \Sigma'$. Therefore, there exists an isomorphism $\phi_0 : \tilde{G}_0 \to G'_0$. Assume the inductive hypothesis at time t, that $\tilde{\mathsf{M}}_t = \mathsf{M}'_t$ and that we have an isomorphism $\phi_t : \tilde{G}_t \to G'_t$. Pick $op \in \tilde{\Omega}_t$ with time t and let $op' = \phi_t(op)$. We aim to prove that op and op' execute in the same way. Because our functions are all deterministic, it suffices to show that op and op' read the same data. There are three cases: op is either in $\Omega_t^{\oplus}$, or in $\Omega_t^{\otimes}$, or otherwise op is an operation that has not been modified.

Assume that op is in $\Omega_t^{\oplus}$. We know $\tilde{\Sigma} = \Sigma'$, therefore, op and op' traverse the same quadtree structure in their execution. For a vertex v that op reads, v cannot be in $\mathsf{M}_t^{\ominus}$ because the vertices in $\mathsf{M}_t^{\ominus}$ are removed at time $< t$. Thus, op reads only the vertices in $\tilde{\mathsf{M}}_t = \mathsf{M}'_t$, in other words op reads the same data as op' does. The case that $op \in \Omega_t^{\otimes}$ is similar, because the re-execution of the inconsistent operations follow the same rules. In the remaining case, op is not modified. Consider a square s that op accesses. Because the update algorithm did not schedule op for re-execution, we know that s is not in Σ^{-}. Furthermore, for the same reason, s does not contain a vertex in $\mathsf{M}_t^{\ominus} \cup \mathsf{M}_t^{\oplus}$. Therefore, op only reads vertices in $\mathsf{M}'_t \cap \mathsf{M}_t$; op reads the same data as op' does. Hence, in all cases, op and op' execute similarly.

Figure 11: Inserting the dynamic vertex x (the un-filled black point) into the set of solid black points creates $\Omega(\log \Delta)$ fresh Steiner vertices (the gray points).



We have a natural correspondence between the operations that op and op' create and the Steiner vertices they insert (in any). Therefore, $\tilde{\mathsf{M}}_{t+1} = \mathsf{M}'_{t+1}$. Furthermore, because op and op' read and write the same squares the edges incident to these operations have natural correspondences as well. Extending ϕ_t to ϕ_{t+1} by adding these correspondences completes proof of the inductive step. □

Theorem 10.3. *The* `Insert` *and* `Delete` *functions modify the output in $O(\log \Delta)$ time and maintain a ρ-well-spaced output of optimal-size with respect to the updated input.*

Proof. By Lemma 10.2, we know that the output is the same as what would have been generated by executing from scratch `StableWS` with the new input, therefore, Theorem 7.4 applies. As discussed in Section 4, the quadtree can be updated in $O(\log \Delta)$ time. Also, Lemma 10.1 relates the runtime of the update algorithm to the distance between the executions with the old and new inputs. Finally, Lemma 9.5 bounds the runtime of `PropagateWS` as desired. □

11. Lower bound

We present a lower bound proving that any algorithm which explicitly maintains a well-spaced superset requires $\Omega(\log \Delta)$ time per dynamic update. Consider dynamically inserting a new point very close to an existing input vertex. Even the optimal dynamic algorithm is forced to insert geometrically growing rings of new Steiner vertices around the dynamically inserted vertex. We prove that we can iterate this process using a gadget. This shows that our algorithm is worst-case optimal compared to all other explicit algorithms, even in an amortized setting.

We define a gadget (see Figure 11) consisting of points in the hypercube $[0, n^{-1/d}]^d$. Consider two vertices at distance $\delta = 1/\Delta$ from each other in the

middle of the box; let one of them be the *dynamic vertex* x which will be inserted later. Also, consider a grid of $O(1)$ vertices on each of the faces of the hypercube, chosen according to the scheme of Hudson [15, p.79]. The input N consists of tiling $[0,1]^d$ with the gadgets, $n^{1/d}$ for each dimension, without any dynamic vertex. The dynamic modification sequence consists of inserting n dynamic vertices, one for each gadget.

Lemma 11.1. *Inserting the dynamic vertex to a single gadget requires inserting* $\Omega(\log\Delta)$ *Steiner vertices.*

Proof. Let N be the input before adding the dynamic vertex x. Any size-optimal output M of N has $O(1)$ Steiner vertices inside the gadget box. Consider inserting x and let $\mathsf{N}' = \mathsf{N} \cup \{x\}$ and $\delta = \mathrm{NN}_{\mathsf{N}'}(x)$. Draw the segment from x to the farthest point in $\mathrm{Vor}_{\mathsf{N}'}(x)$. This segment has length at least $\ell = \Omega(n^{-1/d})$. Consider the Voronoi diagram of a ρ-well-spaced superset M' of N' and consider the Voronoi cells that this segment cuts. Let $v_1, v_2, \ldots$ be the vertices of those Voronoi cells, in order. We know that the vertices in M' are ρ-well-spaced, therefore, $|v_1 x| \le 2\rho\,\mathrm{NN}_{\mathsf{N}'}(x) = 2\rho\delta$. Also, the nearest neighbor distance of v_1 is at most $|v_1 x|$. We can use the same argument to get $|v_1 v_2| \le 2\rho|v_1 x|$ and repeat. In other words, distance from x grows only geometrically as we walk down the segment: covering the distance ℓ requires $\Omega(\log n^{-1/d}/\delta) = \Omega(\log n^{-1/d}\Delta)$ many Steiner vertices. Choosing δ to be sufficiently small, e.g., $\delta = O(n^{-(1+\varepsilon)/d})$ for some constant $\varepsilon > 0$, we have $\Omega(\log n^{-1/d}\Delta) = \Omega(\log n^{\varepsilon/d}) = \Omega(\log\Delta)$. This implies that M differs from M' in at least $\Omega(\log\Delta)$ vertices. □

Theorem 11.2 (Lower Bound)**.** *There exists an initial input and a set of n dynamic insertions that forces any algorithm to insert* $\Omega(n\log\Delta)$ *new Steiner vertices.*

Proof. In the above scheme, we would like to prove that inserting n dynamic vertices requires inserting $\Omega(n\log\Delta)$ Steiner vertices. We refer to a technique of inserting vertices to the hypercube faces [15]. It was developed precisely to make sure that certain algorithms need not add vertices outside the hypercube when making the interior ρ-well-spaced. Contrapositively, adding vertices outside a gadget does not help make the gadget, with its dynamic vertex, be ρ-well-spaced. Thus the prior lemma applies to each gadget individually, showing that the final ρ-well-spaced superset must contain at least $\Omega(n\log\Delta)$ Steiner vertices, for a carefully selected ρ. Since there exists a constant $\rho > 1$ such that the original input of n gadgets is ρ-well-spaced, the initial output must be of size $O(n)$. This completes our proof. □

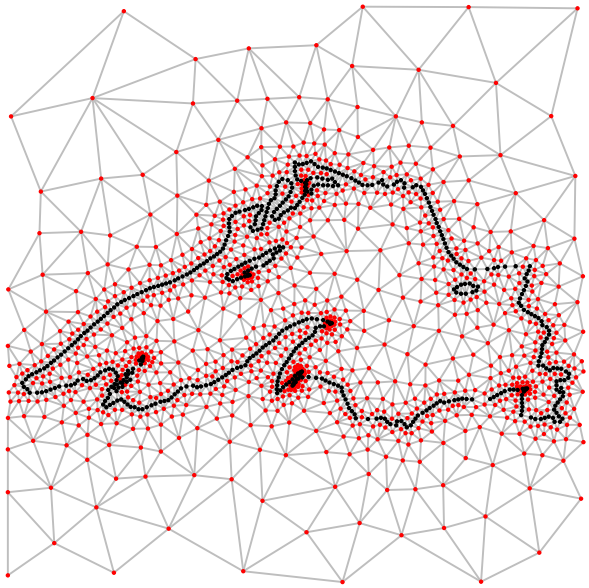


Figure 12: A model of Lake Superior meshed by `StableWS`.

12. Experiments

We have implemented[1] the proposed algorithm, and performed an experimental analysis using both synthetic and real datasets with parameters $\rho = \sqrt{2}$, and $\beta = 2$ in 2D or $\beta = 2\sqrt{2}/\sqrt{3}$ in 3D. As we report below, our experiments confirm our asymptotic bounds and give strong evidence that they can be realized efficiently in practice. Specifically, considering synthetic datasets, we show that the practical efficiency of our algorithm matches the theoretical analysis. We also consider real data sets of varying sizes in 2D and 3D, where our algorithm handles dynamic changes significantly faster than a full recomputation.

In all experiments, we measure both the time required for generating a well-spaced superset from scratch, and for performing point insertions into and deletions from this set. We report wall-clock times for from-scratch runs, and speedups, measured relative to these times, for point changes. Our testing machine has an Intel Core i5 750 CPU, 8G of memory, and runs Ubuntu 10.04.

We note that developing practically efficient mesh generators is a huge engi-

[1] `http://ttic.uchicago.edu/~cotter/projects/dynamic_wsp`

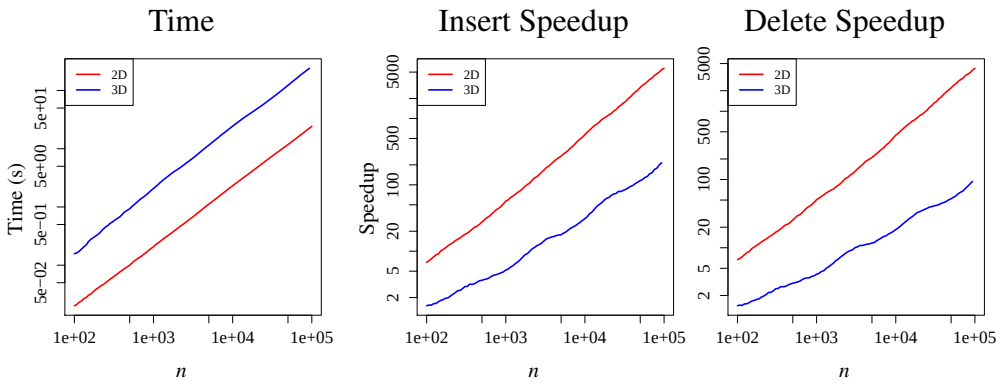


Figure 13: Left: time, in seconds, required to generate a well-spaced superset of a size-n set of uniformly random inputs. Right: speedups, relative to the time required by a from-scratch run, observed for random single-point insertions into and deletions from an existing well-spaced point set. All plots are in log-log scale.

neering challenge (e.g., Triangle [28, 30] in 2D and SVR [16, 5] in 3D), especially because of the care needed to ensure that arithmetic operations on floating-point numbers can be performed accurately and efficiently. Such an engineering effort is beyond the scope of this paper; we simply wish to confirm that there are no major obstacles in making our algorithm practical. In our implementation we therefore employ exact floating-point arithmetic operations with some simple, easy to implement optimization.

12.1. Uniform random data

In our first set of experiments, we generated uniformly random data of various sizes and dimensions, and measured the time required for from-scratch runs, and point changes. The results of these experiments are reported in Figure 13. As can be seen in the left-hand plot, the runtime required to create a well-spaced superset from scratch using our algorithm is roughly linear in the input size n, indicating that our algorithm scales very well. Theorem 8.7 upper bounds the runtime as $O(n \log \Delta)$, and while $\log(\Delta)$ is an increasing function of n for uniformly random data, the effects of this additional term are not noticeable in these experiments.

Theorem 10.3 indicates that the time required for each point change, whether an insertion or deletion, should be $O(\log \Delta)$, indicating that we should experience a speedup for each dynamic change, over re-running from scratch, which is linear in n. The two right-hand plots in figure 13 illustrate the speedups, relative to

Application			Our Implementation			
Dataset	d	Inputs	Outputs	Time	Insert	Delete
SD Bay	2	1823	8469	0.672s	52.5×	47.9×
New Zealand	2	18595	97952	8.69s	483×	417×
Cape Cod	2	20930	85571	7.24s	459×	387×
Lake Superior	2	33487	160004	14.3s	634×	538×
SF Bay	2	85910	341962	30.1s	1710×	1420×
Bunny	3	35947	145018	4.76s	22.9×	18.5×
Armadillo	3	172974	608342	384s	55.1×	42.8×

Table 1: Properties of a number of well-known meshing datasets, as well as the sizes of the well-spaced supersets found by our algorithm, time required to find them, and speedups experienced when performing single-point insertions into or deletions from these well-spaced supersets.

the time of a from-scratch run, and confirm that the observed performance of our algorithm matches these bounds. In fact, the speedups experienced for the largest point sets on which we experimented, containing 100000 points, are quite large, approaching a factor of 5000 in 2D and 100 in 3D.

12.2. Real-world Datasets

In order to explore further the performance of our algorithm, we performed experiments on several well-known 2D and 3D datasets. Table 1 reports the wall-clock time required to find well-spaced supersets, from scratch, of these datasets, as well as the speedups experienced when performing random point insertions or deletions. These results are roughly in line with those of figure 13, and show that the results of these earlier experiments did not depend on the fact that the data was taken to be uniformly random.

13. Conclusion

We present a dynamic algorithm for computing a well-spaced superset of a dynamically changing set of input points. Our algorithm is efficient, finds an optimal-size output, consumes linear space, and responds to dynamic modifications in worst-case optimal time. The underlying technique is a stable algorithm for computing well-spaced point sets whose executions can be represented with computation graphs that remain similar when the input sets themselves are similar.

Our dynamic update algorithm takes advantage of stability to update the output efficiently by propagating the input modification through the computation graph. To assess the practicality of our approach we present a prototype implementation. Our experiments show that the algorithm can be implemented efficiently such that it delivers performance consistent with our theoretical bounds. We expect a well-polished implementation will provide static performance comparable to the state of the art, and dynamic performance orders of magnitude faster.

References

[1] Acar, U.A., 2005. Self-Adjusting Computation. Ph.D. thesis. Department of Computer Science, Carnegie Mellon University.

[2] Acar, U.A., Blelloch, G.E., Blume, M., Tangwongsan, K., 2006. An experimental analysis of self-adjusting computation, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.

[3] Acar, U.A., Blelloch, G.E., Tangwongsan, K., Türkoğlu, D., 2008. Robust Kinetic Convex Hulls in 3D, in: 16th Annual European Symposium on Algorithms.

[4] Acar, U.A., Cotter, A., Hudson, B., Türkoğlu, D., 2010. Dynamic well-spaced point sets, in: SCG '10: Proceedings of the 26th Annual Symposium on Computational Geometry.

[5] Acar, U.A., Hudson, B., Miller, G.L., Phillips, T., 2007. SVR: Practical engineering of a fast 3D meshing algorithm, in: International Meshing Roundtable, pp. 45–62.

[6] Bern, M., Eppstein, D., Gilbert, J.R., 1994. Provably Good Mesh Generation. Journal of Computer and System Sciences 48, 384–409.

[7] Boissonnat, J.D., Devillers, O., Schott, R., Teillaud, M., Yvinec, M., 1992. Applications of random sampling to on-line algorithms in computational geometry. Discrete Computional Geometry 8, 51–71.

[8] Cheng, S.W., Dey, T.K., Edelsbrunner, H., Facello, M.A., Teng, S.H., 2000. Sliver Exudation. Journal of the ACM 47, 883–904.

[9] Chentanez, N., Alterovitz, R., Ritchie, D., Cho, L., Hauser, K.K., Goldberg, K., Shewchuk, J.R., O'Brien, J.F., 2009. Interactive simulation of surgical needle insertion and steering, in: Proceedings of ACM SIGGRAPH.

[10] Chew, L.P., 1989. Guaranteed-quality triangular meshes. Technical Report TR-89-983. Department of Computer Science, Cornell University.

[11] Clarkson, K.L., Mehlhorn, K., Seidel, R., 1993. Four results on randomized incremental constructions. Computational Geometry Theory and Application 3, 185–212.

[12] Coll, N., Guerrieri, M., Sellarès, J.A., 2006. Mesh modification under local domain changes, in: 15th International Meshing Roundtable, pp. 39–56.

[13] Hammer, M.A., Acar, U.A., Chen, Y., 2009. CEAL: A C-based language for self-adjusting computation, in: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation.

[14] Har-Peled, S., Üngör, A., 2005. A time-optimal Delaunay refinement algorithm in two dimensions., in: 21st Symposium on Computational Geometry, pp. 228–236.

[15] Hudson, B., 2007. Dynamic Mesh Refinement. Ph.D. thesis. School of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-07-162.

[16] Hudson, B., Miller, G.L., Phillips, T., 2006. Sparse Voronoi Refinement, in: 15th International Meshing Roundtable, pp. 339–356. Long version in Carnegie Mellon University Tech. Report CMU-CS-06-132.

[17] Hudson, B., Türkoğlu, D., 2008. An efficient query structure for mesh refinement, in: Canadian Conference on Computational Geometry.

[18] Jampani, R., Üngör, A., 2007. Construction of sparse well-spaced point sets for quality tetrahedralizations, in: IMR, pp. 63–80.

[19] Ley-Wild, R., Acar, U.A., Fluet, M., 2009. A cost semantics for self-adjusting computation, in: Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages.

[20] Li, X.Y., Teng, S.H., 2001. Generating well-shaped Delaunay meshes in 3D, in: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 28–37.

[21] Li, X.Y., Teng, S.H., Üngör, A., 1999. Simultaneous refinement and coarsening for adaptive meshing. Engineering with Computers 15, 280–291.

[22] Miller, G.L., Talmor, D., Teng, S.H., Walkington, N., Wang, H., 1996. Control Volume Meshes Using Sphere Packing: Generation, Refinement and Coarsening, in: 5th Intl. Meshing Roundtable, pp. 47–61.

[23] Molino, N., Bao, Z., Fedkiw, R., 2004. A virtual node algorithm for changing mesh topology during simulation, in: SIGGRAPH.

[24] Mulmuley, K., 1991. Randomized multidimensional search trees (extended abstract): dynamic sampling, in: Proceedings of the 7th Annual Symposium on Computational Geometry, pp. 121–131.

[25] Nienhuys, H.W., van der Stappen, A.F., 2004. A Delaunay approach to interactive cutting in triangulated surfaces, in: fifth Intl. Workshop on Algorithmic Foundations of Robotics.

[26] Ruppert, J., 1995. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. J. Algorithms 18, 548–585.

[27] Schwarzkopf, O., 1991. Dynamic maintenance of geometric structures made easy, in: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, pp. 197–206.

[28] Shewchuk, J.R., 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. volume 1148. Springer-Verlag, Berlin.

[29] Shewchuk, J.R., 1998. Tetrahedral mesh generation by Delaunay refinement, in: SCG '98: Proceedings of the Fourteenth Annual Symposium on Computational Geometry, ACM Press, New York, NY, USA. pp. 86–95.

[30] Shewchuk, J.R., 2002. Delaunay refinement algorithms for triangular mesh generation. Computational Geometry: Theory and Applications 22, 21–74.

[31] Spielman, D., Teng, S.H., Üngör, A., 2007. Parallel Delaunay refinement: Algorithms and analyses. IJCGA 17, 1–30.

[32] Talmor, D., 1997. Well-Spaced Points for Numerical Methods. Ph.D. thesis. Carnegie Mellon University. Available as Technical Report CMU-CS-97-164.